5-2023

A Bit-Exact Hierarchical SystemC Model of pPIM for Verification and Performance Modeling

Dhana Lavanya Balasubramanian
lb1310@rit.edu

A Bit-exact Hierarchical SystemC Model of pPIM for Verification and Performance Modeling

Dhana Lavanya Balasubramanian

A Bit-exact Hierarchical SystemC Model of pPIM for Verification and Performance Modeling

BY

Dhana Lavanya Balasubramanian

A 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

May, 2023

A BIT-EXACT HIERARCHICAL SYSTEMC MODEL OF PPIM FOR VERIFICATION AND PERFORMANCE MODELING

DHANA LAVANYA BALASUBRAMANIAN

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

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

---

Dr. Amlan Ganguly, *Department Head* Date
Professor, Department of Computer Engineering

Dr. Dorin Patru Date
Associate Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Daniel B. Phillips Date
Associate Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Ferat Sahin, *Department Head* Date
Professor, Department of Electrical and Microelectronic Engineering

Dedication

I would like to dedicate this work to my family, my grandparents, my father Balasubramanian, my mother Shyamala, my sister Suganya, my brother-in-law Ram, and my friends Hadlee and Christie for their unconditional love, support, and motivation during my thesis. I would also like to dedicate this thesis to Professor Indovina for patiently teaching me the skills I have today and guiding me.

Dhana Lavanya Balasubramanian

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Thesis is 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 Thesis 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.

Dhana Lavanya Balasubramanian

May, 2023

Acknowledgements

I would like to thank Dr. Amlan Ganguly and Professor Mark Indovina for giving me the opportunity to be part of their research team and providing guidance on my thesis. I would like to express my gratitude to Professor Indovina for all his advice during my master's. He is one of the best professors I have had so far because all of his coursework provides hands-on experience and encourages me to figure things out independently. With extensive industry experience, he has guided me in choosing a good career path for which I am filled with gratitude. I would also like to thank the thesis committee members Dr. Dorin Patru and Dr. Dan Phillips for their invaluable feedback. Special thanks to Purab Sutradhar, Dr. Sai Manoj Pudukotai Dinakarrao, Sathwika Bavikadi, Namita Bhosle, and Allen Su for their work, feedback and participation in this work. Getting to this stage would not have been possible without the support of my family and friends, therefore I would like to thank them.

Dhana Lavanya Balasubramanian

Abstract

The semiconductor industry has seen tremendous advancement lately and is expected to continue to grow since multiple industries like automobile, health care, meteorology, etc are developing dedicated chips with advancements in Artificial Intelligence (AI). These improvements also come with increased complexities. Engineers now require advanced ways to model the designs in addition to existing Hardware Description Languages (HDLs) such as Very High-Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog. One such way to model the designs is by using SystemC, which is a library of C++ classes and macros that can be used similar to an HDL for modeling hardware for functional verification and performance modeling of the design. This paper discusses a bit-exact hierarchical SystemC model written for a multi-core programmable Processor-In-Memory (pPIM) and explores the advantages it has over existing HDLs such as Verilog, SystemVerilog, or VHDL. The pPIM is a Look Up Table (LUT) based Processing In Memory (PIM) architecture that can perform parallel processing with ultra-low-latency for implementing data-intensive applications like Deep Neural Networks (DNN) and Convolution Neural Networks (CNN).

Contents

List of Figures

List of Tables

Glossary

Acronyms

AI	Artifitial Intelligence
ASIC	Application-Specific Integrated Circuit
AsmL	Abstract State Machine Language
CNN	Convolutional Neural Network
DDR4	Double Data Rate 4
DMA	Direct Memory Access
DNN	Deep Neural Network
DPI	Direct Programming Interface
DRAM	Dynamic Random Access Memory
EDA	Electronic Design Automation
FF	Full Function
FIFO	First-In-First-Out

FPGA	Field Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
HDL	Hardware Descriptive Language
HLS	High Level Synthesis
HMC	Hybrid Memory Cube
HPC	High Performance Computing
I/O	Input/Output
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IMC	In-Memory Computing
IoT	Internet of Things
IP	Intellectual Property
LISA	Low-cost Inter-Linked Subarray
LRM	Language Reference Manual
LSTM	Long short-term memory
LUT	Look Up Table
MAC	Multiply Accumulate
mem	Memory subarrays

ML	Machine Learning
MLP	Multi-Layer Perceptron
MP-SoC	Multiprocessor System on Chip
MRAM	Magnetoresistive Random Access Memory
NN	Neural Network
NVM	Non-Volatile Memory
OOP	Object Oriented Programming
PCU	PEI Computation Unit
PEI	PIM-Enabled Instruction
pPIM	Programmable Processing in Memory
ReRAM	Resistive Random Access Memory
RTL	Register Transfer Level
SoC	System on Chip
SOT	Spin-Orbit-Torque
STT-PIM	Spim Transfer Torque - Processing-In-Memory
SystemC-AMS	SystemC Analog/Mixed-Signal
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

Artificial Intelligence is growing at a rapid rate which has a tremendous impact on our lifestyle and work life. Recently, AI has become extremely powerful and is being implemented in various fields such as in automobiles for achieving autonomous driving, in Finance for trading and fraud detection, in healthcare for medical image analysis, as an assistant during surgery, in Agriculture for water monitoring, etc. With the usage of AI proliferating at this rapid rate, complexity increases and dedicated chips are being designed to deal with this growth. To manage this complexity, effective design and verification languages are required to be used that are also cost-effective.

SystemC is one such language that is an extension of C++ that can be used as a Hardware Description Language to design, verify and model different architectures in digital systems. The growth of numerous ideas in research and EDAsectors led to SystemC. SystemC was originally developed by the SystemC Language Working Group and is now maintained by Accellera Systems Initiative, a consortium of leading electronics and semiconductor companies. The industry has used this language widely, especially for the design and verification of complex digital systems like SoCs, FPGAs, and ASICs.

A wide range of modeling structures, such as channels, events, processes, and modules

are offered by SystemC allowing designers to accurately depict the functional and temporal behavior of a digital system. It also supports TLM which enables designers to simulate complex interactions between modules. SystemC's flexibility and efficiency make it an essential tool for digital system designers enabling them to build precise, effective, and scalable models of complex digital systems.

AI chips are often found performing complex calculations for machine learning and deep learning algorithms. These calculations are also required to perform at a very high efficiency. To achieve this, a certain architecture of chip that can do this with low power and low latency is used. A promising strategy for enhancing the performance and energy efficiency in complex digital systems is the PIM architecture. Traditional computing systems are having a difficult time keeping up with the amount of data being generated in terms of data transportation, latency, and energy usage. The PIM architecture is capable of addressing these challenges by integrating processing modules in the memory subsystem.

In comparison to conventional architectures, PIM architecture can provide several advantages. For instance, it can minimize the volume of data that needs to be moved between memory and processor units which will reduce latency and power consumption. Furthermore, PIM can support novel applications and algorithms that were previously impractical because of the constraints of conventional architectures.

Gianfranco Bonanome compared the two HDLs Verilog and SystemC in [15], and the design used to base this comparison is an alarm clock which is a very simple design without any hierarchy. This paper discusses a more complex design with a hierarchy which is a multi-core pPIM architecture for efficient AI processing. The focus of the work the research and development of a bit-exact hierarchical SystemC model of pPIM for verification and performance modeling.

1.1 Research Goals

The goal of this research is to research and model the pPIM architecture in SystemC. The following objectives are the primary objects of this research:

- To understand the LUT base pPIM core and the pPIM Cluster of these cores.
- To modify the existing Verilog testbench for the pPIM core and the pPIM Cluster to make them functional.
- To research the SystemC library and study the semantics required for bit-exact, hierarchical modeling.
- To isolate and model each module in the pPIM core with a stimulus for each module in SystemC.
- To model the pPIM core separately with a stimulus in SystemC.
- To model the pPIM Cluster with a separate stimulus for verification and performance modeling in SystemC.
- The capture trace signal dumps of the system, subsystem, and all the modules to observe and confirm operation using the resulting waveforms.

1.2 Thesis Organization

The structure of the thesis is as follows:

- Chapter 2 provides a literature review of related work in SystemC and PIM architectures
- Chapter 3 gives an overview of SystemC

- Chapter 4 explains the pPIM architecture being modeled in detail.
- Chapter 5 focuses on the pPIM model written in SystemC.
- Chapter 6 provides the results and contributions of the thesis
- Chapter 7 concludes the thesis with areas of improvement.

Chapter 2

Background Research

This Chapter covers background information about SystemC and the PIM architecture.

2.1 Preliminary Research on SystemC

SystemC is a library of C++ [16] classes and macros that can be used as an HDL for modeling hardware designs equipped with object-oriented programming features of SystemC. According to [12], the first version of SystemC was released in September 1999 which was a cycle-based simulator and was similar to other HDLs at that time. March 2000 saw a major release of the library that was widely accessed by engineers. The library went through a lot of bug fixes and improvements the following year. In August 2002, the concept of channels and events was added to SystemC in addition to more lucid syntax. A standard LRM was submitted for IEEE standard review in 2004 and is now standardized over the years. Today, there is widespread use of SystemC for modeling and simulation in the design and verification of digital systems. It is used by both hardware and software engineers and has become an important tool in the development of digital systems. Not requiring any expensive tools like Verilog does make the library very cost-effective.

[17] introduces an approach for formal verification of SystemC designs. SystemC was used because it allows the compilation of hardware models using a standard C++ compiler and simulates efficiently. The SystemC simulation kernel is discussed in this study, which uses the idea of delta-time delays to represent the concurrent operation of hardware systems. These designs are converted into Rebeca models that act as intermediate code and they have been checked by model checkers.

In [18] several techniques for analyzing SystemC designs are explained. These techniques include the evaluation of various parsers and the specifics of their implementations which are primarily used for analyzing static code. Additionally in this paper, several applications that combine static and dynamic analysis are discussed which combine system development before simulation with source code analysis. Other methods include Aspect-Oriented analysis and machine learning. This paper concludes that there is no one best method for SystemC analysis.

[19] presented a library reimplementing SystemC data types to achieve a faster simulation while still retaining semantic equivalence. Type abstraction was used to develop this methodology to increase the simulation performance. This paper is an improvisation to an already fast simulation-producing language SystemC.

[20] explains the need for formal verification tools to be developed for verification using SystemC models. The author points out that having high-level models of microarchitectures of processors is crucial. This paper mostly starts the validation at the micro-architectural level rather than the RTL level. [20] also talks about different formal verification techniques like Explicit-state Model Checking, Assertion-based validation, Symbolic Model Checking, and Symbolic simulation.

[21] designed custom hierarchical channels that can be used in high-level communication. The sc_channel, channel interface, and data payload interface are combined with the SystemVerilog layered testbench. Also discussed is how multiple inheritances in the OOP technique are beneficial

for creating class types that blend the characteristics of two or more class types. Because SystemVerilog OOP techniques do not permit multiple inheritances, the authors use SystemC to design a component of the verification environment that has multiple inheritances. They then use SystemVerilog DPI and ModelSim macro to combine the SystemC design unit with the SystemVerilog-based layered testbench.

[22] proposed semantics for SystemC and offered a method for formal verification of SystemC designs. Razavi et al successfully programmed the MIPS, tested the program that was written on it, and came to the conclusion that the strategy utilized in this work can be used for the verification of designs (both hardware and software) written in SystemC.

In order for hardware and software to be specified in the same language and to be built on a communication architecture that complies with the AMBA bus, [23] created a simulation environment for MP-SoC based on SystemC. It is discovered that this SystemC built environment is a potent tool for MP-SoC design and offers a great degree of flexibility.

[24] discussed several methods ways to implement a high-level reference model for the UVM environment and find that SystemC has the capability to enable replication of the RTL behavior by being equipped with many hardware data types very close to those used on widely used HDL languages and SystemC. The downside found was that a workaround is required to overcome the payload length restriction.

[25] presented a method to design and verify the SystemC model at the transaction level. For efficient verification with assertions in SystemC, a static code analysis technique was carried out that generates an abstract version of the initial design. The SystemC to AsmL transformation was used to reduce the complexity of SystemC models and were able to implement AsmL in Formal verification.

[26] focused on improving SystemC simulation speed for an HLS. To improve the speed of reset parts and enhance busy loops, special waiting methods were applied. It is found that the

SystemC source code required small modification to use the simulation tool FastSim but it can be done quickly. Modification of the SystemC kernel is being carried out to improve the speed of the simulation without having to modify the source code.

[12] mentions that SystemC does not support analog hardware but is a work-in-progress and research groups are working on this. This kind of research is seen in [27] and was able to build a precise mixed-signal SystemC/ SystemC-AMS model the result of which was also verified by contrasting the simulation results given by Matlab model, ADI SimPLL simulations tool, Cadence simulations, and a real chip implementation. The outcomes showed that the SystemC model could accurately and quickly simulate the behavior of the real system. The research also demonstrates the effectiveness and strength of SystemC/SystemC-AMS as a mixed-signal modeling language.

2.2 Preliminary Research on Processing-in-memory (PIM) Architectures

With the advancement in technology and discoveries of successful highly efficient processing methods in computational devices, there is still room for improvement in the area of placement of the subsystems in the design. Contribution to high latency is mostly done by the movement of data from one subsystem to another within the design. This requires an architecture that takes into consideration the distance between the subsystems with optimized placements between the subsystems especially the processing subsystem and the memory subsystem. The near-memory approaches are more efficient compared to traditional approaches where the architecture comprises the cache coupled with processors as discussed in [1]. The author discusses processing options in the memory hierarchy as shown in Figure 2.1. To increase the efficiency even further, PIM architectures have been used in recent times to decrease the power consumption and latency associated with the large-scale data transfer between processor units and memory. For effective

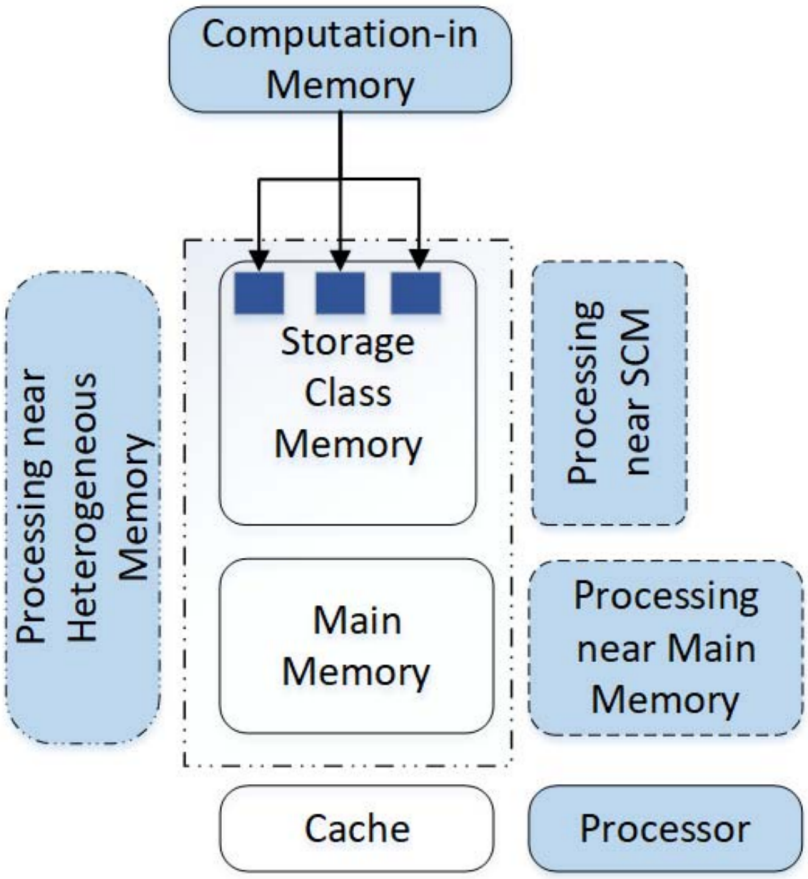

Figure 2.1: Processing options in the memory hierarchy [1]

performance of data-intensive applications, a processing-in-memory architecture is one in which the processing units are positioned close to the memory subsystems. The memory architecture houses the computations. Some of the PIM architectures created by researchers are covered in this section.

[2] present a PIM architecture that is capable of deciding if the PIM operations should be executed in memory or in the processor based on the locality of data. The architecture also does not alter the already in-use sequential programming model. The in-memory computations are invoked by special PIM-enabled instruction which enables PIM operations to be able to

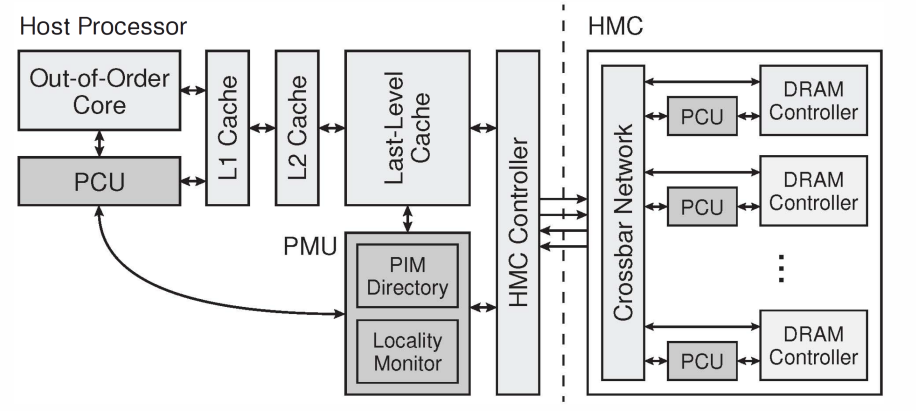

Figure 2.2: Arhcitecture of PIM in [2]

switch between existing programming models, cache coherence protocols, and virtual memory mechanisms with no alterations at all. For finding the locality of data simple hardware is used that is capable of monitoring the data during the execution of PIM-enabled instructions at runtime and switching to processing the computations in the host processor instead of performing the processing inside the memory. Figure 2.2 shows the architecture of the PIM designed in this paper.

The authors have used HMC for their memory technology which comprises a set of multiple DRAM partitions called vaults[2]. As seen in the figure, a DRAM controller is mounted on each vault. The host processor and the HMC communicate based on a packet-based protocol that supports all kinds of operations. The PCU is the PEI computation unit that executes PEIs. This architecture is found to combine the best parts of traditional processor architectures and PIM architectures in terms of performance and power consumption with low overhead.

[3] proposed a PIM architecture with a modification of the memory technology used. The memory technology used in this paper is a Non-volatile memory to overcome the limitation of having to use different metal layers for DRAM and logic technology. The authors claim that one of the NVMs that has the most potential is SOT MRAM. In light of this, this study suggests

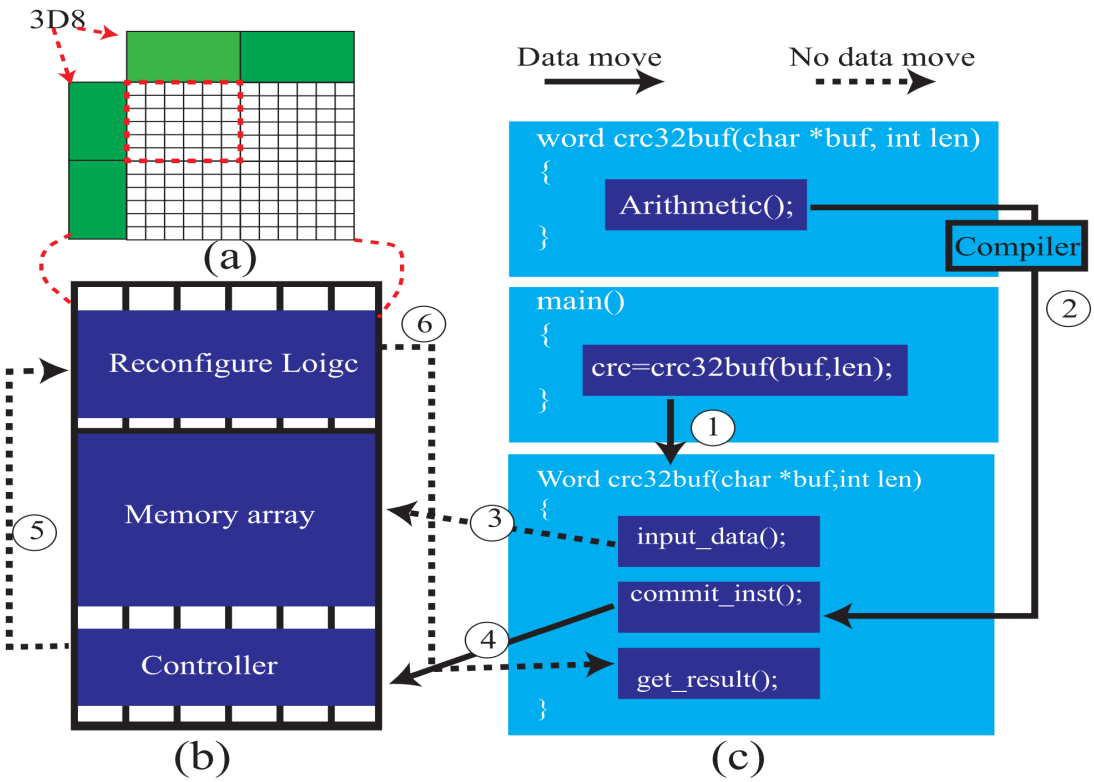

Figure 2.3: Architecture and compile operation of PISOTM [3]

the reconfigurable processing-in-SOT MRAM (PISTOM) architecture. Figure 2.3 shows the architecture and compile operation of CRC32 arithmetic taken as a benchmark program running on PISOTM architecture. 3D8 is a 3-8 decoder. (a) is a small memory array with 3-8 decoders. (b) is the PISOTM architecture and (c) is the CRC32 arithmetic. The architecture consists of traditional SOT memory, reconfigurable SOT-logic, interconnection, and a controller. The PISOTM was compared with DRAM and STT-PIM and found to have better speed than both of them. However, the PISOTM has been put to use only for simpler benchmarks. Research is yet to be done for more complex benchmarks.

In [4], a Silent-PIM that does the PIM computation with conventional DRAM memory requests has been presented. While handling memory requests from programs that do not require PIM,

Silent-PIM enables the PIM memory device to carry out computations without requiring any changes to the hardware. By adhering to the regular DRAM interface, the PIM device can be operated by the user and conduct PIM computations with merely standard memory requests. The Silent-PIM's ability to compute PIM using only conventional DRAM memory requests and without requiring any changes to the hardware is the paper's primary strength. In this architecture, the DDR4 DRAM was configured on FPGA as uncacheable system memory and so it was considered a memory controller on the processor side. As a memory controller, the Xilinx DDR4 was used as it is without any modifications. Doing so allowed the authors to use DMA as the Silent-PIM offloading engine that reduced the memory request overhead significantly. The performance of the Silent-PIM was analyzed by executing it modeled in the FPGA platform, CPU, and GPU using three types of LSTM kernels. The Silent-PIM was found to achieve significant performance in all cases without exploiting data locality. Figure 2.4 shows the architecture of the Silent-PIM.

[5] proposes a method to use the PIM to its full potential. To do this, two steps were taken. First, the data flow in memory was planned so that in-memory computation with a high degree of proximity is possible, and second, computations were made to be dispatched to the regions where the input data are located. The fulfillment of PIM capabilities was made possible by programming models and systems software both of which are essential. Therefore, to draw attention to the issues that the architecture and software communities must resolve to fully realize the potential of PIM. Figure 2.5 shows a high-level point of view of PIM used for HPC. Many instances of the same or similarly configured nodes may make up an overall HPS system[5]. Each node in this system has a high-performance host processor and stacked DRAM with PIM.Indirect memory access techniques like pointer chasing and random gather/scatter are made more effective by switching from read-update-store to op-and-store to prevent synchronization. These are only a few examples of application scenarios and advantages of PIM. This stops receiving data from the

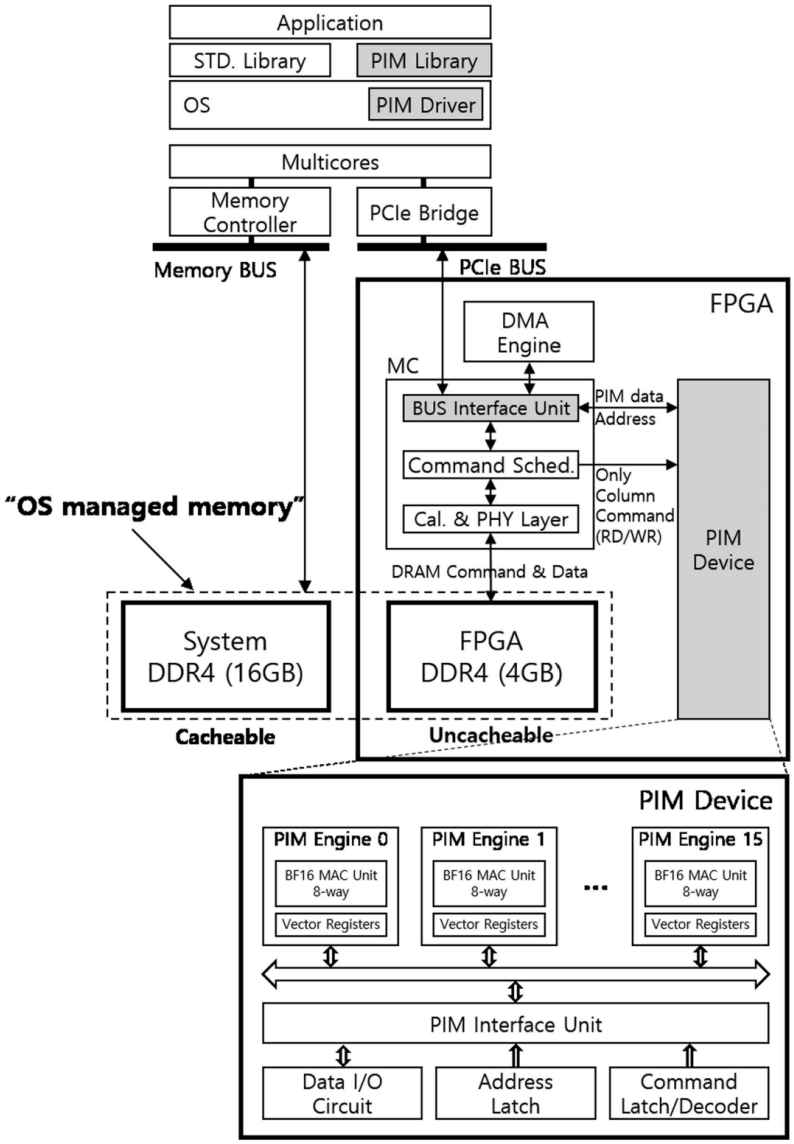

Figure 2.4: Architecture of Silent-PIM [4]

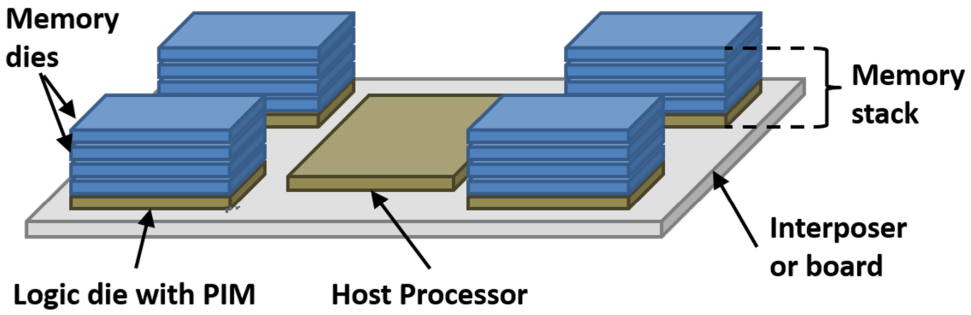

Figure 2.5: Compute node design with PIM [5]

host while predefined memory-intensive processes, such as matrix transpose and convolutions, are being performed.

In [6], the PIM processing units inside the DRAM are designed and operated by successfully coordinating with regular DRAM operations delivering full computing performance and reducing implementation costs. To achieve these objectives, a standard DRAM state diagram was used to show PIM behavior like how standard DRAM commands are scheduled and executed on DRAM devices. Various levels of parallelism were taken advantage of to combine memory and processing activities. By putting these methods to use on the platform developed by the authors, they demonstrate how the operating systems, memory controllers, and PIM devices cooperate for efficient execution. Figure 2.6 shows the PIM architecture discussed in this paper. The architecture has three parts which include a software stack that holds the PIM application, the PIM library and the PIM device driver, second, a memory controller and the third part is the PIM device.

The software stack maps the PIM data in an application to the PIM device. In order to have the PIM device driver unload them into the memory controller, it is necessary to write PIM instructions that are compatible with the PIM library. In terms of performance and energy usage, this PIM architecture worked well.

[7] proposed a PIM architecture called PRIME. This acts as an accelerator for Neural Network

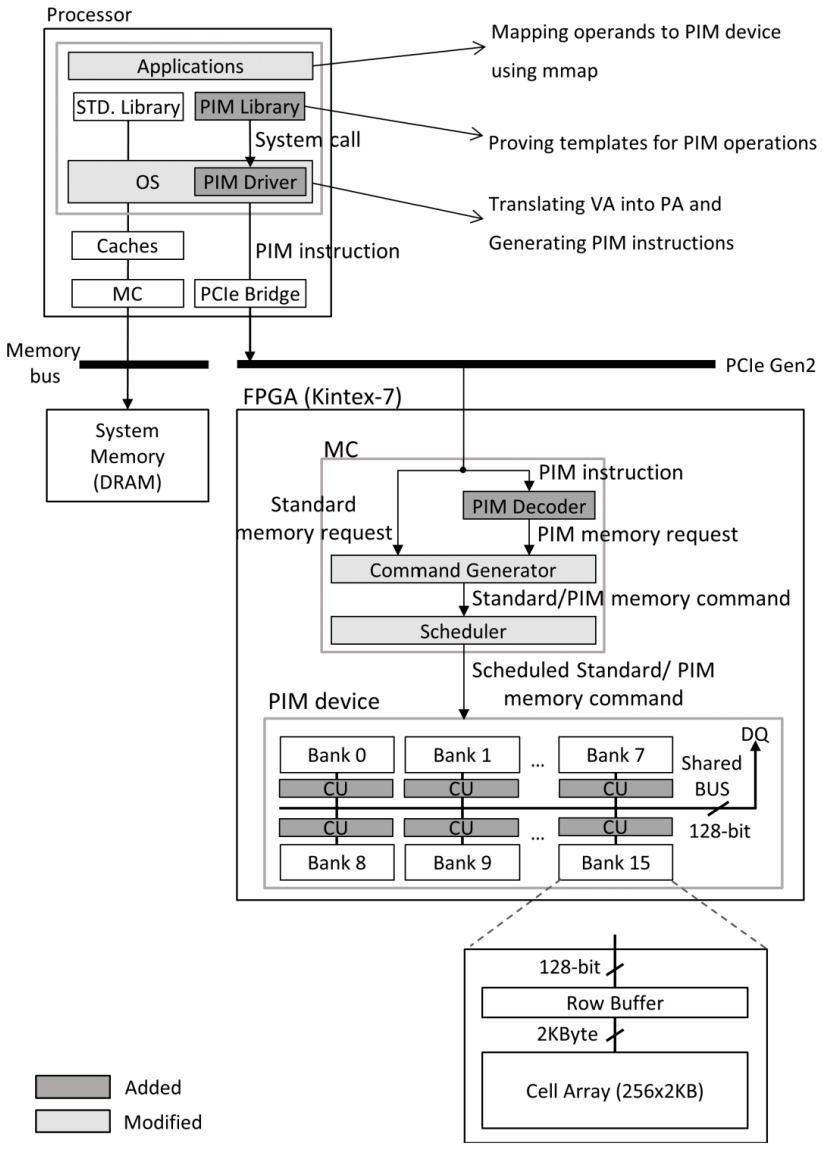

Figure 2.6: PIM architecture from [6]

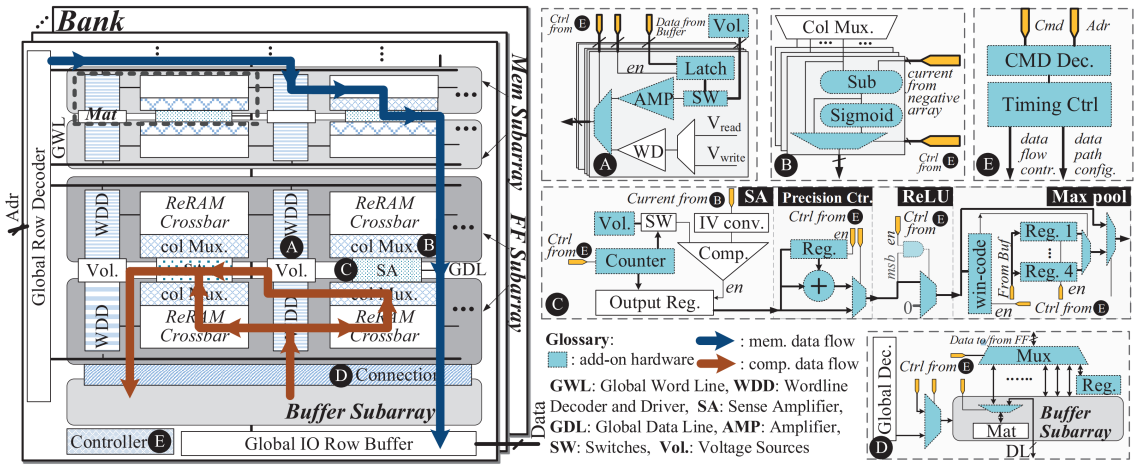

Figure 2.7: PRIME architecture [7]

applications in ReRAM-based memory. Figure 2.7 the architecture of PRIME. The ReRAM memory has a crossbar array structure that can perform matrix-vector multiplication efficiently for Neural Networks applications. This design provides negligible area overhead because of circuit reuse. PRIME uses ReRAM without needing additional processing units. This is done by dividing the ReRAM bank into three parts: mem subarray, FF subarrays, and Buffer subarrays. Using both the features of PIM architecture and the effectiveness of computation of ReRAM-based Neural Networks, this paper introduced a new processing method in ReRAM-based main memory design that increases the energy efficiency and performance of neural network applications by a significant amount. In this design, the ReRAM memory arrays have the ability to perform computations for Neural Networks. This ability of the ReRAM enables it to be used to perform computations quickly for Neural Networks and also act as memory when larger space is required for storing. PRIME was found to reduce energy consumption while significantly speeding up a variety of Neural Network applications employing MLP and CNN.

[28] proposed GraphH an HMC array architecture for large-scale graph processing problems. Two PIM-based graph processing architectures based on HMCs are Tesseract and GraphPIM. However, HMC in GraphPIM just serves as a stand-in for traditional memory in a graph processing

system without the architecture of the cube's connections. Graph scales to enormous graphs, performs up to two orders of magnitude better than DDR-based graph processing systems, and speeds up to 5.12 times faster than Tesseract.

[29] presents a thorough overview of the study of ML-oriented IMC/PIM designs. IoT applications, real-time applications including facial/voice recognition, pattern matching, object detection, and digital support are all expected to leverage IMC in mobile and edge devices. It is possible to assume that IMCs will make greater use of DNN and CNN's newly developed low bit-precision inference and training techniques. Although there is not significant scope for innovation in the crossbar array architecture of ReRAM, Architectural advancements for ReRAM-based IMCs are anticipated.

In [8] the authors study the computer architecture of large in-memory data processing and large-scale graph processing, and demonstrate that the primary limitation for these workloads is memory bandwidth. In this work, the Tesseract, a new programmable accelerator for in-memory graph processing that can effectively employ PIM, design and programming interface are described. Tesseract makes use of 3D layered memory. When constructing a PIM architecture for massively parallel graph processing, the authors argue that specialized in-order cores are preferable than high-end CPUs or GPGPUs. This is due to the fact that such workloads involve maximizing stacked DRAM capacity while adhering to strict chip thermal constraints, which in turn necessitates reducing the energy in-take of in-memory calculation units. Figure 2.8 shows the architecture of Tesseract. Tesseract has three major parts in it. An effective method of communicating across various memory partitions, new hardware that completely exploits the memory bandwidth, and a programming interface that benefits from the unique hardware architecture. Furthermore, it includes two hardware prefetchers created specifically for the memory access patterns needed in graph analysis. These prefetchers run in the indications given by the programming paradigm. For analyses, five cutting-edge graph processing workloads with sizable real-world graphs were

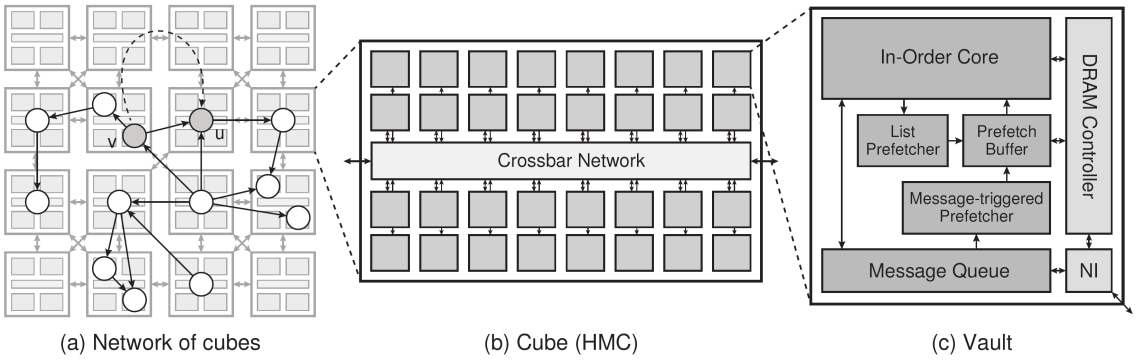

Figure 2.8: Architecture of Tesseract [8]

used and they demonstrate that the suggested architecture delivers an average system performance improvement by a factor of 10 and an average energy reduction of 87% compared to traditional systems.

The inference phase of neural networks inside the memory is greatly sped up by the unique processing in-memory architecture in [9], known as NNPIM. To make quick addition, multiplication, and searches inside the memory, the authors first develop a crossbar memory architecture. Second, they present straightforward optimization methods that considerably raise the performance of NNs and lower their overall energy usage. Parallel in-memory components were used to map all NN features. The proposed design provides weight sharing to lower the amount of computations in memory and thus speed up NNPIM computation to further increase efficiency. The effectiveness of the suggested NNPIM is evaluated in comparison to GPU and cutting-edge PIM architectures. Figure shows an overview of the architecture of NNOIM discussed in this paper.

[10] discusses an alternative to using DRAM memory with 3D die-stacking technology for the memory architecture used in PIM. The authors use features of NVM, like resistance-based storage and current sensing, to come with an efficient design using PIM and NVM. This paper presents Pinatubo, a processing architecture for bulk bitwise operations in NVM. Pinatubo transforms the read circuitry to be able to perform bitwise logic of two or more memory rows effectively and

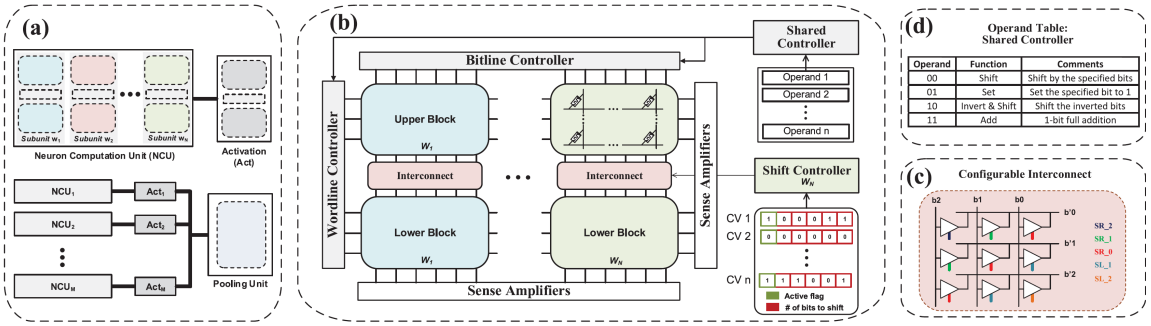

Figure 2.9: Architecture overview of NNPIM [9]

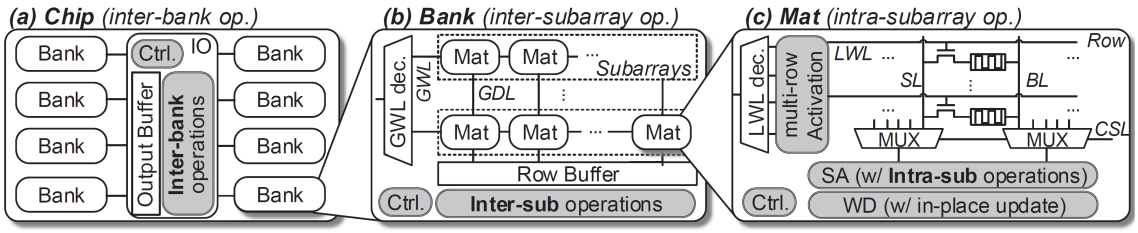

Figure 2.10: The Pinatubo Architecture [10]

support one-step multi-row operations, as opposed to integrating sophisticated logic inside the cost-sensitive memory. Pinatubo outperforms the conventional CPU, as shown by test results on database applications and data-intensive graph processing. Figure 2.10 shows the architecture of Pinatubo.

Chapter 3

Overview of SystemC

Chapters 1 and 2 discussed the purpose and application of SystemC. This chapter discusses the SystemC library elements to familiarize readers with the basics of this library to follow this thesis with ease.

3.1 SystemC Library Architecture

Figure 3.1 shows the major components of SystemC that are discussed in this chapter.

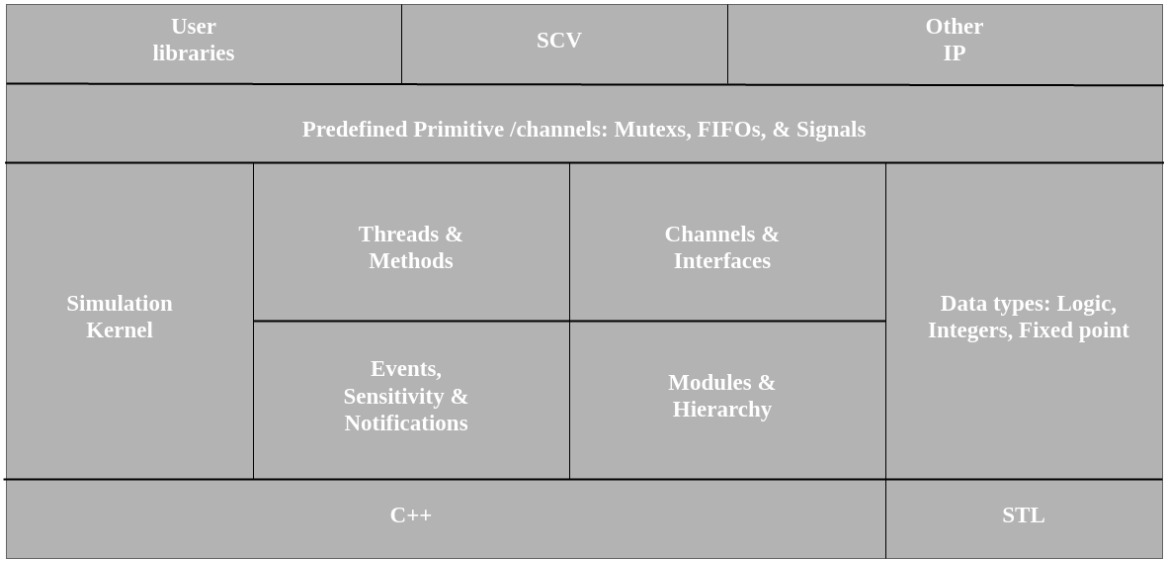

Figure 3.1: SystemC library Architecture

Before diving deep into the details of SystemC, it is important to talk about modules and hierarchy in SystemC. A module is a building block of SystemC just like it is in any other Hardware Description Language like Verilog and is used to represent a component and also has hierarchical connectivity. The modules in SystemC are found to be written using different syntax's or structures. In some places 'sc_module' is used and 'SC_MODULE' in others. 'sc_module' and 'SC_MODULE' both mean the same generally speaking since they both represent a module. 'sc_module' is a class used in SystemC to represent a module[12]. It is a C++ class that can inherit all the Object Oriented Programming features. Figure 3.2 shows an example of how to use sc_module to define a module in SystemC.

```
#include <systemc.h>
#include <iomanip>
#include <iostream>
#include <string>
#include <fstream>

template <int size, int reg_count> class register256 : public sc_module {
public:
    sc_in<sc_bv<size> > DATA_IN;
    sc_in<bool> WRITE;
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_out<sc_bv<size> > DATA_OUT;
    sc_bv<size> DATA;
// private:
//  static sc_signal<sc_bv<1> > one = 1;


    SC_HAS_PROCESS(register256);
    register256(sc_module_name name) : sc_module(name) //constructor that takes paprameter
    {
        SC_METHOD(prc_register256);
        sensitive << clk << reset;


    }

    void prc_register256 ()
    {
        if(reset)
            DATA = 0;
        else if(WRITE)
            DATA = DATA_IN;
        DATA_OUT = DATA;
        cout<< "at time: " << sc_time_stamp()<< endl << "DATA_OUT = "<< DATA_OUT <<endl;

    }

};
```

Figure 3.2: Example for sc_module

SC_MODULE on the other hand is a macro used to represent a module[30]. Although using a macro is advised, it is not required. The macro is used to improve the readability and comprehension of the code. Figure 3.3 shows an example of how to use SC_MODULE to define a module in SystemC:

```
#include "systemc.h"
#include <iomanip>
#include <iostream>
#include <string>
#include <fstream>

SC_MODULE(register256) {

  sc_in<sc_bv<4> > DATA_IN;
  sc_in<bool> WRITE;
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_out<sc_bv<4> > DATA_OUT;
  sc_bv<4> DATA;

  SC_CTOR(register256) {
    // constructor code here
    SC_METHOD(prc_register256);
    sensitive << clk << reset;
  }
  // module code here

  void prc_register256 ()
  {
    if(reset)
      DATA = 0;
    else if(WRITE)
      DATA = DATA_IN;
    DATA_OUT = DATA;
    cout<< "at time: " << sc_time_stamp()<< endl << "DATA_OUT = "<< DATA_OUT <<endl;

  }
};
```

Figure 3.3: Example for SC_MODULE

In this thesis, “sc_module” and SC_MODULE are used interchangeably and the pPIM model uses “sc_module” because of its flexibility which allows the usage of the template for parameterization.

The simulation with SystemC is explained in [31]. The high-level design specification is established along with the system requirements. This entails identifying the system’s interfaces, parts, and functions. The modeling of the design in SystemC takes place after this. Here, the SystemC constructs like modules, processes, and channels are used. The models are created using a text editor like Sublime or in an IDE that supports C++, including SystemC. The SystemC models are compiled and linked to create an executable using a C++ Compiler like GNU’s g++ as

shown in 3.4. The compiled program is then executed standalone, or using a SystemC simulator such as Cadence Xcelium or Mentor's QuestaSim. Following the simulation, the results are used to analyze and verify the design. The model can be integrated with SystemVerilog/UVM and various verification steps like assertions, coverage, and checking system behavior, performance, and functionality can be done. After verification, the low lower level HDL design can be synthesized and can proceed to further steps like physical design, place and route, and other implementation-related tasks.

