Modeling SYSTEMC and TLM2.0 models for various elements of PercSense Architecture for Machine learning IP

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Electronics & Communication Engineering

(Communication Engineering)

BY

Kasturi D. Patil

(12MECC34)

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2014

Modeling SYSTEMC and TLM2.0 models for various elements of PercSense Architecture for Machine learning IP

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Electronics & Communication Engineering

(Communication Engineering)

BY

Kasturi D. Patil

(12MECC34)

Under The Guidance Of

Dr. N.P Gajjar

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2014

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of **Master of Technology in Communication Engineering** at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Kasturi D.Patil

Certificate

This is to certify that the Major Project entitled **"Modeling SYSTEMC and TLM2.0 models for various elements of PercSense Architecture for Machine learning IP"** submitted by **Kasturi D. Patil (12MECC34)**, towards the partial fulfillment of the requirements for the degree of **Master of Technology in Communication Engineering** of Nirma University, Ahmedabad is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this thesis, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date :

Place : Ahmedabad

Guide :
Dr. N. P. Gajjar
(Professor,EC)

Program Coordinator:
Dr. D. K Kothari
(Professor and section head,EC)

HOD,EE :
Dr. P. N. Tekwani
(Professor,EE)

Director
Dr. K Kotecha
(Director, IT, NU)

ACKNOWLEDGEMENT

I would like to express my gratitude and sincere thanks to my guides and thesis supervisors **Dr.N. P. Gajjar** and **Krishnakumar Nair** for their constant guidance and motivation. I am deeply indebted to **Dr. P.N Tekwani**,Head of Electrical Engineering Department and **Dr. D. K. Kothari**, Coordinator M.Tech Communication Engineering and **Dr. K Kotecha**, Director, Institute of Technology Nirma University, for allowing me to undertake this thesis work and for his guidelines during the review process. Without their experience and insights, it would have been very difficult to do quality work. I wish to thank my friends of my class for their delightful company which kept me in good humor throughout the year. Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete the degree program successfully.

Kasturi D. Patil
12MECC34

Abstract

With computing systems these days having the capability of being driven by natural interfaces such as gestures, speech recognition, etc., there is an ever growing need for the systems being built in the future to have more complexity in different domains and hence heavy computing capability to support the underlying Perceptual Computing algorithms. In particular Object detection and recognition of Machine Vision are heavy on compute, also ISP/GPU processors are not very efficient for data manipulation to feed into detection/extraction stages since these stages require data to be extracted from different locations of the integral image. There is a need for efficient extraction of these data elements. Most of the fast implementations of machine vision algorithms for Object recognition make use of Integral Image based (32 bit) processing, and this is not very efficient to be processed on an ISP.
The Project aims at improvement of power-performance and alleviating workloads for machine learning algorithms like GMM(Gaussian Mixture Model), KNN(K-Nearest Neighbor), DBNN(Deep Belief Neural Network), CDBNN(Convolutional Deep Belief Neural Network) by providing custom accelerator.

The algorithms that have been identified with a common structure for acceleration are GMM for speech based Senone classifier, , K Nearest Neighbor (KNN) based Classifier (Object). Some of the motivating factors for an accelerator of this nature is to be very efficient for a class of Detection/Feature Extraction, Classification and Machine Learning algorithms including Speech based, with the help of this project much better metrics for performance/power/price can be obtained as compared to ISP/GPU based general purpose based vector DSP systems, and this proposed architecture is highly scalable and configurable.

Abbreviation Notation and Nomenclature

TLM	Transaction Level Modeling
IP	Intellectual Property
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ISP	Internet Service Provider
GMM	Gaussian Mixture Model
KNN	K-Nearest Neighbor
DBNN	Deep Belief Neural Network
CDBNN	Convolutional Deep Belief Neural Network
PE	Processing Engine
ML	Machine Learning
FSM	Finite State Machine
RBM	Restricted Boltzmann Machine
SIMD	Single Instruction Multiple Data
MAC	Multiply And Accumulate
GDB	GNU Debugger
RTL	Register Transfer Level
CPP	C Plus Plus

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Definition

The purpose of the project as a whole is to implement the PercSense Architecture for speeding up Perceptual Computing and Machine learning based algorithms. Perceptual Computing (PerC) is a project undertaken within Intel Corporation tasked with the research, development, and productization of technologies for Natural User Interaction. [1] The objective of PerC is to explore consumer-focused hardware and software applications of close-range hand and finger gestures, speech recognition, face recognition and tracking, and augmented reality.PercSense is a configurable and scalable fixed function accelerator that yields high performance at low power consumption for Computer Vision, Speech and in general Machine Learning (ML) workloads. PercSense ML IP in combination with a CPU/GPU/ISP is a unique approach wherein we get the benefits of a programmable approach as well as a HW fixed function approach. So, The PerC System Level Architechture aims at improvement of power-performance and alleviating workloads for machine learning algorithms like GMM, KNN, DBNN, CDBNN by providing custom accelerator.
The PerC System Level Architechture has a small part called Processing engine(PE). The Processing engine has the following features:

- It is based on SIMD(Single instruction multiple data) architecture.
- It has mathematics which is common to various Machine Learning algorithms.
- It has fixed point arithmetic.

The Project aims at improvement of power-performance and alleviating workloads for machine learning algorithms like GMM, KNN, DBNN, CDBN by providing custom accelerator.

1.2 Objective of Study

The main objective of the Study is as Follows :

a. Ownership of Processing Engine Modeling
    (1) Work closely with architecture team to understand Processing Engine (PE)
    (2) Model Processing Engine using SystemC
    (3) Compare PE cycle accuracy with RTL Feedback into SystemC

b. Functional verification of Processing Engine

c. Ownership of TLM2.0 model

1.3 Scope of Work

The Project focuses on the Machine Learning IPs and how they are modelled using SystemC. The preceding chapters also explains the advantges of SystemC. Furthermore the scope of the project is extended to make the simulations faster. To make them faster, TLM2.0 IEEE standard is used. How does TLM2.0 advantageous is explained in the chapters to come. Furthermore, The project is based on artificial intelligence and explores various Machine Learning Algorithms.