According to [31], the class library supports the implementation of numerous hardware-specific object types like a clock, ports, concurrent and hierarchical modules and also contains a lightweight kernel for scheduling processes which means it's designed to have a minimal memory footprint and consumes fewer resources which is what makes it faster than other HDLs like Verilog. With SystemC, it is possible to model complex chip designs at a high level of abstraction.

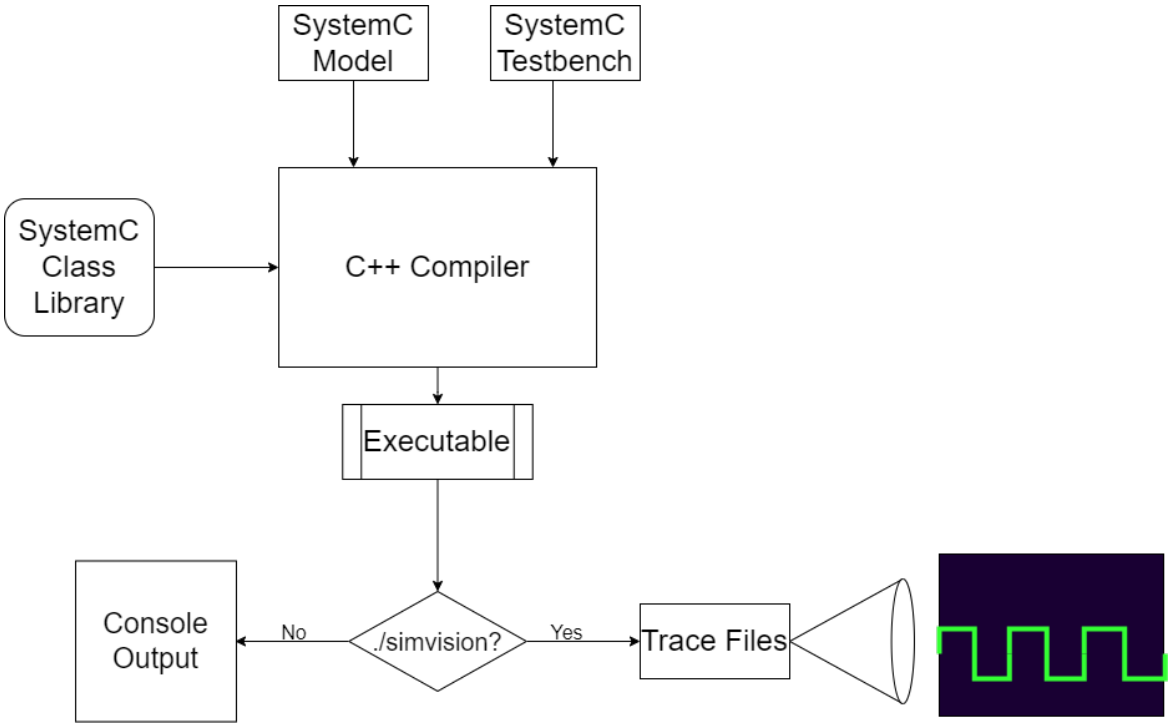

Figure 3.4: SystemC Design Flow adapted from [11]

SystemC is commonly used in the design and verification of complex digital systems as it allows verification engineers to model digital systems at different levels of abstraction. In verification, SystemC is used to write test benches. The SystemC test benches simulate a digital system, produce stimuli, and enable engineers to watch and examine the output signals and verify the behavior of the design. SystemC can be integrated with other simulation tools to view all the signals which is very useful while working with larger systems like AI chips. It can also be used in verification as a reference model since it can be wrapped in SystemVerilog using the DPI function [32]. By doing so SystemC model can be used as a reference while formal verification uses mathematical algorithms to compare the outputs from the reference model and the Design Under Test.

3.1.1 Semantics in SystemC

One of the most important pieces of knowledge required about SystemC is its semantics. As discussed previously, SystemC is used for system-level design and enables engineers to model designs with a high level of abstraction. The hierarchical modules are required to communicate with each other. To control how these modules communicate with each other during simulation, SystemC has a set of semantics consisting of rules for timing, synchronization, and signal propagation. For example, if the value of a certain signal changes in one module, the values in other modules, that are dependent on this signal are all updated. This feature of SystemC ensures that the hardware is simulated accurately. Before going into the semantics of SystemC in detail, it is important to know some SystemC-specific statements like watching statements, signal assignment, and wait statements that will be used to explain the SystemC semantics.

3.1.1.1 Signal Assignment

To understand signal assignment, consider an example of a shift register shown in Figure 3.5 where Process_A represents reg4, Process_B represents reg3, Process_C represents reg2 and Process_D represents reg1.

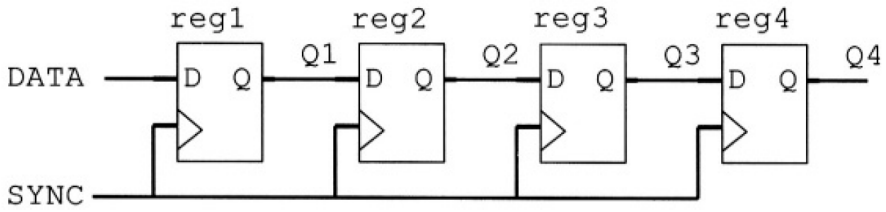

Figure 3.5: Shift registers

When the working of the hardware is considered, the DATA moves from left to right at the clock edge produced by CLK. But when we represent this in software these are just four ordinary assignments that get executed sequentially as shown below.

$$Q_4 = Q_3;$$

$$Q_3 = Q_2;$$

$$Q_2 = Q_1;$$

$$Q_1 = DATA;$$

From a hardware perspective, each register is an independent concurrent process [12]. A new idea is required as one assignment doesn't need to be completed before the other. The use of events to force an ordering is one possibility. If this is implemented, Process_A has to wait for an event from Process_B before assigning its register, and so on. This means that the 'wait' and 'notify' code has to be manually written for this mechanism to take place which is a tedious

process. To solve this problem, simulators use a feature called “the evaluate-update paradigm”. This can be explained with the help of Figure 3.6. A new phase called the update phase has been added to the SystemC simulation kernels from the past. In this new kernel, it is possible to switch between the update and evaluate phases. This is called the delta cycle. The data storage for signal assignment is shown in Figure 3.7. There are two sets of data storage allocations when an entity is declared as a signal. These locations are for storing the current value and new value. When a value is written to a signal, the data is stored in the new value instead of the current value, and request_update() method is called to make the kernel call update() method in the update phase. Once all the processes have been completed in the evaluation phase, the update() method is called for all the signal channels that requested an update. It is worth noting that, the current value remains unaltered. When a process writes to an evaluate-update channel, and immediately accesses it, the current value is unchanged [12].

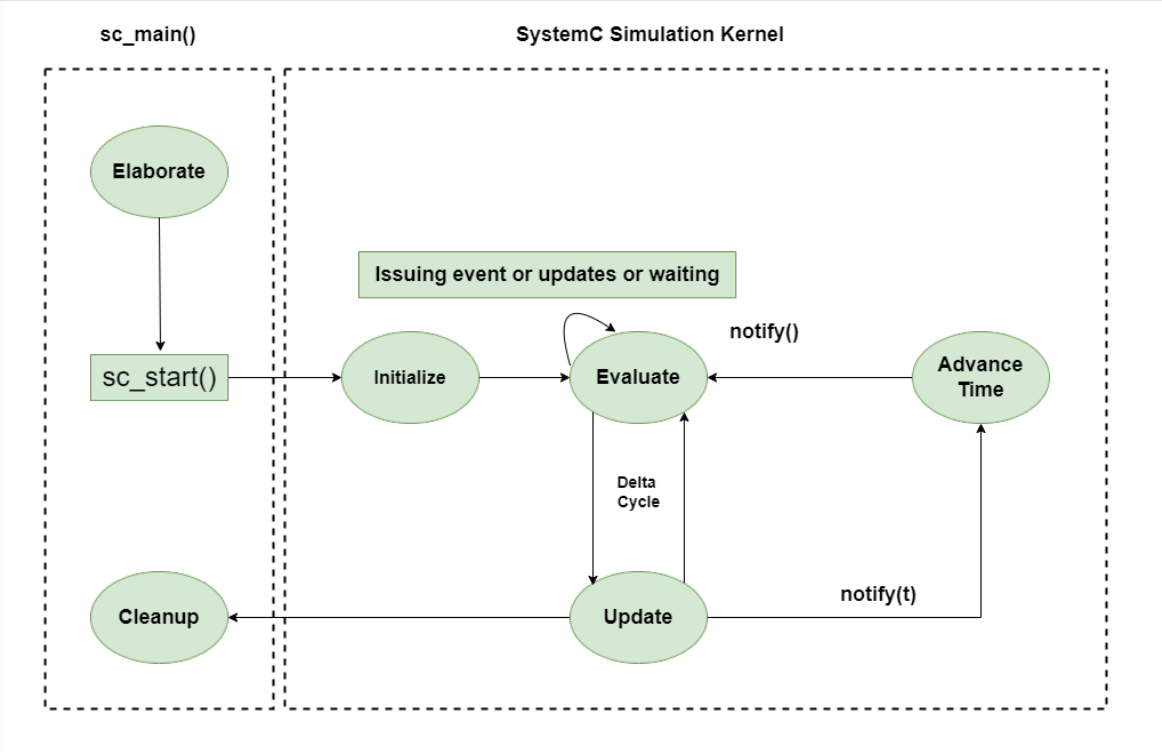

Figure 3.6: SystemC simulation engine adapted from [12]

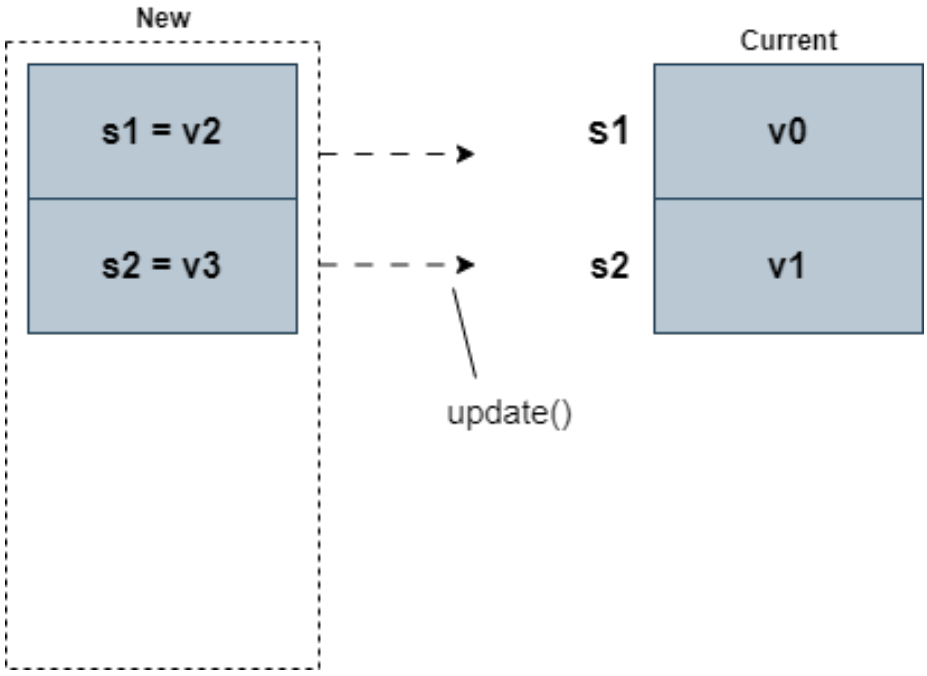

Figure 3.7: Signal Channel Data Storage adapted from [12]

3.1.1.2 Watching Statements

In SystemC, a construct called watching statements is used to monitor and respond to changes in the signals [12]. The watching statement is used in the body of a method and takes a signal or a variable as an argument. The method that contains the watching statement is triggered when the value of the signal or variable changes. As a result, the module can respond to changes in dependent variables or signals and take necessary actions. For example, consider the SystemC code in Figure 3.8.

```
#include <systemc.h>

SC_MODULE(example_module){
    sc_in<bool> input1;
    sc_in<bool> input2;

    void process1()
    {
        watching(input1.value())
        {
            //Do something when input1 changes
        }

        watching(input2.value())
        {
            //Do something when input2 changes
        }

    }

    SC_CTOR(example_module){
        SC_METHOD(process1);
        sensitive << input1 << input2;
    }
};
```

Figure 3.8: SystemC example code for watching statements

In this example, “example_module” is a SystemC process that is sensitive to changes in the inputs “input1” and “input2”. The method “process1” has two watching statements that monitor the values of “input1” and “input2”. When “input1” or “input2” change, the corresponding watching statement is triggered and the code inside is executed. Building reactive systems in SystemC that react to environmental changes makes good use of watching statements. The “*sensitive*” statement seen in the above example triggers the process “process1” when there is a change in either “input1” or “input2”. Watching statements have been deprecated according to the latest SystemC LRM [33] but are used in some rare cases.

3.1.1.3 Wait statements

The wait() statement in SystemC is used to delay the execution of a SC_THREAD (thread) or a SC_CTHREAD (clocked thread) process instance. The wait() method is called from SC_THREAD or SC_CTHREAD for the very next event on which the process was made to wait. For example, wait(clk.posedge_event) will wait till the next positive edge is encountered. The control is returned to the simulator by invoking the wait method. The wait statement suspends the SC_THREAD process [12]. The wait statement is also indirectly invoked when sc_fifo is used. The "read()" and "write()" methods are used to read and write from the FIFO respectively as their names suggest. When a "read" method is used with a FIFO while it is empty, it invokes the wait method. Similarly, when the 'write' method is used with a FIFO when it is full, the wait method is used to suspend the process until the FIFO is emptied similar to invoking wait directly.

The wait method is a member of the "sc_module" class and has many syntax's which will be discussed in the next chapter 4. The current state of the thread which is being suspended by wait is first saved when it is called. The simulation kernel is in control now and activates a process that is in the ready state. When the wait statement has completed or reached the amount of time it was made to delay, the thread continues to execute. Before this, the scheduler restores the state of the original thread to pick up from where it was left.

3.1.2 Threads and Methods

The author in [13] explains the different processes in SystemC namely SC_METHOD, SC_THREAD, and SC_CTHREAD, and their simulation cycles. These processes are important to achieve concurrency and it is important to understand their simulation cycles to use them in appropriate instances. Figure 3.9 shows the SystemC Simulation Cycles. As discussed already signal assignments, it is worth noting that Signals don't change their values right away, and their

assignments aren't put into action until the following simulation cycle[13]. In SystemC, "sc_start" starts the simulation and the initialization of the entities of every class with the values given in constructors takes place. Next, the events that are the clock is generated, the simulation kernel processes, and all the user-defined concurrent processes start running but these don't happen simultaneously. A more detailed view of the different phases of the SystemC simulation Kernel can be seen in Figure 3.10.

The kernel processes start executing only when all the user-defined processes are suspended and the user-defined processes execute only when kernel processes are suspended. Figure 3.11 shows the life cycle of a process. A process is said to be active when it is invoked and it is invoked by a sensitivity list in most cases. It is suspended when a wait statement is invoked in that process or when it executes the last statement of the process. Figure3.11shows the different states that a process goes through. The different statuses are *checkLocalWatching*, *checkGlobalWatching*, *active*, and *suspended*. Before going to the active state, watching conditions like the Global and Local Watching conditions are checked and the program counter is set as required.

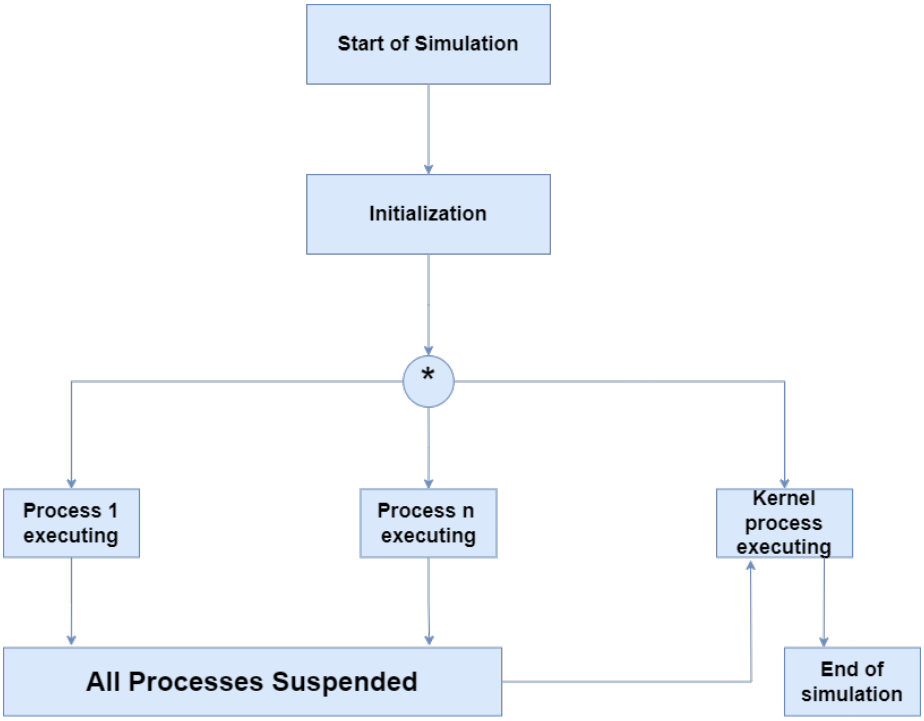

Figure 3.9: SystemC simulation cycle adapted from [13]

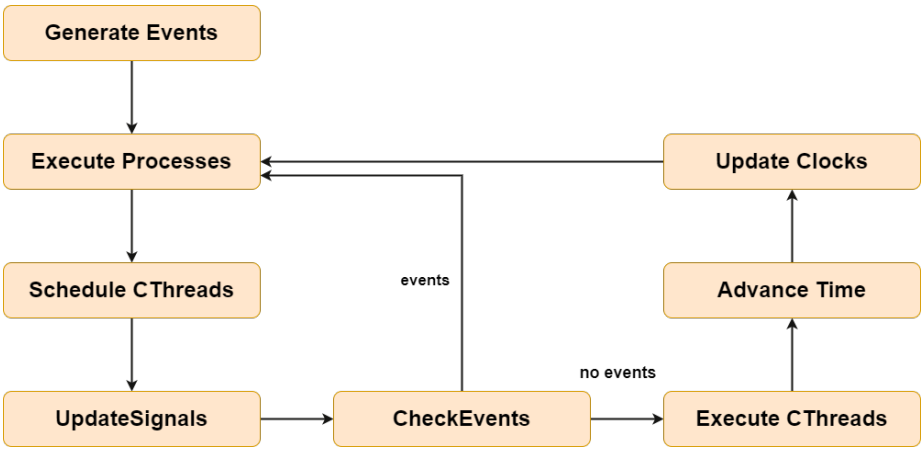

Figure 3.10: Phases of the Simulation Kernel [13]

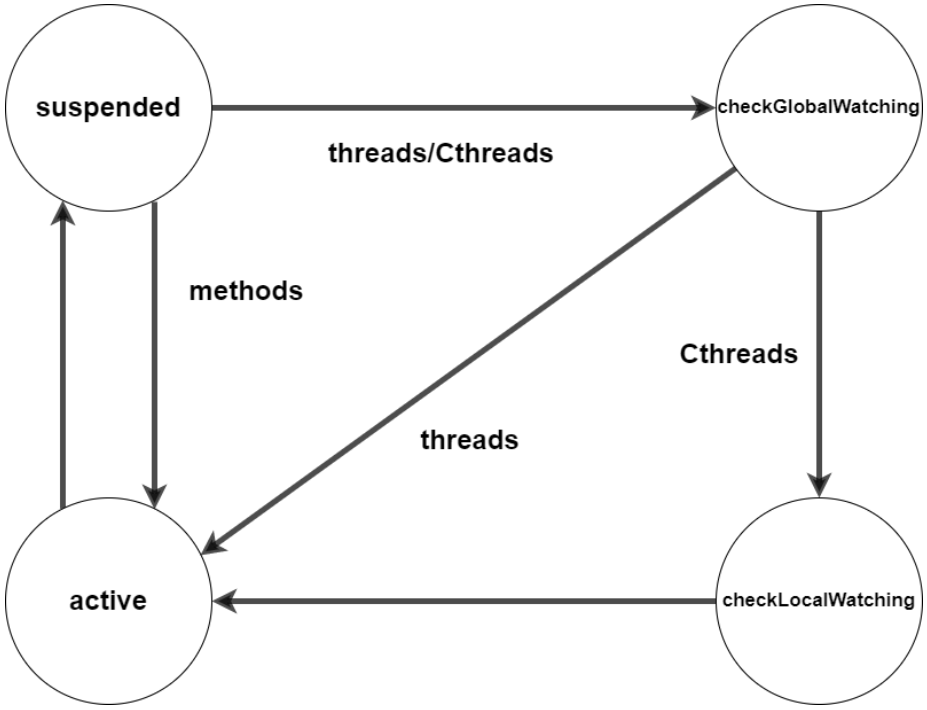

Figure 3.11: Life Cycle of a Process [13]

3.1.3 Data types in SystemC

In addition to C++ data types like integers, float, double, bool, etc., SystemC has separate set of data types that are hardware-compatible. sc_int<>, sc_fixed<>, sc_bv<> are some of the SystemC-specific data types that have been implemented using the template class and operator overloading features of C++. To accommodate the tri-state logic or four-state logic (0,1,X,Z) that hardware engineers are familiar with, SystemC also provides sc_logic<> data type. The value given withing the '<>' is the size or number of bits of the variable.

3.2 SystemC Compilation

For compiling the source code of SystemC, it does not require any expensive tools. It requires an environment that is easy and cheap to establish. This is what makes the library very cost effective and portable.

The environment should contain the following as mentioned in [12]:

- SystemC-supported platform
- SystemC-supported compiler
- SystemC library
- Compiler command sequence, *makefile* or equivalent

Figure 3.12 shows the SystemC Compilation Flow. [12] has explained the SystemC compilation in great detail. In this thesis, GNU's GCC g++ compiler is used with a *makefile* containing instructions for building the software into an executable program using the "make" command. Once the source program is written, the "make" command typed on the command line compile the SystemC source files. Once they are free of errors, the executable is run to obtain outputs. Further, the Cadence Simvision tool or an open source VCD waveform viewer such as GTKWave can be used to open the dump files to view the waveforms.

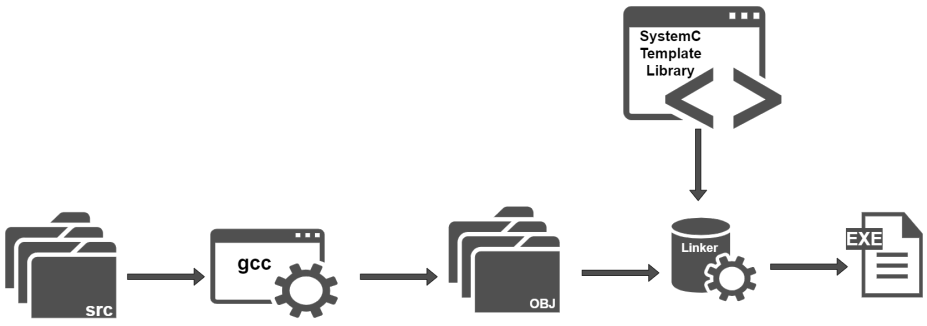

Figure 3.12: SystemC Compilation Flow adapted from [12]

3.3 Analogy between SystemC and Verilog

Assuming that most of the hardware engineers are familiar with the HDL Verilog, Table 3.1 shows SystemC equivalents of Verilog to help the reader get familiarize with the library quickly.

SystemC	Verilog
sc_bv<5> A;	[4:0] A;
sc_in<sc_bv<5> > A;	input [4:0] A;
class clock_gen : public sc_module{};	module clock_gen();
template <typename T>	parameter
sc_vector<>	Variables in generate statements
SC_THREAD()	initial block
SC_METHOD()	always block
sc_signal<>	reg that uses non-blocking assignments (<=) [12]

Table 3.1: SystemC Equivalents of Verilog

The Table 3.1 provides a brief idea of how a SystemC code would look like in reference to a Verilog code. Elements do not completely translate to their equivalents mentioned in Table 3.1. For example, the table shows that SC_METHOD is an equivalent of always block. While this is true, there is a slight difference in that SC_METHOD cannot have delays or wait statements in them but always blocks in Verilog can contain delay statements. The reader will still need go through SystemC books like [30], [12], [34], [35], and LRM [36] in detail to learn the library

completely.

Chapter 4

pPIM Architecture

The PIM architecture modeled in this paper is the pPIM (programmable Processing In Memory) architecture researched and discussed in [37], [14], and [38] . It is designed for ML models like CNN and DNN that involve processing a large amount of data. The architecture is capable of ultra-low-latency parallel processing and is also an alternative to Von Neumann's architecture [14]. pPIM is a LUT (Look Up Table) based PIM architecture in which pre-calculated values for calculations can be done on the DRAM memory platform. This architecture can be programmed for various operations like addition, multiplication, subtraction, etc., by reprogramming the LUT, the arithmetic operation can be changed. Additionally, this architecture can be altered to a number of combinations of bit-widths of the operand data which makes it flexible across precision, performance, and efficiency.

The data movement to and from the memory subsystem and the processing unit attributes to a large amount of power consumption. To overcome this, the data movement between the processing nodes is done along with existing data movement mechanisms in DRAM. This architecture also operates with a variety of data-precision combinations that are achieved by operand-decomposition and multi-step processing algorithms using a multi-core processor with nine pPIM cores forming

a cluster. This multi-core setup makes it possible to achieve parallelization of operations. During parallelization, multiple operation steps are completed at a single machine cycle which gives great throughput and excellent resource utilization.

Figure 4.1 shows a top-level view of the pPIM cluster placed in the DRAM bank. Within the top level of this design, is the pPIM cluster. Each cluster consists of nine pPIM cores that are reprogrammable which makes this system design very flexible. In Figure 4.1 we can see that the clusters are placed between the memory subarray. The important feature to note is how close the clusters are placed to the memory subarray which make the data transaction fast and efficient. The DRAM memory banks were enhanced using Low-cost Inter-Linked Subarray (LISA) for establishing communication among the clusters adapted from [39]. Further, the subarray is connected to a decoder circuit for selecting and accessing the subarray in use. As can be seen, the distance between the cluster, which is the processing unit of this design, and the memory subarray is very minimal. This setup minimizes the distance that the data has to travel to and from both the units and hence power consumption due to wiring overhead is largely reduced.

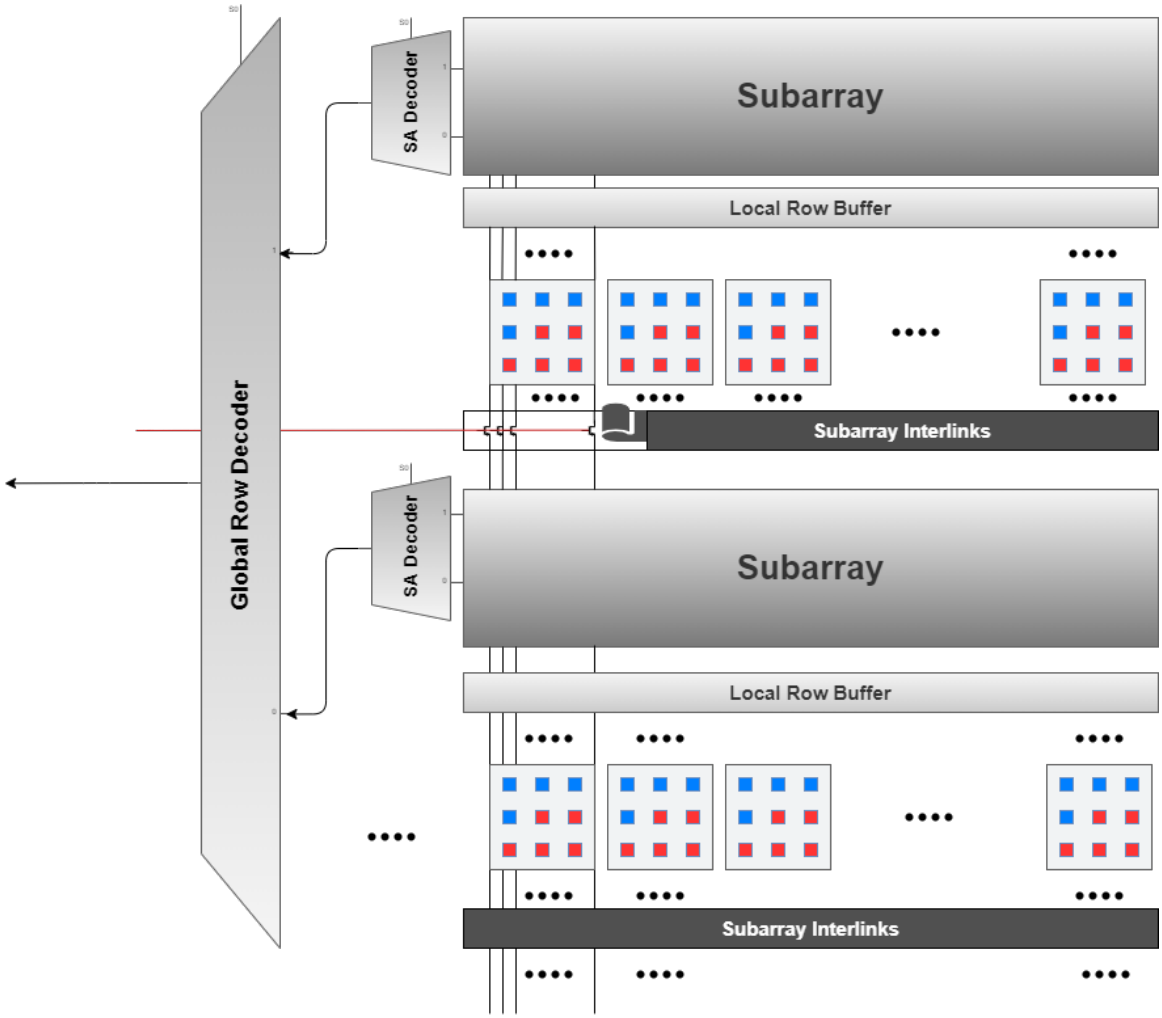

Figure 4.1: Top level view of pPIM cluster placed in a DRAM bank

4.1 pPIM Cluster

The pPIM Cluster is a multi-core setup of nine pPIM cores that communicate with each other to achieve parallel processing at a reduced number of clock cycles for finishing the computation efficiently. Each core can perform different operations simultaneously. Each pPIM core can be made to perform the different operations by loading the corresponding LUTs with appropriate function words. These cores are connected by a router that helps in the data transfers among the

cores. Figure 4.2 shows general the layout of the pPIM Cluster including the nine pPIM cores and a router.

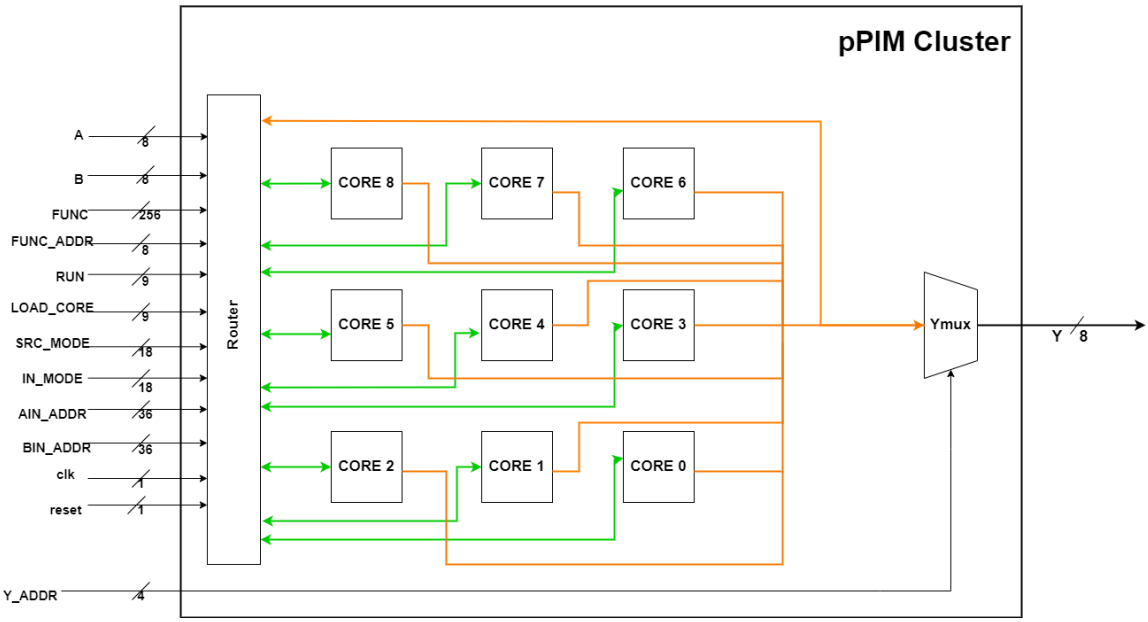

Figure 4.2: Layout of pPIM Cluster

More detail can be found in Figure 4.3 which shows how the router is placed to communicate with the cores, also note in this figure an accumulator is included within the cluster to support complex operations such as Multiply-Accumulate (MAC).

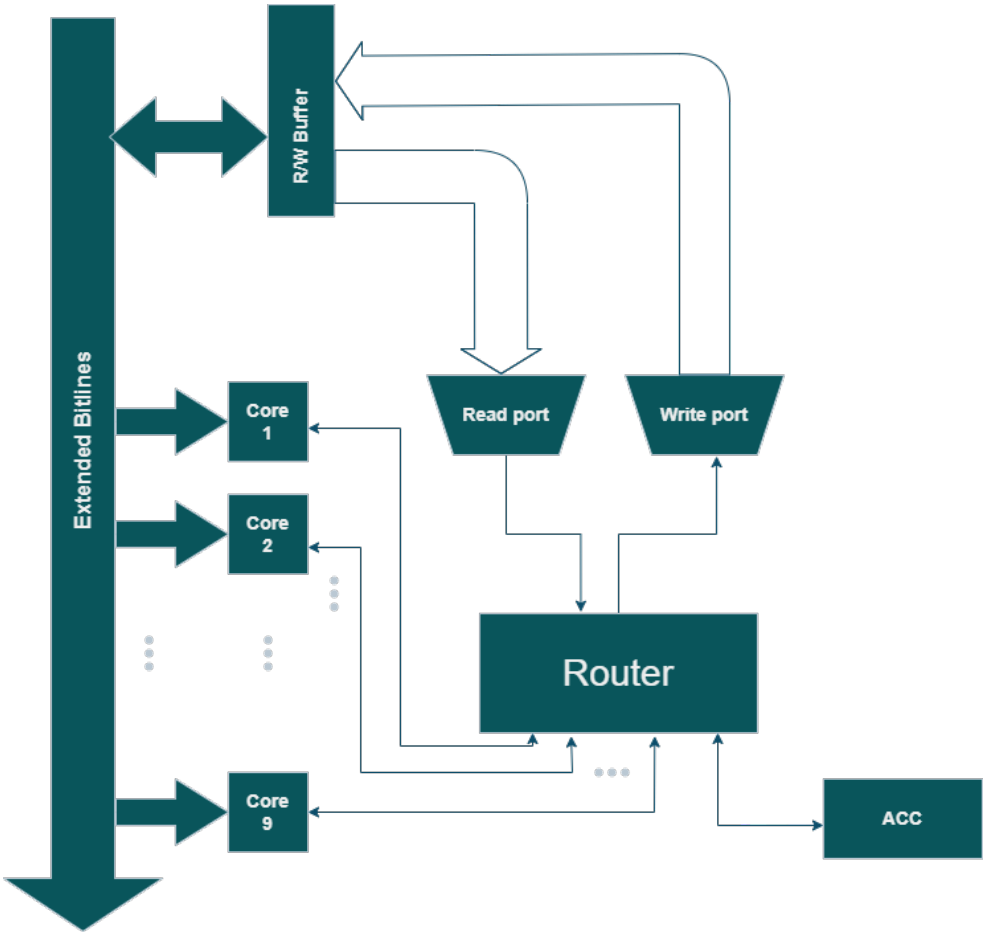

Figure 4.3: pPIM cluster

Since the pPIM cluster is designed for data-intensive applications, all the cores in the network should be able to communicate with every other core with minimal overhead. This is also called all-to-all communication in parallel computing. This is done by a set of multiplexers in the router shown in Figure 4.4. The router has a cross-bar switch architecture. In the cross-bar switch architecture, a pPIM core can receive inputs from outputs of every other pPIM core in the cluster and vice versa. Note that n cannot be equal to k because the core is not connected to itself. To match the size of the inputs and outputs of the pPIM cores in the router, the 8-bit outputs are regarded as two 4-bit segments.

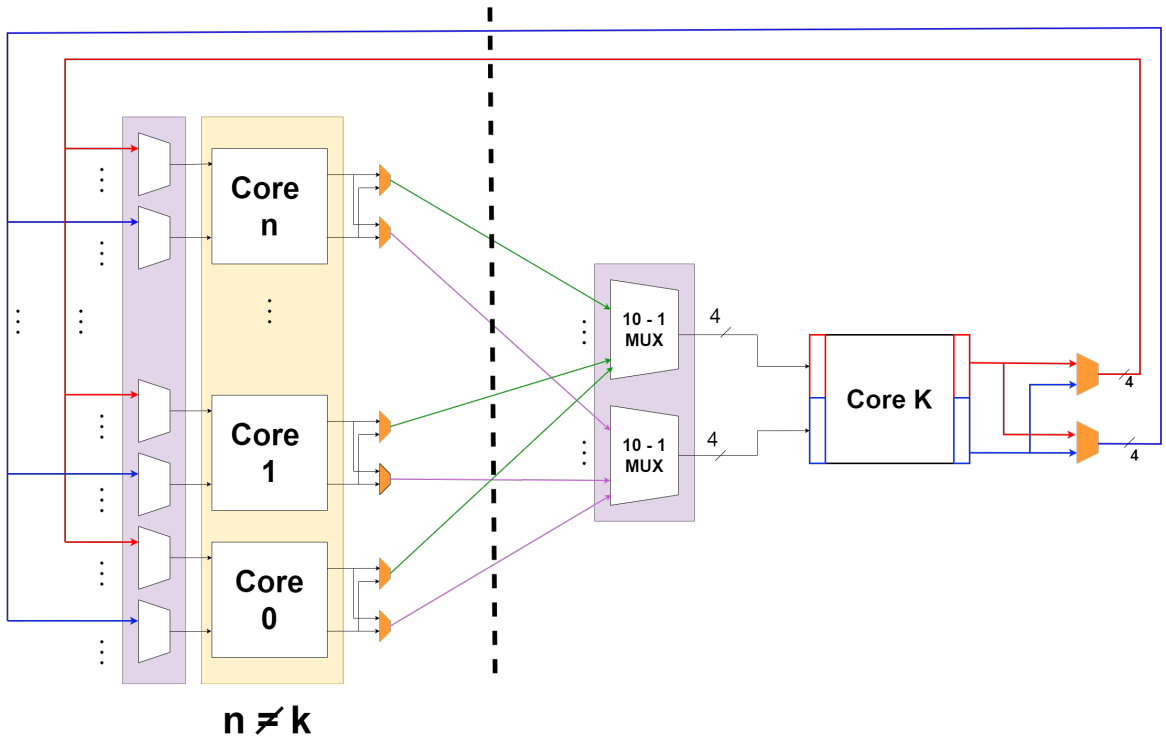

Figure 4.4: The interconnect Router architecture for n cores in a cluster

4.2 pPIM core

The main processing unit of the system is the pPIM core. It consists of multiplexers, registers, and register files. All of these component sizes are reprogrammable. This means the number of bits of the input and output ports and the width of the registers can be changed. The set of parameters given for a configuration remains fixed at implementation. This is very helpful for the performance analysis of different sizes of the components. Figure 4.5 shows the architecture of the pPIM core. Most of the computational process is carried out by the pPIM core in this design. Processing is done in the core by using the pre-calculated values in the LUT called function words. Function words are loaded into the registers of the register file, and the function word values are accessed from the LUT based on the input operand values in Register A and Register B that

select the required bits for further processing. The final output is seen in port Y. The pPIM core is capable of performing various operations that can take 4-bit inputs and gives an 8-bit output. The two inputs for a single operand function can be the high and low input segments.

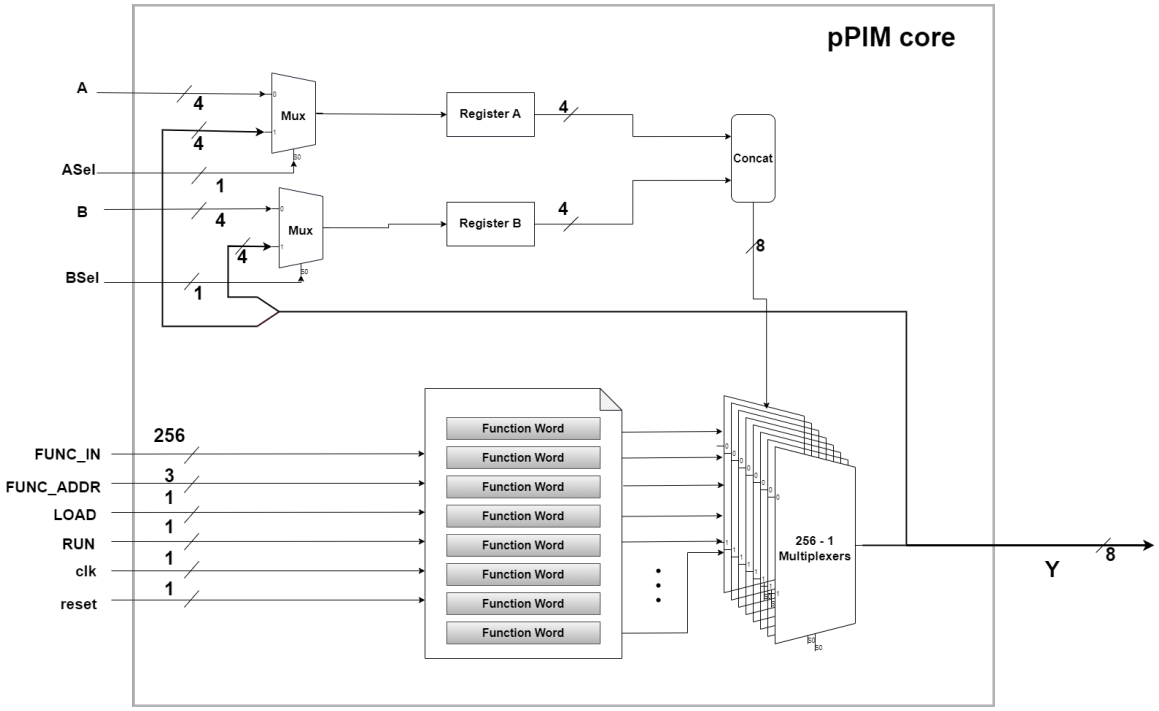

Figure 4.5: Architecture of pPIM core

4.3 pPIM Multiply-Accumulate (MAC) operation

The MAC operation is used to exercise the various features of pPIM architecture like being able to reprogram the cores to perform different operations and parallel processing of these cores to complete the computations efficiently. pPIM architecture is also capable of switching to different precision modes for multiple data types. pPIM can perform MAC operations on both signed and unsigned integers at 4-bit and 8-bit precision.

Figure 4.6 shows the calculation for MAC operation. To work with both 4-bit and 8-bit inputs,

the partial product method is used as shown in Figure 4.6 where partial products are obtained by multiplying half segments of the numbers as shown, and then these partial products are added together. By doing so, the upper nibble which is always 0 in 4-bit inputs separated, and their partial products become zero. The summation results are accumulated into the accumulator to obtain the final result of the MAC operation.

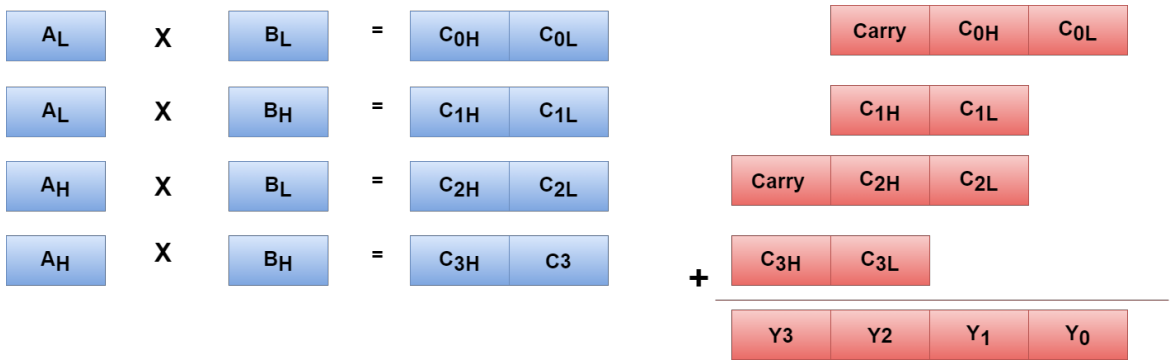

Figure 4.6: Calculation for MAC operation

Figure 4.8 shows how the cores are arranged as per the core numbers for easier understanding. The colors correspond to the corresponding waveforms for different cores used in later sections. Figure 4.9 shows how the nine cores of the cluster work together to perform the MAC operation. In the Figure, the nine small squares represent the pPIM cores of the pPIM cluster. The blue-colored cores perform multiplication and the red-colored cores perform addition. t is the clock cycle. V_0, V_1 ,V_2, and V_3 are the partial products produced by pPIM cores N, 7, 6, and 5 respectively. For a 8-bit input, a_H and b_H are the upper bytes of the inputs in register A and B respectively. $a_L$and $b_L$is the lower bytes of the registers A and B respectively. The equations for the partial products are as shown below:

$$V_0 = a_L b_L \tag{4.1}$$

$$V_1 = a_L b_H \tag{4.2}$$

$$V_2 = a_H b_L \tag{4.3}$$

$$V_3 = a_H b_H \tag{4.4}$$

The partial products obtained in 4.1, 4.2, 4.3, and 4.4 are then added by the rest of the cores as shown in Figure 4.9. The letters I, J, K, L, M, N, and F represent the clock steps of addition cores, and 0,1, 2, and 3 represent parallel operations during each step [14]. A0, A1, A2, and A3 are accumulator values from the previous cycle. Is also important to note that the accumulator value is initialized to 0. The texts in red represent the value from the next cycle of operation.

$$|a_H a_L| = c_H c_L \tag{4.5}$$

$$|b_H b_L| = d_H d_L \tag{4.6}$$

$$c_L \times d_L = e_{0H} e_{0L} \tag{4.7}$$

$$c_L \times d_H = e_{1H} e_{1L} \tag{4.8}$$

$$c_H \times d_L = e_{2H} e_{2L} \tag{4.9}$$

$$c_H \times d_H = e_{3H}e_{3L} \tag{4.10}$$

The partial products from Equations 4.7, 4.8, 4.9, and 4.10 are added as shown below:

		Carry	e_{0H}	e_{0L}
		e_{1H}	e_{2L}	
+	*Carry*	e_{2H}	e_{2L}	
	e_{3H}	e_{3L}		
	F_3	F_2	F_1	F_0

Figure 4.7: Addition of the partial products

Signed inputs have additional steps for performing MAC operations. First, the magnitude of signed inputs should be taken, which means the sign has to be ignored. Then the MAC operation is performed by treating the inputs like unsigned numbers. Once the MAC operation is complete, the sign of the result is determined based on the sign of the inputs and is computed as follows:

If $a_H \& b_H \geq 0$	$Y = F_3 \quad F_2 \quad F_1 \quad F_0$
If $a_H \& b_H < 0$	$Y = F_3 \quad F_2 \quad F_1 \quad F_0$
If $a_H \| b_H < 0$	$Y = -F_3 \quad F_2 \quad F_1 \quad F_0$

Figure 4.8: pPIM core arrangement in the cluster with core numbers and color code

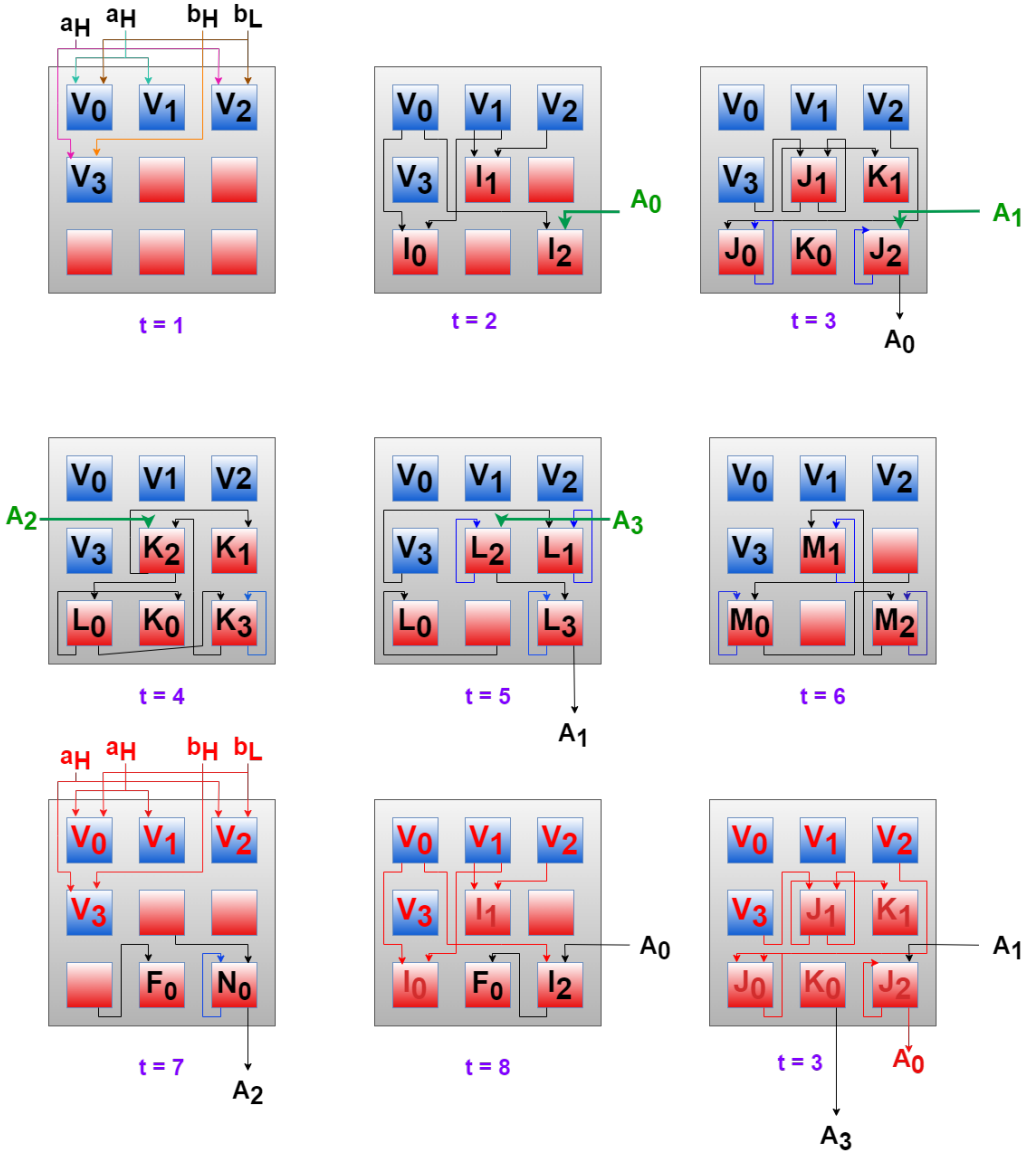

Figure 4.9: pPIM cluster performing MAC operation for 8-bit full precision [14]

For more clarity, the data flow model of the pPIM cluster can be seen in Figure 4.10. It can be seen that at time-step 7 the new values of inputs A and B are taken into cores N, 7, 6, and 5 for multiplication while finishing up the MAC operation for the previous inputs. This overlapping of operations enabled scaling down the operational steps and reduced the latency by 33.3% [14].

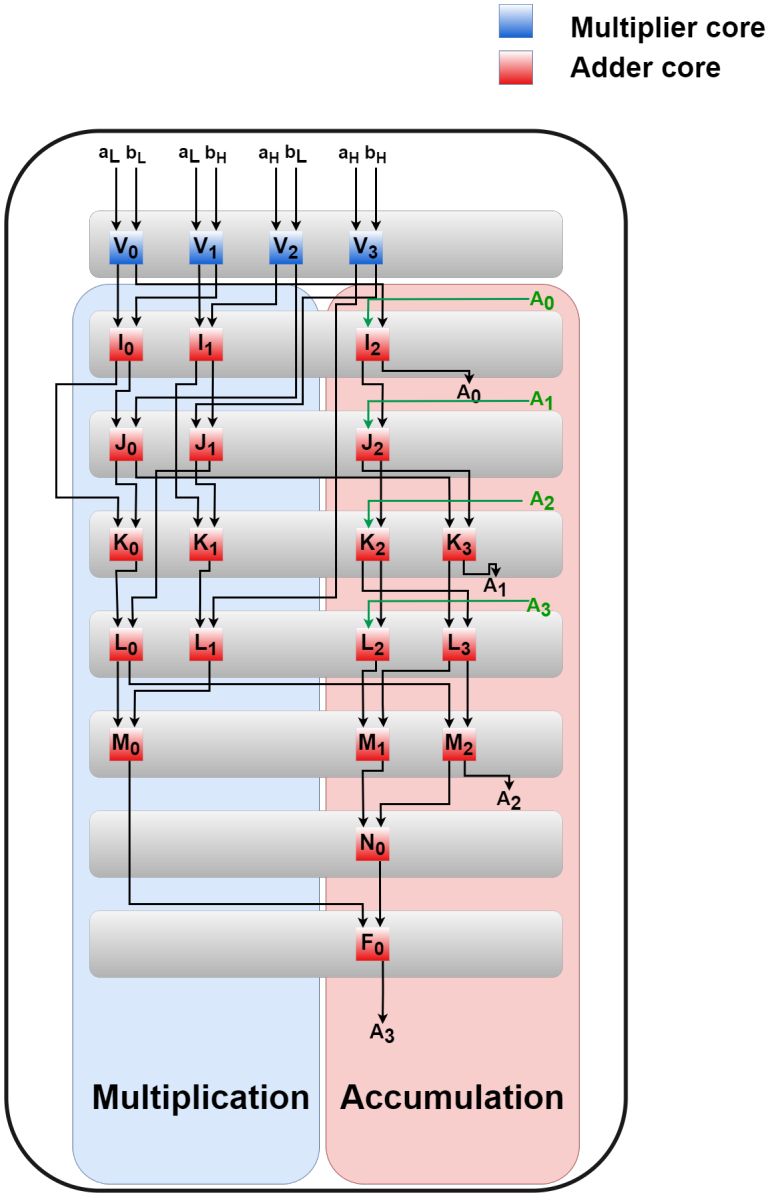

Figure 4.10: Data flow model of pPIM [14]

As an example, Table 4.1 provides a time step analysis of how the MAC is performed on the pPIM Cluster to understand the calculation details discussed earlier. The calculation sequence starts with the following 8-bit unsigned inputs, $A = 36_{10}$, $0010\,0100_2$, and $B = 129_{10}$, $1000\,0001_2$, and the accumulator A, comprised of A_3, A_2, A_1, A_0, is initially 0. To highlight back to back

MAC calculations, at step $t = 7$, new inputs $A = 9_{10}$, $0000\ 1001_2$, and $B = 99_{10}$, $0110\ 0011_2$ are provided to the cluster.

Table 4.1: pPIM Cluster MAC Calculation Details

Step	**Calculation**
At $t = 1$	$A = 36_{10}$, $0010\ 0100_2$, and $B = 129_{10}$, $1000\ 0001_2$
	$V_0 = V_{0H}V_{0L} = A_L \times B_L = 0100 \times 0001 = 0000\,0100$
	$V_1 = V_{1H}V_{1L} = A_L \times B_H = 0100 \times 1000 = 0010\,0000$
	$V_2 = V_{2H}V_{2L} = A_H \times B_L = 0010 \times 0001 = 0000\,0010$
	$V_3 = V_{3H}V_{3L} = A_H \times B_H = 0010 \times 1000 = 0001\,0000$
At $t = 2$	$I_0 = V_{0H} + V_{1L} = 0000 + 0000 = 0000\ 0000$
	$I_1 = V_{1H} + V_{2H} = 0010 + 0000 = 0000\ 0010$
	$I_2 = V_{0L} + A_0 = 0100 + 0000 = 0000\ 0100$
At $t = 3$	$J_0 = I_{0L} + V_{2L} = 0000 + 0010 = 0000\ 0010$
	$J_1 = V_{3L} + I_{1L} = 0000 + 0010 = 0000\ 0010$
	$J_2 = A_1 + I_{2H} = 0000 + 0000 = 0000\ 0000$
	$A_0 = I_{2L} = 0100$
At $t = 4$	$K_2 = A_2 + J_{2L} = 0000 + 0000 = 0000\ 0000$
	$K_3 = J_{0L} + J_{2H} = 0010 + 0000 = 0000\ 0010$
	$K_0 = I_{0H} + J_{0H} = 0000 + 0000 = 0000\ 0000$
	$K_1 = I_{1H} + J_{1H} = 0000 + 0000 = 0000\ 0000$

Table 4.1: pPIM Cluster MAC Calculation Details

Step	**Calculation**
At $t = 5$	$L_0 = K_{0L} + J_{1L} = 0000 + 0010 = 0000\,0010$
	$L_1 = K_{1L} + V_{3H} = 0000 + 0001 = 0000\,0001$
	$L_2 = A_3 + K_{2L} = 0000 + 0000 = 0000\,0000$
	$L_3 = K_{3H} + K_{2H} = 0000 + 0000 = 0000\,0000$
	$A_1 = K_{3L} = 0010$
At $t = 6$	$M_0 = L_{0H} + L_{1L} = 0000 + 0001 = 0000\,0001$
	$M_1 = L_{2L} + L_{3H} = 0000 + 0000 = 0000\,0000$
	$M_2 = L_{0L} + L_{3L} = 0010 + 0000 = 0000\,0010$
At $t = 7$	$A = 9_{10}$, $0000\ 1001_2$, and $B = 99_{10}$, $0110\,0011_2$
	$V_0 = V_{0H}V_{0L} = A_L \times B_L = 1001 \times 0011 = 0001\,1011$
	$V_1 = V_{1H}V_{1L} = A_L \times B_H = 1001 \times 0110 = 0011\,0110$
	$V_2 = V_{2H}V_{2L} = A_H \times B_L = 0000 \times 0011 = 0000\,0000$
	$V_3 = V_{3H}V_{3L} = A_H \times B_H = 0000 \times 0110 = 0000\,0000$
	$N_0 = M_{1L} + M_{2H} = 0000 + 0000 = 0000\,0000$
	$A_2 = M_{2L} = 0010$
At $t = 8$	$I_0 = V_{0H} + V_{1L} = 0001 + 0110 = 0000\,0111$
	$I_1 = V_{1H} + V_{2H} = 0011 + 0000 = 0000\,0011$
	$I_2 = V_{0L} + A_0 = 0010 + 1011 = 0000\,1101$
	$F_0 = M_{0L} + N_{0L} = 0001 + 0000 = 0000\,0001$

Table 4.1: pPIM Cluster MAC Calculation Details

Step	**Calculation**
At $t = 9$	$J_0 = I_{0L} + V_{2L} = 0111 + 0000 = 0000\ 0111$
	$J_1 = V_{3L} + I_{1L} = 0000 + 0011 = 0000\ 0011$
	$A_3 = F_{0L} = 0001$

As shown, the expected final result in accumulator A, comprised of A_3, A_2, A_1, A_0, is $4,644_{10}$, $0001\ 0010\ 0010\ 0100_2$.

Chapter 5

Bit-exact Hierarchical SystemC Model of pPIM

This chapter covers the SystemC bit-exact hierarchical models of the pPIM core and pPIM cluster. The SystemC models of the pPIM core and the pPIM cluster are bit-exact hierarchical models that match the signal bit width and hierarchical structure of the Verilog RTL unlike most of the other SystemC models which are flat algorithmic models. The complex systems designed today evolve with a rapid evolution in technology, and bit-exact hierarchical models have the ability to be shared as Intellectual Property (IP) library components. Once compiled, the SystemC model protects the structure of the design from being exposed to potential users. In this case, the purpose of model is to act as a reference for the verification of the pPIM core and cluster and it will also be used for performance modeling and architecture exploration since the model is parameterized and can be used to explore different architectures to observe the change in performance factors. The parameters that can be configured are the bit width of registers and the I/O ports, the number of registers, and the size of all the components. The opcode can also be changed by changing the Look Up Table loaded in the test bench. A test bench is written to provide input signals to

the design, which is the same as the inputs given to the test bench for the Verilog RTL design to compare both outputs. The most common method seen in SystemC is the construct SC_MODULE. Although this is widely used, it does not have a lot of flexibility in terms of parameterization and making the code reusable. This rises a requirement to know different ways of using the constructs based on the SystemC semantics and when the parameter values are required to be resolved. Unlike the commonly used HDL, different aspects of the design can be parameterized using SystemC.

5.1 Parameterization in SystemC

According to [34], the various aspects that can be parameterized are:

- Values within a design: The values are parameters that can be computed or declared within the design and set as parameters when the parameterized module is instantiated.
- Data types and type attributes: Unlike other HDLs, the data type of an entity in a design can be parameterized. The type attributes like the width of a channel and the width of a vector can be parameterized.
- Design structure: The structure of the design can be changed by passing a parameter that alters the width of other dependent components of the design. The pPIM is also modeled with this flexibility.

In fact, the pPIM models exercise all of the above-mentioned parameterization methods.

Depending on when the parameter values are resolved, there are three ways to specify parameters in SystemC. The three ways are:

- At compile time: In this method, the parameter values or types are resolved when the C++

compiler is run. This is done by using the template parameters in C++. This is also the method used to model the pPIM in this thesis.

- At elaboration time: In this method, the parameters are resolved before the start of the simulation. This is when the design hierarchy is constructed. The C++ constructor arguments are used to declare the parameters along with "sc_module". In this method, the user can specify the parameters from the command line.

- At simulation time: The parameter values are resolved during the simulation. This is done by using channels like sc_signal<>.

The above-mentioned methods and their applications are explained in detail in [34]. This section discusses the compile time resolution method since this is used to model pPIM in this work.

5.2 Compile-Time Resolution

Compile-Time Resolution is the best method for parameterization preferred over elaboration-time resolution and simulation-time resolution [34]. This is because errors that occur because of faulty parameters can be found at an earlier stage. When data type compatibility issues exist in the code, the C++ compiler will report issues at compile time. Better code optimization can be done by C++ compilers at compile time. In the case of elaboration time resolution and simulation time resolution, there may be portability issues because not all synthesis tools support these two methods of resolution of parameter values and types. As mentioned earlier, SC_MODULE and SC_CTOR macros cannot be used for parameterization. Instead, C++ constructors are used along with the SC_HAS_PROCESS macro because the pPIM model uses lots of processes like SC_METHOD, SC_THREAD, and SC_CTHREAD.

Figure 5.1 is the module written in SystemC for register256 written in a ".h" file and shows a

```
template <int size, int reg_count> class register256 : public sc_module {
public:
    sc_in<sc_bv<size> > DATA_IN;
    sc_in<bool> WRITE;
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_out<sc_bv<size> > DATA_OUT;
    sc_bv<size> DATA;


    SC_HAS_PROCESS(register256);
    register256(sc_module_name name) : sc_module(name)
    {
        SC_METHOD(prc_register256);
        sensitive << clk << reset;


    }

    void prc_register256 ()
    {
        if(reset)
            DATA = 0;
        else if(WRITE)
            DATA = DATA_IN;
        DATA_OUT = DATA;
        cout<< "at time: " << sc_time_stamp()<< endl << "DATA_OUT = "<< DATA_OUT <<endl;

    }

};
```

Figure 5.1: Example code for Compile-Time Resolution

model for a parameterized register using compile time resolution method. As can be seen that the module has a template for parameterization. In this design, the register size is parameterized. Note that the size is declared in the template and not in the constructor. The prc_register256 is the process of type SC_METHOD that is invoked when the members in the sensitivity list change. This is where all the computations of the design take place. To assist debugging, values can be printed out just like it is done in C++ using *cout* as shown in Figure 5.1.

The "sc_main" function is a starting point like the main function in C++. This contains all the ports declared as sc_signal and connected to the members of the instantiated instances similar to Verilog. It can be seen in Figure 5.2 that a register size of 64 is passed to the register256 module with an instance name "r0". Similarly, the values are given in the stimulus which is also treated as a module and instantiated. The values from the stimulus are given to the design via ports that are declared as "sc_signal" data type. This ensures that the values are concurrently given to the design and reflect the values immediately when there is any change. The stimulus is also set to a size of 64 and is passed to the template with the instance name "stim0". The method "sc_start" starts the simulation phase where the initialization and execution take place. The number passed as an argument is the amount of time the simulation is required to run and the second argument is the unit of time. In Figure 5.2 the simulation is made to run for 10ns.

Since the pPIM core and cluster are hierarchical modules, it is important to know the instantiation of modules within modules. This can be seen implemented in the register file module which is a part of the pPIM core. Figure 5.3 shows the code snippet for the register file. The module "register_file" has two parameters N and M used to calculate the sizes of the I/O ports as shown in 5.3. The register file is a module containing a set of registers in it. This is done by instantiating the register module discussed in 5.1 multiple times. For multiple instantiations of the same module or for replicating generate statements in Verilog, vectors from C++ can be used. Vectors can be declared as a type of module that needs to be instantiated because a module in SystemC is

```
int sc_main(int argc, char* argv[]) {


    sc_signal<sc_bv<64> > DATA_IN0;
    sc_signal<sc_bv<6> > WRITE_ADDR0;
    sc_signal<bool> WRITE_EN0;
    sc_signal<bool> READ_EN0;
    sc_clock clk("c1", 1, SC_NS);
    sc_signal<bool> reset0;
    sc_signal<sc_bv<64> > DATA_OUT0;

    sc_signal<bool> And_sig;
    WRITE_ADDR0 = 111011;
    WRITE_EN0 = true;
    And_sig = WRITE_EN0 and WRITE_ADDR0[0];

    //create instance

    register256<64> r0("r0");


    r0.DATA_IN(DATA_IN0);
    r0.WRITE(And_sig);
    r0.clk(clk);
    r0.reset(reset0);
    r0.DATA_OUT(DATA_OUT0);

    stimulus<64> stim0("stim0");
    stim0.Data_in(DATA_IN0);
    stim0.write_en(WRITE_EN0);
    stim0.reset(reset0);
    stim0.write_addr(WRITE_ADDR0);



    //invoke the simulator

    sc_start(10,SC_NS);
    return 0;



}
```

Figure 5.2: Main Function for register256

nothing but a class. In addition to multiple instantiations of modules, vectors can also be used as a container to store the values that come from these multiple instances. This is implemented in the methods of the register_file module shown in Figure 5.4. The method “combine_output” is written to flatten out the outputs received from M registers into one long bit vector of size M*N and produced as the final output of the register_file module.

An important point to keep in mind is that the method range() cannot be used with the I/O ports directly. The method range in SystemC is used for bit slicing. Note that the combine_output method has a local declaration of a bit vector “dout” to first store the contents of data_out so range() can be used and then written to the final output DATA_OUT.

These methods are then triggered using the sensitivity list as shown in Figure 5.3. It is critical to choose the correct member for the sensitivity list for different SC_METHODS of the module. If this is not decided correctly, the module will not behave as expected. Figure 5.5 shows the result on the console and Figure 5.6 shows the result observed on the waveform window. The result can be observed either way using SystemC depending on the requirement or the complexity of the system design.

Figure 5.7 shows a block diagram of the pPIM core from a SystemC perspective to get an overview about the ports and their data types. In SystemC, not all data types are compatible with every other data type. So this requires consciousness of which two data types are being used together. In Figure 5.7 the blue block is a top-level view of the pPIM core and the blocks on the left are the components inside the pPIM core. The white block is the Decoder, and the bright green block is the register_file with a smaller sap green block of register256. The yellow block is the “nbit_multiplexer” which is a module containing a set of multiplexers and the blue block inside it is a simple multiplexer. As can be seen that the output ports of some modules are required to be connected to other ports that are of different data types. For example, the output port of register256 is of type “sc_biguint” and the output port of the register file is “sc_bv”. It is not

```
template <unsigned N, unsigned M> class register_file : public sc_module
{
public:

    sc_in<sc_bv<N> > DATA_IN;
    sc_in<sc_bv<2*M> > WRITE_ADDR;
    sc_in<bool> WRITE_EN;
    sc_in<bool> READ_EN;
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_out<sc_bv<M*N> > DATA_OUT; //384

    sc_bv<2*M> write_addr_var;
    bool write_en_var;


    sc_vector<register256<N>> regs;
    sc_vector<sc_signal<sc_bv<N>>> data_out;
    sc_vector<sc_signal<bool>> AND_sig;
    unsigned i;

    SC_HAS_PROCESS(register_file);

  register_file(sc_module_name name): sc_module(name), regs("regs", M), data_out("data_out", N), AND_sig("AND_sig", M)
    {
        SC_METHOD(combine_output);
        sensitive << data_out;
        SC_METHOD(addr_computation);
        sensitive << WRITE_ADDR << WRITE_EN;

        for(auto i=0U; i< M ; ++i)
        {

          // connect the register
            regs[i].clk(clk);
            regs[i].reset(reset);
            regs[i].DATA_IN(DATA_IN);
            regs[i].WRITE_EN(AND_sig[i]);
            regs[i].DATA_OUT(data_out[i]);

        }
    }
```

Figure 5.3: Code snippet for register file

```
        for(auto i=0U; i< M ; ++i)
        {

          // connect the register
            regs[i].clk(clk);
            regs[i].reset(reset);
            regs[i].DATA_IN(DATA_IN);
            regs[i].WRITE_EN(AND_sig[i]);
            regs[i].DATA_OUT(data_out[i]);

        }
    }

  void combine_output() {
        sc_bv<M*N> dout;
        //dout = 0;

        for(auto i=0U; i< M; ++i)
        {
          dout.range(((i+1)*N)-1,i*N)=data_out[i].read();
        }
        DATA_OUT.write(dout);

    }

    void addr_computation()
    {
        write_en_var = WRITE_EN;

        for(auto i=0U; i<M; ++i)
        {
            (AND_sig[i] = write_en_var && WRITE_ADDR.read()[i]);
        }
    }

};
```

Figure 5.4: Methods of register_file

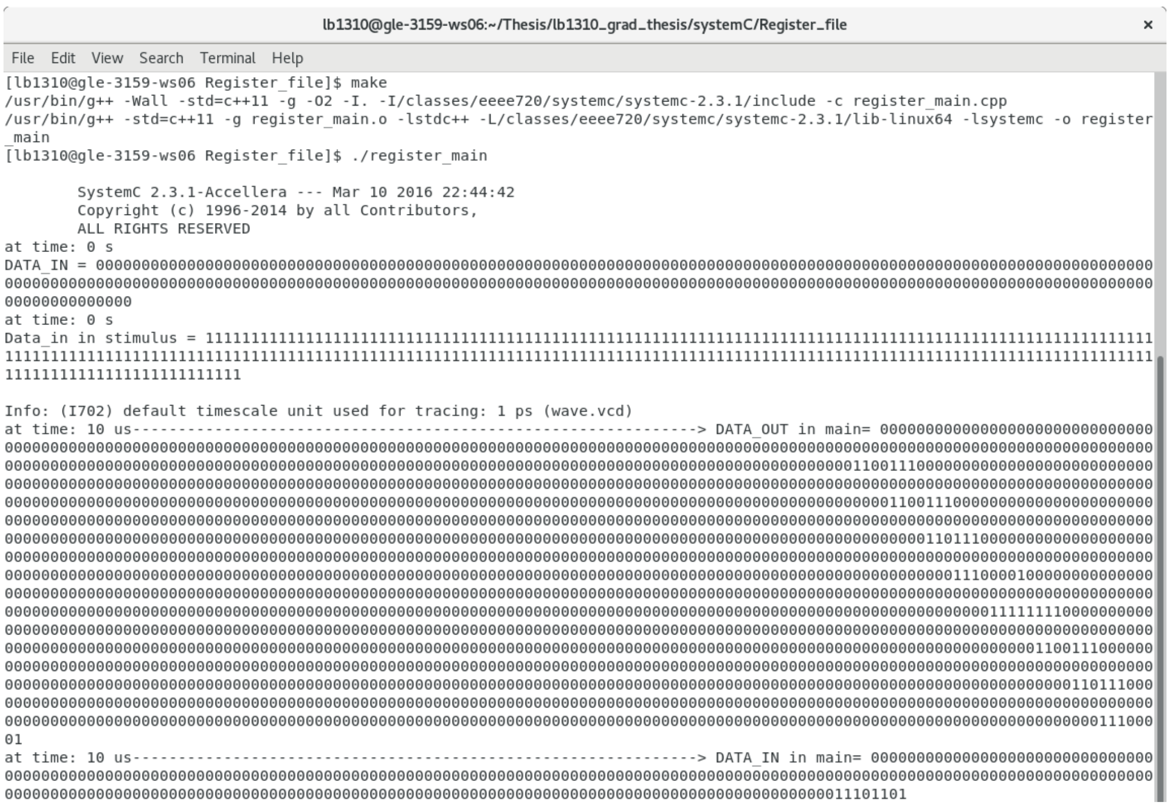

Figure 5.5: Result on terminal

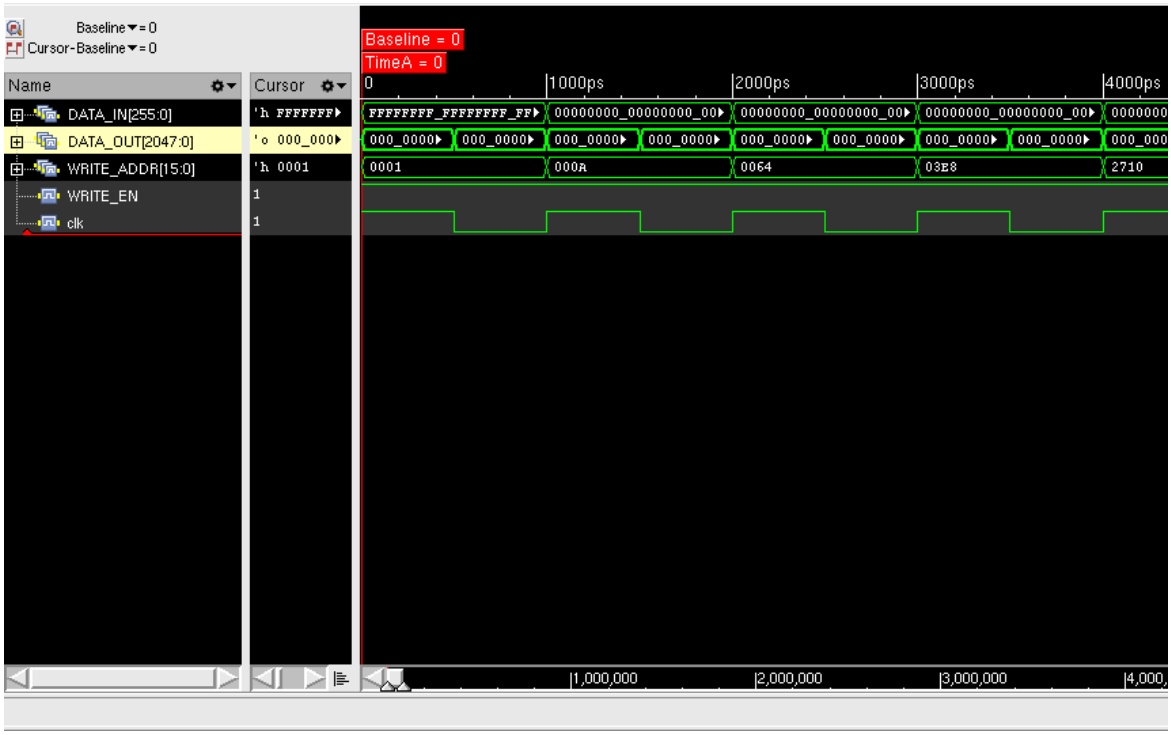

Figure 5.6: Result on waveform window

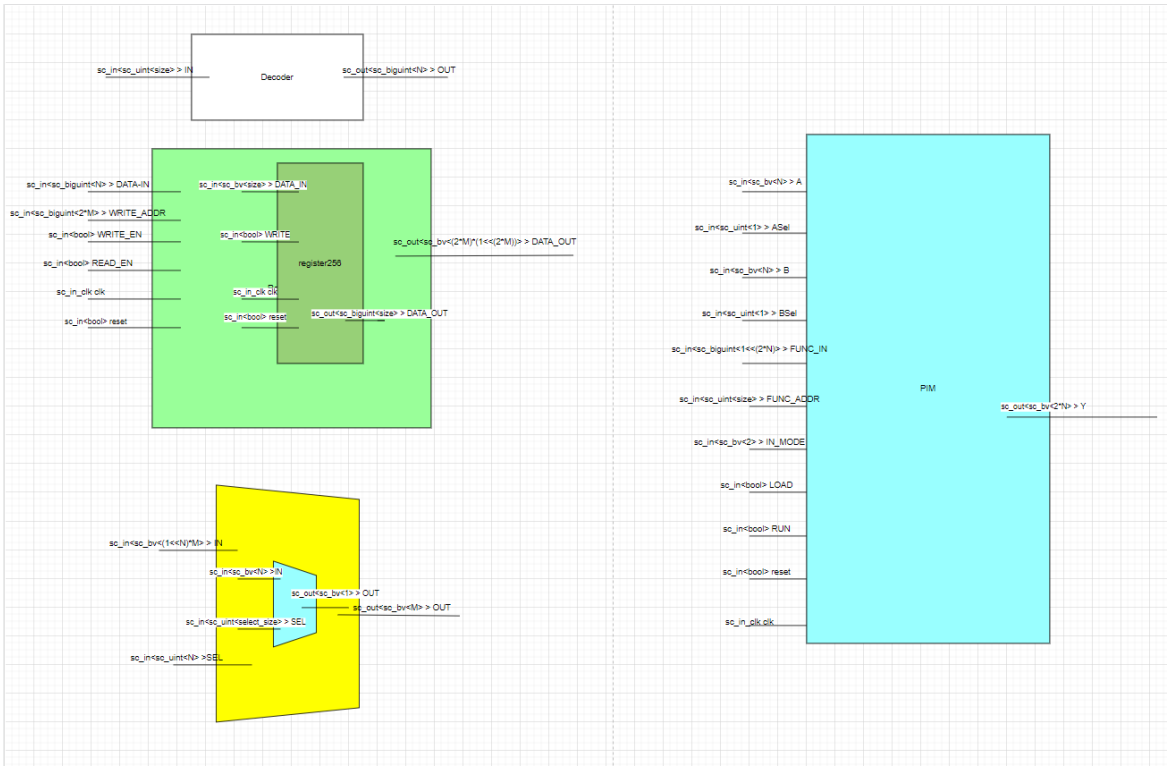

Figure 5.7: pPIM block diagram from SystemC perspective

possible to directly assign “sc_biguint” to “sc_bv”. This will require a separate method written for this kind of typecasting with an intermediate local temporary variable in the method.

The SystemC model for the cluster is a more complex code compared to the simpler modules discussed so far. This is because the cluster integrates all nine cores and the communication between them is critical. Moreover, some of the features of SystemC do not behave as they do in Verilog. For example, the way the cluster establishes communication between the cores is by having the same variables shared amongst them by using bit slicing. In the cluster, the outputs from all the cores are stored in different parts of a variable called “sY”. Unlike Verilog, some operations like bit select and bit slicing cannot be directly written by using the square brackets. Certain rules are required to be followed in terms of data types as discussed earlier. So each of the operations needs to be carried out using separate methods. These methods need

```
#define SC_INCLUDE_FX
#include "systemc.h"
#include "PIMC_stimulus.h"
#include "PIM_Cluster.h"

int sc_main(int argc, char* argv[])
{
    sc_clock clk("clock", 6, SC_NS);

    PIMC_stimulus<8,9> cluster_stim("cluster_stim");
    cluster_stim.clk(clk);
    sc_start(3000,SC_NS);

    return 0;

}
```

Figure 5.8: Code snippet for main function of the pPIM cluster

to be sensitive only to the corresponding bit slices and not the other slices. The same method needs to be followed even for updating these variables. The corresponding bit slices have to be updated without disturbing the other parts of the variable. The update of outputs from all the cores and retrieval of bits from these commonly shared variables as input to the cores needs to be synchronous. If they are mismatched, it is not possible to obtain the desired output.

The stimulus of the pPIM cluster is also complex and requires the implementation of the semantics discussed in the previous chapters. The stimulus is also slightly different from the ones written for the smaller modules of the pPIM. For the pPIM cluster, the main function is used only to generate the system clock using “sc_clock”, to start the simulation using “sc_start”, and to instantiate the cluster stimulus as shown in figure 5.8. Typically the instantiation of the module and the stimulus happen in the main function along with the “sc_trace” used for dumping signals for waveforms. The reason for this difference is that for the cluster, it is critical for the inputs to be given to the design concurrently whenever there is a change in the stimulus.

Figure 5.9 shows a snippet of the test bench stimulus of the pPIM cluster. The PIMC_stimulus

```
PIM_Cluster<8,9> clusters{"clusters"};

SC_HAS_PROCESS(PIMC_stimulus);

PIMC_stimulus(sc_module_name name) : sc_module(name)
{

    SC_CTHREAD(test_bench,clk.neg());

    //Cluster instantiation
    clusters.A(A);
    clusters.B(B);
    clusters.FUNC(FUNC);
    clusters.FUNC_ADDR(FUNC_ADDR);
    clusters.LOAD_CORE(LOAD_CORE);
    clusters.AIN_ADDR(AIN_ADDR);
    clusters.BIN_ADDR(BIN_ADDR);
    clusters.SRC_MODE(SRC_MODE);
    clusters.IN_MODE(IN_MODE);
    clusters.RUN(RUN);
    clusters.reset(reset);
    clusters.clk(clk);
    clusters.Y_ADDR(Y_ADDR);
    clusters.Y(Y);

}

void start_of_simulation()
{
    fp = sc_create_vcd_trace_file("PIM_Cluster_waveform");
    if(!fp) cout << "There was an error!!" << endl;
        sc_trace(fp,A,"A");
        sc_trace(fp,B,"B");
        sc_trace(fp,FUNC,"FUNC");
        sc_trace(fp,FUNC_ADDR,"FUNC_ADDR");
        sc_trace(fp,LOAD_CORE,"LOAD_CORE");
        sc_trace(fp,AIN_ADDR,"AIN_ADDR");
        sc_trace(fp,BIN_ADDR,"BIN_ADDR");
        sc_trace(fp,SRC_MODE,"SRC_MODE");
        sc_trace(fp,IN_MODE,"IN_MODE");
        sc_trace(fp,RUN,"RUN");
        sc_trace(fp,Y_ADDR,"Y_ADDR");
        sc_trace(fp,reset,"reset");
        sc_trace(fp,Y,"Y");
        sc_trace(fp,ACC,"ACC");
        sc_trace(fp,clk,"clk");

        sc_trace(fp,clusters.pimcore0.A,"PIMCORE_0.A");
        sc_trace(fp,clusters.pimcore0.B,"PIMCORE_0.B");
        sc_trace(fp,clusters.pimcore0.Y,"PIMCORE_0.Y");
```

Figure 5.9: Code snippet for stimulus of the pPIM cluster

is a module that acts as the stimulus. Inside the stimulus is the instantiation of the PIM_cluster as shown in 5.9. The actual test bench is written as a clocked thread using SC_CTHREAD since the cluster is a fully synchronous design that is statically sensitive to the clock. The SC_CTHREAD is also faster than SC_THREAD because SC_CTHREAD does not suspend and resume at each wait() method. When SC_THREAD was used instead of SC_CTHREAD, the simulation would run only for the first set of values. This is because once the SC_THREAD exits, it never runs again. The SC_CTHREAD on the other hand restarts on the next clock edge. The SC_CTHREAD can be stopped by using the method called halt().

It can also be seen that the "sc_trace" method is also used here inside the stimulus to write the signal data to the dump file. Since this is being done inside a module and not in the main function, this has to be written inside the start_of_simulation() method to force it to write the dump file during the simulation phase and close the trace file in the end_of_simulation() method as shown in figure 5.10.

Similar to the RTL, the test bench contains a LUT that consists of pre-calculated results for addition and multiplication. The LUTs are loaded to the corresponding cores by writing appropriate values to the LOAD_CORE port. The other stimulus values are written at consecutive wait() methods as shown in Figure 5.11. The stimulus values are given for all nine cycles for each value of A and B, unlike the RTL. The test bench is now complete and the results can be seen in the next chapter 6.

```
void start_of_simulation()
{
    fp = sc_create_vcd_trace_file("PIM_Cluster_waveform");
    if(!fp) cout << "There was an error!!" << endl;
        sc_trace(fp,A,"A");
        sc_trace(fp,B,"B");
        sc_trace(fp,FUNC,"FUNC");
        sc_trace(fp,FUNC_ADDR,"FUNC_ADDR");
        sc_trace(fp,LOAD_CORE,"LOAD_CORE");
        sc_trace(fp,AIN_ADDR,"AIN_ADDR");
        sc_trace(fp,BIN_ADDR,"BIN_ADDR");
        sc_trace(fp,SRC_MODE,"SRC_MODE");
        sc_trace(fp,IN_MODE,"IN_MODE");
        sc_trace(fp,RUN,"RUN");
        sc_trace(fp,Y_ADDR,"Y_ADDR");
        sc_trace(fp,reset,"reset");
        sc_trace(fp,Y,"Y");
        sc_trace(fp,ACC,"ACC");
        sc_trace(fp,clk,"clk");

        sc_trace(fp,clusters.pimcore0.A,"PIMCORE_0.A");
        sc_trace(fp,clusters.pimcore0.B,"PIMCORE_0.B");
        sc_trace(fp,clusters.pimcore0.Y,"PIMCORE_0.Y");

        sc_trace(fp,clusters.pimcore[0].A,"PIMCORE_1.A");
        sc_trace(fp,clusters.pimcore[0].B,"PIMCORE_1.B");
        sc_trace(fp,clusters.pimcore[0].Y,"PIMCORE_1.Y");

        sc_trace(fp,clusters.pimcore[1].A,"PIMCORE_2.A");
        sc_trace(fp,clusters.pimcore[1].B,"PIMCORE_2.B");
        sc_trace(fp,clusters.pimcore[1].Y,"PIMCORE_2.Y");

        sc_trace(fp,clusters.pimcore[2].A,"PIMCORE_3.A");
        sc_trace(fp,clusters.pimcore[2].B,"PIMCORE_3.B");
        sc_trace(fp,clusters.pimcore[2].Y,"PIMCORE_3.Y");

        sc_trace(fp,clusters.pimcore[3].A,"PIMCORE_4.A");
        sc_trace(fp,clusters.pimcore[3].B,"PIMCORE_4.B");
        sc_trace(fp,clusters.pimcore[3].Y,"PIMCORE_4.Y");

        sc_trace(fp,clusters.pimcore[4].A,"PIMCORE_5.A");
        sc_trace(fp,clusters.pimcore[4].B,"PIMCORE_5.B");
        sc_trace(fp,clusters.pimcore[4].Y,"PIMCORE_5.Y");

        sc_trace(fp,clusters.pimcore[5].A,"PIMCORE_6.A");
        sc_trace(fp,clusters.pimcore[5].B,"PIMCORE_6.B");
        sc_trace(fp,clusters.pimcore[5].Y,"PIMCORE_6.Y");

        sc_trace(fp,clusters.pimcore[6].A,"PIMCORE_7.A");
        sc_trace(fp,clusters.pimcore[6].B,"PIMCORE_7.B");
        sc_trace(fp,clusters.pimcore[6].Y,"PIMCORE_7.Y");

        sc_trace(fp,clusters.pimcoreN.A,"PIMCORE_N.A");
        sc_trace(fp,clusters.pimcoreN.B,"PIMCORE_N.B");
        sc_trace(fp,clusters.pimcoreN.Y,"PIMCORE_N.Y");



}

void end_of_simulation()
{
    sc_close_vcd_trace_file(fp);
}
```

Figure 5.10: Trace files in simulation phase

```
wait();//1
A.write(36);
B.write(129);
IN_MODE.write(0x3FC00);
SRC_MODE.write(0x3FC00);
AIN_ADDR.write(0x223300000);
BIN_ADDR.write(0x010100000);


wait();//2
//wait(6,SC_NS);
B_copy = B.read();
B_copy.range(3,0) = ACC.range(3,0);
B.write(B_copy);
IN_MODE.write(0x00333);
SRC_MODE.write(0x00001);
AIN_ADDR.write(0x0B0C00);
BIN_ADDR.write(0xD0F0E);
RUN.write(0x1E0);


wait();//3
Y_copy = Y.read();
ACC = Accumulator.read();
ACC.range(3,0) = Y_copy.range(3,0);
Accumulator.write(ACC);
Y_copy = Y.read();
B_copy.range(3,0) = ACC.range(7,4);
B.write(B_copy);
IN_MODE.write(0x003BB);
SRC_MODE.write(0x00113);
AIN_ADDR.write(0x80800);
BIN_ADDR.write(0x87A38);
RUN.write(0x015);

wait();//4
ACC = Accumulator.read();
B_copy.range(3,0) = ACC.range(11,8);
B.write(B_copy);
IN_MODE.write(0x00357);
SRC_MODE.write(0x00201);
AIN_ADDR.write(0x17638);
BIN_ADDR.write(0x00002);
RUN.write(0x015);
Y_ADDR.write(0x0000);

wait();//5
Y_copy = Y.read();
ACC = Accumulator.read();
ACC.range(7,4) = Y_copy.range(3,0);
Accumulator.write(ACC);
B_copy.range(3,0) = ACC.range(15,12);
B.write(B_copy);
IN_MODE.write(0x003E3);
SRC_MODE.write(0x00342);
AIN_ADDR.write(0x08006);
BIN_ADDR.write(0x89208);
RUN.write(0x01B);

wait();//6
IN_MODE.write(0x00333);
SRC_MODE.write(0x00121);
AIN_ADDR.write(0x80408);
BIN_ADDR.write(0x10802);
RUN.write(0x01D);
```

Figure 5.11: Stimulus for pPIM cluster

Chapter 6

Results and Discussion

In this thesis, the system level hierarchical module of the pPIM Cluster was modeled after verifying the subsystem pPIM core and all the modules in it by writing isolated test benches for each of the SystemC models. The results in this chapter include screenshots of the waveform and console outputs obtained from the RTL test bench and SystemC model of the pPIM core and the pPIM cluster. The results also include outputs of the SystemC models of individual isolated modules.

6.1 Results for register file

Since the register file is a simple design, the output can be observed on the console without requiring tools to observe the outputs on waveforms. The figure 6.2 shows the output on the console for the SystemC model of the register file. For easier readability, the output can also be seen in the form of simulation waveform as shown in Figure 6.1. Here DATA_IN is the input signal and DATA_OUT is the output signal. The eight set of registers of width 256 bits each, are flattened out into one single output DATA_OUT of width 2048 bits in this configuration. However

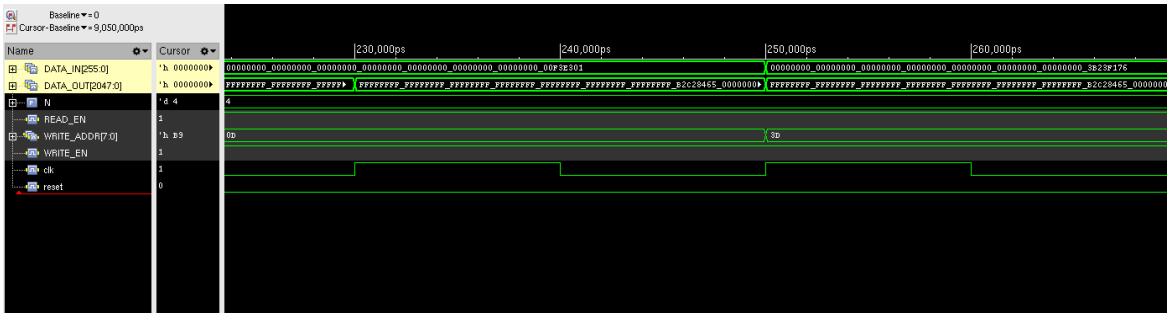

Figure 6.1: SystemC waveform for register file

the number of registers, the width of input and output vary for different parameter values.

6.2 Results for register256

The simulation waveform results for the isolated register256 module is seen in Figure 6.3. The size of the register is parameterized in this model. The configuration of the model output in the figure is a register of width 256 bits. The input signal is DATA_IN and the output is DATA_OUT. Figure 6.4 shows the console output for the SystemC model of register256.