1.4 Thesis Organization

The title of the project says " Modeling SYSTEMC and TLM2.0 models for various elements of PercSense Architecture for Machine learning IP".First chapter of the thesis includes all the literature survey and theories studied required for the project. This survey includes study about SystemC, TLM2.0, python,various algorithms of machine learning etc.Chapter two includes Modeling and verification of PE for different Machine learning Algorithms. The algorithms taken under consideration are DBNN, CNN,RBM. Chapter three includes exercises done in TLM2.0 and work done on Percsense architechture using TLM2.0. The final chapter is conclusion and future scope wherein all the work is concluded and future scope is cited.

Chapter 2

Literature Survey

2.1 MACHINE LEARNING

Computer learns from a series of events. When a new event is placed before it, it makes a prediction based on what it has learnt. [1] Machine learning is Field of study that gives computers the ability to learn without being explicitly programmed.

Tom Mitchell (1998) Well-posed Learning Problem:
A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Machine Learning grew out of Artificial intelligence (AI). There are the following machine learning algorithms:

- Supervised learning
- Unsupervised learning

Supervised Learning: It is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal).

A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way.

Unsupervised Learning: In machine learning, the problem of unsupervised learning is that of trying to find hidden structure in unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. This distinguishes unsupervised learning from supervised learning and reinforcement learning.
Unsupervised learning is closely related to the problem of density estimation in statistics. However unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data. Many methods employed in unsupervised learning are based on data mining methods used to preprocess data.

2.1.1 Applications of Machine Learning

- Database mining: Large datasets from growth of automation/web. E.g., Web click data, medical records, biology, engineering.
- Applications cant program by hand. E.g., Autonomous helicopter, most of Natural Language Processing (NLP), Computer Vision.
- Handwriting Recognition.
- Speech recognition.

- Face Recognition.
- Stock market prediction.
- Autonomous driving.

2.1.2 Machine Learning Algorithms

a. Gaussian Mixture Model

b. K-Nearest Neighbor

c. Deep Belief Networks

a. Gaussian Mixture Model : A Gaussian Mixture Model (GMM) is a parametric probability density function represented as a weighted sum of Gaussian component densities. GMMs are commonly used as a parametric model of the probability distribution of continuous measurements or features in a biometric system, such as vocal-tract related spectral features in a speaker recognition system. GMM parameters are estimated from training data using the iterative Expectation-Maximization (EM) algorithm or Maximum a Posteriori (MAP) estimation from a well-trained prior model.

A Gaussian mixture model is a weighted sum of M component Gaussian densities as given by the equation,

$$P(x|\lambda) = \sum_{i=1}^{M} wi\, g(x|\mu i, \Sigma_i) \qquad (2.1)$$

Where x is a D-dimensional continuous-valued data vector (i.e. measurement or features), wi, i = 1, . . . ,M, are the mixture weights, and g(x|ui,i) i = 1, . . . ,M, are the component Gaussian densities. Each component density is a D-variate Gaussian function of the form,

$$G(x/\mu_i, \Sigma_i) = 1/\left((2\pi)^{D/2}|\Sigma_i|^{1/2}\right) \exp\left\{\frac{-1}{2}(x-\mu i)' \, \Sigma_i^{-1} \, (x-\mu i)\right\} \qquad (2.2)$$

The complete Gaussian mixture model is parameterized by the mean vectors, covariance matrices and mixture weights from all component densities. These parameters are collectively represented by the notation,

$$\lambda = \{ wi, \mu i, \Sigma_i \} \qquad (2.3)$$

b. K-Nearest Neighbor: The intuition underlying Nearest Neighbor Classification is quite straightforward, examples are classified based on the class of their nearest neighbors. It is often useful to take more than one neighbor into account so the technique is more commonly referred to as k-Nearest Neighbor (k-NN) Classification where k nearest neighbors are used in determining the class. Since the training examples are needed at run-time, i.e. they need to be in memory at run-time, it is sometimes also called Memory-Based Classification. Because induction is delayed to run time, it is considered a Lazy Learning technique. Because classification is based directly on the training examples it is also called Example-Based Classification or Case-Based Classification.

The basic idea is as shown in Figure which depicts a 3-Nearest Neighbor Classifier on a two-class problem in a two-dimensional feature space. In this example the decision for q1 is straightforward all three of its nearest neighbors are of class O so it is classified as an O. The situation for q2 is a bit more complicated at it has two neighbors of class X and one of class O. So kNN classification has two stages; the first is the determination of the nearest neighbors and the second is the determination of the class using those neighbors.

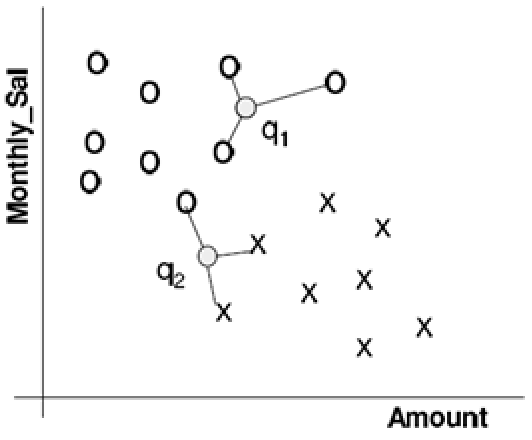


Figure 2.1: A simple example of 3- Nearest Neighbor Classification)

c. Deep belief networks: Learning is difficult in densely connected, directed belief nets that have many hidden layers because it is difficult to infer the conditional distribution of the hidden activities when given a data vector. Variational methods use simple approximations to the true conditional distribution, but the approximations may be poor, especially at the deepest hidden layer, where the prior assumes independence. Also, variational learning still requires all of the parameters to be learned together and this makes the learning time scale poorly as the number of parameters increases.