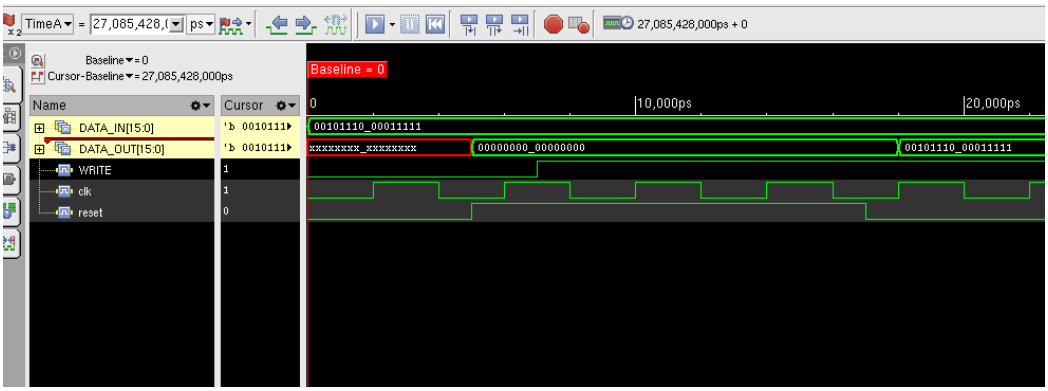

Figure 6.3: SystemC waveform for register256

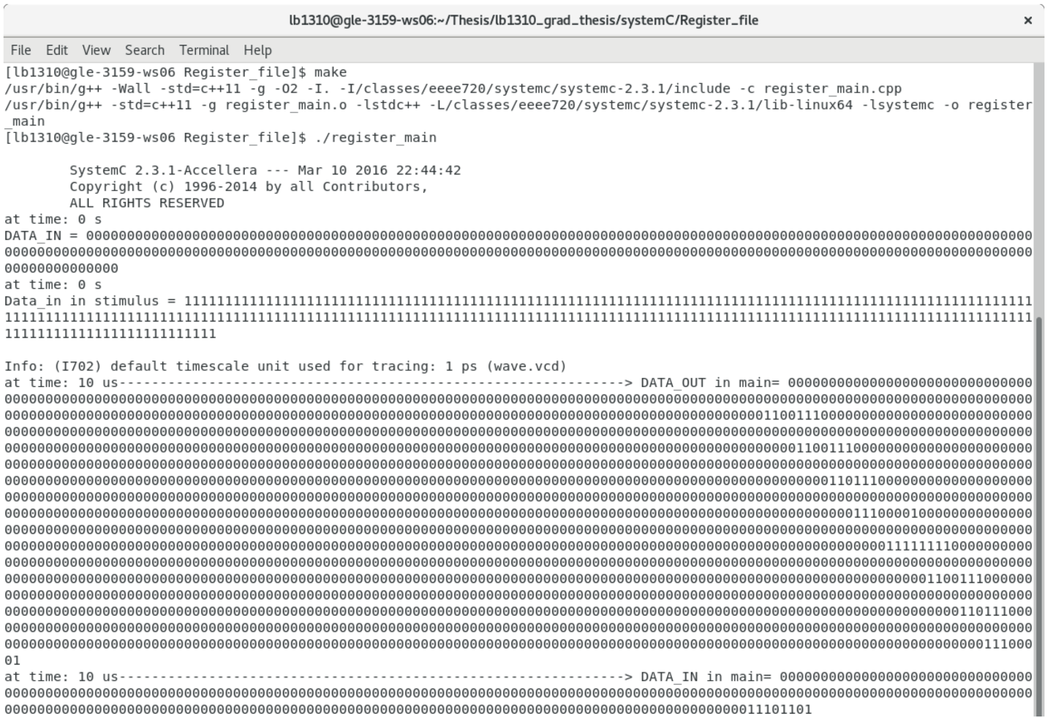

Figure 6.2: SystemC output on console

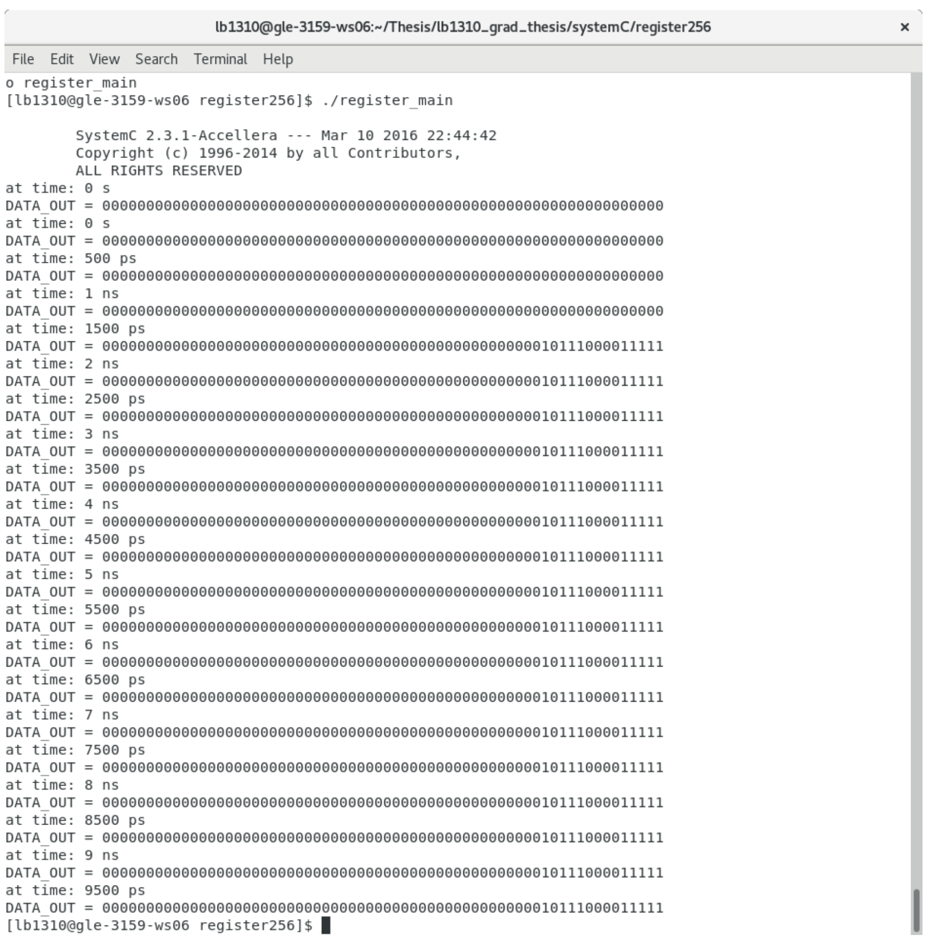

Figure 6.4: SystemC output on console for register256

6.3 Results for Decoder

The simulation waveform results for the SystemC model of the decoder is shown in Figure 6.5. The module decodes the input and gives a corresponding value as output. The number of bits of the input and output ports are parameterized in this model.

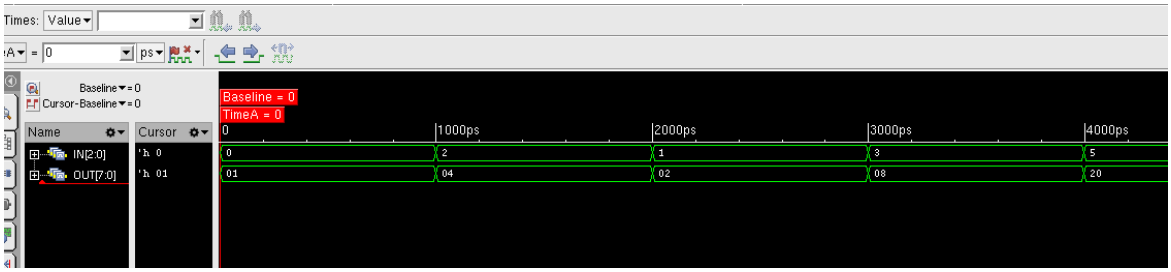

Figure 6.5: SystemC output for Decoder

6.4 Results for nbit_multiplexer

The simulation waveform result for the SystemC model of the nbit_multiplexer is shown in Figure 6.6. The nbit_multiplexer is a module consisting of a set of multiplexers used in the pPIM core and the pPIM cluster. Number of multiplexers and the width of the I/O ports are parameterized in this model.

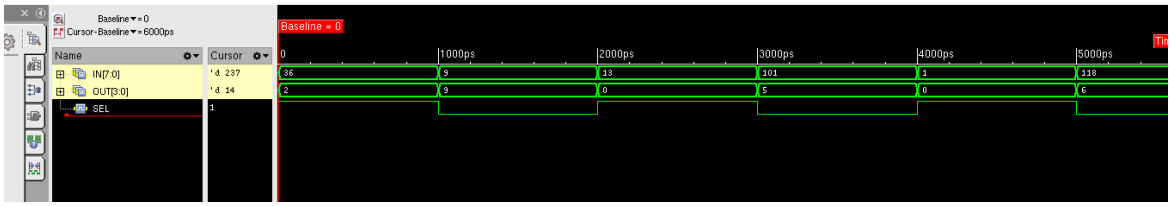

Figure 6.6: SystemC waveform for nbit_multiplexer

6.5 Result for In_multiplexer

The simulation waveform result for the SystemC model of the In_multiplexer is shown in Figure 6.7. The In_multiplexer is a module consisting of a set of nbit_multiplexer modules used in the pPIM cluster. Number of nbit_multiplexers and the width of the I/O ports are parameterized in this model.

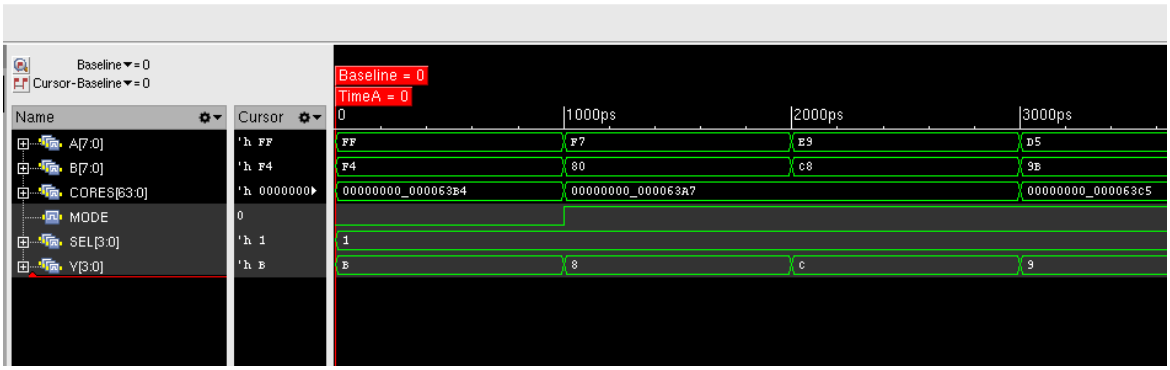

Figure 6.7: SystemC waveform for In_multiplexer

6.6 Results for pPIM core

Figure 6.8 shows the waveform result for the pPIM core. Registers A and B have randomized values given from the test bench. The core is made to perform addition by loading the LUT for addition. The sum of A and B is observed in Y as seen in the waveform. Figure 6.9 shows the waveform obtained from the SystemC model of the pPIM core. From the waveforms, the model is also able to perform addition like the RTL.

The pPIM core is made to perform multiplication by loading the LUT for multiplication. The product of A and B is observed in Y as seen in the waveform. Figure 6.10 shows the waveform obtained from the SystemC model of the pPIM core performing multiplication correctly.

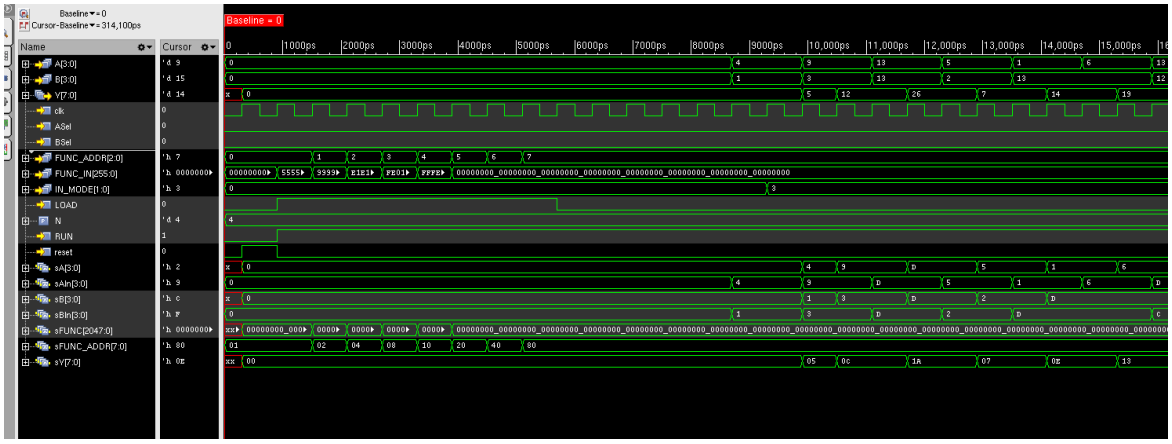

Figure 6.8: RTL waveform for pPIM core

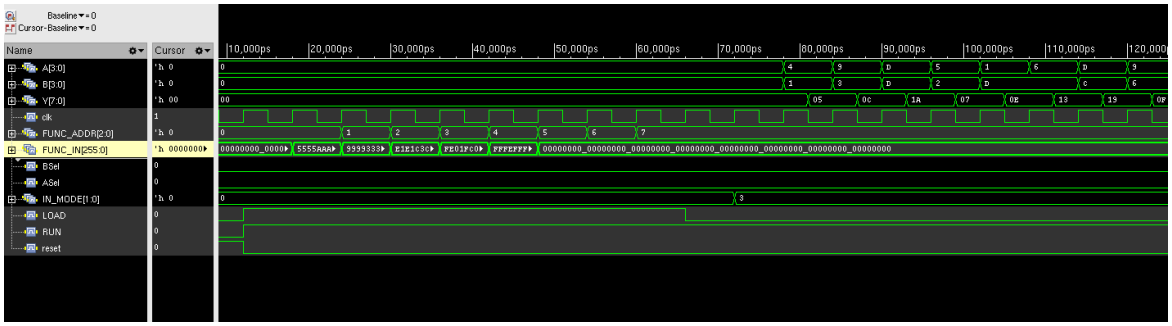

Figure 6.9: SystemC waveform for pPIM core performing addition

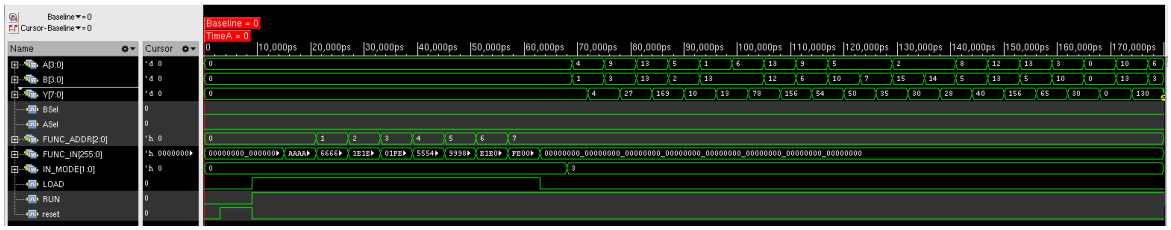

Figure 6.10: SystemC waveform for pPIM core performing multiplication

6.7 Results for pPIM cluster

Figure 6.12 shows the waveform loading the LUT for addition through the port FUNC which stands for the Function word. The LUT is pre-calculated values for all combinations of the

operands. The number of combinations varies based on the number of bits of the operands. The LUT function words are generated using a nested for-loop in which the pre-calculation takes place as shown in Figure 6.14; in this case the for-loop generating the LUT values is included in the Verilog testbench. Figure 6.13 shows the waveform loading the LUT for addition in the SystemC testbench. Figure 6.11 shows the color code followed to identify waveforms of the different cores. The green waveforms on the top are values from the top level of the design/model. Figure 6.15 shows the for loop generating the LUT in the SystemC testbench. Note that some cores are loaded with addition LUT and others are loaded with multiplication LUT. This is done by giving the correct values in the LOAD port. One of the reasons the RTL testbench was not working correctly was that it had wrong values in the LOAD port while loading the LUT.

Figure 6.11: Color code for waveforms of different cores

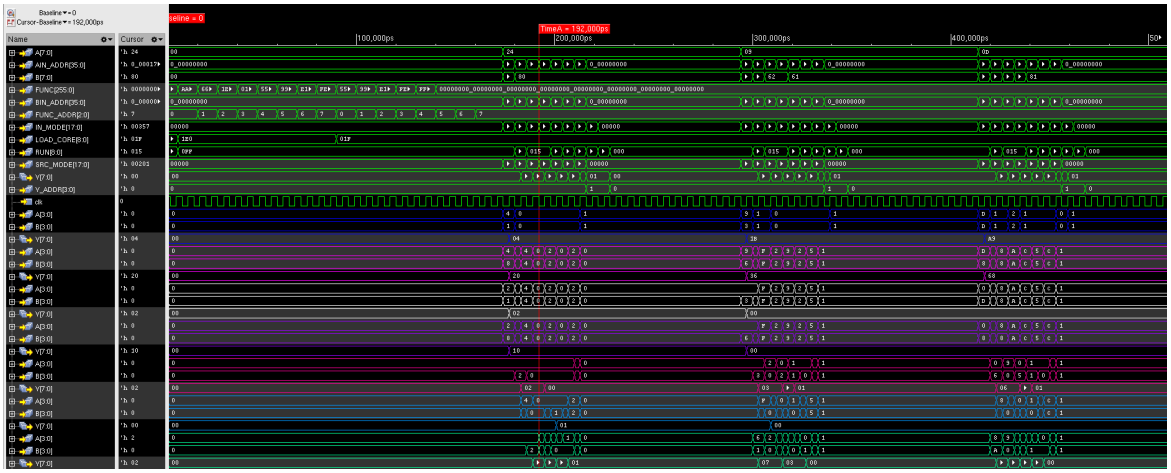

Figure 6.12: RTL waveform showing loading Look Up Table

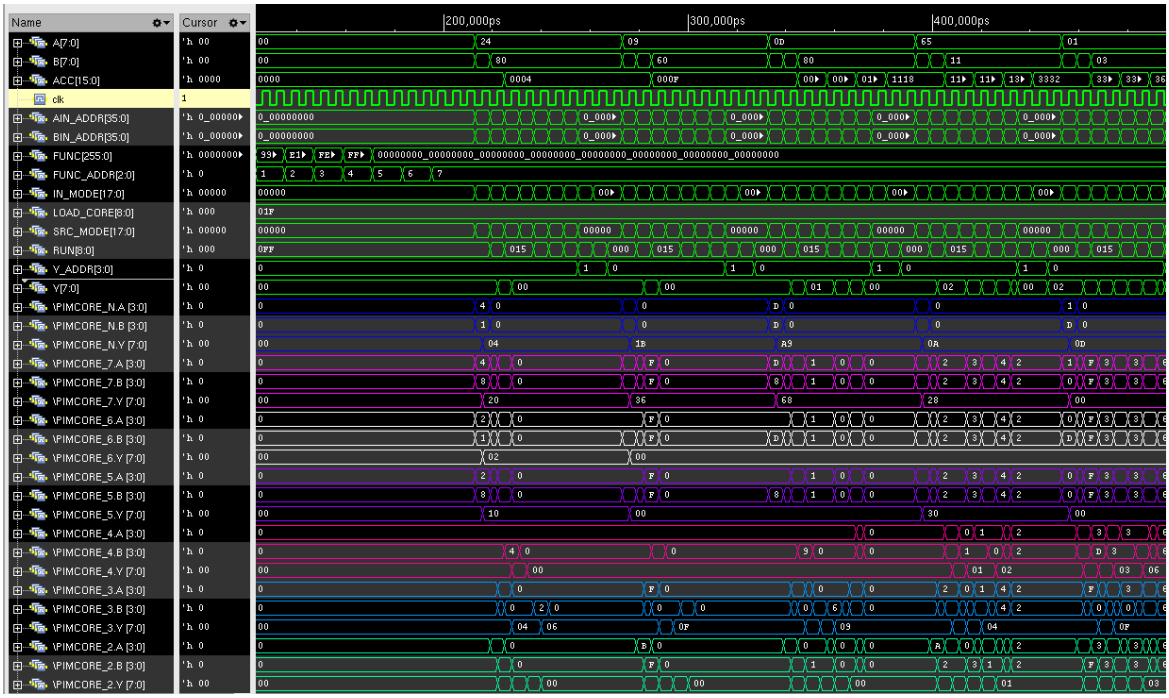

Figure 6.13: SystemC waveform showing loading Look Up Table

```
//Generate Function Words for Multiplication and Addition
for (a=0;a<16;a=a+1) begin
    for (b=0;b<16;b=b+1) begin
        add = a + b;
        mult = a * b;
        ADD_tmp = add;
        MULT_tmp = mult;
        for (kk=0;kk<8;kk=kk+1) begin

            FUNC_MULT[(kk*256)+index] = MULT_tmp[kk];
            FUNC_ADD[(kk*256)+index] = ADD_tmp[kk];
        end
        index = index + 1;
    end
end


#5;
reset = 1;
#5;
reset = 0;

RUN = 9'hff;


//Load Multiplication Function Worxds into cores 8:0
for (ii=0;ii<8;ii=ii+1) begin
  FUNC = FUNC_MULT[(ii*256) +: 256];
  $display("FUNC in mult= 0x%0h",FUNC);
  FUNC_ADDR = ii;
 LOAD_CORE = 9'h1E0;
  #10;
end


//Load Addition Function Words into cores 4:0
for (ii=0;ii<8;ii=ii+1) begin
  FUNC = FUNC_ADD[(ii*256) +: 256];
  FUNC_ADDR = ii;
  LOAD_CORE = 9'h1F;
  #10;
end
```

Figure 6.14: LUT in RTL testbench

```
void test_bench()
{

    for( a=0; a<16; ++a)
    {

        for( b=0; b<16; ++b)
        {

            add = a + b;
            mult = a*b;

            ADD_tmp = add;
            MUL_tmp = mult;

            for( auto kk=0U;kk<8;++kk)
            {
                FUNC_MULT[(kk*256)+index] = MUL_tmp[kk];
                FUNC_ADD[(kk*256)+index] = ADD_tmp[kk];

            }
            ++index;

        }
    }
            func_add_copy = FUNC_ADD;
            func_mult_copy = FUNC_MULT;


    wait();
    reset.write(1);
    wait();
    reset.write(0);


    RUN.write(0xff);


    for(auto i = 0; i<8; ++i)
    {
        func_mult_slice = FUNC_MULT.range(((i*256)+256-1),i*256);


        FUNC.write(func_mult_slice);
        FUNC_ADDR.write(i);
        LOAD_CORE.write(0x1E0);
        wait(10,SC_NS);
    }

    for(auto j = 0; j< 8; ++j)
    {
        func_add_slice = func_add_copy.range(((j*256)+256-1),j*256);
        FUNC.write(func_add_slice);
        FUNC_ADDR.write(j);
        LOAD_CORE.write(0x1F);
        wait(10,SC_NS);
    }
```

Figure 6.15: LUT in SystemC testbench

Figure 6.16 shows the RTL waveforms focusing on the cores N, 7, 6, and 5 that perform the multiplication operation. Figure 6.17 shows the SystemC waveforms for the same cores performig multiplication correctly. Initially, the values are correctly multiplied and the cores N, 7, 6, and 5 produce the same and correct result in the RTL design and the SystemC model since the SystemC pPIM core and RTL pPIM core work correctly. After 4 cycles, the write-back to the accumulator is not happening. It can be seen that the intermediate results are written to the accumulator and eventually the values in the accumulator are used for the following calculations by other cores

and so this is an important part of the communication between the cores. Since the accumulator is part of the test bench and all the transactions between the accumulator and the top-level registers happen in the test bench, there is a timing issue happening in the synchronization of intermediate results among the cores. The Figure 6.18 shows RTL waveforms focusing on the cores 4, 3, 2, 1, and 0 that perform the addition operation. The figure 6.20 shows the SystemC waveforms for the same cores.

The figures 6.19 and 6.21 are the zoomed view of the addition cores of the pPIM cluster. Since these cores depend on the values in the accumulator, they do not produce correct outputs after about the first 4 cycles. The accumulator is found to behave anomalously in both RTL and SystemC test benches which needs to be fixed as part of future work.

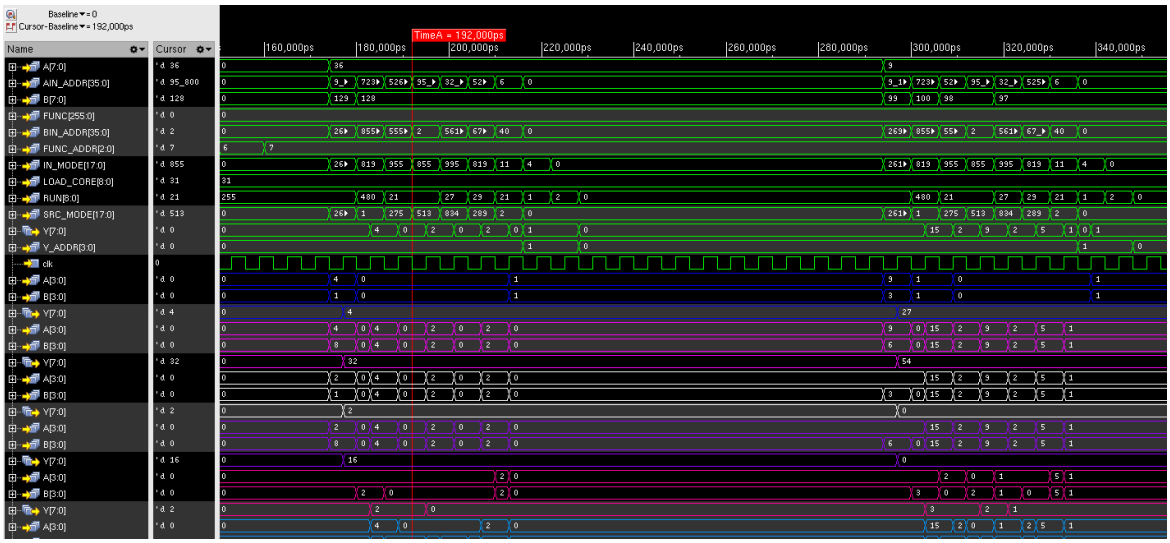

Figure 6.16: RTL waveforms for multiplication cores

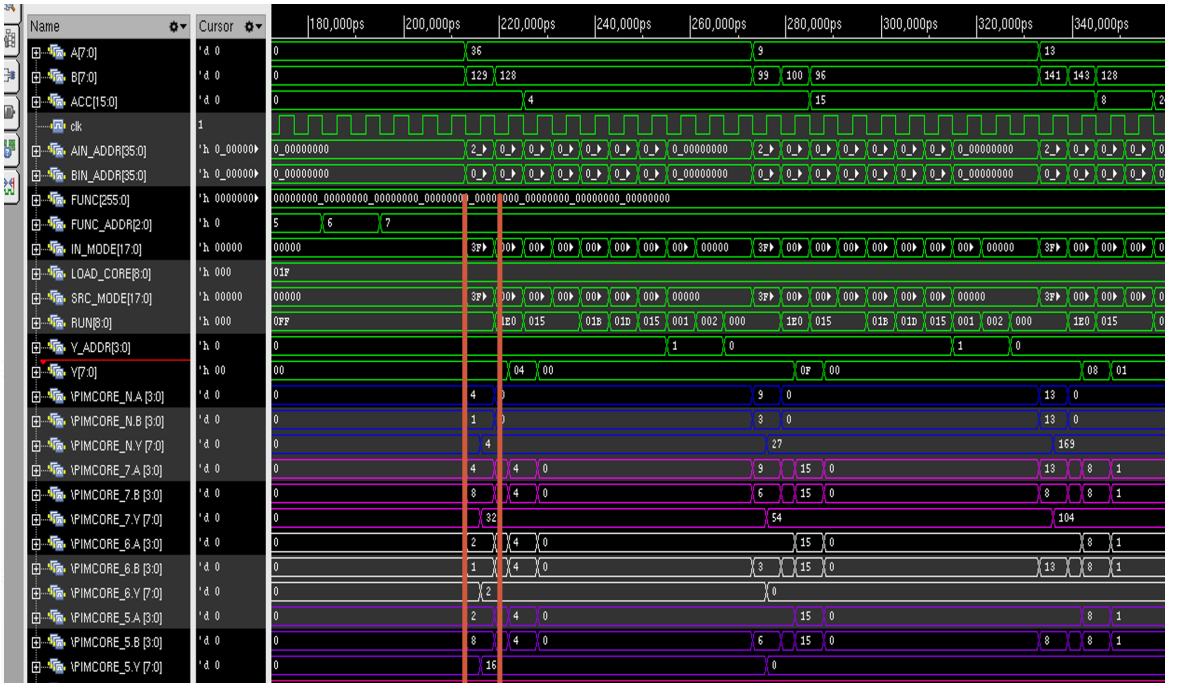

Figure 6.17: SystemC waveforms for multiplication cores

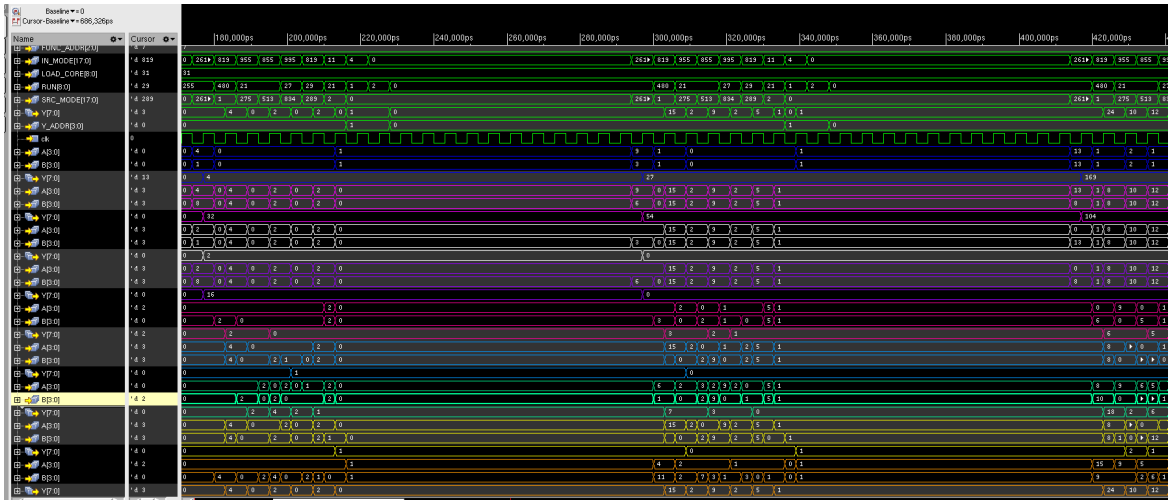

Figure 6.18: RTL pPIM cluster waveforms for addition cores

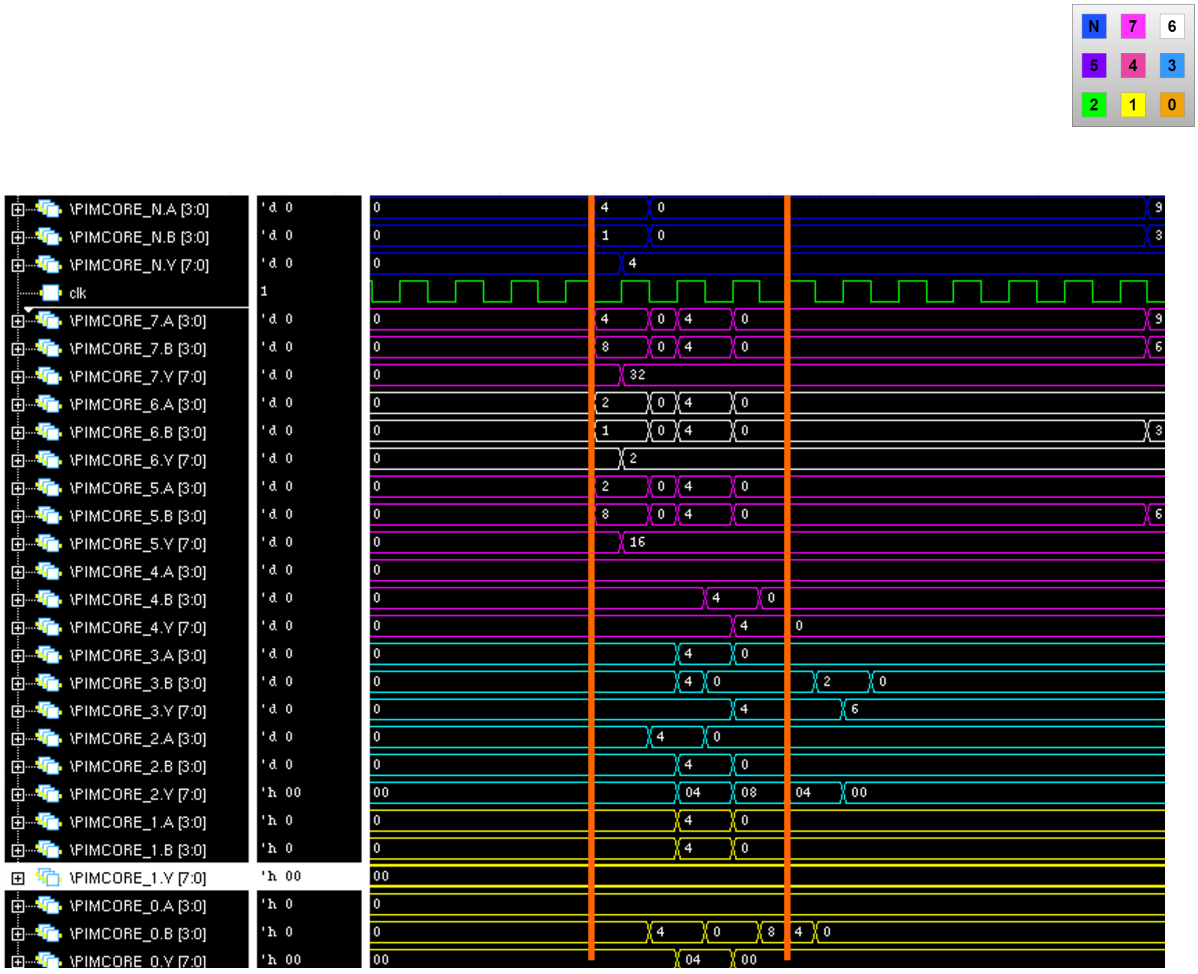

Figure 6.19: RTL pPIM cluster waveforms zoomed

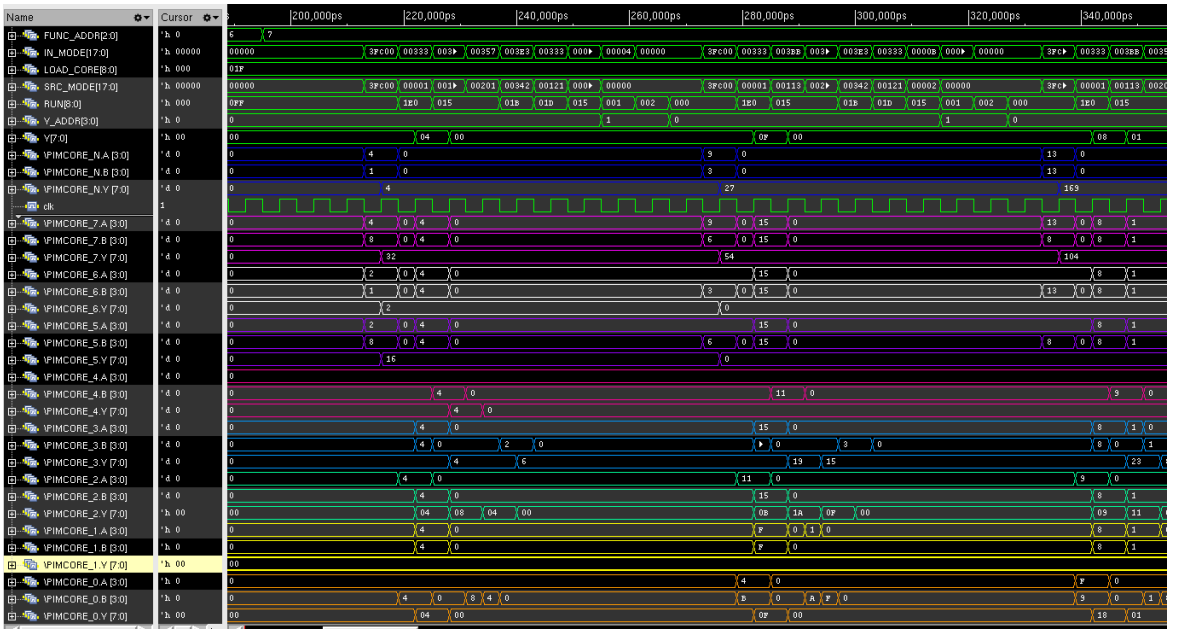

Figure 6.20: SystemC pPIM cluster waveforms for addition cores

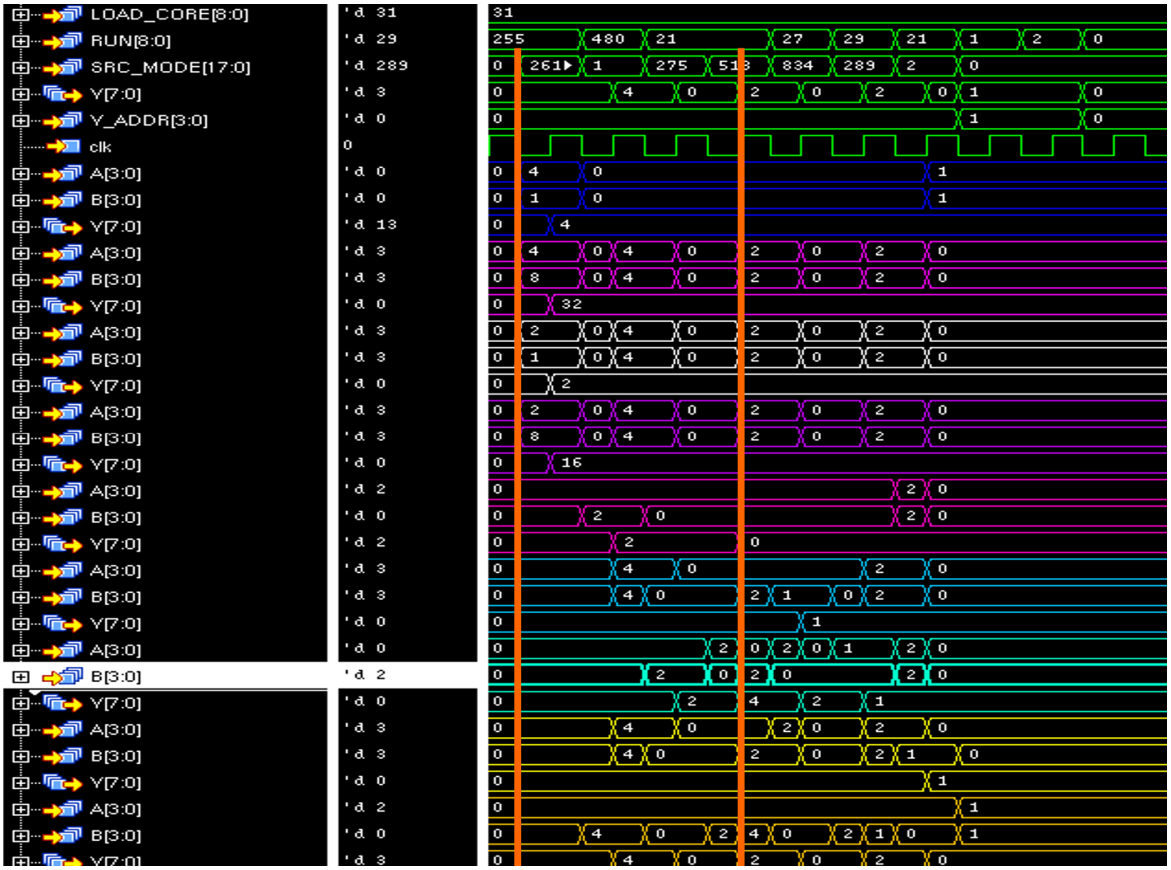

Figure 6.21: SystemC pPIM cluster wavefor zoomed

6.8 Comparison of Simulation Runtime for Verilog vs SystemC

Table 6.1 provides a comparison of the pPIM core and pPIM cluster simulation runtimes for Verilog using Cadence Design System Xcelium verses the same simulation using SystemC.

Table 6.1: Comparison of Simulation Runtime for Verilog vs SystemC

Target	**Verilog (Xcelium), seconds**	**SystemC, seconds**	**Improvement, %**
pPIM core	2.287	1.678	-26.629
pPIM cluster	1.992	0.195	-90.211

6.9 Discussion

The pPIM core and pPIM cluster were bit-exact hierarchically modeled in SystemC with the goal of flexible for architecture exploration or performance modeling. Unlike most of the work found in SystemC in which the models are flat algorithmic models that do not match the exact implementation details of the design at the Verilog RTL or structural hierarchy, the pPIM core and the pPIM cluster models are parameterized, bit-exact hierarchical models of the pPIM core and pPIM cluster Verilog models that can be used to create a platform and methodology for current and future pPIM cores and pPIM clusters. They are modeled in SystemC as opposed to Verilog to allow quick exploration of different design architectures without the use of commercial EDA tools which saves time and effort. The reusability also avoids human errors caused by performing repetitive tasks, and SystemC models of the pPIM core and pPIM cluster form a library of reusable components. If needed, compiled versions of the models can be shared protecting the intellectual property by not exposing the structural details of the design which cannot easily be done using the Verilog models[1]. The SystemC model of the pPIM core simulation waveforms match exactly with the simulation waveforms of the Verilog model. The pPIM cluster however matches only for about four clock cycles. The mismatch in the pPIM cluster waveforms happen because of two reasons. First, the accumulation of the intermediate results happen in the pPIM cluster Verilog testbench. Since

[1] Verilog models can be encrypted, however there are known compatibility issues between vendor simulators, and also freely available well known exploits for decrypting.

the values in the accumulator are continuously used by the cores for further computation, the communication between the cores to and from the testbench faces synchronization issues. Second, the number of clock cycles in the pPIM cluster Verilog testbench are around 20 clock cycles, which is a more than the maximum of nine clock cycles for a single cluster operation.

Chapter 7

Conclusion

The SystemC models of the pPIM core and the pPIM cluster are bit-exact hierarchical models built in this thesis. They are exact to the bit to the Verilog models and have a hierarchical structure unlike most of the other SystemC models which are flat algorithmic models. The complex systems designed today with a rapid evolution in technology are all hierarchical models that should have the ability to be shared by protecting its Itellectual Property(IP). This work provides a basis for learning to build bit-exact parameterized secure hierarchical models for Verification, Architecture Exploration and Performance Modeling.

In this thesis, the LUT-based pPIM core and the pPIM cluster were studied from the ongoing research on these architectures. The Verilog test benches were modified to make them functional. SystemC was learned from scratch by building experimental modules and testbenches in SystemC. A testbench was written for each SystemC model by isolating them from the System design and verifying the result. Each sub-module was bit-exact modeled in SystemC along with separate testbench. The pPIM core was bit-exact hierarchically modeled in SystemC and a separate test bench was written for it and tested. The pPIM cluster was bit-exact hierarchically modeled in SystemC and a test bench was written for it to test it along with waveforms. Each sub-module

and the pPIM core has SystemC simulation waveform results which match exactly with the results of the verilog simulation. The pPIM cluster SystemC simulation results partially match the pPIM cluster Verilog results. From debugging, it is believed the pPIM cluster Verilog testbench needs improvement. The SystemC pPIM core model can now be used as reference models for verification and performance modeling as the design evolves during the research.

7.1 Future Work

Although the pPIM core SystemC model matches the behavior of the pPIM Verilog test bench, minor discrepancies were found during testing of the SystemC pPIM cluster verses the Verilog pPIM cluster Verilog test bench. The pPIM cluster Verilog testbench requires fixes in terms of number of clock cycles. Having the accumulator as part of the pPIM cluster instead of having it in the testbench would avoid complications in using the intermediate results from the accumulator. Further testing is required to determine improvements needed in either pPIM cluster SystemC model or the test bench to ensure the outputs produced for all the cycles to complete the MAC operation correctly. The SystemC model can be enhanced by using advanced SystemC concepts like SystemC Transaction-Level-Modeling(TLM).

References

[1] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A Review of Near-Memory Computing Architectures: Opportunities and Challenges," *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 608–617, 2018.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 336–348.

[3] L. Chang, Z. Wang, Y. Zhang, and W. Zhao, "Reconfigurable processing in memory architecture based on spin orbit torque," in *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2017, pp. 95–96.

[4] C. H. Kim, W. J. Lee, Y. Paik, K. Kwon, S. Y. Kim, I. Park, and S. W. Kim, "Silent-PIM: Realizing the Processing-in-Memory Computing With Standard Memory Requests," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 251–262, 2022.

[5] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, "A New Perspective on Processing-in-Memory Architecture Design," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and*

Correctness, ser. MSPC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2492408.2492418

[6] W. J. Lee, C. H. Kim, Y. Paik, J. Park, I. Park, and S. W. Kim, "Design of Processing-Inside-Memory Optimized for DRAM Behaviors," *IEEE Access*, vol. 7, pp. 82 633–82 648, 2019.

[7] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, jun 2016. [Online]. Available: https://doi.org/10.1145/3007787.3001140

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 105–117. [Online]. Available: https://doi.org/10.1145/2749469.2750386

[9] S. Gupta, M. Imani, H. Kaur, and T. S. Rosing, "NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1325–1337, 2019.

[10] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2897937.2898064

[11] J. Gerlach and W. Rosenstiel, *System Level Design Using the SystemC Modeling Platform*. Boston, MA: Springer US, 2001, pp. 27–34. [Online]. Available: https://doi.org/10.1007/978-1-4757-6666-0_2

[12] D. C. Black and J. Donovan, "SystemC: From the Ground Up," *Springer*, 2004.

[13] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The simulation semantics of SystemC," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001, pp. 64–70.

[14] 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.

[15] G. Bonanome, "Hardware Description Languages Compared : Verilog and SystemC," 2001.

[16] B. S. Amal Banerjee, "SystemC and SystemC-AMS in Practice," *Springer Cham*, 2014.

[17] R. Behjati, H. Sabouri, N. Razavi, and M. Sirjani, "An effective approach for model checking SystemC designs," in *2008 8th International Conference on Application of Concurrency to System Design*, 2008, pp. 56–61.

[18] J. Stoppe and R. Drechsler, "Analyzing SystemC Designs: SystemC Analysis Approaches for Varying Applications," *Sensors (Basel, Switzerland)*, vol. 15, pp. 10 399 – 10 421, 2015.

[19] N. Bombieri, F. Fummi, V. Guarnieri, F. Stefanni, and S. Vinco, "Efficient implementation and abstraction of SystemC data types for fast simulation," in *FDL 2011 Proceedings*, 2011, pp. 1–7.

[20] M. Y. Vardi, "Formal Techniques for SystemC Verification," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 188–192. [Online]. Available: https://doi.org/10.1145/1278480.1278527

[21] M.-K. You and G.-Y. Song, "SystemVerilog-based verification environment using SystemC custom hierarchical channel," in *2009 IEEE 8th International Conference on ASIC*, 2009, pp. 1310–1313.

[22] N. Razavi, R. Behjati, H. Sabouri, E. Khamespanah, A. Shali, and M. Sirjani, "Sysfier: Actor-based formal verification of SystemC," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 1–35, 2010. [Online]. Available: https://doi.org/10.1145/1880050.1880055

[23] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *J. VLSI Signal Process. Syst.*, vol. 41, no. 2, pp. 169–182, sep 2005. [Online]. Available: https://doi.org/10.1007/s11265-005-6648-1

[24] A. Moursi, R. Samhoud, Y. Kamal, M. Magdy, S. El-Ashry, and A. Shalaby, "Different Reference Models for UVM Environment to Speed Up the Verification Time," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2018, pp. 67–72.

[25] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 1, pp. 57–68, 2006.

[26] M. Glukhikh and M. Moiseev, "Fast Simulation of SystemC Synthesizable Subset," in *2015*

IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, 2015, pp. 103–106.

[27] K. Ma, R. van Leuken, M. Vidojkovic, J. Romme, S. Rampu, H. Pflug, L. Huang, and G. Dolmans, "A fast and accurate SystemC-AMS model for PLL," in *Proceedings of the 18th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2011*, 2011, pp. 411–416.

[28] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2019.

[29] S. Bavikadi, P. R. Sutradhar, K. N. Khasawneh, A. Ganguly, and S. M. Pudukotai Dinakarrao, "A Review of In-Memory Computing Architectures for Machine Learning Applications," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 89–94. [Online]. Available: https://doi.org/10.1145/3386263.3407649

[30] J. Bhasker, "A SystemC Primer," *Star Galaxy Publishing*, 2002.

[31] P. Panda, "SystemC - a modeling platform supporting multiple design abstractions," in *International Symposium on System Synthesis (IEEE Cat. No.01EX526)*, 2001, pp. 75–80.

[32] S. Sutherland, "Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface," *Sutherland HDL*.

[33] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[34] *Parameterized Modules and Channels*. Boston, MA: Springer US, 2002, pp. 87–108. [Online]. Available: https://doi.org/10.1007/0-306-47652-5_6

[35] W. Muller, W. Rosenstiel, and J. Ruf, Eds., *SystemC: Methodologies and Applications*. USA: Kluwer Academic Publishers, 2003.

[36] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[37] 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.

[38] S. Bavikadi, P. R. Sutradhar, M. A. Indovina, A. Ganguly, and S. M. P. Dinakarrao, "POLAR: Performance-aware On-device Learning Capable Programmable Processing-in-Memory Architecture for Low-Power ML Applications," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 889–898.

[39] K. Chang, P. Nair, D. Lee, S. Ghose, M. Qureshi, and O. Mutlu, "Low cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *Proceedings of the 2016 IEEE International Symposium on High-Performance Computer Architecture, HPCA 2016*, ser. Proceedings - International Symposium on High-Performance Computer Architecture. IEEE Computer Society, Apr. 2016, pp. 568–580, funding Information: We acknowledge the support of Google, Intel, Nvidia, Samsung, and VMware. This research was supported in part by the ISTC-CC, SRC, CFAR, and NSF (grants 1212962, 1319587, and 1320531). Kevin Chang is supported in part by the SRCEA/Intel Fellowship. Publisher

Appendix I

SystemC code for pPIM core

I.1 In_multiplexer model

```
#ifndef IN_MUX_H
#define IN_MUX_H
#define SC_INCLUDE_FX
#include <systemc.h>
#include <iomanip>
#include <iostream>
#include <string>
#include <fstream>
#include "nbit_multiplexer.h"

template <int M, int N> class In_multiplexer : sc_module
{
public:
```

```
14    //PORTS
15    sc_in<sc_bv<2*N> > A{"A"};
16    sc_in<sc_bv<2*N> > B{"B"};
17    sc_in<sc_bv<2*N*(M-1)> > CORES{"CORES"};
18    sc_in<bool> MODE{"MODE"};
19    constexpr static int size = ceil(log2(2*(M-1)));
20    sc_in<sc_bv<size> > SEL{"SEL"};
21    sc_out<sc_bv<N> > Y{"Y"};
22
23    //Signal Declaratoin
24
25    sc_signal<sc_bv<N> > sInt;
26    sc_signal<sc_bv<N> > sExt;
27    sc_signal<sc_bv<2> > sel_range;
28    sc_signal<sc_bv<4*N> > A_B;
29    sc_signal<sc_bv<2*N> > Ext_Int;
30
31
32    //Global declarations of instantiated modules
33       nbit_multiplexer<size,N> IntMux{"IntMux"};
34       nbit_multiplexer<2,N> ExtMux{"ExtMux"};
35       nbit_multiplexer<1,N> ModeMux{"ModeMux"};
36
37    SC_HAS_PROCESS(In_multiplexer);
38
```

```
39  In_multiplexer(sc_module_name name): sc_module(name)
40  {
41
42    SC_METHOD(Select_Range);
43    sensitive << SEL;
44
45    SC_METHOD(concat_AB);
46    sensitive << A << B;
47
48    SC_METHOD(concat_ExtInt);
49    sensitive << sInt << sExt;
50
51    IntMux.IN(CORES);
52    IntMux.SEL(SEL);
53    IntMux.OUT(sInt);
54
55    ExtMux.IN(A_B);
56    ExtMux.SEL(sel_range);
57    ExtMux.OUT(sExt);
58
59    ModeMux.IN(Ext_Int);
60    ModeMux.SEL(MODE);
61    ModeMux.OUT(Y);
62  }
63
```

```
64    void Select_Range()
65    {
66      sc_bv<size> > select_copy;
67
68      select_copy = SEL.read();
69      sel_range = select_copy.range(1,0);
70
71    }
72
73    void concat_AB()
74    {
75      sc_bv<2*N> copy_A;
76      sc_bv<2*N> copy_B;
77      sc_bv<4*N> copy_AB;
78
79
80      copy_A = A.read();
81      copy_B = B.read();
82
83      copy_AB = (copy_A,copy_B);
84      A_B.write(copy_AB);
85    }
86
87    void concat_ExtInt()
88    {
```

```
89      sc_bv<N> copy_Ext;
90      sc_bv<N> copy_Int;
91      sc_bv<2*N> copy_ExtInt;
92
93
94      copy_Ext = sExt.read();
95      copy_Int = sInt.read();
96
97      copy_ExtInt = (copy_Ext,copy_Int);
98      Ext_Int.write(copy_ExtInt);
99    }
100
101
102
103 };
104
105 #endif
```

I.2 Decoder model

```
1  #ifndef DECODER_H
2   #define DECODER_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10
11 template <int N> class decoder : public sc_module {
12      public:
13          constexpr static int size = ceil(log2(N));
14          sc_in<sc_uint<size> > IN{"IN"};
15          sc_out<sc_bv<N> > OUT;
16
17          //sc_in_clk clk;
18          sc_bv<N> sOUT;
19
20          unsigned i;
21
22          SC_HAS_PROCESS(decoder);
23          decoder(sc_module_name name) : sc_module(name)
```

```
24          {
25              SC_METHOD(prc_decoder);
26              sensitive << IN;
27          }
28
29          void prc_decoder()
30          {
31                  sc_uint<3> INcopy;
32                  INcopy = IN->read();
33              switch(INcopy)
34              {
35                  case 0 : sOUT=0x1;
36                           break;
37                  case 1 : sOUT=0x2;
38                           break;
39                  case 2 : sOUT=0x4;
40                           break;
41                  case 3 : sOUT=0x8;
42                           break;
43                  case 4 : sOUT=0x10;
44                           break;
45                  case 5 : sOUT=0x20;
46                           break;
47                  case 6 : sOUT=0x40;
48                           break;
```

```
49                      case 7 : sOUT=0x80;
50                               break;
51                      default : sOUT=0x0;
52                               break;
53                  }
54
55                  OUT = sOUT;
56                  //cout<< "at time: " << sc_time_stamp()<<
                        "----------> "<< "OUT in decoder= "<< OUT.read()
                        <<endl;
57              }
58  };
59
60  #endif
```

I.3 Multiplexer model

```
1  #ifndef MUX_H
2  #define MUX_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10
11 //N=2
12 template <int N, int select_size> class multiplexer : public
       sc_module
13 {
14 public:
15   sc_in<sc_bv<N> > IN;
16   sc_in<sc_bv<select_size> > SEL; //8 bit from PIM core
17   sc_out<bool> OUT;
18
19   //sc_vector<sc_signal<sc_biguint<N> > > IN_copy{"IN_copy", N
         };
20   sc_bv<select_size> IN_select;
21
```

```
22    bool out_copy;
23    sc_bv<N> IN_copy;
24    //
25    SC_HAS_PROCESS(multiplexer);
26
27    multiplexer(sc_module_name name) : sc_module(name)
28    {
29      SC_METHOD(prc_mux);
30       sensitive<<IN<<SEL;
31
32    }
33
34    void prc_mux()
35    {
36      IN_copy = IN;
37      IN_select = SEL.read();
38      //mask = 1;
39      //s = IN_select;
40      //cout<< "at time: " << sc_time_stamp()<< endl << "
          IN_select = "<< IN_select <<endl;
41      for(auto i=0U; i<N; ++i)
42      {
43        if(i==IN_select)
44        {
45          out_copy = bool (IN_copy[i]);
```

```
46          }
47      }
48      OUT.write(out_copy);
49
50      // cout<< "at time: " << sc_time_stamp()<< endl << "SEL =
            "<< SEL.read() <<endl;
51      // cout<< "at time: " << sc_time_stamp()<< endl << "IN =
            "<< IN <<endl;
52      // cout<< "at time: " << sc_time_stamp()<< endl << "IN_copy
             = "<< IN_copy <<endl;
53      // //cout<< "at time: " << sc_time_stamp()<< endl << "
            out_copy = "<< out_copy <<endl;
54      //cout<< "at time: " << sc_time_stamp()<< endl << "OUT mux
            = "<< OUT.read() <<endl;
55
56    }
57 };
58
59 #endif
```

I.4 nbit_multiplexer model

```
1  #ifndef NBIT_MUX_H
2  #define NBIT_MUX_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10 #include "multiplexer.h"
11 #include "sensitive_def.h"
12 #include<math.h>
13
14 // template <typename T>
15 // sc_sensitive& operator << (sc_sensitive& sensitive, const
       sc_vector<T>& vec )
16 // {
17 //   for (auto & el : vec )
18 //     sensitive << el;
19 //   return sensitive;
20 // }
21 //N=1, M=4
```

```
22 template<int N, int M> class nbit_multiplexer : public
       sc_module
23 {
24 public:
25
26    //sc_in<sc_biguint<pow(2,N)*M> > IN;
27    sc_in<sc_bv<(1<<N)*M> > IN; //8 bits
28    sc_in<sc_bv<N> > SEL; //1 bit
29    sc_out<sc_bv<M> > OUT; //4 bits
30
31    sc_bv<(1<<N)*M> sIN;
32    //sc_vector<sc_signal<sc_bv<1>>> sOUT;
33    //sc_bv<M> sOUT;
34
35    unsigned G;
36
37    constexpr static int select_size = ceil(log2(1<<N)); //
          constexpr is used to make it a compiletime statement
38
39   SC_HAS_PROCESS(nbit_multiplexer);
40
41    sc_vector<multiplexer<(1<<N),select_size> > bit_mux{"bit_mux"
          ,M};
42    sc_vector<sc_signal<sc_bv<(1<<N)>>> sIN_segment{"sIN_segment"
          ,M};
```

```
43    sc_vector<sc_signal<bool> > sOUT_bit_select{"sOUT_bit_select"
          ,M};
44
45
46    nbit_multiplexer(sc_module_name name): sc_module(name)
47    {
48
49      SC_METHOD(reorganization);
50       sensitive << IN;
51
52       // SC_METHOD(selection);
53       // sensitive << sIN_segment;
54       //cout<< "at time: " << sc_time_stamp()<< endl << "IN in
             nbit_multiplexer= "<< IN  <<endl;
55
56      SC_METHOD(sOUT_select);
57       sensitive << sOUT_bit_select;
58
59       for (auto G = 0U; G<M ; ++G)
60       {
61          bit_mux[G].IN(sIN_segment[G]);
62          bit_mux[G].SEL(SEL);
63          bit_mux[G].OUT(sOUT_bit_select[G]);
64       }
65
```

```
66
67    }
68    void reorganization()
69    {
70       sc_bv<(1<<N)*M> din;
71       //cout<< "at time: " << sc_time_stamp()<< endl << "din =
             "<< din <<endl;
72       //reorganization
73       for(G=0; G<((1<<N)*M); ++G)
74       {
75
76          sIN[((G%M)*(1<<N))+(G/M)] = IN.read()[G];
77          for(auto i=0U; i<M; ++i)
78          {
79             din=sIN.range(((1<<N)*(i+1))-1,(1<<N)*i);
80             sIN_segment[i]=din;
81             // cout<< "at time: " << sc_time_stamp()<< endl << "sIN
                   = "<< sIN <<endl;
82             // cout<< "at time: " << sc_time_stamp()<< endl << "din
                   = "<< din <<endl;
83          }
84
85       }
86
87
```

```
88    }
89
90    // void selection()
91    // {
92    //   sc_bv<(1<<N)*M> din;
93
94    //   for(auto i=0U; i<M; ++i)
95    //   {
96    //     din=sIN.range((1<<N)*(i+1)-1,(1<<N)*i);
97    //     sIN_segment[i]=din;
98    //   }
99
100
101   // }
102
103   void sOUT_select()
104   // {
105   //   for(auto i=0U; i<M; ++i)
106   //   {
107   //     sOUT[i] = sOUT_bit_select[i];
108   //     OUT.write(sOUT);
109   //   }
110
111   // }
112
```

```
113     {
114         sc_bv<M> write_value;
115         for(auto i=0U; i<M; ++i)
116         {
117             write_value[i] = sOUT_bit_select[i].read();
118 
119         }
120         //sOUT = write_value;
121         OUT.write(write_value);
122     }
123 
124 
125 
126 };
127 
128 #endif
```

I.5 register256 model

```
1  #ifndef REGISTER_256_H
2  #define REGISTER_256_H
3 #define SC_INCLUDE_FX
4 #include <systemc.h>
5 #include <iomanip>
6 #include <iostream>
7 #include <string>
8 #include <fstream>
9
10 template <int size> class register256 : public sc_module {
11 public:
12    sc_in<sc_bv<size> > DATA_IN;
13    sc_in<bool> WRITE;
14       //sc_in<sc_biguint<6> > WRITE_ADDR;
15
16    sc_in_clk clk;
17    sc_in<bool> reset;
18    sc_out<sc_bv<size> > DATA_OUT;
19    sc_bv<size> DATA;
20
21    SC_HAS_PROCESS(register256);
22    register256(sc_module_name name) : sc_module(name) //
         constructor that takes paprameter
```

```
23   {
24     SC_METHOD(prc_register256);
25     sensitive << clk << reset;//poedge
26   }
27
28   void prc_register256 ()
29   {
30
31     if(reset)
32       DATA = 0;
33     else if(WRITE.read())
34               DATA=DATA_IN.read();
35    DATA_OUT.write(DATA);
36
37     //cout<< "at time: " << sc_time_stamp()<< endl << "DATA_OUT
           = "<< DATA_OUT.read() <<endl;
38     // // cout<< "at time: " << sc_time_stamp()<< endl << "
          DATA_IN = "<< DATA_IN.read() <<endl;
39     //  cout<< "at time: " << sc_time_stamp()<< endl << "reset
          = "<< reset <<endl;
40     // cout<< "at time: " << sc_time_stamp()<< endl << "
          write_addr = "<< write_addr[i] <<endl;
41     //cout<< "at time: " << sc_time_stamp()<< endl << "WRITE_EN
           = "<< WRITE_EN.read() <<endl;
42     //cout<< "WRITE_EN = "<< WRITE_EN.read() <<endl;
```

```
43      //cout<< "at time: " << sc_time_stamp()<< "---------> "<< "
           WRITE_EN in reg256 = "<< WRITE_EN.read()  <<endl;
44      //cout<< "at time: " << sc_time_stamp()<< "---------> "<< "
           DATA_OUT in reg256 = "<< DATA_OUT.read()  <<endl;
45      //cout<< "at time: " << sc_time_stamp()<< endl << "DATA_IN
           = "<< DATA_IN.read() <<endl;
46    }
47
48 };
49
50 #endif
```

I.6 register_file model

```
1  #ifndef REG_FILE_IF
2  #define REG_FILE_IF
3  #define SC_INCLUDE_FX
4  #include <iomanip>
5  #include <iostream>
6  #include <string>
7  #include <fstream>
8  #include "register256.h"
9  #include "sensitive_def.h"
10
11 // template <typename T>
12 // sc_sensitive& operator << (sc_sensitive& sensitive, const
       sc_vector<T>& vec )
13 // {
14 //   for (auto & el : vec )
15 //     sensitive << el;
16 //   return sensitive;
17 // }
18 //256,4
19 template <unsigned N, unsigned M> class register_file : public
       sc_module
20 {
21 public:
```

```
22
23    //  int K;
24    // K = M*N;
25    //  int L;
26    // L = 2*M;
27    sc_in<sc_bv<N> > DATA_IN;
28    sc_in<sc_bv<2*M> > WRITE_ADDR;
29    sc_in<bool> WRITE_EN;
30    sc_in<bool> READ_EN;
31    sc_in_clk clk;
32    sc_in<bool> reset;
33    sc_out<sc_bv<(2*M)*(1<<(2*M))> > DATA_OUT;  //384
34
35    sc_bv<2*M> write_addr_var;
36    bool write_en_var;
37    sc_signal<sc_bv<N> > DATA_IN_copy;
38
39
40      sc_vector<register256<N>> regs;
41    sc_vector<sc_signal<sc_bv<N>>> data_out;
42    sc_vector<sc_signal<bool>> AND_sig;
43    unsigned i;
44
45   SC_HAS_PROCESS(register_file);
46
```

```
47    register_file(sc_module_name name): sc_module(name), regs("
          regs", 2*M), data_out("data_out", N), AND_sig("AND_sig", 2*
          M)
48    {
49      SC_METHOD(convert_to_bv)
50      sensitive << DATA_IN;
51      SC_METHOD(combine_output);
52      sensitive << data_out;
53      SC_METHOD(addr_computation);
54      sensitive << WRITE_ADDR << WRITE_EN;
55
56      for(auto i=0U; i< 2*M ; ++i)
57      {
58
59
60            // make the method react on changes of data_out[i]
61
62            // connect the register
63        //auto& reg1 = regs[i];
64        regs[i].clk(clk);
65        regs[i].reset(reset);
66        regs[i].DATA_IN(DATA_IN_copy);
67        regs[i].WRITE(AND_sig[i]);
68        regs[i].DATA_OUT(data_out[i]);
69
```

```
70
71          //reg.DATA_OUT(DATA_OUT);
72
73           //cout<< "at time: " << sc_time_stamp()<< endl << "
                data_out"<< i << " = " << data_out[i] <<endl;
74           // cout<< "at time: " << sc_time_stamp()<< endl << "
                AND_sig[i] = "<< AND_sig[i] <<endl;
75
76          //cout<< "at time: " << sc_time_stamp()<< endl << "
              WRITE_ADDR = "<< WRITE_ADDR.read() <<endl;
77           //cout<< "at time: " << sc_time_stamp()<< endl << "flag
                 = "<< flag.read() <<endl;
78
79
80
81
82      }
83
84
85      }
86
87      void convert_to_bv()
88      {
89        sc_bv<N> D;
90
```

```
91          D= DATA_IN;
92
93          DATA_IN_copy = D;
94      }
95
96    void combine_output() {
97          sc_bv<(2*M)*(1<<(2*M))> dout;
98          //dout = 0;
99
100     for(auto i=0U; i< 2*M; ++i)
101     {
102             dout.range(((i+1)*N)-1,i*N)=data_out[i].read();
103
104             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    data_out[i] = "<< data_out[i].read() <<endl;
105             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    write_addr_var = "<< write_addr_var[i] <<endl;
106             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    flag = "<< flag <<endl;
107            //cout<< "at time: " << sc_time_stamp()<<
                  "----------------------------------------> "<< "
                    dout = " << dout <<endl;
108
109           }
110           DATA_OUT.write(dout);
```

```
111         //cout<< "at time: " << sc_time_stamp()<<
              "------------------------------------------> "<< "
              DATA_OUT = "<< DATA_OUT.read() <<endl;
112          //cout<< "at time: " << sc_time_stamp()<< endl << "
               WRITE_EN = "<< WRITE_EN <<endl;
113     }
114
115     void addr_computation()
116     {
117       write_en_var = WRITE_EN;
118
119       for(auto i=0U; i<2*M; ++i)
120       {
121         AND_sig[i] = write_en_var && WRITE_ADDR.read()[i];
122       }
123
124       //cout<< "at time: " << sc_time_stamp()<< "---------> "<<
            "write_en_var = "<< write_en_var <<endl;
125       //cout<< "at time: " << sc_time_stamp()<< "---------> "<<
            "WRITE_ADDR = "<< WRITE_ADDR.read()[i] <<endl;
126     }
127
128
129     //Destructor
130
```

```
131       // ~register_file()
132       // {
133       //   for(i=0; i<M;++i)
134       //   {
135       //     delete regs[i];
136       //   }
137
138       // }
139
140
141
142
143
144 };
145
146 #endif
```

I.7 pPIM core model

```
1  #ifndef PIM_H
2  #define PIM_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include "multiplexer.h"
10 #include "decoder.h"
11 #include "nbit_multiplexer.h"
12 #include "register256.h"
13 #include "register_file.h"
14 #include "sensitive_def.h"
15
16 template <int N> class PIM : public sc_module
17 {
18 public:
19     //PORTS
20     sc_in<sc_bv<N> > A{"A"}; //4
21     sc_in<sc_bv<1> > ASel{"ASel"};
22     sc_in<sc_bv<N> > B{"B"};//4
23     sc_in<sc_bv<1> > BSel{"BSel"};
```

```
24      sc_in<sc_bv<1<<(2*N)> >FUNC_IN{"FUNC_IN"}; //256
25      constexpr static int size = ceil(log2(2*N)); //3
26      sc_in<sc_uint<size> > FUNC_ADDR{"FUNC_ADDR"}; //3
27      sc_in<sc_bv<2> > IN_MODE{"IN_MODE"};
28      sc_in<bool> LOAD{"LOAD"};
29      sc_in<bool> RUN{"RUN"};
30      sc_in<bool> reset{"reset"};
31      sc_in_clk clk{"clk"};
32      sc_out<sc_bv<2*N> > Y{"Y"}; //8
33
34      //SIGNALS
35      sc_signal<sc_bv<(2*N)*(1<<(2*N))> > sFUNC{"sFUNC"};
36      sc_signal<sc_bv<N> > sA{"sA"};
37      sc_signal<sc_bv<N> > sB{"sB"};
38      sc_signal<sc_bv<N> > sAIn{"sAIn"};
39      sc_signal<sc_bv<N> > sBIn{"sBIn"};
40      sc_bv<2*N> sY;
41      sc_signal<sc_bv<2*N> > sFUNC_ADDR{"sFUNC_ADDR"};
42      sc_signal<sc_bv<2*N> > sY_A{"sY_A"};
43      sc_signal<sc_bv<2*N> > sY_B{"sY_B"};
44
45      sc_signal<sc_bv<2*N> > sA_sB{"sA_sB"};
46      sc_signal<bool> mode0{"mode0"};
47      sc_signal<bool> mode1{"mode1"};
48
```

```
49      //Global declarations of instantiated modules
50      decoder<2*N> decoder_addr{"decoder_addr"};
51      register_file<1<<(2*N),N> func_regs{"func_regs"};
52      nbit_multiplexer<1,N> AMux{"AMux"};
53      nbit_multiplexer<1,N> BMux{"BMux"};
54      register256<N> Areg{"Areg"};
55      register256<N> Breg{"Breg"};
56
57
58      int G;
59
60      SC_HAS_PROCESS(PIM);
61
62      //VECTORS
63      constexpr static int sel_size = ceil(log2(1<<(2*N))); //8
64      sc_vector<multiplexer<(1<<(N*2)),sel_size> > mux256to1{"
            mux256to1", 2*N};
65      sc_vector<sc_signal<bool> > sY_bit{"sY_bit",2*N};
66      sc_vector<sc_signal<sc_bv<1<<(N*2)> >> sFUNC_range{"
            sFUNC_range", 2*N};
67      //sc_signal<sc_bv<(2*N)*(1<<(2*N))> > sFUNC{"sFUNC"};
68
69      PIM(sc_module_name name): sc_module(name)
70      {
71          SC_METHOD(concat_method_sY);
```

```
72          sensitive << sY_bit << A << B;
73
74          SC_METHOD(sFUNC_range_compute);
75          sensitive << sFUNC;
76
77          SC_METHOD(concat_sA_sB);
78          sensitive << sA << sB;
79
80          SC_METHOD(in_mode_bit_select);
81          sensitive << IN_MODE;
82
83          SC_METHOD(sY_bit_select);
84          sensitive << sY_bit;
85
86          SC_METHOD(print_decoder);
87          sensitive << sAIn<<sBIn;
88
89          //Decoder port connections
90          decoder_addr.IN(FUNC_ADDR);
91          decoder_addr.OUT(sFUNC_ADDR); //works
92
93          //register_file port connections
94          func_regs.DATA_IN(FUNC_IN);
95          func_regs.WRITE_ADDR(sFUNC_ADDR);
96          func_regs.WRITE_EN(LOAD);
```

```
97          func_regs.READ_EN(RUN);
98          func_regs.clk(clk);
99          func_regs.reset(reset);
100         func_regs.DATA_OUT(sFUNC); //works
101
102         //nbit_multiplexer A port connections
103         AMux.IN(sY_A); //sc_in<sc_biguint<(1<<N)*M> > IN;
                concatenating 2 sc_uints
104         AMux.SEL(ASel);
105         AMux.OUT(sAIn);
106
107         //nbit_multiplexer B port connections
108         BMux.IN(sY_B);
109         BMux.SEL(BSel);
110         BMux.OUT(sBIn);
111
112         //mux256to1 port connections
113         for (auto G = 0U; G< 2*N; ++G)
114         {
115             mux256to1[G].IN(sFUNC_range[G]);
116             mux256to1[G].SEL(sA_sB); //SEL is 8 bit sA_sB = 4+4
                    bits ??
117             mux256to1[G].OUT(sY_bit[G]);
118         }
119
```

```
120
121             //register256 Areg port connections
122             Areg.DATA_IN(sAIn);
123             Areg.WRITE(mode0);
124             Areg.clk(clk);
125             Areg.reset(reset);
126             Areg.DATA_OUT(sA);
127
128             //register256 Breg port connections
129             Breg.DATA_IN(sBIn);
130             Breg.WRITE(mode1);
131             Breg.clk(clk);
132             Breg.reset(reset);
133             Breg.DATA_OUT(sB);
134
135
136
137         }
138
139
140
141         void print_decoder()
142         {
143             sc_uint<size> FA;
144             FA = FUNC_ADDR;
```

```
145         //cout<< "at time: " << sc_time_stamp()<< endl << "
               sFUNC_ADDR = "<< sFUNC_ADDR.read() <<endl;
146         cout<< "at time: " << sc_time_stamp()<<"
               ---------------->"<< "sA = "<< sA.read() <<endl;
147        cout<< "at time: " << sc_time_stamp()<<"---------------->
             "<< "sB = "<< sB.read() <<endl;
148
149      }
150
151     void concat_method_sY()
152     {
153         sc_bv<2*N> s;
154         sc_bv<4> first_nib;
155         sc_bv<4> sec_nib;
156         sc_bv<2*N> sYA;
157         sc_bv<2*N> sYB;
158         sc_bv<N> reg_A;
159         sc_bv<N> reg_B;
160         reg_A = A;
161         reg_B = B;
162
163         s = sY;
164         first_nib = s.range(3,0);
165         sec_nib = s.range(7,4);
166
```

```
167             sYA = (first_nib,reg_A);
168
169             sYB = (sec_nib,reg_B);
170             sY_A.write(sYA);
171             sY_B.write(sYB);
172             //cout<< "at time: " << sc_time_stamp()<< endl << "sAIn
                    = "<< sAIn <<endl;
173             //cout<< "at time: " << sc_time_stamp()<< endl << "sBIn
                    = "<< sBIn <<endl;
174
175
176         }
177
178
179         void concat_sA_sB()
180         {
181             sc_bv<2*N> concat_A_B;
182             sc_bv<N> copy_A;
183             sc_bv<N> copy_B;
184
185             copy_A = A;
186             copy_B = B;
187
188             concat_A_B = (copy_A,copy_B);
189             sA_sB.write(concat_A_B);
```

```
190
191         //  cout<< "at time: " << sc_time_stamp()<< endl << "sA
              = "<< sA <<endl;
192         //  cout<< "at time: " << sc_time_stamp()<< endl << "sB
              = "<< sB <<endl;
193         // cout<< "at time: " << sc_time_stamp()<< endl << "
              sA_sB = "<< sA_sB <<endl<<endl;
194
195     }
196
197     void sFUNC_range_compute()
198     {
199         sc_bv<(2*N)*(1<<(2*N))> func;//2048
200         func = sFUNC;
201         sc_bv<1<<(N*2)> sFUNC_range_copy;
202
203         for (auto G = 0U; G< 2*N; ++G)
204         {
205             sFUNC_range_copy = func.range(((1<<(2*N))*(G+1))
                  -1,((1<<(2*N))*G));
206             //sFUNC_range_copy = 0x84928fff892389;
207             sFUNC_range[G] = sFUNC_range_copy;
208         }
209
```

```
210         //cout<< "at time: " << sc_time_stamp()<< endl << "
               sFUNC_range_copy = "<< sFUNC_range_copy <<endl;
211         //cout<< "at time: " << sc_time_stamp()<< endl << "
               sFUNC_range = "<< sFUNC_range <<endl;
212
213
214
215     }
216
217     void in_mode_bit_select()
218     {
219         sc_bv<2> mode;
220         mode = IN_MODE;
221          mode0 = bool (mode[0]);
222          mode1 = bool (mode[1]);
223          //cout<< "at time: " << sc_time_stamp()<< endl << "
                mode0 = "<< mode0 <<endl;
224
225     }
226
227     void sY_bit_select()
228     {
229
230         sc_bv<2*N> write_val;
231         for(auto i=0U; i<2*N; ++i)
```

```
232             {
233                 write_val[i] = sY_bit[i].read();
234
235             }
236             sY = write_val;
237             //cout<< "at time: " << sc_time_stamp()<< endl << "sY =
                    "<< sY <<endl;
238             Y.write(sY);
239             cout<< "at time: " << sc_time_stamp()<< endl << "Y = "
                   << Y <<endl;
240
241         }
242
243
244 };
245
246 #endif
```

I.8 pPIM core testbench

```
1  #ifndef PIM_STI_MULUS_H
2  #define PIM_STI_MULUS_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9
10 template <int N> class PIM_stimulus : public sc_module {
11   public:
12
13     sc_out<sc_bv<N> > A; //4
14     sc_out<sc_bv<1> > ASel;
15     sc_out<sc_bv<N> > B; //4
16     sc_out<sc_bv<1> > BSel;
17     sc_out<sc_bv<(1<<(2*N))> > FUNC_IN; //256
18     constexpr static int size = ceil(log2(2*N));
19     sc_out<sc_uint<size> > FUNC_ADDR; //3
20     sc_out<sc_bv<2> > IN_MODE;
21     sc_out<bool> LOAD;
22     sc_out<bool> RUN;
23     sc_out<bool> reset;
```

```
24      sc_out<bool> clk;
25      //sc_in_clk clk;
26
27    sc_bv<2048> FUNC_ADD =0;
28    sc_bv<2048> FUNC_MULT=0;
29    sc_bv<2048> func_add_copy=0;
30    sc_bv<256> func_add_slice=0;
31    sc_bv<2048> func_mult_copy=0;
32    sc_bv<256> func_mult_slice=0;
33    sc_bv<8> ADD_tmp=0;//8
34    sc_bv<8> MUL_tmp=0;//8
35    sc_bv<8> add=0;//8
36    sc_bv<8> mult=0;
37
38    int index=0;
39    int a =0;
40    int b =0;
41
42     // sc_out<sc_bv<8>> add_port;
43     // sc_out <sc_bv<8>> mult_port;
44
45
46      SC_HAS_PROCESS(PIM_stimulus);
47
48      PIM_stimulus(sc_module_name name) : sc_module(name)
```

```
49      {
50        SC_THREAD(clock_gen1);
51        SC_THREAD(test_bench);
52        sensitive<<clk.neg();
53
54      }
55
56      void clock_gen1()
57        {
58
59          bool stat = true;
60          while(true)
61          {
62
63            clk->write(stat);
64            stat = !stat;
65            wait(3,SC_NS);
66
67          }
68
69        }
70
71      void test_bench()
72      {
73
```

```
74      for( a=0; a<16; ++a)
75      {
76
77        for( b=0; b<16; ++b)
78        {
79
80          add = a + b;
81          mult = a*b;
82
83          ADD_tmp = add;
84          MUL_tmp = mult;
85
86          // add_port = ADD_tmp;
87          // mult_port = MUL_tmp;
88          // cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "ADD_tmp in stimulus= "<< ADD_tmp <<endl;
89          // cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "MUL_tmp in stimulus= "<< MUL_tmp <<endl;
90          for( auto kk=0U;kk<8;++kk)
91          {
92            FUNC_MULT[(kk*256)+index] = MUL_tmp[kk];
93            FUNC_ADD[(kk*256)+index] = ADD_tmp[kk];
94
95          }
96          ++index;
```

```
97
98          }
99      }
100
101             func_add_copy = FUNC_ADD;
102             func_mult_copy = FUNC_MULT;
103
104
105         wait(clk.negedge_event());
106     reset.write(1);
107     wait(clk.negedge_event());
108     reset.write(0);
109
110     RUN.write(1);
111     LOAD.write(1);
112     wait(clk.negedge_event());
113     FUNC_IN.write(FUNC_ADD.range((0+1)*256-1,(0)*256));
114     FUNC_ADDR.write(0x000);
115     wait(clk.negedge_event());
116     FUNC_IN.write(FUNC_ADD.range((1+1)*256-1,(1)*256));
117     FUNC_ADDR.write(0x001);
118     wait(clk.negedge_event());
119     FUNC_IN.write(FUNC_ADD.range((2+1)*256-1,(2)*256));
120     FUNC_ADDR.write(0x002);
121     wait(clk.negedge_event());
```

```
122         FUNC_IN.write(FUNC_ADD.range((3+1)*256-1,(3)*256));
123         FUNC_ADDR.write(0x003);
124
125         wait(clk.negedge_event());
126         FUNC_IN.write(FUNC_ADD.range((4+1)*256-1,(4)*256));
127         FUNC_ADDR.write(0x004);
128
129         wait(clk.negedge_event());
130         FUNC_IN.write(FUNC_ADD.range((5+1)*256-1,(5)*256));
131         FUNC_ADDR.write(0x005);
132
133         wait(clk.negedge_event());
134         FUNC_IN.write(FUNC_ADD.range((6+1)*256-1,(6)*256));
135         FUNC_ADDR.write(0x006);
136
137         wait(clk.negedge_event());
138         FUNC_IN.write(FUNC_ADD.range((7+1)*256-1,(7)*256));
139         FUNC_ADDR.write(0x007);
140
141         wait(clk.negedge_event());
142         FUNC_IN.write(0);
143         LOAD.write(0);
144         wait(5,SC_NS);
145         RUN.write(1);
146         IN_MODE.write(0x3);
```

```
147
148         wait(clk.negedge_event());
149         A.write(4);
150         B.write(1);
151         wait(clk.negedge_event());
152         A.write(9);
153         B.write(3);
154         wait(clk.negedge_event());
155         A.write(13);
156         B.write(13);
157
158         wait(clk.negedge_event());
159         A.write(5);
160         B.write(2);
161
162         wait(clk.negedge_event());
163         A.write(1);
164         B.write(13);
165
166         wait(clk.negedge_event());
167         A.write(6);
168         B.write(13);
169
170         wait(clk.negedge_event());
171         A.write(13);
```

```
172         B.write(12);
173
174         wait(clk.negedge_event());
175         A.write(9);
176         B.write(6);
177
178         wait(clk.negedge_event());
179         A.write(5);
180         B.write(10);
181
182         wait(clk.negedge_event());
183         A.write(5);
184         B.write(7);
185
186         wait(clk.negedge_event());
187         A.write(2);
188         B.write(15);
189
190         wait(clk.negedge_event());
191         A.write(2);
192         B.write(14);
193
194         wait(clk.negedge_event());
195         A.write(8);
196         B.write(5);
```

```
197
198         wait(clk.negedge_event());
199         A.write(12);
200         B.write(13);
201
202         wait(clk.negedge_event());
203         A.write(13);
204         B.write(5);
205
206         wait(clk.negedge_event());
207         A.write(3);
208         B.write(10);
209
210         wait(clk.negedge_event());
211         A.write(0);
212         B.write(0);
213
214         wait(clk.negedge_event());
215         A.write(10);
216         B.write(13);
217
218         wait(clk.negedge_event());
219         A.write(6);
220         B.write(3);
221
```

```
222
223
224
225
226
227
228
229
230
231
232         }
233
234
235 };
236
237
238 #endif
```

I.9 pPIM main

```
1  #define SC_INCLUDE_FX
2  #include "systemc.h"
3  #include "PIM_stimulus.h"
4  #include "PIM.h"
5
6  int sc_main(int argc, char* argv[])
7  {
8    constexpr static int N = 4;
9    sc_signal<sc_bv<N> > A{"A_main"};
10     sc_signal<sc_bv<1> > ASel{"ASel_main"};
11     sc_signal<sc_bv<N> > B{"B_main"};
12     sc_signal<sc_bv<1> > BSel{"BSel_main"};
13     sc_signal<sc_bv<1<<(2*N)> > FUNC_IN{"FUNC_IN_main"};
14     constexpr static int size = ceil(log2(2*N));
15     sc_signal<sc_uint<size> > FUNC_ADDR{"FUNC_ADDR_main"};
16     sc_signal<sc_bv<2> > IN_MODE{"IN_MODE_main"};
17     sc_signal<bool> LOAD{"LOAD_main"};
18     sc_signal<bool> RUN{"RUN_main"};
19     sc_signal<bool> reset{"reset_main"};
20     sc_signal<bool> clk{"clk_main"};
21     //sc_in_clk clk{"clk"};
22     //sc_clock clk("clock", 1, SC_NS);
23     sc_signal<sc_bv<2*N> > Y{"Y_main"};
```

```
24 //sc_signal<
25
26      PIM<4> core_pim("core_pim");
27      core_pim.A(A);
28      core_pim.ASel(ASel);
29      core_pim.B(B);
30      core_pim.BSel(BSel);
31      core_pim.FUNC_IN(FUNC_IN);
32      core_pim.FUNC_ADDR(FUNC_ADDR);
33      core_pim.IN_MODE(IN_MODE);
34      core_pim.LOAD(LOAD);
35      core_pim.RUN(RUN);
36      core_pim.reset(reset);
37      core_pim.clk(clk);
38      core_pim.Y(Y);
39
40      PIM_stimulus<4> stim_pim("stim_pim");
41      stim_pim.A(A);
42      stim_pim.B(B);
43      stim_pim.ASel(ASel);
44      stim_pim.BSel(BSel);
45      stim_pim.FUNC_IN(FUNC_IN);
46      stim_pim.FUNC_ADDR(FUNC_ADDR);
47      stim_pim.IN_MODE(IN_MODE);
48      stim_pim.LOAD(LOAD);
```

```
49      stim_pim.RUN(RUN);
50      stim_pim.reset(reset);
51      // stim_pim.add_port;
52      // stim_pim.mult_port
53      stim_pim.clk(clk);
54
55      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "A in main= "<< A.read() <<endl;
56      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "FUNC_ADDR in main= "<< FUNC_ADDR.read() <<endl;
57      cout<< "at time: " << sc_time_stamp()<< "----------> "<< "Y
            in main= "<< Y.read() <<endl;
58
59    sc_trace_file *fp;
60    fp = sc_create_vcd_trace_file("PIM_wave");
61    if(!fp) cout <<"There was an error!!" << endl;
62      //sc_trace(fp,clk,"clk");
63      sc_trace(fp,B,"B");
64          sc_trace(fp,A,"A");
65      sc_trace(fp,ASel,"ASel");
66      sc_trace(fp,BSel,"BSel");
67      sc_trace(fp,FUNC_IN,"FUNC_IN");
68      sc_trace(fp,FUNC_ADDR,"FUNC_ADDR");
69      sc_trace(fp,IN_MODE,"IN_MODE");
70      sc_trace(fp,LOAD,"LOAD");
```

```
71        sc_trace(fp,RUN,"RUN");
72        sc_trace(fp,reset,"reset");
73             sc_trace(fp,Y,"Y");
74        sc_start(10000000,SC_NS);
75             //sc_start();
76     sc_close_vcd_trace_file(fp);
77
78     return 0;
79
80
81
82  }
```