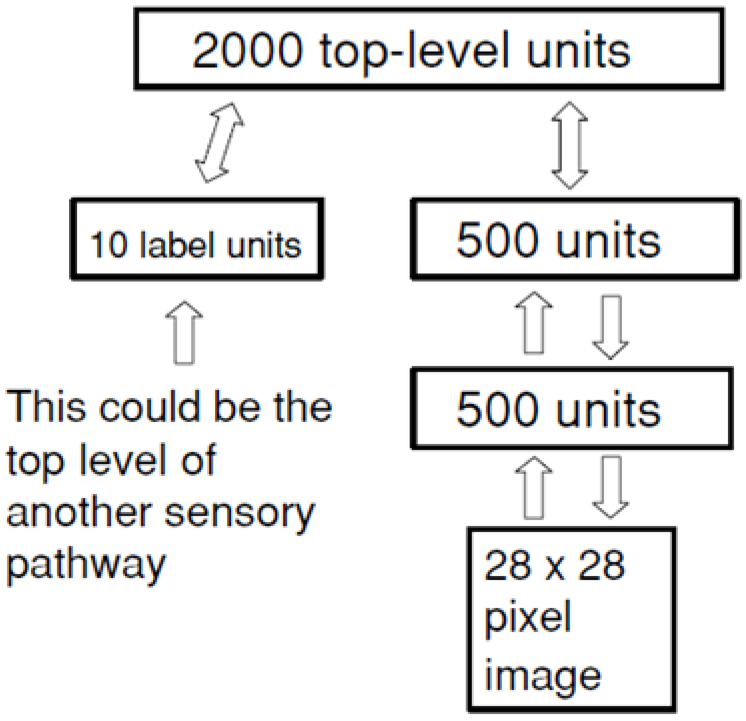


Figure 2.2: Deep belief model
The network used to model the joint distribution of digit images and digit labels. In this letter, each training case of an image and an explicit class label, but work in progress has shown that same learning algorithm can be used if the ”labels” are replaced by a multilayer pathway whose inputs are spectrograms from multiple different speaker saying isolated digits.The network then learns to generate pairs that consists of an image and a spectrogram of the same digit class.

A model is described in which the top two hidden layers form an undirected associative memory (Figure 2.2) and the remaining hidden layers form a directed acyclic graph that converts the representations in the associative memory into observable variables such as the pixels of an image. This hybrid model has some attractive features:

- There is a fast, greedy learning algorithm that can find a fairly good set of parameters quickly, even in deep networks with millions of parameters and many hidden layers.
- The learning algorithm is unsupervised but can be applied to labeled data by learning a model that generates both the label and the data.
- There is a fine-tuning algorithm that learns an excellent generative model that outperforms discriminative methods on the MNIST database of hand-written digits.
- The generative model makes it easy to interpret the distributed representations in the deep hidden layers.
- The inference required for forming a percept is both fast and accurate.
- The learning algorithm is local. Adjustments to a synapse strength depend on only the states of the presynaptic and postsynaptic neuron.
- The communication is simple. Neurons need only to communicate their stochastic binary states.

2.2 SYSTEMC(IEEE-1666)

SystemC [2] is a language that allows designers to develop both the hardware and software Components of their system together. This is all possible to do at a high level of abstraction. Strictly speaking, SystemC is not a language, but rather a library for C++, Containing structures for modeling hardware components and their interactions. SystemC can thus be compared to the hardware description languages VHDL and Verilog. An important aspect that these languages have in common is that they all come with A simulation kernel, which allows the designer to evaluate the system behavior through Simulations. Nowadays, there exist tools for high-level synthesis of SystemC models; this has pushed the industry to adopt this unified hardware − software design language in large scale.

SystemC design flow:

a. **System / Architectural Level**

   (1) Not synthesizable

   (2) Event- Driven

   (3) Abstract communication

   (4) Abstract data types

b. **Behavioral Level**

   (1) synthesizable

   (2) Clocked item I/O cycle accurate

   (3) Algorithm Description

c. **RT level (Register Transfer)**

  (1) synthesizable

  (2) Clocked

  (3) FSM

  (4) Datapath

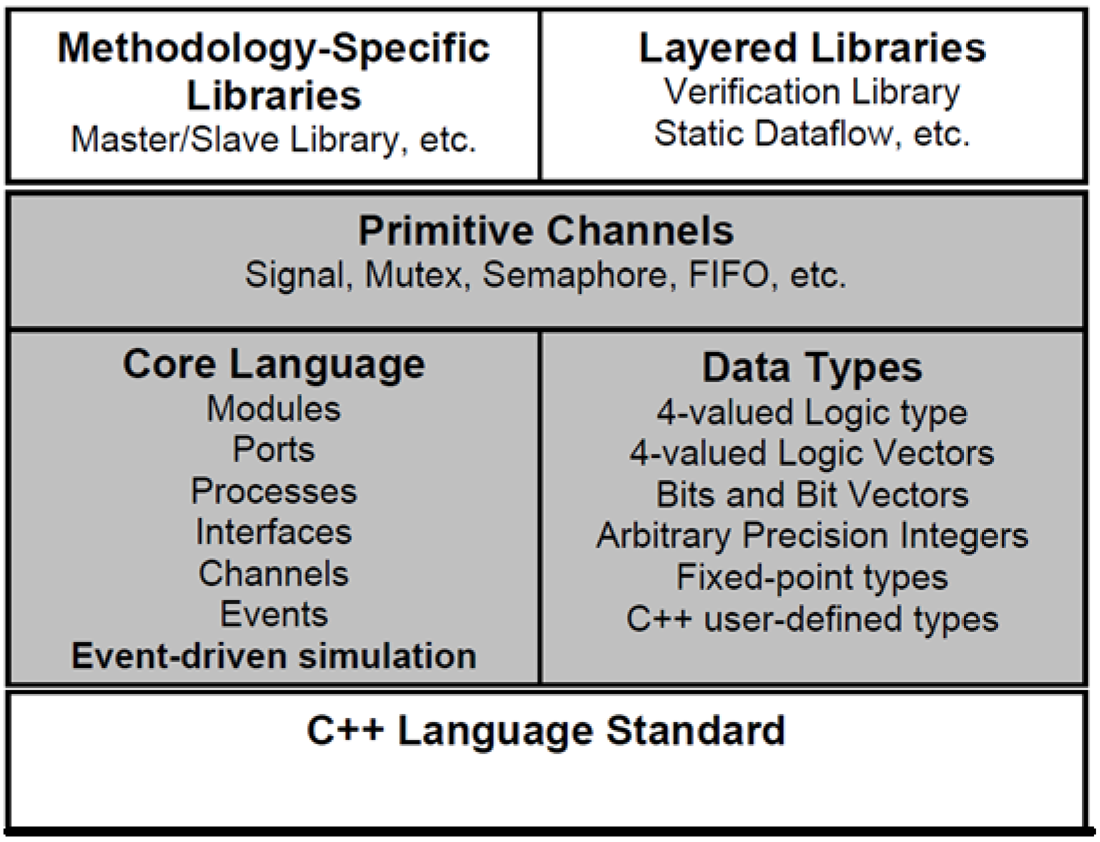


Figure 2.3: SystemC language architechture

2.2.1 MODEL OF TIME :

One of the most important components in the SystemC library is the model of time. The underlying model is based on 64 bits unsigned integer values. However, this is Hidden to the programmer through the data type (class)sc_time. Due to the limits of the underlying implementation of time, we cannot represent continuous time, but only discrete time. Therefore, in SystemC there is a minimum representable time quantum, called the time resolution. This can be set by the user, as we shall demonstrate in later parts of this document. Note that this time resolution limits the maximum representable time, because the underlying data type representing the time is a 64-bit integer value. Thus, any time value smaller than the time resolution will be rounded to zero.

2.2.2 MODULES :

Modules are the basic building blocks in SystemC. A module comprises ports, concurrent processes, and some internal data structures and channels that represent the model state and the communication between processes. A module can also use another module in a hierarchy. A module is described with the macro SC_MODULE. Actually, the macro SC_MODULE (Module) expands to class Module: public sc_module.

2.2.3 PROCESSES AND EVENTS :

The functionality in SystemC is achieved with processes. As opposed to C++ functions, which are used to model sequential system behavior, processes provide the mechanism for simulating concurrency. A process is a C++ member function of the SystemC module. The function is declared to be a process in the constructor. There are two macros (SC_METHOD and SC_THREAD) which can be used in the constructor for such process registration with the simulation kernel.

2.2.4 PORTS, INTERFACES AND CHANNELS :

Communication between processes inside different modules is accomplished using ports, interfaces and channels. The port of a module is the object through which the process accesses a channels interface. The interface defines the set of access functions for a channel while the channel itself provides the implementation of these functions. At elaboration time the ports of a module are connected (bound) to designated channels. The interface, port, channel structure provides for great flexibility in modeling communication and in model refinement.

2.3 PYTHON(SCRIPTING)

Python in the project was just used for scripting[8]. Python is a high-level, interpreted, interactive and object oriented-scripting language. Python was designed to be highly readable which uses English keywords frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.

- **Python is interpreted:** This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

2.3.1 PYTHON FEATURES:

Python's feature highlights include:

- **Easy-to-learn:** Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.
- **Easy-to-read:** Python code is much more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.
- **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of UNIX.

- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

2.3.2 Why PYTHON?:[8]

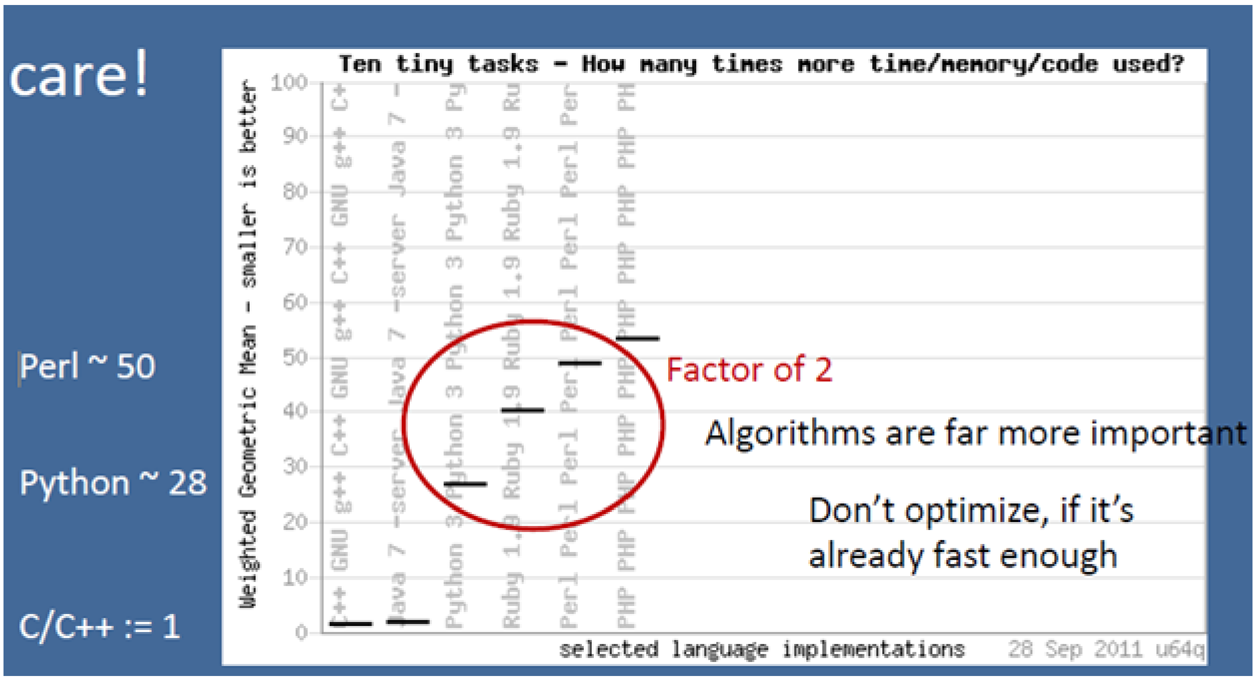


Figure 2.4: Comaparison between Python and other languages

2.4 TRANSACTION LEVEL MODELING (TLM2.0)

Transaction level modeling[2] in SystemC involves communication between SystemC processes using function calls. The focus of TLM is on the communication between the processes rather than the algorithms performed by the processes themselves, so the processes shown in this tutorial will be rather trivial. We assume that in a model of system behavior, some of the SystemC processes will produce data, others will consume data, some will initiate communication, others will passively respond to communication initiated by others. The focus of OSCI TLM-2.0 in particular is the modeling of on-chip memory-mapped busses. This does not mean that TLM-2.0 is dedicated exclusively to memory-mapped busses, just that this is where most of the features are focused. TLM-2.0 has a layered structure, with the lower layers being more flexible and general, and the upper layers being specific to bus modeling. In future, the standard may be re-oriented toward other styles of communication as they emerge, the obvious direction being network-on-chip (NoC) architectures.

TLM-2.0 consists of a set of core interfaces, the global quantum, initiator and target sockets, the generic payload and base protocol, and the utilities. The TLM-1 core interfaces, analysis interface and analysis ports are also included, although they are separate from the main body of the TLM-2.0 standard. The TLM-2.0 core interfaces consist of the blocking and non-blocking transport interfaces, the direct memory interface (DMI), and the debug transport interface. The generic payload supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols whilst maximizing interoperability. TLM 1.0 has three shortcomings with respect to the modeling of memory-mapped buses and other on-chip communication networks:

a. TLM-1 has no standard transaction class, so each application has to create its own non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2.0 addresses this shortcoming with the generic payload.

b. TLM-1 has no support for timing annotation, so no standard way of communicating timing information between models. TLM-1 models would typically implement delays by calling wait, which slows down simulation. TLM-2.0 addresses this shortcoming with the addition of timing annotation to the blocking and non-blocking transport interface.

c. The TLM-1 interfaces require all transaction objects and data to be passed by value or const reference, which slows down simulation. Some applications work around this restriction by embedded pointers in transaction objects, but this is non-standard and non-interoperable. TLM-2.0 addresses this shortcoming with transaction objects whose lifetime extends across several transport calls, supported by a new transport interface.

The TLM-2.0 classes are layered on top of the SystemC class library as shown in the diagram below. For maximum interoperability, and particularly for memory-mapped bus modeling, it is recommended that the TLM-2.0 core interfaces, sockets, generic payload and base protocol be used together in concert. These classes are known collectively as the interoperability layer. In cases where the generic payload is inappropriate, it is possible for the core interfaces and the initiator and target sockets, or the core interfaces alone, to be used with an alternative transaction type. It is even technically possible for the generic payload to be used directly with the core interfaces without the initiator and target sockets, although this approach is not recommended.