Appendix II

SystemC code for pPIM cluster

II.1 pPIM cluster model

```
1  #ifndef PIM_CLUSTER_H
2  #define PIM_CLUSTER_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include "nbit_multiplexer.h"
10 #include "In_multiplexer.h"
11 #include "PIM.h"
12 //N = 9 M = 8
13 template <int M, int N> class PIM_Cluster : public sc_module
```

```
14 {
15 public:
16   //PORTS
17   sc_in<sc_bv<M> >A{"A_Cluster"};
18   sc_in<sc_bv<M> >B{"B_Cluster"};
19   sc_in<sc_bv<1<<M> > FUNC{"FUNC_Cluster"};
20   constexpr static int size = ceil(log2(M));
21   sc_in<sc_uint<size> > FUNC_ADDR{"FUNC_ADDR_Cluster"};
22   sc_in<sc_bv<N> > LOAD_CORE{"LOAD_CORE_Cluster"};
23
24   constexpr static int size_addr = ceil(log2(2*(N-1))); //4
25   sc_in<sc_bv<N*size_addr> > AIN_ADDR{"AIN_ADDR_Cluster"};
26   sc_in<sc_bv<N*size_addr> > BIN_ADDR{"BIN_ADDR_Cluster"};
27
28   sc_in<sc_bv<2*N> > SRC_MODE{"SRC_MODE_Cluster"};
29   sc_in<sc_bv<2*N> > IN_MODE{"IN_MODE_Cluster"};
30
31   sc_in<sc_bv<N> > RUN{"RUN_Cluster"};
32   sc_in<bool> clk{"clk_Cluster"};
33   sc_in<bool> reset{"reset_Cluster"};
34   constexpr static int size_y_addr = ceil(log2(N));
35   sc_in<sc_bv<size_y_addr> > Y_ADDR{"Y_ADDR_Cluster"};
36   sc_out<sc_bv<M> > Y{"Y_Cluster"};
37
38   //SIGNALS
```

```
39    sc_bv<(N*(M/2))> sA; //36
40    sc_bv<(N*(M/2))> sA_collect;
41    sc_bv<(N*(M/2))> sB;
42    sc_bv<(N*(M/2))> sB_collect;
43    sc_bv<N*M> sY;
44    sc_bv<N*M> sY_collect;
45    sc_signal<sc_bv<1> > pimcore0_ASel_and;
46    sc_signal<sc_bv<1> > pimcore0_BSel_and;
47    sc_signal<sc_bv<1> > pimcoreN_ASel_and;
48    sc_signal<sc_bv<1> > pimcoreN_BSel_and;
49    sc_signal<bool> pimcore0_LOAD;
50    sc_signal<bool> pimcore0_RUN;
51    sc_signal<bool> pimcoreN_LOAD;
52    sc_signal<bool> pimcoreN_RUN;
53    sc_signal<sc_bv<1> > AMux0_SRC;
54    sc_signal<sc_bv<1> > BMux0_SRC;
55    sc_signal<sc_bv<1> > AMuxN_SRC;
56    sc_signal<sc_bv<1> > BMuxN_SRC;
57    sc_signal<sc_bv<M*M> > AMux0_sY;
58    sc_signal<sc_bv<M*M> > AMuxN_sY;
59    sc_signal<sc_bv<M/2> > pimcoreN_A;
60    sc_signal<sc_bv<M/2> > pimcoreN_B;
61    sc_signal<sc_bv<(N*M)+56> > sY_concat;
62    sc_signal<sc_bv<(M/2)> > pimcore0_A_part_select;
63    sc_signal<sc_bv<(M/2)> > pimcore0_B_part_select;
```

```
64    sc_signal<sc_bv<(M/2)> > AMuxN_sA_part_select;
65    sc_signal<sc_bv<(M/2)> > BMuxN_sB_part_select;
66    sc_signal<sc_bv<2> > pimcore0_IN_MODE_part_select;
67    sc_signal<sc_bv<2> > pimcoreN_IN_MODE_part_select;
68    sc_signal<sc_bv<M> > pimcore0_sY_part_select;
69    sc_signal<sc_bv<M> > pimcoreN_sY_part_select;
70
71    sc_signal<sc_bv<M/2> > AMux0_AIN_ADDR_part_select;
72    sc_signal<sc_bv<M/2> > BMux0_BIN_ADDR_part_select;
73    sc_signal<sc_bv<M/2> > AMuxN_AIN_ADDR_part_select;
74    sc_signal<sc_bv<M/2> > BMuxN_BIN_ADDR_part_select;
75    sc_signal<sc_bv<M/2> > AMux0_sA_part_select;
76    sc_signal<sc_bv<M/2> > BMux0_sB_part_select;
77
78
79    //Global declarations of instantiated modules
80    PIM<M/2> pimcore0{"pimcore0"};
81
82    PIM<M/2> pimcoreN{"pimcoreN"};
83    In_multiplexer<N,M/2> AMux0{"AMux0"};
84    In_multiplexer<N,M/2> BMux0{"BMux0"};
85    In_multiplexer<N,M/2> AMuxN{"AMuxN"};
86    In_multiplexer<N,M/2> BMuxN{"BMuxN"};
87    nbit_multiplexer<4,M> YMux{"YMux"};
88
```

```
89    //VECTORS
90    sc_vector<PIM<M/2> > pimcore{"pimcore", N-2};
91    sc_vector<In_multiplexer<N,M/2>> AMux{"AMux", N-2};
92    sc_vector<In_multiplexer<N,M/2>> BMux{"BMux", N-2};
93    sc_vector<sc_signal<sc_bv<M/2> >> pimcore_A_vec{"
          pimcore_A_vec",N-1};
94    sc_vector<sc_signal<sc_bv<M/2> >> pimcore_B_vec{"
          pimcore_B_vec",N-1};
95    sc_vector<sc_signal<sc_bv<1> >> pimcore_ASel_and_vec{"
          pimcore_ASel_and_vec",N-1};
96    sc_vector<sc_signal<sc_bv<1> >> pimcore_BSel_and_vec{"
          pimcore_BSel_and_vec",N-1};
97    sc_vector<sc_signal<sc_bv<2> >> pimcore_in_mode_vec{"
          pimcore_in_mode_vec",N-1};
98    sc_vector<sc_signal<bool> > pimcore_LOAD_vec{"
          pimcore_LOAD_vec",N-1};
99    sc_vector<sc_signal<bool> > pimcore_RUN_vec{"pimcore_RUN_vec"
          ,N-1};
100   sc_vector<sc_signal<sc_bv<M> >> pimcore_Y_vec{"pimcore_Y_vec"
          ,N-1};
101   sc_vector<sc_signal<sc_bv<1> >> AMux_mode_vec{"AMux_mode_vec"
          ,N-1};
102   sc_vector<sc_signal<sc_bv<M/2> >> AMux_sel_vec{"AMux_sel_vec"
          , N-1};
```

```
103     sc_vector<sc_signal<sc_bv<M*M> >> AMux_core_vec{"
            AMux_core_vec",N-1};
104     sc_vector<sc_signal<sc_bv<1> >> BMux_mode_vec{"BMux_mode_vec"
            ,N-1};
105     sc_vector<sc_signal<sc_bv<M/2> >> BMux_sel_vec{"BMux_sel_vec"
            , N-1};
106     sc_vector<sc_signal<sc_bv<M*M> >> BMux_core_vec{"
            BMux_core_vec",N-1};
107     sc_vector<sc_signal<sc_bv<M/2> >> part_sA{"part_sA",N-1};
108     sc_vector<sc_signal<sc_bv<M/2> >> part_sB{"part_sB",N-1};
109
110
111
112     SC_HAS_PROCESS(PIM_Cluster);
113
114     PIM_Cluster(sc_module_name name): sc_module(name)
115     {
116         //INVOKING PIMCORE0 METHODS
117         SC_METHOD(pimcore0_A);
118         sensitive << A<<B;
119
120         SC_METHOD(pimcore0_B);
121         sensitive << A<<B;
122
123         SC_METHOD(pimcore0_in_mode);
```

```
124         sensitive << IN_MODE;
125
126         SC_METHOD(pimcore0_sY);
127         sensitive <<A<<B;
128
129         SC_METHOD(pimcore0_add_A);
130         sensitive << AIN_ADDR;
131
132         SC_METHOD(pimcore0_add_B);
133         sensitive << BIN_ADDR;
134
135         SC_METHOD(pimcore0_LOAD_select);
136         sensitive << LOAD_CORE;
137
138         SC_METHOD(pimcore0_RUN_select);
139         sensitive << RUN;
140
141         //INVOKING AMux0 METHODS
142
143         SC_METHOD(AMux0_SRC_MODE_select);
144         sensitive << SRC_MODE<<A<<B;
145
146         SC_METHOD(AMux0_CORES_range);
147         sensitive <<A<<B;
148
```

```
149         SC_METHOD(AMux0_SEL);
150         sensitive << AIN_ADDR<<A<<B;
151
152         SC_METHOD(AMux0_Y);
153         sensitive << AMux0_sA_part_select<<A<<B;
154
155         //INVOKING BMux0 METHODS
156
157         SC_METHOD(BMux0_SEL);
158         sensitive << BIN_ADDR<<A<<B;
159
160         SC_METHOD(BMux0_Y);
161         sensitive << BMux0_sB_part_select<<A<<B;
162
163         //INVOKING pimcoreN METHODS
164
165         SC_METHOD(pimcoreN_A_range);
166         sensitive << A<<B;
167
168         SC_METHOD(pimcoreN_B_range);
169         sensitive << B<<A;
170
171         SC_METHOD(pimcoreN_IN_MODE);
172         sensitive << IN_MODE;
173
```

```
174         SC_METHOD(pimcoreN_sY);
175          sensitive << A<<B;
176
177         SC_METHOD(pimcoreN_add_A);
178          sensitive << AIN_ADDR;
179
180         SC_METHOD(pimcoreN_add_B);
181          sensitive << BIN_ADDR;
182
183         SC_METHOD(pimcoreN_LOAD_select);
184          sensitive << LOAD_CORE;
185
186         SC_METHOD(pimcoreN_RUN_select);
187          sensitive << RUN;
188
189          //INVOKING AMuxN METHODS
190
191         SC_METHOD(AMuxN_SRC_MODE_select);
192          sensitive << SRC_MODE<<A<<B;
193
194         SC_METHOD(AMuxN_CORES_range);
195          sensitive <<A<<B;
196
197         SC_METHOD(AMuxN_AIN_ADDR);
198          sensitive << AIN_ADDR<<A<<B;
```

```
199
200         SC_METHOD(AMuxN_sA);
201         sensitive <<A<<B<<AMuxN_sA_part_select;
202
203         //INVOKING BMuxN METHODS
204
205         SC_METHOD(BMuxN_BIN_ADDR);
206         sensitive << BIN_ADDR<<A<<B;
207
208         SC_METHOD(BMuxN_sB);
209         sensitive <<A<<B<<BMuxN_sB_part_select;
210
211         //INVOKING YMux METHODS
212
213         SC_METHOD(YMux_concatenate);
214         sensitive << A<<B<<Y_ADDR;
215
216         //GENERATE STATEMENTS
217
218         //INVOKING PIMCORE METHODS
219
220         SC_METHOD(pimcore_A_range);
221         sensitive << A<<B;
222
223         SC_METHOD(pimcore_B_range);
```

```
224         sensitive << A<<B;
225
226         SC_METHOD(pimcore_in_mode);
227         sensitive << IN_MODE;
228
229         SC_METHOD(pimcore_Y);
230         sensitive << A<<B;
231
232         SC_METHOD(pimcore_AIN_ADDR);
233         sensitive << AIN_ADDR;
234
235         SC_METHOD(pimcore_BIN_ADDR);
236         sensitive << BIN_ADDR;
237
238         SC_METHOD(pimcore_LOAD);
239         sensitive << LOAD_CORE;
240
241         SC_METHOD(pimcore_RUN);
242         sensitive << RUN;
243
244         //INVOKING AMux METHODS
245
246         SC_METHOD(AMux_mode);
247         sensitive << SRC_MODE;
248
```

```
249         SC_METHOD(AMux_core);
250          sensitive << A<<B;
251
252         SC_METHOD(AMux_sel);
253          sensitive << AIN_ADDR;
254
255         SC_METHOD(AMux_Y);
256          sensitive <<A<<B;
257
258          //INVOKING BMux METHODS
259
260         SC_METHOD(BMux_mode);
261          sensitive << SRC_MODE;
262
263         SC_METHOD(BMux_core);
264          sensitive << A<<B;
265
266         SC_METHOD(BMux_sel);
267          sensitive << BIN_ADDR;
268
269         SC_METHOD(BMux_Y);
270          sensitive <<A<<B;
271
272          //Accumulating methods
273         SC_METHOD(sY_accum);
```

```
274     sensitive << pimcore0_sY_part_select << pimcore_Y_vec <<
            pimcoreN_sY_part_select<<A<<B;
275
276     SC_METHOD(sA_accum);
277     sensitive << AMux0_sA_part_select<< AMuxN_sA_part_select <<
             part_sA <<A<<B;
278
279     SC_METHOD(sB_accum);
280     sensitive << BMux0_sB_part_select<< part_sB <<
            BMuxN_sB_part_select<<A<<B;
281
282     SC_METHOD(print_Y);
283     sensitive << Y;
284
285     SC_METHOD(print_A_B);
286     sensitive <<A<<B<<Y_ADDR<<FUNC;
287
288     //Port connections of modules
289
290     pimcore0.A(pimcore0_A_part_select);
291     pimcore0.B(pimcore0_B_part_select);
292     pimcore0.ASel(pimcore0_ASel_and);
293     pimcore0.BSel(pimcore0_BSel_and);
294     pimcore0.FUNC_IN(FUNC);
295     pimcore0.FUNC_ADDR(FUNC_ADDR);
```

```
296     pimcore0.IN_MODE(pimcore0_IN_MODE_part_select);
297     pimcore0.LOAD(pimcore0_LOAD);
298     pimcore0.RUN(pimcore0_RUN);
299     pimcore0.Y(pimcore0_sY_part_select);
300     pimcore0.clk(clk);
301     pimcore0.reset(reset);
302
303
304     AMux0.A(A);
305     AMux0.B(B);
306     AMux0.MODE(AMux0_SRC);
307     AMux0.CORES(AMux0_sY);
308     AMux0.SEL(AMux0_AIN_ADDR_part_select);
309     AMux0.Y(AMux0_sA_part_select);
310
311     BMux0.A(A);
312     BMux0.B(B);
313     BMux0.MODE(BMux0_SRC);
314     BMux0.CORES(AMux0_sY);
315     BMux0.SEL(BMux0_BIN_ADDR_part_select);
316     BMux0.Y(BMux0_sB_part_select);
317
318     for (auto G = 0; G<N-2; ++G)
319     {
320       pimcore[G].A(pimcore_A_vec[G]); //done
```

```
321         pimcore[G].B(pimcore_B_vec[G]  ); //done
322         pimcore[G].ASel(pimcore_ASel_and_vec[G]);  //done
323         pimcore[G].BSel(pimcore_BSel_and_vec[G]);
324         pimcore[G].FUNC_IN(FUNC);
325         pimcore[G].FUNC_ADDR(FUNC_ADDR);
326         pimcore[G].IN_MODE(pimcore_in_mode_vec[G]);  //done
327         pimcore[G].LOAD(pimcore_LOAD_vec[G]); //done
328         pimcore[G].RUN(pimcore_RUN_vec[G]); //done
329         pimcore[G].Y(pimcore_Y_vec[G]); //done
330         pimcore[G].clk(clk);
331         pimcore[G].reset(reset);
332
333         AMux[G].A(A);
334         AMux[G].B(B);
335         AMux[G].MODE(AMux_mode_vec[G]); //done
336         AMux[G].CORES(AMux_core_vec[G]); //done
337         AMux[G].SEL(AMux_sel_vec[G]); //done
338         AMux[G].Y(part_sA[G]); //done
339
340         BMux[G].A(A);
341         BMux[G].B(B);
342         BMux[G].MODE(BMux_mode_vec[G]);; //done
343         BMux[G].CORES(BMux_core_vec[G]);; //done
344         BMux[G].SEL(BMux_sel_vec[G]);; //done
345         BMux[G].Y(part_sB[G]); //DONE
```

```
346
347         }
348
349         pimcoreN.A(pimcoreN_A);
350         pimcoreN.B(pimcoreN_B);
351         pimcoreN.ASel(pimcoreN_ASel_and);
352         pimcoreN.BSel(pimcoreN_BSel_and);
353         pimcoreN.FUNC_IN(FUNC);
354         pimcoreN.FUNC_ADDR(FUNC_ADDR);
355         pimcoreN.IN_MODE(pimcoreN_IN_MODE_part_select);
356         pimcoreN.LOAD(pimcoreN_LOAD);
357         pimcoreN.RUN(pimcoreN_RUN);
358         pimcoreN.Y(pimcoreN_sY_part_select);
359         pimcoreN.clk(clk);
360         pimcoreN.reset(reset);
361
362         AMuxN.A(A);
363         AMuxN.B(B);
364         AMuxN.MODE(AMuxN_SRC);
365         AMuxN.CORES(AMuxN_sY);
366         AMuxN.SEL(AMuxN_AIN_ADDR_part_select);
367         AMuxN.Y(AMuxN_sA_part_select);
368
369         BMuxN.A(A);
370         BMuxN.B(B);
```

```
371         BMuxN.MODE(BMuxN_SRC);
372         BMuxN.CORES(AMuxN_sY);
373         BMuxN.SEL(BMuxN_BIN_ADDR_part_select);
374         BMuxN.Y(BMuxN_sB_part_select);
375
376         YMux.IN(sY_concat);
377         YMux.SEL(Y_ADDR);
378         YMux.OUT(Y);
379       }
380
381     //pimcore0 methods
382
383       void print_A_B()
384       {
385         //  cout<< "at time: " << sc_time_stamp()<< "---------->
              "<< "A = "<< A <<endl;
386         // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
               "B = "<< B <<endl;
387         // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
               "FUNC = "<< FUNC <<endl;
388         // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
               "RUN = "<< RUN <<endl;
389         // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
               "LOAD_CORE = "<< LOAD_CORE <<endl;
```

```
390     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "SRC_MODE = "<< SRC_MODE <<endl;
391     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "IN_MODE = "<< IN_MODE <<endl;
392     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "AIN_ADDR = "<< AIN_ADDR <<endl;
393     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "BIN_ADDR = "<< BIN_ADDR <<endl;
394     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "Y_ADDR = "<< Y_ADDR <<endl;
395     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "clk = "<< clk <<endl;
396   }
397
398   void pimcore0_A()
399   {
400     sc_bv<(N*(M/2))> sA_copy;
401     sA_copy = sA;
402     pimcore0_A_part_select = sA_copy.range((0+M/2)-1,0);
403     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
          "pimcore0_A_part_select = "<< pimcore0_A_part_select <<
          endl;
404     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
          "sA = "<< sA <<endl;
405
```

```
406
407     }
408
409     void pimcore0_B()
410     {
411       sc_bv<(N*(M/2))> sB_copy;
412       sB_copy = sB;
413       pimcore0_B_part_select = sB_copy.range((0+M/2)-1,0);
414       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "pimcore0_B_part_select = "<< pimcore0_B_part_select <<
              endl;
415       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "FUNC_ADDR = "<< FUNC_ADDR<<endl;
416       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "pimcore0_B_part_select = "<< pimcore0_B_part_select <<
              endl;
417
418     }
419
420     void pimcore0_in_mode()
421     {
422       sc_bv<2*N> in_mode_copy;
423       in_mode_copy = IN_MODE.read();
424       pimcore0_IN_MODE_part_select = in_mode_copy.range(0+2-1,0);
```

```
425     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "pimcore0_IN_MODE_part_select = "<<
            pimcore0_IN_MODE_part_select <<endl;
426     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "FUNC = "<< FUNC<<endl;
427
428   }
429
430   // void pimcore0_sY()
431   // {
432   //   sc_bv<N*M> sY_copy;
433   //   sY_copy = sY;
434   //   pimcore0_sY_part_select = sY_copy.range(0,0+M);
435   // }
436
437   void pimcore0_sY()
438   {
439     sc_bv<N*M> sY_copy;
440     sY_copy = sY;
441     sY_copy.range((0+M/2)-1,0) = pimcore0_sY_part_select.read()
            ;
442     sY_collect = sY_copy;
443     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sY_collect = "<< sY_collect <<endl;
```

```
444       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "pimcore0_sY_part_select = "<< pimcore0_sY_part_select.
              read() <<endl;
445
446
447     }
448
449
450     void pimcore0_add_A()
451     {
452       sc_bv<N*size_addr> AIN_ADDR_copy;
453       AIN_ADDR_copy = AIN_ADDR;
454       sc_bv<2*N> SRC_MODE_copy;
455       SRC_MODE_copy = SRC_MODE;
456       bool temp;
457
458       temp = AIN_ADDR_copy[size_addr-1] && SRC_MODE_copy[0];
459
460       pimcore0_ASel_and = temp;
461       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "pimcore0_ASel_and = "<< pimcore0_ASel_and <<endl;
462     }
463
464     void pimcore0_add_B()
465     {
```

```
466         sc_bv<N*size_addr> BIN_ADDR_copy;
467         BIN_ADDR_copy = BIN_ADDR;
468         sc_bv<2*N> SRC_MODE_copy;
469         SRC_MODE_copy = SRC_MODE;
470
471         pimcore0_BSel_and = BIN_ADDR_copy[size_addr-1] &&
                SRC_MODE_copy[1];
472     }
473
474     void pimcore0_LOAD_select()
475     {
476         sc_bv<N> LOAD_CORE_copy;
477         LOAD_CORE_copy = LOAD_CORE;
478         pimcore0_LOAD.write(bool(LOAD_CORE_copy[0]));
479
480     }
481
482     void pimcore0_RUN_select()
483
484     {
485         sc_bv<N> RUN_copy;
486         RUN_copy = RUN;
487
488         pimcore0_RUN.write(bool(RUN_copy[0]));
489
```

```
490     }
491
492     //AMux0 methods
493
494     void AMux0_SRC_MODE_select()
495     {
496       sc_bv<2*N> SRC_MODE_copy;
497
498       SRC_MODE_copy = SRC_MODE;
499
500       AMux0_SRC = bool(SRC_MODE_copy[0]);
501       BMux0_SRC = bool(SRC_MODE_copy[1]);
502
503
504     }
505
506     void AMux0_CORES_range()
507     {
508       sc_bv<N*M> sY_copy;
509       sY_copy = sY;
510       AMux0_sY = sY_copy.range((M*N)-1,M);
511
512     }
513
514     void AMux0_SEL()
```

```
515     {
516       sc_bv<N*size_addr> AIN_ADDR_copy;
517       AIN_ADDR_copy = AIN_ADDR.read();
518       AMux0_AIN_ADDR_part_select = AIN_ADDR_copy.range(0+
              size_addr-1,0);
519     }
520
521     void AMux0_Y()
522     {
523       sc_bv<N*M/2> sA_copy;
524       sA_copy = sA;
525       sA_copy.range(0+M/2,0) = AMux0_sA_part_select.read();
526       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "AMux0_sA_part_select= "<< AMux0_sA_part_select <<endl;
527       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sA= "<< sA <<endl;
528       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sA_copy= "<< sA_copy <<endl;
529       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
             "sA_collect= "<< sA_collect <<endl;
530       sA_collect = sA_copy;
531
532     }
533     //BMux0 methods
534
```

```
535     void BMux0_SEL()
536     {
537       sc_bv<N*size_addr> BIN_ADDR_copy;
538       BIN_ADDR_copy = BIN_ADDR.read();
539       BMux0_BIN_ADDR_part_select = BIN_ADDR_copy.range(0+
              size_addr-1,0);
540     }
541     void BMux0_Y()
542     {
543       sc_bv<N*M/2> sB_copy;
544       sB_copy = sB;
545       sB_copy.range((0+M/2)-1,0) = BMux0_sB_part_select.read();
546       sB_collect = sB_copy;
547       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sB_collect= "<< sB_collect <<endl;
548
549     }
550     // Generate statements - pimcore methods
551     void pimcore_A_range()
552     {
553       sc_bv<(N*(M/2))> sA_copy;
554       sA_copy = sA;
555
556       for(auto i =1; i<N-1; ++i)
557       {
```

```
558         pimcore_A_vec[i] = sA_copy.range(((i+1)*(M/2))-1,(i*M/2))
                ;
559         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "pimcore_A_vec" << "["<< i <<"] =" << pimcore_A_vec
                [i] <<endl;
560       }
561
562
563     }
564
565     void pimcore_B_range()
566     {
567       sc_bv<(N*(M/2))> sB_copy;
568       sB_copy = sB;
569
570       for(auto i =1; i<N-1;++i)
571       {
572         pimcore_B_vec[i] = sB_copy.range(((i+1)*(M/2))-1,(i*M/2))
                ;
573         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "pimcore_B_vec" << "["<< i <<"] =" << pimcore_B_vec
                [i] <<endl;
574       }
575
576
```

```
577     }
578
579     void pimcore_AIN_ADDR()
580     {
581       sc_bv<N*size_addr> AIN_ADDR_copy;
582       sc_bv<2*N> src_mode_copy;
583       src_mode_copy = SRC_MODE.read();
584       AIN_ADDR_copy = AIN_ADDR.read();
585
586       for(auto i=1; i<N-1; ++i)
587       {
588         pimcore_ASel_and_vec[i] = AIN_ADDR_copy[((i+1)*size_addr)
              -1] && src_mode_copy[2*i];
589       }
590
591     }
592
593     void pimcore_BIN_ADDR()
594     {
595       sc_bv<N*size_addr> BIN_ADDR_copy;
596       sc_bv<2*N> src_mode_copy;
597       src_mode_copy = SRC_MODE.read();
598       BIN_ADDR_copy = BIN_ADDR.read();
599
600       for(auto i=1; i<N-1; ++i)
```

```
601         {
602             pimcore_BSel_and_vec[i] = BIN_ADDR_copy[((i+1)*size_addr)
                    -1] && src_mode_copy[2*i+1];
603         }
604
605     }
606
607     void pimcore_in_mode()
608     {
609         sc_bv<2*N> in_mode_copy;
610         in_mode_copy = IN_MODE.read();
611         for(auto i=1; i<N-1; ++i)
612         {
613             pimcore_in_mode_vec[i] = in_mode_copy.range((i*2)+2-1,i
                    *2);
614         }
615
616     }
617
618     void pimcore_LOAD()
619     {
620         sc_bv<N> LOAD_CORE_copy;
621         LOAD_CORE_copy = LOAD_CORE.read();
622         for(auto i=1; i<N-1; ++i)
623         {
```

```
624         pimcore_LOAD_vec[i].write(bool(LOAD_CORE_copy[i]));
625       }
626
627     }
628
629     void pimcore_RUN()
630     {
631       sc_bv<N> RUN_copy;
632       RUN_copy = RUN.read();
633       for(auto i = 1; i<N-1; ++i)
634       {
635         pimcore_RUN_vec[i].write(bool(RUN_copy[i]));
636       }
637     }
638
639     void pimcore_Y()
640     {
641       sc_bv<N*M> sY_copy;
642       sY_copy = sY;
643       for(auto i = 1; i<N-1; ++i)
644       {
645         sY_copy.range((i*M)+M-1,i*M) = pimcore_Y_vec[i].read() ;
646         //cout<< "at time: " << sc_time_stamp()<< "---------->
              "<< "pimcore_Y_vec" << "["<< i <<"] =" << pimcore_Y_vec
              [i] <<endl;
```

```
647       }
648       sY_collect = sY_copy;
649
650   }
651   // Generate statements - AMux methods
652
653   void AMux_mode()
654   {
655       sc_bv<2*N> src_mode_copy;
656       sc_bv<1> single_bit;
657       src_mode_copy = SRC_MODE.read();
658       for(auto i = 1; i<N-1; ++i)
659       {
660           single_bit = bool(src_mode_copy[i*2]);
661           AMux_mode_vec[i] = single_bit;
662           //AMux_mode_vec[i] = 1;
663       }
664
665   }
666
667   void AMux_core()
668   {
669       sc_bv<N*M> sY_copy;
670       sY_copy = sY;
671       sc_bv<((N*M)-M)> range1;
```

```
672       sc_bv<M> range2;
673
674       for(auto i = 1; i<N-1; ++i)
675       {
676         range1 = sY_copy.range((N*M)-1,(i+1)*M);
677         range2 = sY_copy.range(i*M-1,0);
678         AMux_core_vec[i] = (range1, range2);
679         //AMux_core_vec[i] = 1;
680
681       }
682     }
683
684     void AMux_sel()
685     {
686       sc_bv<N*size_addr> AIN_ADDR_copy;
687       AIN_ADDR_copy = AIN_ADDR.read();
688       for(auto i = 1; i<N-1; ++i)
689       {
690         AMux_sel_vec[i] = AIN_ADDR_copy.range(i*size_addr+
              size_addr-1,i*size_addr);
691         //AMux_sel_vec[i] = 1;
692
693       }
694     }
695
```

```
696   void AMux_Y()
697   {
698     sc_bv<N*(M/2)> write_val;
699     for(auto i= 1; i < N-1; ++i)
700     {
701       write_val.range((i*M/2)-1+M/2,i*(M/2)) = part_sA[i].read
            ();
702       //cout<< "at time: " << sc_time_stamp()<< "---------->
            "<< "part_sA" << "["<< i <<"] =" << part_sA[i] <<endl;
703       //cout<< "at time: " << sc_time_stamp()<< "---------->
            "<< "write_val= "<< write_val <<endl;
704     }
705     sA_collect = write_val;
706     //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
          "sA_collect = "<< sA_collect <<endl;
707   }
708
709   //Generate statements - BMux methods
710
711   void BMux_mode()
712   {
713     sc_bv<2*N> src_mode_copy;
714     sc_bv<1> single_bit;
715     src_mode_copy = SRC_MODE.read();
716     for(auto i = 1; i< N-1; ++i)
```

```
717       {
718         single_bit = bool (src_mode_copy[(i*2)+1]);
719         BMux_mode_vec[i] = single_bit;
720       }
721
722     }
723
724     void BMux_core()
725     {
726       sc_bv<N*M> sY_copy;
727       sY_copy = sY;
728       sc_bv<((N*M)- M)> range1;
729       sc_bv<M> range2;
730
731       for(auto i = 1; i<N-1; ++i)
732       {
733         range1 = sY_copy.range((N*M)-1,(i+1)*M);
734         range2 = sY_copy.range(i*M-1,0);
735         BMux_core_vec[i] = (range1, range2);
736
737       }
738     }
739
740     void BMux_sel()
741     {
```

```
742     sc_bv<N*size_addr> BIN_ADDR_copy;
743     BIN_ADDR_copy = BIN_ADDR.read();
744     for(auto i = 1; i<N-1; ++i)
745     {
746       BMux_sel_vec[i] = BIN_ADDR_copy.range(i*size_addr+
            size_addr-1,i*size_addr);
747     }
748   }
749
750   void BMux_Y()
751   {
752     sc_bv<N*(M/2)> write_val;
753     for(auto i= 1; i < N-1; ++i)
754     {
755       write_val.range((i*M/2)-1+M/2,i*(M/2)) = part_sB[i].read
            ();
756       //cout<< "at time: " << sc_time_stamp()<< "---------->
            "<< "part_sB" << "["<< i <<"] =" << part_sB[i] <<endl;
757     }
758     sB_collect = write_val;
759   }
760
761   //pimcoreN methods
762   void pimcoreN_A_range()
763   {
```

```
764     sc_bv<(N*(M/2))> sA_copy;
765     sA_copy = sA;
766     pimcoreN_A = sA_copy.range((N*(M/2))-1,((N-1)*(M/2)));
767     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sA = "<< sA <<endl;
768     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sA_copy= "<< sA_copy <<endl;
769      //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
             "pimcoreN_A = "<< pimcoreN_A <<endl;
770
771   }
772
773   void pimcoreN_B_range()
774   {
775     sc_bv<(N*(M/2))> sB_copy;
776     sB_copy = sB;
777     pimcoreN_B = sB_copy.range((N*(M/2))-1,((N-1)*(M/2)));
778     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sB = "<< sB <<endl;
779     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sB_copy = "<< sB_copy <<endl;
780     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "pimcoreN_B = "<< pimcoreN_B <<endl;
781
782   }
```

```
783
784     void pimcoreN_add_A()
785     {
786       sc_bv<N*size_addr> AIN_ADDR_copy;
787       AIN_ADDR_copy = AIN_ADDR;
788       sc_bv<2*N> src_mode_copy;
789       src_mode_copy = SRC_MODE.read();
790
791       pimcoreN_ASel_and = AIN_ADDR_copy[N*size_addr-1] &&
              src_mode_copy[2*(N-1)];
792     }
793
794     void pimcoreN_add_B()
795     {
796       sc_bv<N*size_addr> BIN_ADDR_copy;
797       BIN_ADDR_copy = BIN_ADDR;
798       sc_bv<2*N> src_mode_copy;
799       src_mode_copy = SRC_MODE.read();
800
801       pimcoreN_BSel_and = BIN_ADDR_copy[N*size_addr-1] &&
              src_mode_copy[2*(N-1)+1];
802     }
803
804     void pimcoreN_IN_MODE()
805     {
```

```
806       sc_bv<2*N> in_mode_copy;
807       in_mode_copy= IN_MODE.read();
808       pimcoreN_IN_MODE_part_select = in_mode_copy.range(((N-1)*2)
              +2-1,(N-1)*2);
809     }
810
811     void pimcoreN_sY()
812     {
813       sc_bv<N*M> sY_copy;
814       sY_copy = sY;
815       sY_copy.range(((N-1)*M)+M-1,(N-1)*M) =
              pimcoreN_sY_part_select.read();
816       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sY= "<< sY <<endl;
817       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sY_copy= "<< sY_copy <<endl;
818       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "pimcoreN_sY_part_select= "<< pimcoreN_sY_part_select <<
             endl;
819       sY_collect = sY_copy;
820     }
821
822     void pimcoreN_LOAD_select()
823     {
824       sc_bv<N> LOAD_CORE_copy;
```

```
825         LOAD_CORE_copy = LOAD_CORE;
826
827         pimcoreN_LOAD.write(bool(LOAD_CORE_copy[N-1]));
828
829     }
830
831     void pimcoreN_RUN_select()
832
833     {
834         sc_bv<N> RUN_copy;
835         RUN_copy = RUN;
836
837         pimcoreN_RUN.write(bool(RUN_copy[N-1]));
838
839     }
840
841     //AMuxN methods
842
843     void AMuxN_SRC_MODE_select()
844     {
845         sc_bv<2*N> SRC_MODE_copy;
846
847         SRC_MODE_copy = SRC_MODE;
848         //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "SRC_MODE= "<< SRC_MODE.read() <<endl;
```

```
849       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "SRC_MODE_copy= "<< SRC_MODE_copy <<endl;
850       AMuxN_SRC = bool (SRC_MODE_copy[2*N-2]);
851       BMuxN_SRC = bool (SRC_MODE_copy[2*N-1]);
852
853       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "AMuxN_SRC= "<< AMuxN_SRC <<endl;
854       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "BMuxN_SRC= "<< BMuxN_SRC <<endl;
855    }
856
857    void AMuxN_CORES_range()
858    {
859       sc_bv<N*M> sY_copy;
860       sY_copy = sY;
861       AMuxN_sY = sY_copy.range((M*N)-1,M);
862       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sY= "<< sY <<endl;
863       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sY_copy= "<< sY_copy <<endl;
864       // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "AMuxN_sY= "<< AMuxN_sY <<endl;
865
866    }
867
```

```
868     void AMuxN_AIN_ADDR()
869     {
870       sc_bv<N*size_addr> AIN_ADDR_copy;
871       AIN_ADDR_copy = AIN_ADDR.read();
872       AMuxN_AIN_ADDR_part_select = AIN_ADDR_copy.range(((N-1)*
              size_addr)+size_addr-1,(N-1)*size_addr);
873     }
874
875     void AMuxN_sA()
876     {
877       sc_bv<(N*(M/2))> sA_copy;
878       sA_copy = sA;
879       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "sA = "<< sA <<endl;
880       sA_copy.range(((N-1)*(M/2)) + (M/2)-1,(N-1)*(M/2)) =
              AMuxN_sA_part_select.read();
881       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "AMuxN_sA_part_select = "<< AMuxN_sA_part_select <<endl;
882       sA_collect = sA_copy;
883     }
884
885
886     //BMuxN methods
887
888     void BMuxN_BIN_ADDR()
```

```
889     {
890       sc_bv<N*size_addr> BIN_ADDR_copy;
891       BIN_ADDR_copy = BIN_ADDR.read();
892       BMuxN_BIN_ADDR_part_select = BIN_ADDR_copy.range(((N-1)*
              size_addr)+size_addr-1,(N-1)*size_addr);
893     }
894
895     void BMuxN_sB()
896     {
897       sc_bv<(N*(M/2))> sB_copy;
898       sB_copy = sB;
899       sB_copy.range(((N-1)*(M/2)) + (M/2)-1,(N-1)*(M/2)) =
              BMuxN_sB_part_select.read();
900       //cout<< "at time: " << sc_time_stamp()<< "----------> "<<
              "BMuxN_sB_part_select = "<< BMuxN_sB_part_select <<endl;
901       sB_collect = sB_copy;
902     }
903     //YMux methods
904
905     void YMux_concatenate()
906     {
907
908       sc_bv<N*M> sY_copy;
909       sY_copy = sY;
910       sc_bv<56> append_zeros;
```

```
911       append_zeros = 0;
912
913       sY_concat = (append_zeros,sY_copy);
914     }
915
916     void sY_accum()
917     {
918       sY = sY_collect;
919     }
920
921     void sA_accum()
922     {
923       sA = sA_collect;
924       //cout<< "at time: " << sc_time_stamp()<< "---------->" <<
              "sA = "<< sA <<endl;
925
926     }
927
928     void sB_accum()
929     {
930       sB = sB_collect;
931       //cout<< "at time: " << sc_time_stamp()<< "---------->"
             << "sB = "<< sB <<endl;
932     }
933
```

```
934     void print_Y()
935     {
936       //cout<< "at time: " << sc_time_stamp()<< "------------>"
              << "Y PIM Cluster = "<< Y <<endl;
937     }
938 };
939
940 #endif
```

II.2 pPIM core model

```
1  #ifndef PIM_H
2  #define PIM_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include "multiplexer.h"
10 #include "decoder.h"
11 #include "nbit_multiplexer.h"
12 #include "register256.h"
13 #include "register_file.h"
14 #include "sensitive_def.h"
15 //N=4
16 template <int N> class PIM : public sc_module
17 {
18 public:
19     //PORTS
20     sc_in<sc_bv<N> > A{"A"}; //4
21     sc_in<sc_bv<1> > ASel{"ASel"};
22     sc_in<sc_bv<N> > B{"B"};//4
23     sc_in<sc_bv<1> > BSel{"BSel"};
```

```
24      sc_in<sc_bv<1<<(2*N)> >FUNC_IN{"FUNC_IN"}; //256
25      constexpr static int size = ceil(log2(2*N)); //3
26      sc_in<sc_uint<size> > FUNC_ADDR{"FUNC_ADDR"}; //3
27      sc_in<sc_bv<2> > IN_MODE{"IN_MODE"};
28      sc_in<bool> LOAD{"LOAD"};
29      sc_in<bool> RUN{"RUN"};
30      sc_in<bool> reset{"reset"};
31      //sc_in_clk clk{"clk"};
32      sc_in<bool> clk;
33      sc_out<sc_bv<2*N> > Y{"Y"}; //8
34
35      //SIGNALS
36      sc_signal<sc_bv<(2*N)*(1<<(2*N))> > sFUNC{"sFUNC"};
37      sc_signal<sc_bv<N> > sA{"sA"};
38      sc_signal<sc_bv<N> > sB{"sB"};
39      sc_signal<sc_bv<N> > sAIn{"sAIn"};
40      sc_signal<sc_bv<N> > sBIn{"sBIn"};
41      sc_bv<2*N> sY;
42      sc_signal<sc_bv<2*N> > sFUNC_ADDR{"sFUNC_ADDR"};
43      sc_signal<sc_bv<2*N> > sY_A{"sY_A"};
44      sc_signal<sc_bv<2*N> > sY_B{"sY_B"};
45      //sc_signal<sc_bv<1<<(N*2)>> sFUNC_range_copy;
46
47      sc_signal<sc_bv<2*N> > sA_sB{"sA_sB"};
48      sc_signal<bool> mode0{"mode0"};
```

```
49      sc_signal<bool> mode1{"mode1"};
50
51      //Global declarations of instantiated modules
52      decoder<2*N> decoder_addr{"decoder_addr"};
53      register_file<1<<(2*N),N> func_regs{"func_regs"};
54      nbit_multiplexer<1,N> AMux{"AMux"};
55      nbit_multiplexer<1,N> BMux{"BMux"};
56      register256<N> Areg{"Areg"};
57      register256<N> Breg{"Breg"};
58
59
60      int G;
61
62      SC_HAS_PROCESS(PIM);
63
64      //VECTORS
65      constexpr static int sel_size = ceil(log2(1<<(2*N))); //8
66      sc_vector<multiplexer<(1<<(N*2)),sel_size> > mux256to1{"
           mux256to1", 2*N};
67      sc_vector<sc_signal<bool> > sY_bit{"sY_bit",2*N};
68      sc_vector<sc_signal<sc_bv<1<<(N*2)> >> sFUNC_range{"
           sFUNC_range", 2*N};
69      //sc_signal<sc_bv<(2*N)*(1<<(2*N))> > sFUNC{"sFUNC"};
70
71      PIM(sc_module_name name): sc_module(name)
```

```
72      {
73          SC_METHOD(concat_method_sY);
74          sensitive << sY_bit << A << B;
75
76          SC_METHOD(sFUNC_range_compute);
77          sensitive << sFUNC;
78
79          SC_METHOD(concat_sA_sB);
80          sensitive << sA << sB;
81
82          SC_METHOD(in_mode_bit_select);
83          sensitive << IN_MODE;
84
85          SC_METHOD(sY_bit_select);
86          sensitive << sY_bit<<A<<B;
87
88          SC_METHOD(print_ports);
89         // sensitive << FUNC_ADDR<<A<<B<<ASel<<BSel<<FUNC_IN<<
              IN_MODE<<LOAD<<RUN<<Y<<clk;
90          sensitive << sFUNC;
91
92          //Decoder port connections
93          decoder_addr.IN(FUNC_ADDR);
94          decoder_addr.OUT(sFUNC_ADDR); //works
95
```

```
96          //register_file port connections
97          func_regs.DATA_IN(FUNC_IN);
98          func_regs.WRITE_ADDR(sFUNC_ADDR);
99          func_regs.WRITE_EN(LOAD);
100         func_regs.READ_EN(RUN);
101         func_regs.clk(clk);
102         func_regs.reset(reset);
103         func_regs.DATA_OUT(sFUNC); //works
104
105         //nbit_multiplexer A port connections
106         AMux.IN(sY_A); //sc_in<sc_biguint<(1<<N)*M> > IN;
               concatenating 2 sc_Auints
107         AMux.SEL(ASel);
108         AMux.OUT(sAIn);
109
110         //nbit_multiplexer B port connections
111         BMux.IN(sY_B);
112         BMux.SEL(BSel);
113         BMux.OUT(sBIn);
114
115         //mux256to1 port connections
116         for (auto G = 0U; G< 2*N; ++G)
117         {
118             mux256to1[G].IN(FUNC_IN);
```

```
119             mux256to1[G].SEL(sA_sB); //SEL is 8 bit sA_sB = 4+4
                    bits ??
120             mux256to1[G].OUT(sY_bit[G]);
121         }
122
123
124         //register256 Areg port connections
125         Areg.DATA_IN(sAIn);
126         Areg.WRITE(mode0);
127         Areg.clk(clk);
128         Areg.reset(reset);
129         Areg.DATA_OUT(sA);
130
131         //register256 Breg port connections
132         Breg.DATA_IN(sBIn);
133         Breg.WRITE(mode1);
134         Breg.clk(clk);
135         Breg.reset(reset);
136         Breg.DATA_OUT(sB);
137
138
139
140     }
141
142
```

```
143
144         void print_ports()
145         {
146             sc_uint<size> FA;
147             FA = FUNC_ADDR;
148             cout<< "at time: " << sc_time_stamp()<<"
                    ---------------->"<< "sFUNC= "<< sFUNC <<endl;
149           //  cout<< "at time: " << sc_time_stamp()
                  <<"--------------------------------->"<< "clk = "<<
                  clk <<endl<<endl;
150           // cout<< "at time: " << sc_time_stamp()
                  <<"---------------->"<< "sAIn= "<< sAIn.read() <<endl;
151           //  cout<< "at time: " << sc_time_stamp()<<
                  "--------------->" << "mode0 = "<< mode0 <<endl;
152           // cout<< "at time: " << sc_time_stamp()
                  <<"---------------->"<< "sA = "<< sA.read() <<endl<<
                  endl;
153
154           // cout<< "at time: " << sc_time_stamp()
                  <<"----------------->"<< "sBIn = "<< sBIn.read() <<endl
                  ;
155           //  cout<< "at time: " << sc_time_stamp()<<
                  "---------------->" << "mode1 = "<< mode1 <<endl;
156           // cout<< "at time: " << sc_time_stamp()
                  <<"----------------->"<< "sB = "<< sB.read() <<endl<<
```

```
            endl;
157
158         //  cout<< "at time: " << sc_time_stamp()<<
              "----------->" << "sY_A = "<< sY_A <<endl;
159         //  cout<< "at time: " << sc_time_stamp()<<
              "----------->" << "sY_B = "<< sY_B <<endl;
160
161
162         //cout<< "at time: " << sc_time_stamp()<<
              "--------------->"<<"A = "<< A.read() <<endl;
163         // cout<< "at time: " << sc_time_stamp() <<
              "------------->"<<"B = "<< B.read() <<endl;
164          // cout<< "at time: " << sc_time_stamp()<<
               "------------->"<<"ASel = "<< ASel.read() <<endl;
165          // cout<< "at time: " << sc_time_stamp()<<
              "------------->"<<"BSel = "<< BSel.read() <<endl;
166          // cout<< "at time: " << sc_time_stamp()<<
              "------------->"<<"FUNC_IN = "<< FUNC_IN.read() <<
              endl;
167          // cout<< "at time: " << sc_time_stamp()<<
              "------------->"<< "FUNC_ADDR = "<< FUNC_ADDR.read()
              <<endl;
168          // cout<< "at time: " << sc_time_stamp()<<
              "------------->"<< "IN_MODE = "<< IN_MODE.read() <<
              endl;
```

```
169         // cout<< "at time: " << sc_time_stamp()<<
               "-------------->"<<"LOAD = "<< LOAD.read() <<endl;
170         // cout<< "at time: " << sc_time_stamp()<<
               "-------------->"<< "RUN = "<< RUN.read() <<endl;
171         // cout<< "at time: " << sc_time_stamp()<<
               "-------------->"<< "clk = "<< clk.read() <<endl;
172
173       }
174
175     void concat_method_sY()
176     {
177         sc_bv<2*N> s;
178         sc_bv<4> first_nib;
179         sc_bv<4> sec_nib;
180         sc_bv<2*N> sYA;
181         sc_bv<2*N> sYB;
182         sc_bv<N> reg_A;
183         sc_bv<N> reg_B;
184         reg_A = A;
185         reg_B = B;
186
187         s = sY;
188         first_nib = s.range(3,0);
189         sec_nib = s.range(7,4);
190
```

```
191         sYA = (first_nib,reg_A);
192
193         sYB = (sec_nib,reg_B);
194         sY_A.write(sYA);
195         sY_B.write(sYB);
196         // cout<< "at time: " << sc_time_stamp()<< endl << "
               sAIn = "<< sAIn <<endl;
197         // cout<< "at time: " << sc_time_stamp()<< endl << "
               sBIn = "<< sBIn <<endl;
198
199
200     }
201
202
203     void concat_sA_sB()
204     {
205         sc_bv<2*N> concat_A_B;
206         sc_bv<N> copy_A;
207         sc_bv<N> copy_B;
208
209         copy_A = A;
210         copy_B = B;
211
212         concat_A_B = (copy_A,copy_B);
213         sA_sB.write(concat_A_B);
```

```
214
215         //  cout<< "at time: " << sc_time_stamp()<< endl << "sA
              = "<< sA <<endl;
216         //  cout<< "at time: " << sc_time_stamp()<< endl << "sB
              = "<< sB <<endl;
217         // cout<< "at time: " << sc_time_stamp()<< endl << "
              sA_sB = "<< sA_sB <<endl<<endl;
218
219     }
220
221     void sFUNC_range_compute()
222     {
223         sc_bv<(2*N)*(1<<(2*N))> func;//2048
224         func = sFUNC;
225         sc_bv<1<<(N*2)> sFUNC_range_copy;
226         cout<< "INVOKED"<< endl;
227
228         for (auto G = 0U; G< 2*N; ++G)
229         {
230             sFUNC_range_copy = func.range(((1<<(2*N))*(G+1))
                  -1,((1<<(2*N))*G));
231             //sFUNC_range_copy = 0x84928fff892389;
232             sFUNC_range[G] = sFUNC_range_copy;
233            // cout<< "at time: " << sc_time_stamp()<<
                "------------>" << "sFUNC_range_copy = "<<
```

```
                sFUNC_range_copy <<endl;
234             //cout<< "at time: " << sc_time_stamp()<<
                  "------------> "<< "sFUNC_range" << "["<< G <<"]
                  =" << sFUNC_range[G] <<endl;
235         }
236
237         //cout<< "at time: " << sc_time_stamp()<< endl << "
              sFUNC_range_copy = "<< sFUNC_range_copy <<endl;
238         //cout<< "at time: " << sc_time_stamp()<< endl << "
              sFUNC_range = "<< sFUNC_range <<endl;
239
240
241
242     }
243
244     void in_mode_bit_select()
245     {
246         sc_bv<2> mode;
247         mode = IN_MODE;
248          mode0 = bool (mode[0]);
249          mode1 = bool (mode[1]);
250
251
252     }
253
```

```
254     void sY_bit_select()
255     {
256
257         sc_bv<2*N> write_val;
258         for(auto i=0U; i<2*N; ++i)
259         {
260             write_val[i] = sY_bit[i].read();
261
262         }
263         sY = write_val;
264         // cout<< "at time: " << sc_time_stamp()<< endl << "sY =
                "<< sY <<endl;
265         Y.write(sY);
266         //cout<< "at time: " << sc_time_stamp()<< endl << "Y
                PIM  = "<< Y <<endl;
267
268     }
269
270
271 };
272
273 #endif
```