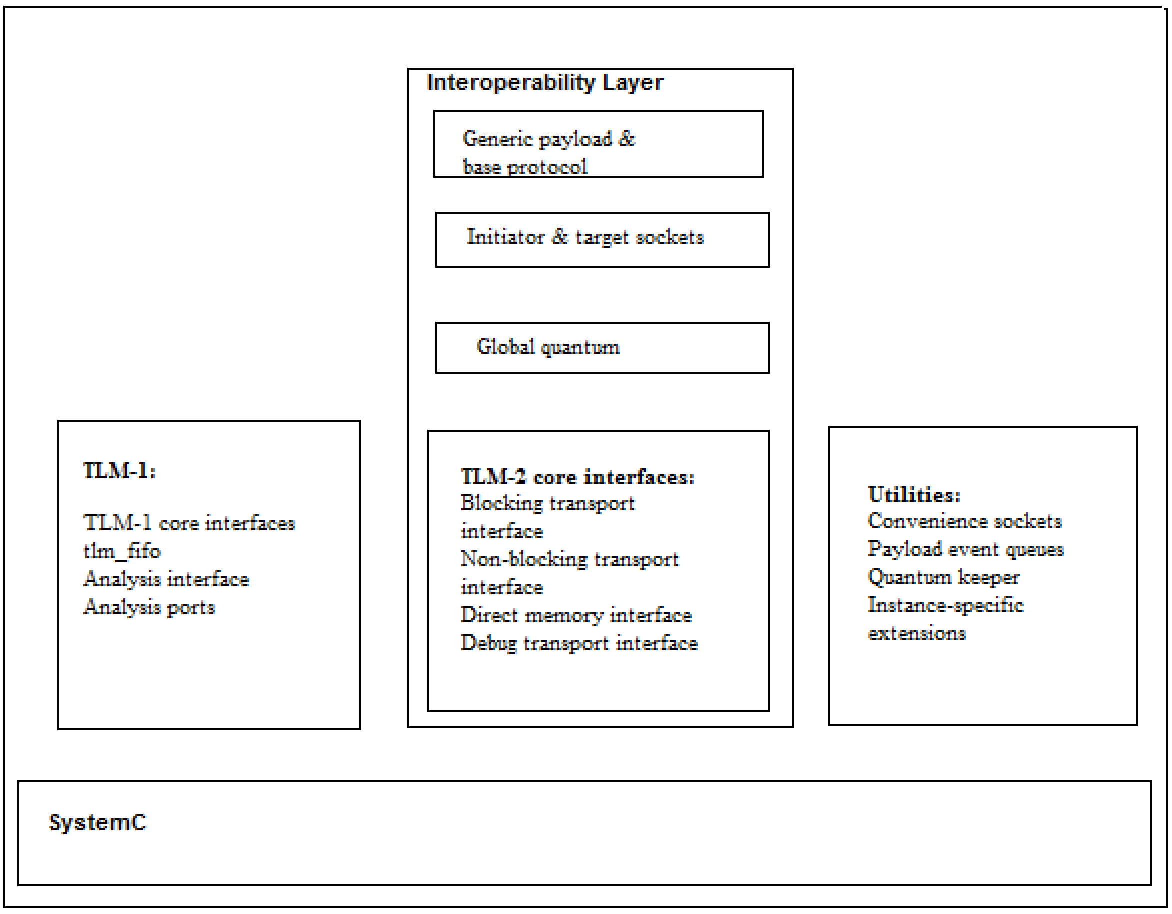


Figure 2.5: TLM2.0 classes

Transaction Level Modeling (TLM) is motivated by a number of practical problems. These include:

- Providing an early platform for software development
- System Level Design Exploration and Verification
- The need to use System Level Models in Block Level Verification.

A commonly accepted industry standard for TLM would help to increase the productivity of software engineers, architects, and implementation and verification engineers. However, the improvement in productivity promised by such a standard can only be achieved if the standard meets a number of criteria:

- It must be easy, efficient and safe to use in a concurrent environment.
- It must enable reuse between projects and between abstraction levels within the same project.
- It must easily model hardware, software and designs which cross the hardware / software boundary.
- It must enable the design of generic components such as routers and arbiters.

Since the release of version 2.0, it has been possible to do TLM using SystemC. However, the lack of established standards and methodologies has meant that each TLM effort has had to invent its own methodologies and APIs to do TLM. In addition to the cost of reinventing the wheel, these methodologies all differed slightly, making IP exchange difficult.

2.4.1 Keypoints

There are key concepts required to understand this proposal:

- Interfaces
- Blocking vs. Non-Blocking
- Bidirectional vs. Uni Directionl
- Sockets
- Coding styles

a. **Interface :** The emphasis on interfaces rather than implementation flows from the fact that SystemC is a C++ class library, and that C++ (when used properly) is an object orientated language. First we need to rigorously define the key interfaces, and then we can go on to discuss the various ways these may be implemented in a TLM design. It is crucial for the reader to understand that the TLM interface classes form the heart of the TLM standard, and that the implementations of those interfaces (e.g. tlm_fifo) are not as central. In SystemC, all interfaces should inherit from the class sc_interface.

b. **Blocking and Non-Blocking:** In SystemC, there are two basic kinds of processes: SC_THREAD and SC_METHOD. The key difference between the two is that it is possible to suspend an SC_THREAD by calling wait(). SC_METHODs on the other hand can only be synchronized by making them sensitive to an externally defined sc_event. Calling wait() inside an SC_METHOD leads to a runtime error. Using SC_THREAD is in many ways more natural, but it is slower because wait() induces a context switch in the SystemC scheduler. Using SC_METHOD is more constrained but more efficient, because it avoids the context switching.

c. **Bidirectional and Unidirectional Transfers:** Some common transactions are clearly bidirectional, for example a read across a bus. Other transactions are clearly unidirectional, as is the case for most packet based communication mechanisms. Where there is a more complicated protocol, it is always possible to break it down into a sequence of bidirectional or unidirectional transfers. For example, a complex bus with address, control and data phases may look like a simple bidirectional read/write bus at a high level of abstraction, but more like a sequence of pipelined unidirectional transfers at a more detailed level. Any TLM standard must have both bidirectional and unidirectional interfaces. The standard should have a common look and feel for bidirectional and unidirectional interfaces, and it should be clearly shown how the two relate.

d. **Initiators, Targets, and Sockets:** In TLM-2.0, an initiator is a module that initiates new transactions, and a target is a module that responds to transactions initiated by other modules. A transaction is a data structure (a C++ object) passed between initiators and targets using function calls. The same module can act both as an initiator and as a target, and this would typically be the case for a model of an arbiter, a router, or a bus. In order to pass transactions between initiators and targets, TLM-2.0 uses sockets. An initiator sends transactions out through an initiator socket, and a target receives incoming transactions through a target socket. A module that merely forwards transactions without modifying their content is known as an interconnect component. An interconnect component would have both a target socket and an initiator socket.

e. **Coding styles:** A coding style is a set of programming language idioms that work well together, not a specific abstraction level or software programming interface. For simplicity and clarity, this document restricts itself to elaborating two specific named coding styles; loosely-timed and approximately-timed. By their nature the coding styles are not precisely defined, and the rules governing the TLM-2.0 core interfaces are defined independently from these coding styles. In principle, it would be possible to define other coding styles based on the TLM-1 and TLM-2.0 mechanisms.

    (1) **Untimed coding style** TLM-2.0 does not make explicit provision for an untimed coding style, because all contemporary bus-based systems require some notion of time in order to model software running on one or more embedded processors. However, untimed modeling is supported by the TLM-1 core interfaces. (The term untimed is sometimes used to refer to models that contain a limited amount of timing information of unspecified accuracy. In TLM-2.0, such models would be termed loosely-timed.)

    (2) **Loosely-timed coding style and temporal decoupling** The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to the call to and return from the blocking transport function. In the case of the base protocol, the first timing point marks the beginning of the request, and the second marks the beginning of the

response. These two timing points could occur at the same simulation time or at different times. The loosely-timed coding style is appropriate for the use case of software development using a virtual platform model of an MPSoC, where the software content may include one or more operating systems. The loosely-timed coding style supports the modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on the target machine. The loosely-timed coding style also supports temporal decoupling, where individual SystemC processes are permitted to run ahead in a local time warp without actually advancing simulation time until they reach the point when they need to synchronize with the rest of the system. Temporal decoupling can result in very fast simulation for certain systems because it increases the data and code locality and reduces the scheduling overhead of the simulator. Each process is allowed to run for a certain time slice or quantum before switching to the next, or instead may yield control when it reaches an explicit synchronization point.

(3) **Approximately-timed coding style** The approximately-timed coding style is supported by the non-blocking transport interface, which is appropriate for the use cases of architectural exploration and performance analysis. The non-blocking transport interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction. For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the base protocol there are exactly four timing points marking the beginning and the end of the request and the beginning and the end of the response. Specific protocols may need to add further timing points, which may possibly cause the loss of direct compatibility with the generic payload. Although it is possible to use the non-blocking transport interface with just two phases to indicate the start

and end of a transaction, the blocking transport interface is generally preferred for loosely-timed modeling. The approximately-timed coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step with the SystemC scheduler. Process interactions are annotated with specific delays. To create an approximately-timed model, it is generally sufficient to annotate delays representing the data transfer times for write and read commands and the latency of the target. For the base protocol, the data transfer times are effectively the same as the minimum initiation interval or accept delay between two successive requests or two successive responses. The annotated delays are implemented by making calls to the SystemC scheduler, that is, wait(delay) or notify(delay).

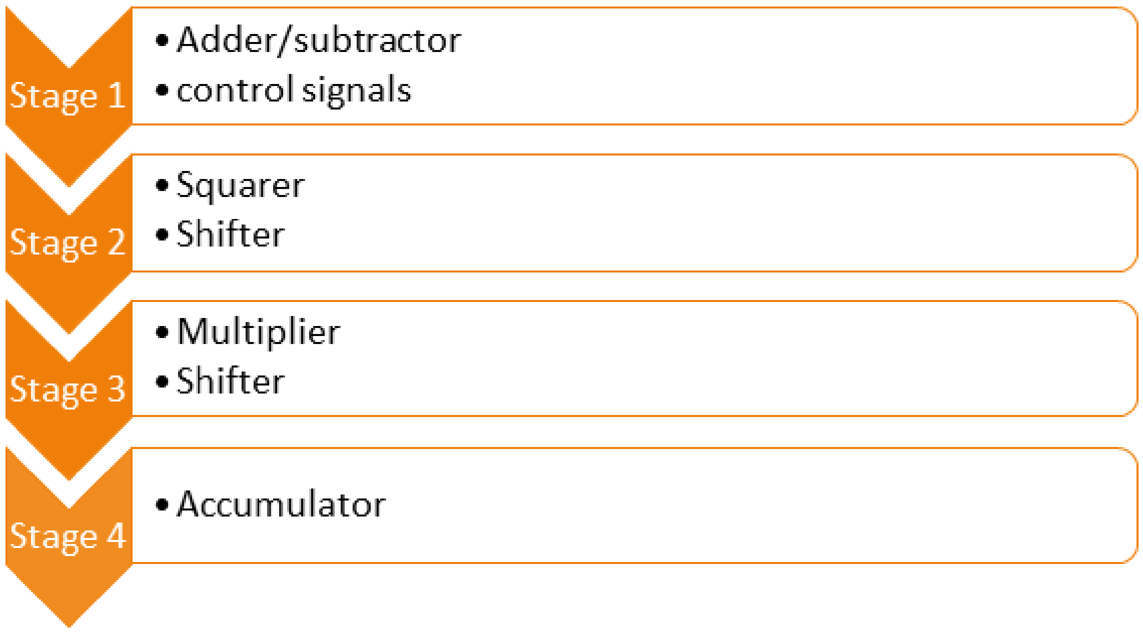
Chapter 3

Modeling and verification of PE

In Percsense system level architechture,there are different classifier stages, one of which is Feature classifier. The feature classifier encompasses processing engine or PE module which is responsible for parallel computations. There are 16 PEs in all in the architechture, processing parallely.Each processing element has a control signal that is coming in and determines whether it is part of the Data Path or is it is passed through.The PE can run various different Machine learning algorithms like DBNN, CNN,RBM used for feature extraction.

The objective of this chapter hence, focusses on modeling and verification of these PEs. The language used for modeling and verification is IEEE-1666 SystemC. Now, Let's discuss in depth what actually is processing engine, how it works, what all it supports, why is it written only in systemC in the further sections to come.

3.1 PROCESSING ENGINE FLOW DIAGRAM



Properties of PE [1]:

- Based on Single instruction multiple data (SIMD) architecture.
- Has mathematics which is common to various machine learning algorithms.
- Has fixed point arithmetic.
- Implements Math required for some common machine learning algorithms.
- Fixed point mathematics.
- Multiple such PE blocks to achieve SIMD.

The PE is actually a small part in the Percsense system level architechture which drives all the machine learning algorithms. PEs exist in the from of what is called as

Tile. Each tile consists of 16 PEs working parallely and hence, SIMD architechture. The basic architecture of PE encompasses four pipeline stages. Each stage or layer has an arithematic block assigned to it.Also, there are multiplexers and and register storage at every layer. As seen in the flow diagram, the first stage consists of adder/subtraction, its the control signal which decides what to choose according to the algorithm demand.Similarly, next is the squarer shifter , then the multiplier then accumulator.