II.3 pPIM cluster testbench

```
1  #ifndef PIMC_STI_MULUS_H
2  #define PIMC_STI_MULUS_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  //#include <stdio>
10 #include <cstdio>
11 //8,9
12 template <int M_stim, int N_stim> class PIMC_stimulus : public
       sc_module {
13   public:
14
15   sc_out<sc_bv<M_stim> >A{"A_stimC"};
16   sc_out<sc_bv<M_stim> >B{"B_stimC"};
17
18   sc_out<sc_bv<1<<M_stim> > FUNC{"FUNC_stimC"}; //256
19   constexpr static int size = ceil(log2(M_stim));
20   sc_out<sc_uint<size> > FUNC_ADDR{"FUNC_ADDR_stimC"}; //3
21   sc_out<sc_bv<N_stim> > LOAD_CORE{"LOAD_CORE_stimC"};
22
```

```
23    constexpr static int size_addr = ceil(log2(2*(N_stim-1)));
24    sc_out<sc_bv<N_stim*size_addr> > AIN_ADDR{"AIN_ADDR_stimC"};
        //36
25    sc_out<sc_bv<N_stim*size_addr> > BIN_ADDR{"BIN_ADDR_stimC"};
26
27    sc_out<sc_bv<2*N_stim> > SRC_MODE{"SRC_MODE_stimC"};
28    sc_out<sc_bv<2*N_stim> > IN_MODE{"IN_MODE_stimC"};
29    //sc_clock clk("clock", 6, SC_NS);
30    //sc_out<bool> clk{"Clock_stim"};
31    sc_out<bool> clk;
32    //sc_port<sc_signal_in_if<bool>> clock_port{"clock_port"};
33    //sc_in<bool> clk;
34
35    sc_out<sc_bv<N_stim> > RUN{"RUN_stimC"};
36    //sc_out<bool> clk{"clk_stimC"};
37    sc_out<bool> reset{"reset_stimC"};
38    constexpr static int size_y_addr = ceil(log2(N_stim));
39    sc_out<sc_bv<size_y_addr> > Y_ADDR{"Y_ADDR_stimC"};
40     sc_in<sc_bv<M_stim> >Y;
41     sc_bv<M_stim>Y_copy;
42
43     //ports created for waveform
44     sc_out<sc_bv<8>> add_port;
45     sc_out <sc_bv<8>> mult_port;
46     sc_out <sc_bv<16> > Accumulator;
```

```
47      //sc_out<bool> clock_port;
48
49    //bit vector
50
51    //  sc_vector<sc_signal<sc_bv<2048>>> FUNC_MULT;
52    //  sc_vector<sc_signal<sc_bv<2048>>> FUNC_ADD;
53
54
55    sc_bv<16> ACC = 0;
56    sc_bv<M_stim> B_copy;
57    //integers
58
59
60    //int index =0;
61    //Function word generator variable
62    int index=0;
63    int a =0;
64    int b =0;
65    //int A_reg;
66    //int B_reg;
67    //  sc_uint<M_stim> A_reg = 0;
68    //  sc_uint<M_stim> B_reg = 0;
69    //  sc_uint<M_stim> A_reg_copy = 0;
70    //  sc_uint<M_stim> B_reg_copy = 0;
71    //  sc_uint<M_stim> add_one = 1;
```

```
72    sc_bv<2048> FUNC_ADD =0;
73    sc_bv<2048> FUNC_MULT=0;
74    sc_bv<2048> func_add_copy=0;
75    sc_bv<256> func_add_slice=0;
76    sc_bv<2048> func_mult_copy=0;
77    sc_bv<256> func_mult_slice=0;
78    sc_bv<8> ADD_tmp=0;//8
79    sc_bv<8> MUL_tmp=0;//8
80    sc_bv<8> add=0;//8
81    sc_bv<8> mult=0;
82
83
84    SC_HAS_PROCESS(PIMC_stimulus);
85
86    PIMC_stimulus(sc_module_name name) : sc_module(name)
87    {
88       SC_THREAD(clock_gen1);
89       //sensitive << clock_port;
90       SC_THREAD(test_bench);
91       sensitive << clk.neg();
92       dont_initialize();
93
94
95    }
96
```

```
97    void test_bench()
98    {
99
100     for( a=0; a<16; ++a)
101     {
102
103       for( b=0; b<16; ++b)
104       {
105
106         add = a + b;
107         mult = a*b;
108
109         ADD_tmp = add;
110         MUL_tmp = mult;
111
112         add_port = ADD_tmp;
113         mult_port = MUL_tmp;
114         // cout<< "at time: " << sc_time_stamp()<< "---------->
              "<< "ADD_tmp in stimulus= "<< ADD_tmp <<endl;
115         // cout<< "at time: " << sc_time_stamp()<< "---------->
              "<< "MUL_tmp in stimulus= "<< MUL_tmp <<endl;
116         for( auto kk=0U;kk<8;++kk)
117         {
118           FUNC_MULT[(kk*256)+index] = MUL_tmp[kk];
119           FUNC_ADD[(kk*256)+index] = ADD_tmp[kk];
```

```
120
121             }
122             ++index;
123
124         }
125     }
126
127
128             func_add_copy = FUNC_ADD;
129             func_mult_copy = FUNC_MULT;
130
131
132         wait(clk.negedge_event());
133     reset.write(1);
134     wait(clk.negedge_event());
135     reset.write(0);
136
137
138     RUN.write(0xff);
139
140     //wait(150,SC_NS);
141
142     for(auto i = 0; i<8; ++i)
143     {
```

```
144         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "FUNC_MULT= "<< FUNC_MULT <<endl;
145         func_mult_slice = FUNC_MULT.range(((i*256)+256-1),i*256);
146
147
148         FUNC.write(func_mult_slice);
149         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "FUNC(mult)= "<< FUNC <<endl;
150         FUNC_ADDR.write(i);
151         LOAD_CORE.write(0x1E0);
152         wait(10,SC_NS);
153       }
154
155       for(auto j = 0; j< 8; ++j)
156       {
157         func_add_slice = func_add_copy.range(((j*256)+256-1),j
                *256);
158         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "func_add_slice= "<< func_add_slice <<endl;
159         FUNC.write(func_add_slice);
160         //cout<< "at time: " << sc_time_stamp()<< "---------->
                "<< "FUNC(add)= "<< FUNC <<endl;
161         FUNC_ADDR.write(j);
162         LOAD_CORE.write(0x1F);
163         wait(10,SC_NS);
```

```
164         }
165
166
167         //for (auto i = 0; i < 16; ++i)
168         //{
169            wait(clk.negedge_event());
170         //wait(6,SC_NS);
171               Accumulator.write(ACC);
172               A.write(36);
173               B.write(129);
174               IN_MODE.write(0x3FC00);
175               SRC_MODE.write(0x3FC00);
176               AIN_ADDR.write(0x223300000);
177                  BIN_ADDR.write(0x010100000);
178
179
180            wait(clk.negedge_event());
181                  //wait(6,SC_NS);
182               B_copy = B.read();
183               B_copy.range(3,0) = ACC.range(3,0);
184               B.write(B_copy);
185               IN_MODE.write(0x00333);
186                  SRC_MODE.write(0x00001);
187                  AIN_ADDR.write(0x0B0C00);
188                  BIN_ADDR.write(0xD0F0E);
```

```
189             RUN.write(0x1E0);
190
191
192              wait(clk.negedge_event());
193            Y_copy = Y.read();
194            // ACC.range(3,0) = Y_copy.range(3,0);
195            Accumulator.write(ACC);
196            B_copy.range(3,0) = ACC.range(7,4);
197            B.write(B_copy);
198              IN_MODE.write(0x003BB);
199              SRC_MODE.write(0x00113);
200              AIN_ADDR.write(0x80800);
201              BIN_ADDR.write(0x87A38);
202              RUN.write(0x015);
203               wait(clk.negedge_event());
204               B_copy.range(3,0) = ACC.range(11,8);
205               B.write(B_copy);
206               IN_MODE.write(0x00357);
207               SRC_MODE.write(0x00201);
208               AIN_ADDR.write(0x17638);
209               BIN_ADDR.write(0x00002);
210               RUN.write(0x015);
211               Y_ADDR.write(0x0000);
212                wait(clk.negedge_event());
213            Y_copy = Y.read();
```

```
214         ACC.range(7,4) = Y_copy.range(3,0);
215         Accumulator.write(ACC);
216         B_copy.range(3,0) = ACC.range(15,12);
217         B.write(B_copy);
218            IN_MODE.write(0x003E3);
219            SRC_MODE.write(0x00342);
220            AIN_ADDR.write(0x08006);
221            BIN_ADDR.write(0x89208);
222            RUN.write(0x01B);
223            wait(clk.negedge_event());
224            IN_MODE.write(0x00333);
225            SRC_MODE.write(0x00121);
226            AIN_ADDR.write(0x80408);
227            BIN_ADDR.write(0x10802);
228            RUN.write(0x01D);
229            wait(clk.negedge_event());
230            ACC.range(11,8) = Y_copy.range(3,0);
231            Accumulator.write(ACC);
232         IN_MODE.write(0x0000B);
233            SRC_MODE.write(0x00002);
234            AIN_ADDR.write(0x00006);
235            BIN_ADDR.write(0x00028);
236            RUN.write(0x015);
237            wait(clk.negedge_event());
238         IN_MODE.write(0x00004);
```

```
239             SRC_MODE.write(0x00000);
240             AIN_ADDR.write(0x00000);
241             BIN_ADDR.write(0x00000);
242             RUN.write(0x001);
243             Y_ADDR.write(0x0001);
244              wait(clk.negedge_event());
245             ACC.range(15,12) = Y_copy.range(3,0);
246              Accumulator.write(ACC);
247             IN_MODE.write(0x00000);
248             RUN.write(0x002);
249              wait(clk.negedge_event());
250             Y_ADDR.write(0x0000);
251             IN_MODE.write(0x00000);
252             RUN.write(0x000);
253              wait(clk.negedge_event());
254
255
256             A.write(9);
257             B.write(99);
258              wait(clk.negedge_event());
259           IN_MODE.write(0x3FC00);
260           SRC_MODE.write(0x3FC00);
261           AIN_ADDR.write(0x223300000);
262             BIN_ADDR.write(0x010100000);
263              wait(clk.negedge_event());
```

```
264         B_copy = B.read();
265         B_copy.range(3,0) = ACC.range(3,0);
266         B.write(B_copy);
267         IN_MODE.write(00333);
268            SRC_MODE.write(0x00001);
269            AIN_ADDR.write(0x0B0C00);
270            BIN_ADDR.write(0xD0F0E);
271         RUN.write(0x1E0);
272            wait(clk.negedge_event());
273         Y_copy = Y.read();
274         ACC.range(3,0) = Y_copy.range(3,0);
275         Accumulator.write(ACC);
276         B_copy.range(3,0) = ACC.range(7,4);
277         B.write(B_copy);
278            IN_MODE.write(0x003BB);
279            SRC_MODE.write(0x00113);
280            AIN_ADDR.write(0x80800);
281            BIN_ADDR.write(0x87A38);
282            RUN.write(0x015);
283            wait(clk.negedge_event());
284            B_copy.range(3,0) = ACC.range(11,8);
285            B.write(B_copy);
286            IN_MODE.write(0x00357);
287            SRC_MODE.write(0x00201);
288            AIN_ADDR.write(0x17638);
```

```
289             BIN_ADDR.write(0x00002);
290             RUN.write(0x015);
291             Y_ADDR.write(0x0000);
292             wait(clk.negedge_event());
293           Y_copy = Y.read();
294           ACC.range(7,4) = Y_copy.range(3,0);
295           Accumulator.write(ACC);
296           B_copy.range(3,0) = ACC.range(15,12);
297           B.write(B_copy);
298             IN_MODE.write(0x003E3);
299             SRC_MODE.write(0x00342);
300             AIN_ADDR.write(0x08006);
301             BIN_ADDR.write(0x89208);
302             RUN.write(0x01B);
303             wait(clk.negedge_event());
304             IN_MODE.write(0x00333);
305             SRC_MODE.write(0x00121);
306             AIN_ADDR.write(0x80408);
307             BIN_ADDR.write(0x10802);
308             RUN.write(0x01D);
309             wait(clk.negedge_event());
310             ACC.range(11,8) = Y_copy.range(3,0);
311             Accumulator.write(ACC);
312           IN_MODE.write(0x0000B);
313             SRC_MODE.write(0x00002);
```

```
314             AIN_ADDR.write(0x00006);
315             BIN_ADDR.write(0x00028);
316             RUN.write(0x015);
317              wait(clk.negedge_event());
318           IN_MODE.write(0x00004);
319             SRC_MODE.write(0x00000);
320             AIN_ADDR.write(0x00000);
321             BIN_ADDR.write(0x00000);
322             RUN.write(0x001);
323             Y_ADDR.write(0x0001);
324              wait(clk.negedge_event());
325             ACC.range(15,12) = Y_copy.range(3,0);
326              Accumulator.write(ACC);
327             IN_MODE.write(0x00000);
328             RUN.write(0x002);
329              wait(clk.negedge_event());
330             Y_ADDR.write(0x0000);
331             IN_MODE.write(0x00000);
332             RUN.write(0x000);
333              wait(clk.negedge_event());
334
335
336             A.write(13);
337             B.write(141);
338              wait(clk.negedge_event());
```

```
339         IN_MODE.write(0x3FC00);
340         SRC_MODE.write(0x3FC00);
341         AIN_ADDR.write(0x223300000);
342            BIN_ADDR.write(0x010100000);
343            wait(clk.negedge_event());
344         B_copy = B.read();
345         B_copy.range(3,0) = ACC.range(3,0);
346         B.write(B_copy);
347         IN_MODE.write(00333);
348            SRC_MODE.write(0x00001);
349            AIN_ADDR.write(0x0B0C00);
350            BIN_ADDR.write(0xD0F0E);
351         RUN.write(0x1E0);
352            wait(clk.negedge_event());
353         Y_copy = Y.read();
354         ACC.range(3,0) = Y_copy.range(3,0);
355         Accumulator.write(ACC);
356         B_copy.range(3,0) = ACC.range(7,4);
357         B.write(B_copy);
358            IN_MODE.write(0x003BB);
359            SRC_MODE.write(0x00113);
360            AIN_ADDR.write(0x80800);
361            BIN_ADDR.write(0x87A38);
362            RUN.write(0x015);
363            wait(clk.negedge_event());
```

```
364             B_copy.range(3,0) = ACC.range(11,8);
365             B.write(B_copy);
366             IN_MODE.write(0x00357);
367             SRC_MODE.write(0x00201);
368             AIN_ADDR.write(0x17638);
369             BIN_ADDR.write(0x00002);
370             RUN.write(0x015);
371             Y_ADDR.write(0x0000);
372             wait(clk.negedge_event());
373           Y_copy = Y.read();
374           ACC.range(7,4) = Y_copy.range(3,0);
375           Accumulator.write(ACC);
376           B_copy.range(3,0) = ACC.range(15,12);
377           B.write(B_copy);
378             IN_MODE.write(0x003E3);
379             SRC_MODE.write(0x00342);
380             AIN_ADDR.write(0x08006);
381             BIN_ADDR.write(0x89208);
382             RUN.write(0x01B);
383             wait(clk.negedge_event());
384             IN_MODE.write(0x00333);
385             SRC_MODE.write(0x00121);
386             AIN_ADDR.write(0x80408);
387             BIN_ADDR.write(0x10802);
388             RUN.write(0x01D);
```

```
389             wait(clk.negedge_event());
390             ACC.range(11,8) = Y_copy.range(3,0);
391             Accumulator.write(ACC);
392         IN_MODE.write(0x0000B);
393             SRC_MODE.write(0x00002);
394             AIN_ADDR.write(0x00006);
395             BIN_ADDR.write(0x00028);
396             RUN.write(0x015);
397             wait(clk.negedge_event());
398         IN_MODE.write(0x00004);
399             SRC_MODE.write(0x00000);
400             AIN_ADDR.write(0x00000);
401             BIN_ADDR.write(0x00000);
402             RUN.write(0x001);
403             Y_ADDR.write(0x0001);
404             wait(clk.negedge_event());
405             ACC.range(15,12) = Y_copy.range(3,0);
406             IN_MODE.write(0x00000);
407             RUN.write(0x002);
408             wait(clk.negedge_event());
409             Y_ADDR.write(0x0000);
410             IN_MODE.write(0x00000);
411             RUN.write(0x000);
412             wait(clk.posedge_event());
413
```

```
414
415             A.write(101);
416             B.write(18);
417              wait(clk.negedge_event());
418           IN_MODE.write(0x3FC00);
419           SRC_MODE.write(0x3FC00);
420           AIN_ADDR.write(0x223300000);
421             BIN_ADDR.write(0x010100000);
422              wait(clk.negedge_event());
423           B_copy = B.read();
424           B_copy.range(3,0) = ACC.range(3,0);
425           B.write(B_copy);
426           IN_MODE.write(00333);
427             SRC_MODE.write(0x00001);
428             AIN_ADDR.write(0x0B0C00);
429             BIN_ADDR.write(0xD0F0E);
430           RUN.write(0x1E0);
431              wait(clk.negedge_event());
432           Y_copy = Y.read();
433           ACC.range(3,0) = Y_copy.range(3,0);
434           Accumulator.write(ACC);
435           B_copy.range(3,0) = ACC.range(7,4);
436           B.write(B_copy);
437             IN_MODE.write(0x003BB);
438             SRC_MODE.write(0x00113);
```

```
439             AIN_ADDR.write(0x80800);
440             BIN_ADDR.write(0x87A38);
441             RUN.write(0x015);
442              wait(clk.negedge_event());
443             B_copy.range(3,0) = ACC.range(11,8);
444             B.write(B_copy);
445             IN_MODE.write(0x00357);
446             SRC_MODE.write(0x00201);
447             AIN_ADDR.write(0x17638);
448             BIN_ADDR.write(0x00002);
449             RUN.write(0x015);
450             Y_ADDR.write(0x0000);
451              wait(clk.negedge_event());
452           Y_copy = Y.read();
453           ACC.range(7,4) = Y_copy.range(3,0);
454           Accumulator.write(ACC);
455           B_copy.range(3,0) = ACC.range(15,12);
456           B.write(B_copy);
457             IN_MODE.write(0x003E3);
458             SRC_MODE.write(0x00342);
459             AIN_ADDR.write(0x08006);
460             BIN_ADDR.write(0x89208);
461             RUN.write(0x01B);
462              wait(clk.negedge_event());
463             IN_MODE.write(0x00333);
```

```
464             SRC_MODE.write(0x00121);
465             AIN_ADDR.write(0x80408);
466             BIN_ADDR.write(0x10802);
467             RUN.write(0x01D);
468              wait(clk.negedge_event());
469             ACC.range(11,8) = Y_copy.range(3,0);
470              Accumulator.write(ACC);
471           IN_MODE.write(0x0000B);
472             SRC_MODE.write(0x00002);
473             AIN_ADDR.write(0x00006);
474             BIN_ADDR.write(0x00028);
475             RUN.write(0x015);
476              wait(clk.negedge_event());
477           IN_MODE.write(0x00004);
478             SRC_MODE.write(0x00000);
479             AIN_ADDR.write(0x00000);
480             BIN_ADDR.write(0x00000);
481             RUN.write(0x001);
482             Y_ADDR.write(0x0001);
483              wait(clk.negedge_event());
484             ACC.range(15,12) = Y_copy.range(3,0);
485              Accumulator.write(ACC);
486             IN_MODE.write(0x00000);
487             RUN.write(0x002);
488              wait(clk.negedge_event());
```

```
489             Y_ADDR.write(0x0000);
490             IN_MODE.write(0x00000);
491             RUN.write(0x000);
492              wait(clk.negedge_event());
493
494
495             A.write(1);
496             B.write(13);
497              wait(clk.negedge_event());
498           IN_MODE.write(0x3FC00);
499           SRC_MODE.write(0x3FC00);
500           AIN_ADDR.write(0x223300000);
501              BIN_ADDR.write(0x010100000);
502              wait(clk.negedge_event());
503           B_copy = B.read();
504           B_copy.range(3,0) = ACC.range(3,0);
505           B.write(B_copy);
506           IN_MODE.write(00333);
507              SRC_MODE.write(0x00001);
508              AIN_ADDR.write(0x0B0C00);
509              BIN_ADDR.write(0xD0F0E);
510           RUN.write(0x1E0);
511              wait(clk.negedge_event());
512           Y_copy = Y.read();
513           ACC.range(3,0) = Y_copy.range(3,0);
```

```
514         Accumulator.write(ACC);
515         B_copy.range(3,0) = ACC.range(7,4);
516         B.write(B_copy);
517           IN_MODE.write(0x003BB);
518           SRC_MODE.write(0x00113);
519           AIN_ADDR.write(0x80800);
520           BIN_ADDR.write(0x87A38);
521           RUN.write(0x015);
522           wait(clk.negedge_event());
523           B_copy.range(3,0) = ACC.range(11,8);
524           B.write(B_copy);
525           IN_MODE.write(0x00357);
526           SRC_MODE.write(0x00201);
527           AIN_ADDR.write(0x17638);
528           BIN_ADDR.write(0x00002);
529           RUN.write(0x015);
530           Y_ADDR.write(0x0000);
531           wait(clk.negedge_event());
532         Y_copy = Y.read();
533         ACC.range(7,4) = Y_copy.range(3,0);
534         Accumulator.write(ACC);
535         B_copy.range(3,0) = ACC.range(15,12);
536         B.write(B_copy);
537           IN_MODE.write(0x003E3);
538           SRC_MODE.write(0x00342);
```

```
539             AIN_ADDR.write(0x08006);
540             BIN_ADDR.write(0x89208);
541             RUN.write(0x01B);
542              wait(clk.negedge_event());
543             IN_MODE.write(0x00333);
544             SRC_MODE.write(0x00121);
545             AIN_ADDR.write(0x80408);
546             BIN_ADDR.write(0x10802);
547             RUN.write(0x01D);
548              wait(clk.negedge_event());
549             ACC.range(11,8) = Y_copy.range(3,0);
550              Accumulator.write(ACC);
551          IN_MODE.write(0x0000B);
552             SRC_MODE.write(0x00002);
553             AIN_ADDR.write(0x00006);
554             BIN_ADDR.write(0x00028);
555             RUN.write(0x015);
556              wait(clk.negedge_event());
557          IN_MODE.write(0x00004);
558             SRC_MODE.write(0x00000);
559             AIN_ADDR.write(0x00000);
560             BIN_ADDR.write(0x00000);
561             RUN.write(0x001);
562             Y_ADDR.write(0x0001);
563              wait(clk.negedge_event());
```

```
564             ACC.range(15,12) = Y_copy.range(3,0);
565             Accumulator.write(ACC);
566             IN_MODE.write(0x00000);
567             RUN.write(0x002);
568             wait(clk.negedge_event());
569             Y_ADDR.write(0x0000);
570             IN_MODE.write(0x00000);
571             RUN.write(0x000);
572             wait(clk.negedge_event());
573
574
575             A.write(110);
576             B.write(61);
577             wait(clk.negedge_event());
578           IN_MODE.write(0x3FC00);
579           SRC_MODE.write(0x3FC00);
580           AIN_ADDR.write(0x223300000);
581              BIN_ADDR.write(0x010100000);
582              wait(clk.negedge_event());
583           B_copy = B.read();
584           B_copy.range(3,0) = ACC.range(3,0);
585           B.write(B_copy);
586           IN_MODE.write(00333);
587              SRC_MODE.write(0x00001);
588              AIN_ADDR.write(0x0B0C00);
```

```
589             BIN_ADDR.write(0xD0F0E);
590           RUN.write(0x1E0);
591             wait(clk.negedge_event());
592           Y_copy = Y.read();
593           ACC.range(3,0) = Y_copy.range(3,0);
594           Accumulator.write(ACC);
595           B_copy.range(3,0) = ACC.range(7,4);
596           B.write(B_copy);
597             IN_MODE.write(0x003BB);
598             SRC_MODE.write(0x00113);
599             AIN_ADDR.write(0x80800);
600             BIN_ADDR.write(0x87A38);
601             RUN.write(0x015);
602             wait(clk.negedge_event());
603             B_copy.range(3,0) = ACC.range(11,8);
604             B.write(B_copy);
605             IN_MODE.write(0x00357);
606             SRC_MODE.write(0x00201);
607             AIN_ADDR.write(0x17638);
608             BIN_ADDR.write(0x00002);
609             RUN.write(0x015);
610             Y_ADDR.write(0x0000);
611             wait(clk.negedge_event());
612           Y_copy = Y.read();
613           ACC.range(7,4) = Y_copy.range(3,0);
```

```
614         Accumulator.write(ACC);
615         B_copy.range(3,0) = ACC.range(15,12);
616         B.write(B_copy);
617           IN_MODE.write(0x003E3);
618           SRC_MODE.write(0x00342);
619           AIN_ADDR.write(0x08006);
620           BIN_ADDR.write(0x89208);
621           RUN.write(0x01B);
622           wait(clk.negedge_event());
623           IN_MODE.write(0x00333);
624           SRC_MODE.write(0x00121);
625           AIN_ADDR.write(0x80408);
626           BIN_ADDR.write(0x10802);
627           RUN.write(0x01D);
628           wait(clk.negedge_event());
629           ACC.range(11,8) = Y_copy.range(3,0);
630           Accumulator.write(ACC);
631         IN_MODE.write(0x0000B);
632           SRC_MODE.write(0x00002);
633           AIN_ADDR.write(0x00006);
634           BIN_ADDR.write(0x00028);
635           RUN.write(0x015);
636           wait(clk.negedge_event());
637         IN_MODE.write(0x00004);
638           SRC_MODE.write(0x00000);
```

```
639             AIN_ADDR.write(0x00000);
640             BIN_ADDR.write(0x00000);
641             RUN.write(0x001);
642             Y_ADDR.write(0x0001);
643              wait(clk.negedge_event());
644             ACC.range(15,12) = Y_copy.range(3,0);
645              Accumulator.write(ACC);
646             IN_MODE.write(0x00000);
647             RUN.write(0x002);
648              wait(clk.negedge_event());
649             Y_ADDR.write(0x0000);
650             IN_MODE.write(0x00000);
651             RUN.write(0x000);
652              wait(clk.negedge_event());
653
654
655             A.write(237);
656             B.write(140);
657              wait(clk.negedge_event());
658          IN_MODE.write(0x3FC00);
659          SRC_MODE.write(0x3FC00);
660          AIN_ADDR.write(0x223300000);
661             BIN_ADDR.write(0x010100000);
662              wait(clk.negedge_event());
663          B_copy = B.read();
```

```
664         B_copy.range(3,0) = ACC.range(3,0);
665         B.write(B_copy);
666         IN_MODE.write(00333);
667            SRC_MODE.write(0x00001);
668            AIN_ADDR.write(0x0B0C00);
669            BIN_ADDR.write(0xD0F0E);
670         RUN.write(0x1E0);
671            wait(clk.negedge_event());
672         Y_copy = Y.read();
673         ACC.range(3,0) = Y_copy.range(3,0);
674         Accumulator.write(ACC);
675         B_copy.range(3,0) = ACC.range(7,4);
676         B.write(B_copy);
677            IN_MODE.write(0x003BB);
678            SRC_MODE.write(0x00113);
679            AIN_ADDR.write(0x80800);
680            BIN_ADDR.write(0x87A38);
681            RUN.write(0x015);
682            wait(clk.negedge_event());
683            B_copy.range(3,0) = ACC.range(11,8);
684            B.write(B_copy);
685            IN_MODE.write(0x00357);
686            SRC_MODE.write(0x00201);
687            AIN_ADDR.write(0x17638);
688            BIN_ADDR.write(0x00002);
```

```
689             RUN.write(0x015);
690             Y_ADDR.write(0x0000);
691             wait(clk.negedge_event());
692           Y_copy = Y.read();
693           ACC.range(7,4) = Y_copy.range(3,0);
694           Accumulator.write(ACC);
695           B_copy.range(3,0) = ACC.range(15,12);
696           B.write(B_copy);
697             IN_MODE.write(0x003E3);
698             SRC_MODE.write(0x00342);
699             AIN_ADDR.write(0x08006);
700             BIN_ADDR.write(0x89208);
701             RUN.write(0x01B);
702             wait(clk.negedge_event());
703             IN_MODE.write(0x00333);
704             SRC_MODE.write(0x00121);
705             AIN_ADDR.write(0x80408);
706             BIN_ADDR.write(0x10802);
707             RUN.write(0x01D);
708             wait(clk.negedge_event());
709             ACC.range(11,8) = Y_copy.range(3,0);
710             Accumulator.write(ACC);
711           IN_MODE.write(0x0000B);
712             SRC_MODE.write(0x00002);
713             AIN_ADDR.write(0x00006);
```

```
714             BIN_ADDR.write(0x00028);
715             RUN.write(0x015);
716             wait(clk.negedge_event());
717         IN_MODE.write(0x00004);
718             SRC_MODE.write(0x00000);
719             AIN_ADDR.write(0x00000);
720             BIN_ADDR.write(0x00000);
721             RUN.write(0x001);
722             Y_ADDR.write(0x0001);
723             wait(clk.negedge_event());
724             ACC.range(15,12) = Y_copy.range(3,0);
725             Accumulator.write(ACC);
726             IN_MODE.write(0x00000);
727             RUN.write(0x002);
728             wait(clk.negedge_event());
729             Y_ADDR.write(0x0000);
730             IN_MODE.write(0x00000);
731             RUN.write(0x000);
732             wait(clk.negedge_event());
733
734
735             A.write(249);
736             B.write(198);
737             wait(clk.negedge_event());
738         IN_MODE.write(0x3FC00);
```

```
739         SRC_MODE.write(0x3FC00);
740         AIN_ADDR.write(0x223300000);
741            BIN_ADDR.write(0x010100000);
742            wait(clk.negedge_event());
743         B_copy = B.read();
744         B_copy.range(3,0) = ACC.range(3,0);
745         B.write(B_copy);
746         IN_MODE.write(00333);
747            SRC_MODE.write(0x00001);
748            AIN_ADDR.write(0x0B0C00);
749            BIN_ADDR.write(0xD0F0E);
750         RUN.write(0x1E0);
751            wait(clk.negedge_event());
752         Y_copy = Y.read();
753         ACC.range(3,0) = Y_copy.range(3,0);
754         Accumulator.write(ACC);
755         B_copy.range(3,0) = ACC.range(7,4);
756         B.write(B_copy);
757            IN_MODE.write(0x003BB);
758            SRC_MODE.write(0x00113);
759            AIN_ADDR.write(0x80800);
760            BIN_ADDR.write(0x87A38);
761            RUN.write(0x015);
762            wait(clk.negedge_event());
763            B_copy.range(3,0) = ACC.range(11,8);
```

```
764             B.write(B_copy);
765             IN_MODE.write(0x00357);
766             SRC_MODE.write(0x00201);
767             AIN_ADDR.write(0x17638);
768             BIN_ADDR.write(0x00002);
769             RUN.write(0x015);
770             Y_ADDR.write(0x0000);
771              wait(clk.negedge_event());
772           Y_copy = Y.read();
773           ACC.range(7,4) = Y_copy.range(3,0);
774           Accumulator.write(ACC);
775           B_copy.range(3,0) = ACC.range(15,12);
776           B.write(B_copy);
777             IN_MODE.write(0x003E3);
778             SRC_MODE.write(0x00342);
779             AIN_ADDR.write(0x08006);
780             BIN_ADDR.write(0x89208);
781             RUN.write(0x01B);
782              wait(clk.negedge_event());
783             IN_MODE.write(0x00333);
784             SRC_MODE.write(0x00121);
785             AIN_ADDR.write(0x80408);
786             BIN_ADDR.write(0x10802);
787             RUN.write(0x01D);
788              wait(clk.posedge_event());
```

```
789             ACC.range(11,8) = Y_copy.range(3,0);
790             Accumulator.write(ACC);
791         IN_MODE.write(0x0000B);
792             SRC_MODE.write(0x00002);
793             AIN_ADDR.write(0x00006);
794             BIN_ADDR.write(0x00028);
795             RUN.write(0x015);
796             wait(clk.negedge_event());
797         IN_MODE.write(0x00004);
798             SRC_MODE.write(0x00000);
799             AIN_ADDR.write(0x00000);
800             BIN_ADDR.write(0x00000);
801             RUN.write(0x001);
802             Y_ADDR.write(0x0001);
803             wait(clk.negedge_event());
804             ACC.range(15,12) = Y_copy.range(3,0);
805             Accumulator.write(ACC);
806             IN_MODE.write(0x00000);
807             RUN.write(0x002);
808             wait(clk.posedge_event());
809             Y_ADDR.write(0x0000);
810             IN_MODE.write(0x00000);
811             RUN.write(0x000);
812             wait(clk.negedge_event());
813
```

```
814
815             A.write(197);
816             B.write(170);
817              wait(clk.negedge_event());
818           IN_MODE.write(0x3FC00);
819           SRC_MODE.write(0x3FC00);
820           AIN_ADDR.write(0x223300000);
821             BIN_ADDR.write(0x010100000);
822              wait(clk.negedge_event());
823           B_copy = B.read();
824           B_copy.range(3,0) = ACC.range(3,0);
825           B.write(B_copy);
826           IN_MODE.write(00333);
827             SRC_MODE.write(0x00001);
828             AIN_ADDR.write(0x0B0C00);
829             BIN_ADDR.write(0xD0F0E);
830           RUN.write(0x1E0);
831              wait(clk.negedge_event());
832           Y_copy = Y.read();
833           ACC.range(3,0) = Y_copy.range(3,0);
834           Accumulator.write(ACC);
835           B_copy.range(3,0) = ACC.range(7,4);
836           B.write(B_copy);
837             IN_MODE.write(0x003BB);
838             SRC_MODE.write(0x00113);
```

```
839             AIN_ADDR.write(0x80800);
840             BIN_ADDR.write(0x87A38);
841             RUN.write(0x015);
842              wait(clk.negedge_event());
843             B_copy.range(3,0) = ACC.range(11,8);
844             B.write(B_copy);
845             IN_MODE.write(0x00357);
846             SRC_MODE.write(0x00201);
847             AIN_ADDR.write(0x17638);
848             BIN_ADDR.write(0x00002);
849             RUN.write(0x015);
850             Y_ADDR.write(0x0000);
851              wait(clk.negedge_event());
852           Y_copy = Y.read();
853           ACC.range(7,4) = Y_copy.range(3,0);
854           Accumulator.write(ACC);
855           B_copy.range(3,0) = ACC.range(15,12);
856           B.write(B_copy);
857             IN_MODE.write(0x003E3);
858             SRC_MODE.write(0x00342);
859             AIN_ADDR.write(0x08006);
860             BIN_ADDR.write(0x89208);
861             RUN.write(0x01B);
862              wait(clk.negedge_event());
863             IN_MODE.write(0x00333);
```

```
864             SRC_MODE.write(0x00121);
865             AIN_ADDR.write(0x80408);
866             BIN_ADDR.write(0x10802);
867             RUN.write(0x01D);
868              wait(clk.negedge_event());
869             ACC.range(11,8) = Y_copy.range(3,0);
870              Accumulator.write(ACC);
871         IN_MODE.write(0x0000B);
872             SRC_MODE.write(0x00002);
873             AIN_ADDR.write(0x00006);
874             BIN_ADDR.write(0x00028);
875             RUN.write(0x015);
876              wait(clk.negedge_event());
877         IN_MODE.write(0x00004);
878             SRC_MODE.write(0x00000);
879             AIN_ADDR.write(0x00000);
880             BIN_ADDR.write(0x00000);
881             RUN.write(0x001);
882             Y_ADDR.write(0x0001);
883              wait(clk.negedge_event());
884             ACC.range(15,12) = Y_copy.range(3,0);
885              Accumulator.write(ACC);
886             IN_MODE.write(0x00000);
887             RUN.write(0x002);
888              wait(clk.negedge_event());
```

```
889             Y_ADDR.write(0x0000);
890             IN_MODE.write(0x00000);
891             RUN.write(0x000);
892              wait(clk.negedge_event());
893
894
895             A.write(229);
896             B.write(119);
897              //wait(60,SC_NS);
898              wait(clk.negedge_event());
899           IN_MODE.write(0x3FC00);
900           SRC_MODE.write(0x3FC00);
901           AIN_ADDR.write(0x223300000);
902              BIN_ADDR.write(0x010100000);
903              wait(clk.negedge_event());
904           B_copy = B.read();
905           B_copy.range(3,0) = ACC.range(3,0);
906           B.write(B_copy);
907           IN_MODE.write(00333);
908              SRC_MODE.write(0x00001);
909              AIN_ADDR.write(0x0B0C00);
910              BIN_ADDR.write(0xD0F0E);
911           RUN.write(0x1E0);
912              wait(clk.negedge_event());
913           Y_copy = Y.read();
```

```
914         ACC.range(3,0) = Y_copy.range(3,0);
915         Accumulator.write(ACC);
916         B_copy.range(3,0) = ACC.range(7,4);
917         B.write(B_copy);
918            IN_MODE.write(0x003BB);
919            SRC_MODE.write(0x00113);
920            AIN_ADDR.write(0x80800);
921            BIN_ADDR.write(0x87A38);
922            RUN.write(0x015);
923            wait(clk.negedge_event());
924            B_copy.range(3,0) = ACC.range(11,8);
925            B.write(B_copy);
926            IN_MODE.write(0x00357);
927            SRC_MODE.write(0x00201);
928            AIN_ADDR.write(0x17638);
929            BIN_ADDR.write(0x00002);
930            RUN.write(0x015);
931            Y_ADDR.write(0x0000);
932            wait(clk.negedge_event());
933         Y_copy = Y.read();
934         ACC.range(7,4) = Y_copy.range(3,0);
935         Accumulator.write(ACC);
936         B_copy.range(3,0) = ACC.range(15,12);
937         B.write(B_copy);
938            IN_MODE.write(0x003E3);
```

```
939             SRC_MODE.write(0x00342);
940             AIN_ADDR.write(0x08006);
941             BIN_ADDR.write(0x89208);
942             RUN.write(0x01B);
943              wait(clk.negedge_event());
944             IN_MODE.write(0x00333);
945             SRC_MODE.write(0x00121);
946             AIN_ADDR.write(0x80408);
947             BIN_ADDR.write(0x10802);
948             RUN.write(0x01D);
949              wait(clk.negedge_event());
950             ACC.range(11,8) = Y_copy.range(3,0);
951              Accumulator.write(ACC);
952          IN_MODE.write(0x0000B);
953             SRC_MODE.write(0x00002);
954             AIN_ADDR.write(0x00006);
955             BIN_ADDR.write(0x00028);
956             RUN.write(0x015);
957              wait(clk.negedge_event());
958          IN_MODE.write(0x00004);
959             SRC_MODE.write(0x00000);
960             AIN_ADDR.write(0x00000);
961             BIN_ADDR.write(0x00000);
962             RUN.write(0x001);
963             Y_ADDR.write(0x0001);
```

```
964             wait(clk.negedge_event());
965             ACC.range(15,12) = Y_copy.range(3,0);
966             Accumulator.write(ACC);
967             IN_MODE.write(0x00000);
968             RUN.write(0x002);
969             wait(clk.negedge_event());
970             Y_ADDR.write(0x0000);
971             IN_MODE.write(0x00000);
972             RUN.write(0x000);
973             wait(clk.negedge_event());
974
975
976         A.write(18);
977         B.write(143);
978             wait(clk.negedge_event());
979         IN_MODE.write(0x3FC00);
980         SRC_MODE.write(0x3FC00);
981         AIN_ADDR.write(0x223300000);
982             BIN_ADDR.write(0x010100000);
983             wait(clk.negedge_event());
984         B_copy = B.read();
985         B_copy.range(3,0) = ACC.range(3,0);
986         B.write(B_copy);
987         IN_MODE.write(00333);
988             SRC_MODE.write(0x00001);
```

```
989             AIN_ADDR.write(0x0B0C00);
990             BIN_ADDR.write(0xD0F0E);
991           RUN.write(0x1E0);
992             wait(clk.negedge_event());
993           Y_copy = Y.read();
994           ACC.range(3,0) = Y_copy.range(3,0);
995           Accumulator.write(ACC);
996           B_copy.range(3,0) = ACC.range(7,4);
997           B.write(B_copy);
998             IN_MODE.write(0x003BB);
999             SRC_MODE.write(0x00113);
1000            AIN_ADDR.write(0x80800);
1001            BIN_ADDR.write(0x87A38);
1002            RUN.write(0x015);
1003            wait(clk.negedge_event());
1004            B_copy.range(3,0) = ACC.range(11,8);
1005            B.write(B_copy);
1006            IN_MODE.write(0x00357);
1007            SRC_MODE.write(0x00201);
1008            AIN_ADDR.write(0x17638);
1009            BIN_ADDR.write(0x00002);
1010            RUN.write(0x015);
1011            Y_ADDR.write(0x0000);
1012            wait(clk.negedge_event());
1013          Y_copy = Y.read();
```

```
1014        ACC.range(7,4) = Y_copy.range(3,0);
1015        Accumulator.write(ACC);
1016        B_copy.range(3,0) = ACC.range(15,12);
1017        B.write(B_copy);
1018           IN_MODE.write(0x003E3);
1019           SRC_MODE.write(0x00342);
1020           AIN_ADDR.write(0x08006);
1021           BIN_ADDR.write(0x89208);
1022           RUN.write(0x01B);
1023           wait(clk.negedge_event());
1024           IN_MODE.write(0x00333);
1025           SRC_MODE.write(0x00121);
1026           AIN_ADDR.write(0x80408);
1027           BIN_ADDR.write(0x10802);
1028           RUN.write(0x01D);
1029           wait(clk.negedge_event());
1030           ACC.range(11,8) = Y_copy.range(3,0);
1031           Accumulator.write(ACC);
1032        IN_MODE.write(0x0000B);
1033           SRC_MODE.write(0x00002);
1034           AIN_ADDR.write(0x00006);
1035           BIN_ADDR.write(0x00028);
1036           RUN.write(0x015);
1037           wait(clk.negedge_event());
1038        IN_MODE.write(0x00004);
```

```
1039            SRC_MODE.write(0x00000);
1040            AIN_ADDR.write(0x00000);
1041            BIN_ADDR.write(0x00000);
1042            RUN.write(0x001);
1043            Y_ADDR.write(0x0001);
1044             wait(clk.negedge_event());
1045            ACC.range(15,12) = Y_copy.range(3,0);
1046             Accumulator.write(ACC);
1047            IN_MODE.write(0x00000);
1048            RUN.write(0x002);
1049             wait(clk.negedge_event());
1050            Y_ADDR.write(0x0000);
1051            IN_MODE.write(0x00000);
1052            RUN.write(0x000);
1053             wait(clk.negedge_event());
1054
1055          A.write(242);
1056          B.write(206);
1057             wait(clk.negedge_event());
1058          IN_MODE.write(0x3FC00);
1059          SRC_MODE.write(0x3FC00);
1060          AIN_ADDR.write(0x223300000);
1061             BIN_ADDR.write(0x010100000);
1062             wait(clk.negedge_event());
1063          B_copy = B.read();
```

```
1064        B_copy.range(3,0) = ACC.range(3,0);
1065        B.write(B_copy);
1066        IN_MODE.write(00333);
1067           SRC_MODE.write(0x00001);
1068           AIN_ADDR.write(0x0B0C00);
1069           BIN_ADDR.write(0xD0F0E);
1070        RUN.write(0x1E0);
1071           wait(clk.negedge_event());
1072        Y_copy = Y.read();
1073        ACC.range(3,0) = Y_copy.range(3,0);
1074        Accumulator.write(ACC);
1075        B_copy.range(3,0) = ACC.range(7,4);
1076        B.write(B_copy);
1077           IN_MODE.write(0x003BB);
1078           SRC_MODE.write(0x00113);
1079           AIN_ADDR.write(0x80800);
1080           BIN_ADDR.write(0x87A38);
1081           RUN.write(0x015);
1082           wait(clk.negedge_event());
1083           B_copy.range(3,0) = ACC.range(11,8);
1084           B.write(B_copy);
1085           IN_MODE.write(0x00357);
1086           SRC_MODE.write(0x00201);
1087           AIN_ADDR.write(0x17638);
1088           BIN_ADDR.write(0x00002);
```

```
1089            RUN.write(0x015);
1090            Y_ADDR.write(0x0000);
1091            wait(clk.negedge_event());
1092          Y_copy = Y.read();
1093          ACC.range(7,4) = Y_copy.range(3,0);
1094          Accumulator.write(ACC);
1095          B_copy.range(3,0) = ACC.range(15,12);
1096          B.write(B_copy);
1097            IN_MODE.write(0x003E3);
1098            SRC_MODE.write(0x00342);
1099            AIN_ADDR.write(0x08006);
1100            BIN_ADDR.write(0x89208);
1101            RUN.write(0x01B);
1102            wait(clk.negedge_event());
1103            IN_MODE.write(0x00333);
1104            SRC_MODE.write(0x00121);
1105            AIN_ADDR.write(0x80408);
1106            BIN_ADDR.write(0x10802);
1107            RUN.write(0x01D);
1108            wait(clk.negedge_event());
1109            ACC.range(11,8) = Y_copy.range(3,0);
1110            Accumulator.write(ACC);
1111          IN_MODE.write(0x0000B);
1112            SRC_MODE.write(0x00002);
1113            AIN_ADDR.write(0x00006);
```

```
1114            BIN_ADDR.write(0x00028);
1115            RUN.write(0x015);
1116             wait(clk.negedge_event());
1117          IN_MODE.write(0x00004);
1118            SRC_MODE.write(0x00000);
1119            AIN_ADDR.write(0x00000);
1120            BIN_ADDR.write(0x00000);
1121            RUN.write(0x001);
1122            Y_ADDR.write(0x0001);
1123             wait(clk.negedge_event());
1124            ACC.range(15,12) = Y_copy.range(3,0);
1125             Accumulator.write(ACC);
1126            IN_MODE.write(0x00000);
1127            RUN.write(0x002);
1128             wait(clk.negedge_event());
1129            Y_ADDR.write(0x0000);
1130            IN_MODE.write(0x00000);
1131            RUN.write(0x000);
1132             wait(clk.negedge_event());
1133
1134
1135          A.write(232);
1136          B.write(197);
1137            wait(clk.negedge_event());
1138          IN_MODE.write(0x3FC00);
```

```
1139        SRC_MODE.write(0x3FC00);
1140        AIN_ADDR.write(0x223300000);
1141           BIN_ADDR.write(0x010100000);
1142           wait(clk.negedge_event());
1143        B_copy = B.read();
1144        B_copy.range(3,0) = ACC.range(3,0);
1145        B.write(B_copy);
1146        IN_MODE.write(00333);
1147           SRC_MODE.write(0x00001);
1148           AIN_ADDR.write(0x0B0C00);
1149           BIN_ADDR.write(0xD0F0E);
1150        RUN.write(0x1E0);
1151           wait(clk.negedge_event());
1152        Y_copy = Y.read();
1153        ACC.range(3,0) = Y_copy.range(3,0);
1154        Accumulator.write(ACC);
1155        B_copy.range(3,0) = ACC.range(7,4);
1156        B.write(B_copy);
1157           IN_MODE.write(0x003BB);
1158           SRC_MODE.write(0x00113);
1159           AIN_ADDR.write(0x80800);
1160           BIN_ADDR.write(0x87A38);
1161           RUN.write(0x015);
1162           wait(clk.negedge_event());
1163           B_copy.range(3,0) = ACC.range(11,8);
```

```
1164            B.write(B_copy);
1165            IN_MODE.write(0x00357);
1166            SRC_MODE.write(0x00201);
1167            AIN_ADDR.write(0x17638);
1168            BIN_ADDR.write(0x00002);
1169            RUN.write(0x015);
1170            Y_ADDR.write(0x0000);
1171             wait(clk.negedge_event());
1172          Y_copy = Y.read();
1173          ACC.range(7,4) = Y_copy.range(3,0);
1174          B_copy.range(3,0) = ACC.range(15,12);
1175          B.write(B_copy);
1176            IN_MODE.write(0x003E3);
1177            SRC_MODE.write(0x00342);
1178            AIN_ADDR.write(0x08006);
1179            BIN_ADDR.write(0x89208);
1180            RUN.write(0x01B);
1181             wait(clk.negedge_event());
1182            IN_MODE.write(0x00333);
1183            SRC_MODE.write(0x00121);
1184            AIN_ADDR.write(0x80408);
1185            BIN_ADDR.write(0x10802);
1186            RUN.write(0x01D);
1187             wait(clk.negedge_event());
1188            ACC.range(11,8) = Y_copy.range(3,0);
```

```
1189            Accumulator.write(ACC);
1190        IN_MODE.write(0x0000B);
1191            SRC_MODE.write(0x00002);
1192            AIN_ADDR.write(0x00006);
1193            BIN_ADDR.write(0x00028);
1194            RUN.write(0x015);
1195            wait(clk.negedge_event());
1196        IN_MODE.write(0x00004);
1197            SRC_MODE.write(0x00000);
1198            AIN_ADDR.write(0x00000);
1199            BIN_ADDR.write(0x00000);
1200            RUN.write(0x001);
1201            Y_ADDR.write(0x0001);
1202            wait(clk.negedge_event());
1203            ACC.range(15,12) = Y_copy.range(3,0);
1204            Accumulator.write(ACC);
1205            IN_MODE.write(0x00000);
1206            RUN.write(0x002);
1207            wait(clk.negedge_event());
1208            Y_ADDR.write(0x0000);
1209            IN_MODE.write(0x00000);
1210            RUN.write(0x000);
1211            wait(clk.negedge_event());
1212
1213
```

```
1214        A.write(92);
1215        B.write(189);
1216            wait(clk.negedge_event());
1217        IN_MODE.write(0x3FC00);
1218        SRC_MODE.write(0x3FC00);
1219        AIN_ADDR.write(0x223300000);
1220            BIN_ADDR.write(0x010100000);
1221            wait(clk.negedge_event());
1222        B_copy = B.read();
1223        B_copy.range(3,0) = ACC.range(3,0);
1224        B.write(B_copy);
1225        IN_MODE.write(00333);
1226            SRC_MODE.write(0x00001);
1227            AIN_ADDR.write(0x0B0C00);
1228            BIN_ADDR.write(0xD0F0E);
1229        RUN.write(0x1E0);
1230            wait(clk.negedge_event());
1231        Y_copy = Y.read();
1232        ACC.range(3,0) = Y_copy.range(3,0);
1233        Accumulator.write(ACC);
1234        B_copy.range(3,0) = ACC.range(7,4);
1235        B.write(B_copy);
1236            IN_MODE.write(0x003BB);
1237            SRC_MODE.write(0x00113);
1238            AIN_ADDR.write(0x80800);
```

```
1239            BIN_ADDR.write(0x87A38);
1240            RUN.write(0x015);
1241             wait(clk.negedge_event());
1242            B_copy.range(3,0) = ACC.range(11,8);
1243            B.write(B_copy);
1244            IN_MODE.write(0x00357);
1245            SRC_MODE.write(0x00201);
1246            AIN_ADDR.write(0x17638);
1247            BIN_ADDR.write(0x00002);
1248            RUN.write(0x015);
1249            Y_ADDR.write(0x0000);
1250             wait(clk.negedge_event());
1251          Y_copy = Y.read();
1252          ACC.range(7,4) = Y_copy.range(3,0);
1253          Accumulator.write(ACC);
1254          B_copy.range(3,0) = ACC.range(15,12);
1255          B.write(B_copy);
1256             IN_MODE.write(0x003E3);
1257             SRC_MODE.write(0x00342);
1258             AIN_ADDR.write(0x08006);
1259             BIN_ADDR.write(0x89208);
1260             RUN.write(0x01B);
1261              wait(clk.negedge_event());
1262             IN_MODE.write(0x00333);
1263             SRC_MODE.write(0x00121);
```

```
1264            AIN_ADDR.write(0x80408);
1265            BIN_ADDR.write(0x10802);
1266            RUN.write(0x01D);
1267             wait(clk.negedge_event());
1268            ACC.range(11,8) = Y_copy.range(3,0);
1269             Accumulator.write(ACC);
1270          IN_MODE.write(0x0000B);
1271            SRC_MODE.write(0x00002);
1272            AIN_ADDR.write(0x00006);
1273            BIN_ADDR.write(0x00028);
1274            RUN.write(0x015);
1275             wait(clk.negedge_event());
1276          IN_MODE.write(0x00004);
1277            SRC_MODE.write(0x00000);
1278            AIN_ADDR.write(0x00000);
1279            BIN_ADDR.write(0x00000);
1280            RUN.write(0x001);
1281            Y_ADDR.write(0x0001);
1282             wait(clk.negedge_event());
1283            ACC.range(15,12) = Y_copy.range(3,0);
1284             Accumulator.write(ACC);
1285            IN_MODE.write(0x00000);
1286            RUN.write(0x002);
1287             wait(clk.negedge_event());
1288            Y_ADDR.write(0x0000);
```

```
1289            IN_MODE.write(0x00000);
1290            RUN.write(0x000);
1291             wait(clk.negedge_event());
1292
1293
1294          A.write(45);
1295          B.write(101);
1296             wait(clk.negedge_event());
1297          IN_MODE.write(0x3FC00);
1298          SRC_MODE.write(0x3FC00);
1299          AIN_ADDR.write(0x223300000);
1300             BIN_ADDR.write(0x010100000);
1301             wait(clk.negedge_event());
1302          B_copy = B.read();
1303          B_copy.range(3,0) = ACC.range(3,0);
1304          B.write(B_copy);
1305          IN_MODE.write(00333);
1306             SRC_MODE.write(0x00001);
1307             AIN_ADDR.write(0x0B0C00);
1308             BIN_ADDR.write(0xD0F0E);
1309          RUN.write(0x1E0);
1310             wait(clk.negedge_event());
1311          Y_copy = Y.read();
1312          ACC.range(3,0) = Y_copy.range(3,0);
1313          Accumulator.write(ACC);
```

```
1314        B_copy.range(3,0) = ACC.range(7,4);
1315        B.write(B_copy);
1316          IN_MODE.write(0x003BB);
1317          SRC_MODE.write(0x00113);
1318          AIN_ADDR.write(0x80800);
1319          BIN_ADDR.write(0x87A38);
1320          RUN.write(0x015);
1321           wait(clk.negedge_event());
1322          B_copy.range(3,0) = ACC.range(11,8);
1323          B.write(B_copy);
1324          IN_MODE.write(0x00357);
1325          SRC_MODE.write(0x00201);
1326          AIN_ADDR.write(0x17638);
1327          BIN_ADDR.write(0x00002);
1328          RUN.write(0x015);
1329          Y_ADDR.write(0x0000);
1330           wait(clk.negedge_event());
1331        Y_copy = Y.read();
1332        ACC.range(7,4) = Y_copy.range(3,0);
1333        Accumulator.write(ACC);
1334        B_copy.range(3,0) = ACC.range(15,12);
1335        B.write(B_copy);
1336          IN_MODE.write(0x003E3);
1337          SRC_MODE.write(0x00342);
1338          AIN_ADDR.write(0x08006);
```

```
1339            BIN_ADDR.write(0x89208);
1340            RUN.write(0x01B);
1341             wait(clk.negedge_event());
1342            IN_MODE.write(0x00333);
1343            SRC_MODE.write(0x00121);
1344            AIN_ADDR.write(0x80408);
1345            BIN_ADDR.write(0x10802);
1346            RUN.write(0x01D);
1347             wait(clk.negedge_event());
1348            ACC.range(11,8) = Y_copy.range(3,0);
1349             Accumulator.write(ACC);
1350          IN_MODE.write(0x0000B);
1351            SRC_MODE.write(0x00002);
1352            AIN_ADDR.write(0x00006);
1353            BIN_ADDR.write(0x00028);
1354            RUN.write(0x015);
1355             wait(clk.negedge_event());
1356          IN_MODE.write(0x00004);
1357            SRC_MODE.write(0x00000);
1358            AIN_ADDR.write(0x00000);
1359            BIN_ADDR.write(0x00000);
1360            RUN.write(0x001);
1361            Y_ADDR.write(0x0001);
1362             wait(clk.negedge_event());
1363            ACC.range(15,12) = Y_copy.range(3,0);
```

```
1364            Accumulator.write(ACC);
1365            IN_MODE.write(0x00000);
1366            RUN.write(0x002);
1367            wait(clk.negedge_event());
1368            Y_ADDR.write(0x0000);
1369            IN_MODE.write(0x00000);
1370            RUN.write(0x000);
1371            wait(clk.posedge_event());
1372
1373
1374        A.write(99);
1375        B.write(10);
1376            wait(clk.negedge_event());
1377        IN_MODE.write(0x3FC00);
1378        SRC_MODE.write(0x3FC00);
1379        AIN_ADDR.write(0x223300000);
1380            BIN_ADDR.write(0x010100000);
1381            wait(clk.negedge_event());
1382        B_copy = B.read();
1383        B_copy.range(3,0) = ACC.range(3,0);
1384        B.write(B_copy);
1385        IN_MODE.write(00333);
1386            SRC_MODE.write(0x00001);
1387            AIN_ADDR.write(0x0B0C00);
1388            BIN_ADDR.write(0xD0F0E);
```

```
1389        RUN.write(0x1E0);
1390            wait(clk.negedge_event());
1391        Y_copy = Y.read();
1392        ACC.range(3,0) = Y_copy.range(3,0);
1393        Accumulator.write(ACC);
1394        B_copy.range(3,0) = ACC.range(7,4);
1395        B.write(B_copy);
1396            IN_MODE.write(0x003BB);
1397            SRC_MODE.write(0x00113);
1398            AIN_ADDR.write(0x80800);
1399            BIN_ADDR.write(0x87A38);
1400            RUN.write(0x015);
1401            wait(clk.negedge_event());
1402            B_copy.range(3,0) = ACC.range(11,8);
1403            B.write(B_copy);
1404            IN_MODE.write(0x00357);
1405            SRC_MODE.write(0x00201);
1406            AIN_ADDR.write(0x17638);
1407            BIN_ADDR.write(0x00002);
1408            RUN.write(0x015);
1409            Y_ADDR.write(0x0000);
1410            wait(clk.negedge_event());
1411        Y_copy = Y.read();
1412        ACC.range(7,4) = Y_copy.range(3,0);
1413        Accumulator.write(ACC);
```

```
1414        B_copy.range(3,0) = ACC.range(15,12);
1415        B.write(B_copy);
1416          IN_MODE.write(0x003E3);
1417          SRC_MODE.write(0x00342);
1418          AIN_ADDR.write(0x08006);
1419          BIN_ADDR.write(0x89208);
1420          RUN.write(0x01B);
1421           wait(clk.negedge_event());
1422          IN_MODE.write(0x00333);
1423          SRC_MODE.write(0x00121);
1424          AIN_ADDR.write(0x80408);
1425          BIN_ADDR.write(0x10802);
1426          RUN.write(0x01D);
1427           wait(clk.negedge_event());
1428          ACC.range(11,8) = Y_copy.range(3,0);
1429           Accumulator.write(ACC);
1430        IN_MODE.write(0x0000B);
1431          SRC_MODE.write(0x00002);
1432          AIN_ADDR.write(0x00006);
1433          BIN_ADDR.write(0x00028);
1434          RUN.write(0x015);
1435           wait(clk.negedge_event());
1436        IN_MODE.write(0x00004);
1437          SRC_MODE.write(0x00000);
1438          AIN_ADDR.write(0x00000);
```

```
1439            BIN_ADDR.write(0x00000);
1440            RUN.write(0x001);
1441            Y_ADDR.write(0x0001);
1442             wait(clk.negedge_event());
1443            ACC.range(15,12) = Y_copy.range(3,0);
1444             Accumulator.write(ACC);
1445            IN_MODE.write(0x00000);
1446            RUN.write(0x002);
1447             wait(clk.negedge_event());
1448            Y_ADDR.write(0x0000);
1449            IN_MODE.write(0x00000);
1450            RUN.write(0x000);
1451             wait(clk.negedge_event());
1452
1453    //      cout<< "at time: " << sc_time_stamp()<< "---------->
          "<< "FUNC in stimulus= "<< FUNC.read() <<endl;
1454    // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
          FUNC_ADDR in stimulus= "<< FUNC_ADDR.read() <<endl;
1455            cout<< "at time: " << sc_time_stamp()<< "----------> "
                << "Y in stimulus= "<< Y.read() <<endl;
1456  }
1457
1458
1459
1460
```

```
1461    //cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
           FUNC in stimulus= "<< FUNC.read() <<endl;
1462    //cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
           FUNC_ADDR in stimulus= "<< FUNC_ADDR.read() <<endl;
1463        void clock_gen1()
1464        {
1465
1466          bool stat = true;
1467          while(true)
1468          {
1469
1470            clk->write(stat);
1471            stat = !stat;
1472            wait(3,SC_NS);
1473
1474          }
1475
1476        }
1477
1478
1479 };
1480
1481 #endif
```

II.4 pPIM cluster main

```
1  #define SC_INCLUDE_FX
2  #include "systemc.h"
3  #include "PIMC_stimulus.h"
4  #include "PIM_Cluster.h"
5
6  int sc_main(int argc, char* argv[])
7  {
8    //M = 8; N = 9
9    int const M = 8;
10   int const N = 9;
11   sc_signal<sc_bv<M> >A;
12   sc_signal<sc_bv<M> >B;
13   sc_signal<sc_bv<1<<M> > FUNC;
14   constexpr static int size = ceil(log2(M));
15   sc_signal<sc_uint<size> > FUNC_ADDR;
16   sc_signal<sc_bv<N> > LOAD_CORE;
17
18   constexpr static int size_addr = ceil(log2(2*(N-1)));
19   sc_signal<sc_bv<N*size_addr> > AIN_ADDR;
20   sc_signal<sc_bv<N*size_addr> > BIN_ADDR;
21
22   sc_signal<sc_bv<2*N> > SRC_MODE;
23   sc_signal<sc_bv<2*N> > IN_MODE;
```

```
24
25    sc_signal<sc_bv<N> > RUN;
26    //sc_clock clk("clock", 6, SC_NS);
27    sc_signal<bool> clk;
28    sc_signal<bool> reset;
29    constexpr static int size_y_addr = ceil(log2(N));
30    sc_signal<sc_bv<size_y_addr> > Y_ADDR;
31    sc_signal<sc_bv<M> > Y;
32
33
34    //Extra ports created for waveform tracing
35    sc_signal<sc_bv<8>> add_port;
36    sc_signal<sc_bv<8>> mult_port;
37    sc_signal<bool> clock_port;
38    sc_signal<sc_bv<16>> ACC;
39
40
41    PIM_Cluster<8,9> clusters("clusters");
42    clusters.A(A);
43    clusters.B(B);
44    clusters.FUNC(FUNC);
45    clusters.FUNC_ADDR(FUNC_ADDR);
46    clusters.LOAD_CORE(LOAD_CORE);
47    clusters.AIN_ADDR(AIN_ADDR);
48    clusters.BIN_ADDR(BIN_ADDR);
```

```
49    clusters.SRC_MODE(SRC_MODE);
50    clusters.IN_MODE(IN_MODE);
51    clusters.RUN(RUN);
52    clusters.reset(reset);
53    clusters.clk(clk);
54    clusters.Y_ADDR(Y_ADDR);
55    clusters.Y(Y);
56
57    cout<< "at time: " << sc_time_stamp()<< "----------> "<< "Y
          in main= "<< Y.read() <<endl;
58
59    PIMC_stimulus<8,9> cluster_stim("cluster_stim");
60    cluster_stim.A(A);
61    cluster_stim.B(B);
62    cluster_stim.FUNC(FUNC);
63    cluster_stim.FUNC_ADDR(FUNC_ADDR);
64    cluster_stim.LOAD_CORE(LOAD_CORE);
65    cluster_stim.AIN_ADDR(AIN_ADDR);
66    cluster_stim.BIN_ADDR(BIN_ADDR);
67    cluster_stim.SRC_MODE(SRC_MODE);
68    cluster_stim.IN_MODE(IN_MODE);
69    cluster_stim.RUN(RUN);
70    cluster_stim.reset(reset);
71    cluster_stim.Y_ADDR(Y_ADDR);
72    cluster_stim.Y(Y);
```

```
73    cluster_stim.add_port(add_port);
74    cluster_stim.mult_port(mult_port);
75    cluster_stim.Accumulator(ACC);
76    //cluster_stim.clock_port(clock_port);
77    cluster_stim.clk(clk);
78
79
80
81    sc_trace_file *fp;
82    fp = sc_create_vcd_trace_file("PIM_wave");
83    if(!fp) cout <<"There was an error!!" << endl;
84      //sc_trace(fp,clk,"clk");
85      sc_trace(fp,A,"A");
86      sc_trace(fp,B,"B");
87      sc_trace(fp,FUNC,"FUNC");
88      sc_trace(fp,FUNC_ADDR,"FUNC_ADDR");
89      sc_trace(fp,LOAD_CORE,"LOAD_CORE");
90      sc_trace(fp,AIN_ADDR,"AIN_ADDR");
91      sc_trace(fp,BIN_ADDR,"BIN_ADDR");
92      sc_trace(fp,SRC_MODE,"SRC_MODE");
93      sc_trace(fp,IN_MODE,"IN_MODE");
94      sc_trace(fp,RUN,"RUN");
95      sc_trace(fp,Y_ADDR,"Y_ADDR");
96      sc_trace(fp,reset,"reset");
97
```

```
98      // sc_trace(fp,clusters.pimcore0.A,"PIMCORE_0.A");
99      // sc_trace(fp,clusters.pimcore0.B,"PIMCORE_0.B");
100     // sc_trace(fp,clusters.pimcore0.ASel,"PIMCORE_0.ASel");
101     // sc_trace(fp,clusters.pimcore0.BSel,"PIMCORE_0.BSel");
102     // sc_trace(fp,clusters.pimcore0.FUNC_IN,"PIMCORE_0.FUNC_IN
            ");
103     // sc_trace(fp,clusters.pimcore0.FUNC_ADDR,"PIMCORE_0.
            FUNC_ADDR");
104     // sc_trace(fp,clusters.pimcore0.IN_MODE,"PIMCORE_0.IN_MODE
            ");
105     // sc_trace(fp,clusters.pimcore0.LOAD,"PIMCORE_0.LOAD");
106     // sc_trace(fp,clusters.pimcore0.RUN,"PIMCORE_0.RUN");
107      //sc_trace(fp,clusters.pimcore0.Y,"PIMCORE_0.Y");
108     // sc_trace(fp,clusters.pimcore0.clk,"PIMCORE_0.clk");
109
110     // sc_trace(fp,clusters.AMux0.A,"AMux_0.A");
111     // sc_trace(fp,clusters.AMux0.B,"AMux_0.B");
112     // sc_trace(fp,clusters.AMux0.MODE,"AMux_0.MODE");
113     // sc_trace(fp,clusters.AMux0.CORES,"AMux_0.CORES");
114     // sc_trace(fp,clusters.AMux0.SEL,"AMux_0.SEL");
115     // sc_trace(fp,clusters.AMux0.Y,"AMux_0.Y");
116     // sc_trace(fp,clusters.AMuxN.Y,"AMux_N.Y");
117
118     // sc_trace(fp,clusters.BMux0.A,"BMux0.A");
119     // sc_trace(fp,clusters.BMux0.B,"BMux0.B");
```

```
120     // sc_trace(fp,clusters.BMux0.MODE,"BMux0.MODE");
121     // sc_trace(fp,clusters.BMux0.CORES,"BMux0.CORES");
122     // sc_trace(fp,clusters.BMux0.SEL,"BMux0.SEL");
123     // sc_trace(fp,clusters.BMux0.Y,"BMux0.Y");
124     // sc_trace(fp,clusters.BMuxN.Y,"BMuxN.Y");
125
126
127
128     // sc_trace(fp,clusters.pimcore[0].A,"PIMCORE_1.A");
129     // sc_trace(fp,clusters.pimcore[0].B,"PIMCORE_1.B");
130     // sc_trace(fp,clusters.pimcore[0].Y,"PIMCORE_1.Y");
131
132     // sc_trace(fp,clusters.pimcore[1].A,"PIMCORE_2.A");
133     // sc_trace(fp,clusters.pimcore[1].B,"PIMCORE_2.B");
134     // sc_trace(fp,clusters.pimcore[1].Y,"PIMCORE_2.Y");
135
136     // sc_trace(fp,clusters.pimcore[2].A,"PIMCORE_3.A");
137     // sc_trace(fp,clusters.pimcore[2].B,"PIMCORE_3.B");
138     // sc_trace(fp,clusters.pimcore[2].Y,"PIMCORE_3.Y");
139
140     // sc_trace(fp,clusters.pimcore[3].A,"PIMCORE_4.A");
141     // sc_trace(fp,clusters.pimcore[3].B,"PIMCORE_4.B");
142     // sc_trace(fp,clusters.pimcore[3].Y,"PIMCORE_4.Y");
143
144     // sc_trace(fp,clusters.pimcore[4].A,"PIMCORE_5.A");
```

```
145     // sc_trace(fp, clusters.pimcore[4].B,"PIMCORE_5.B");
146     // sc_trace(fp, clusters.pimcore[4].Y,"PIMCORE_5.Y");
147
148     // sc_trace(fp, clusters.pimcore[5].A,"PIMCORE_6.A");
149     // sc_trace(fp, clusters.pimcore[5].B,"PIMCORE_6.B");
150     // sc_trace(fp, clusters.pimcore[5].Y,"PIMCORE_6.Y");
151
152     // sc_trace(fp, clusters.pimcore[6].A,"PIMCORE_7.A");
153     // sc_trace(fp, clusters.pimcore[6].B,"PIMCORE_7.B");
154     // sc_trace(fp, clusters.pimcore[6].Y,"PIMCORE_7.Y");
155
156     sc_trace(fp, clusters.pimcoreN.A, "PIMCORE_N.A");
157     sc_trace(fp, clusters.pimcoreN.B, "PIMCORE_N.B");
158     sc_trace(fp, clusters.pimcoreN.Y, "PIMCORE_N.Y");
159     sc_trace(fp, clusters.pimcoreN.ASel, "PIMCORE_N.ASel");
160     sc_trace(fp, clusters.pimcoreN.BSel, "PIMCORE_N.BSel");
161     sc_trace(fp, clusters.pimcoreN.FUNC_IN, "PIMCORE_N.FUNC_IN");
162     sc_trace(fp, clusters.pimcoreN.FUNC_ADDR, "PIMCORE_N.
          FUNC_ADDR");
163      sc_trace(fp, clusters.pimcoreN.IN_MODE, "PIMCORE_N.IN_MODE")
          ;
164     sc_trace(fp, clusters.pimcoreN.LOAD, "PIMCORE_N.LOAD");
165     sc_trace(fp, clusters.pimcoreN.RUN, "PIMCORE_N.RUN");
166     sc_trace(fp, clusters.pimcoreN.AMux.IN, "PIMCORE_N.AMux.IN");
```

```
167     sc_trace(fp, clusters.pimcoreN.AMux.SEL, "PIMCORE_N.AMux.SEL"
            );
168     sc_trace(fp, clusters.pimcoreN.AMux.OUT, "PIMCORE_N.AMux.OUT"
            );
169     sc_trace(fp, clusters.pimcoreN.BMux.IN, "PIMCORE_N.BMux.IN");
170     sc_trace(fp, clusters.pimcoreN.BMux.SEL, "PIMCORE_N.BMux.SEL"
            );
171     sc_trace(fp, clusters.pimcoreN.BMux.OUT, "PIMCORE_N.BMux.OUT"
            );
172
173     sc_trace(fp, clusters.pimcoreN.mux256to1[0].IN, "PIMCORE_N.
            mux256to1_0.IN");
174     sc_trace(fp, clusters.pimcoreN.mux256to1[1].IN, "PIMCORE_N.
            mux256to1_1.IN");
175     sc_trace(fp, clusters.pimcoreN.mux256to1[2].IN, "PIMCORE_N.
            mux256to1_2.IN");
176     sc_trace(fp, clusters.pimcoreN.mux256to1[3].IN, "PIMCORE_N.
            mux256to1_3.IN");
177     sc_trace(fp, clusters.pimcoreN.mux256to1[4].IN, "PIMCORE_N.
            mux256to1_4.IN");
178     sc_trace(fp, clusters.pimcoreN.mux256to1[5].IN, "PIMCORE_N.
            mux256to1_5.IN");
179     sc_trace(fp, clusters.pimcoreN.mux256to1[6].IN, "PIMCORE_N.
            mux256to1_6.IN");
```

```
180     sc_trace(fp, clusters.pimcoreN.mux256to1[7].IN, "PIMCORE_N.
            mux256to1_7.IN");
181     sc_trace(fp, clusters.pimcoreN.func_regs.DATA_IN, "REG_FILE.
            DATA_IN");
182     sc_trace(fp, clusters.pimcoreN.func_regs.WRITE_EN, "REG_FILE.
            WRITE_EN");
183     sc_trace(fp, clusters.pimcoreN.func_regs.DATA_OUT, "REG_FILE.
            sFUNC");
184     sc_trace(fp, clusters.pimcoreN.func_regs.WRITE_ADDR, "
            REG_FILE.WRITE_ADDR");
185
186     sc_trace(fp, clusters.pimcoreN.mux256to1[0].SEL, "PIMCORE_N.
            mux256to1_0.SEL");
187     sc_trace(fp, clusters.pimcoreN.mux256to1[0].OUT, "PIMCORE_N.
            mux256to1_0.OUT");
188
189     sc_trace(fp, clusters.pimcoreN.mux256to1[7].IN, "PIMCORE_N.
            mux256to1_7.IN");
190     sc_trace(fp, clusters.pimcoreN.mux256to1[7].SEL, "PIMCORE_N.
            mux256to1_7.SEL");
191     sc_trace(fp, clusters.pimcoreN.mux256to1[7].OUT, "PIMCORE_N.
            mux256to1_7.OUT");
192
193     sc_trace(fp, clusters.pimcoreN.Areg.DATA_IN, "PIMCORE_N.Areg.
            DATA_IN");
```

```
194     sc_trace(fp, clusters.pimcoreN.Areg.WRITE, "PIMCORE_N.Areg.
            WRITE");
195     sc_trace(fp, clusters.pimcoreN.Areg.clk, "PIMCORE_N.Areg.clk"
            );
196     sc_trace(fp, clusters.pimcoreN.Areg.DATA_OUT, "PIMCORE_N.Areg
            .DATA_OUT");
197
198     sc_trace(fp, clusters.pimcoreN.Breg.DATA_IN, "PIMCORE_N.Breg.
            DATA_IN");
199     sc_trace(fp, clusters.pimcoreN.Breg.WRITE, "PIMCORE_N.Breg.
            WRITE");
200     sc_trace(fp, clusters.pimcoreN.Breg.clk, "PIMCORE_N.Breg.clk"
            );
201     sc_trace(fp, clusters.pimcoreN.Breg.DATA_OUT, "PIMCORE_N.Breg
            .DATA_OUT");
202
203     sc_trace(fp, clusters.pimcoreN.clk, "PIMCORE_N.clk");
204
205
206     // sc_trace(fp, clusters.pimcore0.AMux.OUT,"pimcore0.sAIn");
207     // sc_trace(fp, clusters.pimcore[1].AMux.OUT,"pimcore1.sAIn
            ");
208     // sc_trace(fp, clusters.pimcore[2].AMux.OUT,"pimcore2.sAIn
            ");
209     // sc_trace(fp, clusters.pimcoreN.AMux.OUT,"pimcoreN.sAIn");
```

```
210       //  sc_trace(fp,clusters.pimcoreN.func_regs.DATA_OUT,"sFUNC
              ");
211       sc_trace(fp,clk,"clk");
212            sc_trace(fp,Y,"Y");
213            sc_trace(fp,ACC,"ACC");
214            sc_trace(fp,add_port,"add_port");
215            sc_trace(fp,mult_port,"mult_port");
216       sc_start(1250,SC_NS);
217            //sc_start();
218    sc_close_vcd_trace_file(fp);
219
220    return 0;
221
222 }
```

II.5 In_multiplexer model

```
1  #ifndef IN_MUX_H
2  #define IN_MUX_H
3  #define SC_INCLUDE_FX
4  //#include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include "nbit_multiplexer.h"
10 //M = 9
11 //N = 4
12 template <int M, int N> class In_multiplexer : public sc_module
13 {
14 public:
15   //PORTS
16   sc_in<sc_bv<2*N> > A{"A"};
17   sc_in<sc_bv<2*N> > B{"B"};
18   sc_in<sc_bv<2*N*(M-1)> > CORES{"CORES"}; //64
19   sc_in<sc_bv<1> > MODE{"MODE"};
20   constexpr static int size = ceil(log2(2*(M-1))); // 4 when
       called from cluster
21   sc_in<sc_bv<size> > SEL{"SEL"};
22   sc_out<sc_bv<N> > Y{"Y"};
```

```
23
24    //Signal Declaratoin
25
26    sc_signal<sc_bv<N> > sInt;
27    sc_signal<sc_bv<N> > sExt;
28    sc_signal<sc_bv<2> > sel_range;
29    sc_signal<sc_bv<4*N> > A_B;
30    sc_signal<sc_bv<2*N> > Ext_Int;
31
32
33    //Global declarations of instantiated modules
34       nbit_multiplexer<size ,N> IntMux{"IntMux"};
35       nbit_multiplexer <2,N> ExtMux{"ExtMux"};
36       nbit_multiplexer <1,N> ModeMux{"ModeMux"};
37
38    SC_HAS_PROCESS(In_multiplexer);
39
40    In_multiplexer(sc_module_name name): sc_module(name)
41    {
42
43       SC_METHOD(Select_Range);
44       sensitive << SEL;
45
46       SC_METHOD(concat_AB);
47       sensitive << A << B;
```

```
48
49        SC_METHOD(concat_ExtInt);
50        sensitive << sInt << sExt;
51
52        SC_METHOD(print_Y);
53        sensitive <<Y;
54
55        IntMux.IN(CORES);
56        IntMux.SEL(SEL);
57        IntMux.OUT(sInt);
58
59        ExtMux.IN(A_B);
60        ExtMux.SEL(sel_range);
61        ExtMux.OUT(sExt);
62
63        ModeMux.IN(Ext_Int);
64        ModeMux.SEL(MODE);
65        ModeMux.OUT(Y);
66    }
67
68    void Select_Range()
69    {
70        sc_bv<size> select_copy;
71
72        select_copy = SEL.read();
```

```
73      sel_range = select_copy.range(1,0);
74
75    }
76
77    void concat_AB()
78    {
79      sc_bv<2*N> copy_A;
80      sc_bv<2*N> copy_B;
81      sc_bv<4*N> copy_AB;
82
83
84      copy_A = A.read();
85      copy_B = B.read();
86
87      copy_AB = (copy_A,copy_B);
88      A_B.write(copy_AB);
89      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "A= "<< A.read() <<endl;
90      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "B = "<< B.read() <<endl;
91      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "A_B = "<< A_B <<endl;
92    }
93
94    void concat_ExtInt()
```

```
95    {
96      sc_bv<N> copy_Ext;
97      sc_bv<N> copy_Int;
98      sc_bv<2*N> copy_ExtInt;
99
100
101     copy_Ext = sExt.read();
102     copy_Int = sInt.read();
103     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sExt = "<< sExt <<endl;
104     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sInt = "<< sInt <<endl;
105
106     copy_ExtInt = (copy_Ext,copy_Int);
107     Ext_Int.write(copy_ExtInt);
108   }
109
110 void print_Y()
111 {
112   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        Y in In_multiplexer= "<< Y.read() <<endl;
113   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        CORES in In_multiplexer= "<< CORES.read() <<endl;
114   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        MODE in In_multiplexer= "<< MODE.read() <<endl;
```

```
115     // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
           SEL in In_multiplexer= "<< SEL.read() <<endl;
116     //cout<< "at time: " << sc_time_stamp()<< "----------> "<< "Y
            in In_multiplexer= "<< Y.read() <<endl;
117
118
119 }
120
121
122 };
123
124 #endif
```

II.6 Decoder model

```
1  #ifndef DECODER_H
2   #define DECODER_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10
11 template <int N> class decoder : public sc_module {
12      public:
13          constexpr static int size = ceil(log2(N));
14          sc_in<sc_uint<size> > IN{"IN"};
15          sc_out<sc_bv<N> > OUT;
16
17          //sc_in_clk clk;
18          sc_bv<N> sOUT;
19
20          unsigned i;
21
22          SC_HAS_PROCESS(decoder);
23          decoder(sc_module_name name) : sc_module(name)
```

```
24          {
25              SC_THREAD(prc_decoder);
26              sensitive << IN;
27          }
28
29          void prc_decoder()
30          {
31                  sc_uint<3> INcopy;
32                  INcopy = IN->read();
33              switch(INcopy)
34              {
35                  case 0 : sOUT=0x1;
36                           break;
37                  case 1 : sOUT=0x2;
38                           break;
39                  case 2 : sOUT=0x4;
40                           break;
41                  case 3 : sOUT=0x8;
42                           break;
43                  case 4 : sOUT=0x10;
44                           break;
45                  case 5 : sOUT=0x20;
46                           break;
47                  case 6 : sOUT=0x40;
48                           break;
```

```
49                      case 7 : sOUT=0x80;
50                               break;
51                      default : sOUT=0x0;
52                               break;
53                  }
54
55                  OUT = sOUT;
56                  //cout<< "at time: " << sc_time_stamp()<<
                        "----------> "<< "OUT in decoder= "<< OUT.read()
                        <<endl;
57              }
58  };
59
60  #endif
```

II.7 Multiplexer model

```
1  #ifndef MUX_H
2  #define MUX_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10
11 //N=2
12 template <int N, int select_size> class multiplexer : public
       sc_module
13 {
14 public:
15   sc_in<sc_bv<N> > IN;
16   sc_in<sc_bv<select_size> > SEL; //8 bit from PIM core
17   sc_out<bool> OUT;
18
19   //sc_vector<sc_signal<sc_biguint<N> > > IN_copy{"IN_copy", N
         };
20   sc_bv<select_size> IN_select;
21
```

```
22    bool out_copy;
23    sc_bv<N> IN_copy;
24    //
25    SC_HAS_PROCESS(multiplexer);
26
27    multiplexer(sc_module_name name) : sc_module(name)
28    {
29      SC_METHOD(prc_mux);
30       sensitive<<IN<<SEL;
31
32    }
33
34    void prc_mux()
35    {
36      IN_copy = IN;
37      IN_select = SEL.read();
38      //mask = 1;
39      //s = IN_select;
40      //cout<< "at time: " << sc_time_stamp()<< endl << "
          IN_select = "<< IN_select <<endl;
41      for(auto i=0U; i<N; ++i)
42      {
43        if(i==IN_select)
44        {
45          out_copy = bool (IN_copy[i]);
```

```
46          }
47      }
48      OUT.write(out_copy);
49
50      // cout<< "at time: " << sc_time_stamp()<< endl << "SEL =
            "<< SEL.read() <<endl;
51      // cout<< "at time: " << sc_time_stamp()<< endl << "IN =
            "<< IN <<endl;
52      // cout<< "at time: " << sc_time_stamp()<< endl << "IN_copy
             = "<< IN_copy <<endl;
53      // //cout<< "at time: " << sc_time_stamp()<< endl << "
            out_copy = "<< out_copy <<endl;
54      //cout<< "at time: " << sc_time_stamp()<< endl << "OUT mux
            = "<< OUT.read() <<endl;
55
56    }
57 };
58
59 #endif
```

II.8 nbit_multiplexer model

```
1  #ifndef NBIT_MUX_H
2  #define NBIT_MUX_H
3  #define SC_INCLUDE_FX
4  #include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include <cmath>
10 #include "multiplexer.h"
11 #include "sensitive_def.h"
12 #include<math.h>
13
14 // template <typename T>
15 // sc_sensitive& operator << (sc_sensitive& sensitive, const
       sc_vector<T>& vec )
16 // {
17 //  for (auto & el : vec )
18 //    sensitive << el;
19 //  return sensitive;
20 // }
21 //N=1, M=4
22 // parameters sent from cluster: N = 4; M = 8
```

```
23 template<int N, int M> class nbit_multiplexer : public
       sc_module
24 {
25 public:
26
27    //sc_in<sc_biguint<pow(2,N)*M> > IN;
28    sc_in<sc_bv<(1<<N)*M> > IN; //8 bits 128 bits as per PIM
          cluster
29    sc_in<sc_bv<N> > SEL; //1 bit
30    sc_out<sc_bv<M> > OUT; //4 bits
31
32    sc_bv<(1<<N)*M> sIN;
33    //sc_vector<sc_signal<sc_bv<1>>> sOUT;
34    //sc_bv<M> sOUT;
35
36    unsigned G;
37
38    constexpr static int select_size = ceil(log2(1<<N)); //
          constexpr is used to make it a compiletime statement
39
40   SC_HAS_PROCESS(nbit_multiplexer);
41
42    sc_vector<multiplexer<(1<<N),select_size> > bit_mux{"bit_mux"
          ,M};
```

```
43    sc_vector<sc_signal<sc_bv<(1<<N)>>> sIN_segment{"sIN_segment"
          ,M};
44    sc_vector<sc_signal<bool> > sOUT_bit_select{"sOUT_bit_select"
          ,M};
45
46
47    nbit_multiplexer(sc_module_name name): sc_module(name)
48    {
49
50      SC_METHOD(reorganization);
51      sensitive << IN;
52
53      // SC_METHOD(selection);
54      // sensitive << sIN_segment;
55      //cout<< "at time: " << sc_time_stamp()<< endl << "IN in
            nbit_multiplexer= "<< IN  <<endl;
56
57      SC_METHOD(sOUT_select);
58      sensitive << sOUT_bit_select;
59
60      for (auto G = 0U; G<M ; ++G)
61      {
62        bit_mux[G].IN(sIN_segment[G]);
63        bit_mux[G].SEL(SEL);
64        bit_mux[G].OUT(sOUT_bit_select[G]);
```

```
65      }
66
67
68  }
69  void reorganization()
70  {
71      sc_bv<(1<<N)*M> din;
72      //cout<< "at time: " << sc_time_stamp()<< endl << "din =
            "<< din <<endl;
73      //reorganization
74      for(G=0; G<((1<<N)*M); ++G)
75      {
76
77          sIN[((G%M)*(1<<N))+(G/M)] = IN.read()[G];
78          for(auto i=0U; i<M; ++i)
79          {
80              din=sIN.range(((1<<N)*(i+1))-1,(1<<N)*i);
81              sIN_segment[i]=din;
82              // cout<< "at time: " << sc_time_stamp()<< endl << "sIN
                    = "<< sIN <<endl;
83              // cout<< "at time: " << sc_time_stamp()<< endl << "din
                    = "<< din <<endl;
84          }
85
86      }
```

```
87
88
89      }
90
91      // void selection()
92      // {
93      //   sc_bv<(1<<N)*M> din;
94
95      //   for(auto i=0U; i<M; ++i)
96      //   {
97      //     din=sIN.range((1<<N)*(i+1)-1,(1<<N)*i);
98      //     sIN_segment[i]=din;
99      //   }
100
101
102     // }
103
104     void sOUT_select()
105     // {
106     //   for(auto i=0U; i<M; ++i)
107     //   {
108     //     sOUT[i] = sOUT_bit_select[i];
109     //     OUT.write(sOUT);
110     //   }
111
```

```
112     // }
113
114     {
115       sc_bv<M> write_value;
116       for(auto i=0U; i<M; ++i)
117       {
118         write_value[i] = sOUT_bit_select[i].read();
119
120       }
121       //sOUT = write_value;
122       OUT.write(write_value);
123     }
124
125
126
127 };
128
129 #endif
```

II.9 register256 model

```
1  #ifndef REGISTER_256_H
2  #define REGISTER_256_H
3 #define SC_INCLUDE_FX
4 #include <systemc.h>
5 #include <iomanip>
6 #include <iostream>
7 #include <string>
8 #include <fstream>
9
10 template <int size> class register256 : public sc_module {
11 public:
12    sc_in<sc_bv<size> > DATA_IN;
13    sc_in<bool> WRITE;
14       //sc_in<sc_biguint<6> > WRITE_ADDR;
15
16    //sc_in_clk clk;
17
18
19    sc_in<bool> clk;
20    sc_in<bool> reset;
21    sc_out<sc_bv<size> > DATA_OUT;
22    sc_bv<size> DATA;
23
```

```
24    SC_HAS_PROCESS(register256);
25    register256(sc_module_name name) : sc_module(name) //
          constructor that takes paprameter
26    {
27      SC_METHOD(prc_register256);
28      sensitive << clk.pos()<< reset.pos();//poedge
29      //sensitive << DATA_IN<<WRITE;
30      SC_METHOD(print_ports)
31      sensitive<<DATA_OUT;
32    }
33
34    void prc_register256 ()
35    {
36      //wait(clk.posedge_event());
37      if(reset)
38        DATA = 0;
39      else if(WRITE.read())
40      {
41              DATA=DATA_IN.read();
42      }
43      DATA_OUT.write(DATA);
44
45
46
```

```
47      //    cout<< "at time: " << sc_time_stamp()<< endl << "clk =
            "<< clk <<endl;
48      // cout<< "at time: " << sc_time_stamp()<< endl << "
           write_addr = "<< write_addr[i] <<endl;
49      //cout<< "at time: " << sc_time_stamp()<< endl << "WRITE_EN
            = "<< WRITE_EN.read() <<endl;
50      //cout<< "WRITE_EN = "<< WRITE_EN.read() <<endl;
51      //cout<< "at time: " << sc_time_stamp()<< "---------> "<< "
           WRITE_EN in reg256 = "<< WRITE_EN.read()  <<endl;
52      //cout<< "at time: " << sc_time_stamp()<< "---------> "<< "
           DATA_OUT in reg256 = "<< DATA_OUT.read()  <<endl;
53      //cout<< "at time: " << sc_time_stamp()<< endl << "DATA_IN
           = "<< DATA_IN.read() <<endl;
54    }
55
56    void print_ports()
57    {
58    //  cout<< "at time: " << sc_time_stamp()<<
           "---------------->" << "DATA_IN = "<< DATA_IN.read() <<endl
           ;
59    //  cout<< "at time: " << sc_time_stamp()<< "---------->" <<
           "WRITE = "<< WRITE.read() <<endl;
60    //  cout<< "at time: " << sc_time_stamp()<< "---------->" <<
           "clk = "<< clk <<endl;
```

```
61    //   cout<< "at time: " << sc_time_stamp()<< "---------->" <<
         "reset = "<< reset <<endl;
62    //   cout<< "at time: " << sc_time_stamp()<<
         "----------------->" << "DATA_OUT = "<< DATA_OUT.read() <<
         endl;
63     }
64
65
66
67  };
68
69  #endif
```

II.10 register_file model

```
1  #ifndef REG_FILE_IF
2  #define REG_FILE_IF
3  #define SC_INCLUDE_FX
4  #include <iomanip>
5  #include <iostream>
6  #include <string>
7  #include <fstream>
8  #include "register256.h"
9  #include "sensitive_def.h"
10
11 // template <typename T>
12 // sc_sensitive& operator << (sc_sensitive& sensitive, const
       sc_vector<T>& vec )
13 // {
14 //   for (auto & el : vec )
15 //     sensitive << el;
16 //   return sensitive;
17 // }
18 //256,4
19 template <unsigned N, unsigned M> class register_file : public
       sc_module
20 {
21 public:
```

```
22
23    //  int K;
24    // K = M*N;
25    //  int L;
26    // L = 2*M;
27    sc_in<sc_bv<N> > DATA_IN;
28    sc_in<sc_bv<2*M> > WRITE_ADDR;
29    sc_in<bool> WRITE_EN;
30    sc_in<bool> READ_EN;
31    sc_in_clk clk;
32    sc_in<bool> reset;
33    sc_out<sc_bv<(2*M)*(1<<(2*M))> > DATA_OUT;  //384
34
35    sc_bv<2*M> write_addr_var;
36    bool write_en_var;
37    sc_signal<sc_bv<N> > DATA_IN_copy;
38
39
40      sc_vector<register256<N>> regs;
41    sc_vector<sc_signal<sc_bv<N>>> data_out;
42    sc_vector<sc_signal<bool>> AND_sig;
43    unsigned i;
44
45   SC_HAS_PROCESS(register_file);
46
```

```
47    register_file(sc_module_name name): sc_module(name), regs("
          regs", 2*M), data_out("data_out", N), AND_sig("AND_sig", 2*
          M)
48    {
49      SC_METHOD(convert_to_bv)
50      sensitive << DATA_IN;
51      SC_METHOD(combine_output);
52      sensitive << data_out;
53      SC_METHOD(addr_computation);
54      sensitive << WRITE_ADDR << WRITE_EN;
55
56      for(auto i=0U; i< 2*M ; ++i)
57      {
58
59
60            // make the method react on changes of data_out[i]
61
62            // connect the register
63        //auto& reg1 = regs[i];
64        regs[i].clk(clk);
65        regs[i].reset(reset);
66        regs[i].DATA_IN(DATA_IN_copy);
67        regs[i].WRITE(AND_sig[i]);
68        regs[i].DATA_OUT(data_out[i]);
69
```

```
70
71          //reg.DATA_OUT(DATA_OUT);
72
73          //cout<< "at time: " << sc_time_stamp()<< endl << "
               data_out"<< i << " = " << data_out[i] <<endl;
74          // cout<< "at time: " << sc_time_stamp()<< endl << "
               AND_sig[i] = "<< AND_sig[i] <<endl;
75
76          //cout<< "at time: " << sc_time_stamp()<< endl << "
              WRITE_ADDR = "<< WRITE_ADDR.read() <<endl;
77          //cout<< "at time: " << sc_time_stamp()<< endl << "flag
                 = "<< flag.read() <<endl;
78
79
80
81
82      }
83
84
85      }
86
87      void convert_to_bv()
88      {
89        sc_bv<N> D;
90
```

```
91          D= DATA_IN;
92
93          DATA_IN_copy = D;
94      }
95
96    void combine_output() {
97          sc_bv<(2*M)*(1<<(2*M))> dout;
98          //dout = 0;
99
100     for(auto i=0U; i< 2*M; ++i)
101     {
102             dout.range(((i+1)*N)-1,i*N)=data_out[i].read();
103
104             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    data_out[i] = "<< data_out[i].read() <<endl;
105             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    write_addr_var = "<< write_addr_var[i] <<endl;
106             //cout<< "at time: " << sc_time_stamp()<< endl << "
                    flag = "<< flag <<endl;
107            //cout<< "at time: " << sc_time_stamp()<<
                  "----------------------------------------> "<< "
                    dout = " << dout <<endl;
108
109           }
110           DATA_OUT.write(dout);
```

```
111         //cout<< "at time: " << sc_time_stamp()<<
              "------------------------------------------> "<< "
              DATA_OUT = "<< DATA_OUT.read() <<endl;
112          //cout<< "at time: " << sc_time_stamp()<< endl << "
               WRITE_EN = "<< WRITE_EN <<endl;
113     }
114
115     void addr_computation()
116     {
117       write_en_var = WRITE_EN;
118
119       for(auto i=0U; i<2*M; ++i)
120       {
121         AND_sig[i] = write_en_var && WRITE_ADDR.read()[i];
122       }
123
124       //cout<< "at time: " << sc_time_stamp()<< "---------> "<<
            "write_en_var = "<< write_en_var <<endl;
125       //cout<< "at time: " << sc_time_stamp()<< "---------> "<<
            "WRITE_ADDR = "<< WRITE_ADDR.read()[i] <<endl;
126     }
127
128
129     //Destructor
130
```

```
131       // ~register_file()
132       // {
133       //   for(i=0; i<M;++i)
134       //   {
135       //     delete regs[i];
136       //   }
137
138       // }
139
140
141
142
143
144 };
145
146 #endif
```

II.11 In_multiplexer model

```
1  #ifndef IN_MUX_H
2  #define IN_MUX_H
3  #define SC_INCLUDE_FX
4  //#include <systemc.h>
5  #include <iomanip>
6  #include <iostream>
7  #include <string>
8  #include <fstream>
9  #include "nbit_multiplexer.h"
10 //M = 9
11 //N = 4
12 template <int M, int N> class In_multiplexer : public sc_module
13 {
14 public:
15   //PORTS
16   sc_in<sc_bv<2*N> > A{"A"};
17   sc_in<sc_bv<2*N> > B{"B"};
18   sc_in<sc_bv<2*N*(M-1)> > CORES{"CORES"}; //64
19   sc_in<sc_bv<1> > MODE{"MODE"};
20   constexpr static int size = ceil(log2(2*(M-1))); // 4 when
       called from cluster
21   sc_in<sc_bv<size> > SEL{"SEL"};
22   sc_out<sc_bv<N> > Y{"Y"};
```

```
23
24    //Signal Declaratoin
25
26    sc_signal<sc_bv<N> > sInt;
27    sc_signal<sc_bv<N> > sExt;
28    sc_signal<sc_bv<2> > sel_range;
29    sc_signal<sc_bv<4*N> > A_B;
30    sc_signal<sc_bv<2*N> > Ext_Int;
31
32
33    //Global declarations of instantiated modules
34       nbit_multiplexer<size ,N> IntMux{"IntMux"};
35       nbit_multiplexer <2,N> ExtMux{"ExtMux"};
36       nbit_multiplexer <1,N> ModeMux{"ModeMux"};
37
38    SC_HAS_PROCESS(In_multiplexer);
39
40    In_multiplexer(sc_module_name name): sc_module(name)
41    {
42
43       SC_METHOD(Select_Range);
44       sensitive << SEL;
45
46       SC_METHOD(concat_AB);
47       sensitive << A << B;
```

```
48
49      SC_METHOD(concat_ExtInt);
50      sensitive << sInt << sExt;
51
52      SC_METHOD(print_Y);
53      sensitive <<Y;
54
55      IntMux.IN(CORES);
56      IntMux.SEL(SEL);
57      IntMux.OUT(sInt);
58
59      ExtMux.IN(A_B);
60      ExtMux.SEL(sel_range);
61      ExtMux.OUT(sExt);
62
63      ModeMux.IN(Ext_Int);
64      ModeMux.SEL(MODE);
65      ModeMux.OUT(Y);
66  }
67
68  void Select_Range()
69  {
70      sc_bv<size> select_copy;
71
72      select_copy = SEL.read();
```

```
73      sel_range = select_copy.range(1,0);
74
75    }
76
77    void concat_AB()
78    {
79      sc_bv<2*N> copy_A;
80      sc_bv<2*N> copy_B;
81      sc_bv<4*N> copy_AB;
82
83
84      copy_A = A.read();
85      copy_B = B.read();
86
87      copy_AB = (copy_A,copy_B);
88      A_B.write(copy_AB);
89      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "A= "<< A.read() <<endl;
90      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "B = "<< B.read() <<endl;
91      // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "A_B = "<< A_B <<endl;
92    }
93
94    void concat_ExtInt()
```

```
95    {
96      sc_bv<N> copy_Ext;
97      sc_bv<N> copy_Int;
98      sc_bv<2*N> copy_ExtInt;
99
100
101     copy_Ext = sExt.read();
102     copy_Int = sInt.read();
103     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sExt = "<< sExt <<endl;
104     // cout<< "at time: " << sc_time_stamp()<< "----------> "<<
            "sInt = "<< sInt <<endl;
105
106     copy_ExtInt = (copy_Ext,copy_Int);
107     Ext_Int.write(copy_ExtInt);
108   }
109
110 void print_Y()
111 {
112   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        Y in In_multiplexer= "<< Y.read() <<endl;
113   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        CORES in In_multiplexer= "<< CORES.read() <<endl;
114   // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
        MODE in In_multiplexer= "<< MODE.read() <<endl;
```

```
115     // cout<< "at time: " << sc_time_stamp()<< "----------> "<< "
           SEL in In_multiplexer= "<< SEL.read() <<endl;
116     //cout<< "at time: " << sc_time_stamp()<< "----------> "<< "Y
            in In_multiplexer= "<< Y.read() <<endl;
117
118
119 }
120
121
122 };
123
124 #endif
```