3.2 Machine learning algorithms in PE

As mentioned in the above section that PE implements a Math required for some Machine learning algorithms. These algorithms include GMM,DBNN,CDBNN etc. The main advantage being that all the above mentioned machine learning algorithms can be implemented on the same PE just by bypassing or including some blocks according to the need of the algorithm. For instance,GMM has an equation as follows:

$$((A-B)^2 \times C) + ((A-B_2)^2 \times C) + \_\_\_ \quad (3.1)$$

So, as the equation requires an adder/Subtractor, a multiplier, a squarer and a MAC, All the four blocks of PE will be in action.Now, let's take another example of DBNN algorithm. The equation defining DBNN algorithm is as follows:

$$w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 + \_\_\_ \qquad (3.2)$$

As evident from the equation, it has algebra like multiplication and MAC, So the blocks of PE which will come into picture here will be multiplier and the MAC. The other blocks that is, adder/ subtractor and squarer can be bypassed using control signals at the very designing stage.

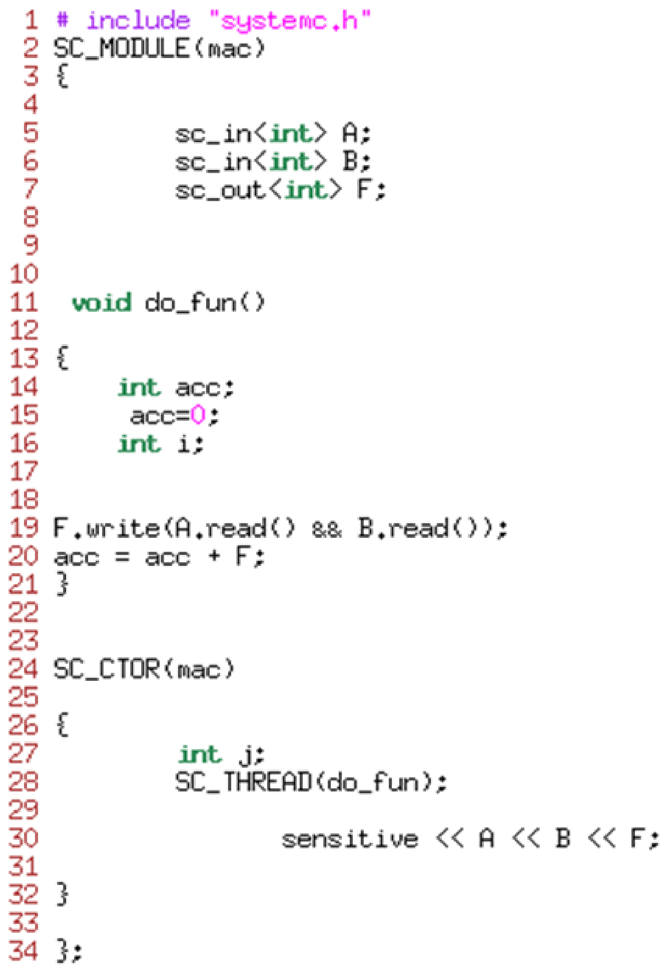
Thus, the PE is designed for such machine learning algorithms and the algorithms can be changed just by changing control signals at the design level.
The design and verification of the PE module is done in IEEE-1666 standard SystemC language. Some of the ramp up task done in SystemC are discussed in sections to come. To get to the verification and modeling of PE, these ramp up tasks were the stepping stone.

3.3 Ramp up tasks in SYSTEMC

- MAC (Multiply and accumulate)[4].
- E-xor gate using NAND gate.
- FIR Filter [6].
- Design and verification of PE without any control signals.
- Code for PE using control signals.
- Verification using Python Script.
- Writing a Make file.
- Debugging different compile and runtime errors.
- Debugging using GDB debugger[1].
- Verification of the Processing Engine(PE) for Fixed point values.
- Adding different control signals.
- Addition of nonlinearity (Sigmoid) and verification for the same.

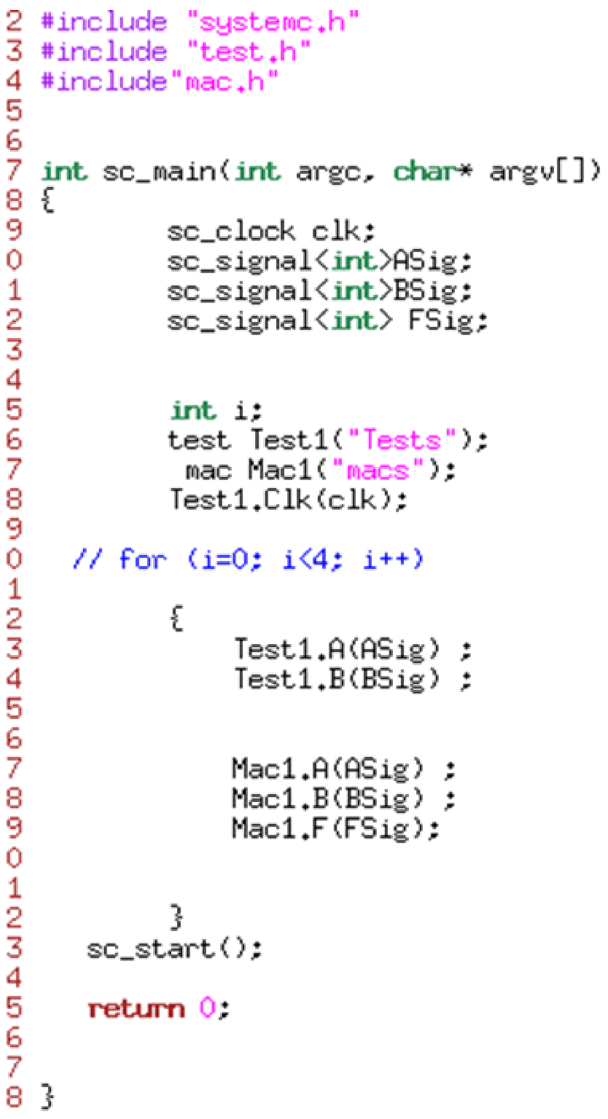
3.3.1 CODE SNIPPET

a. Multiply and Accumulate(MAC)



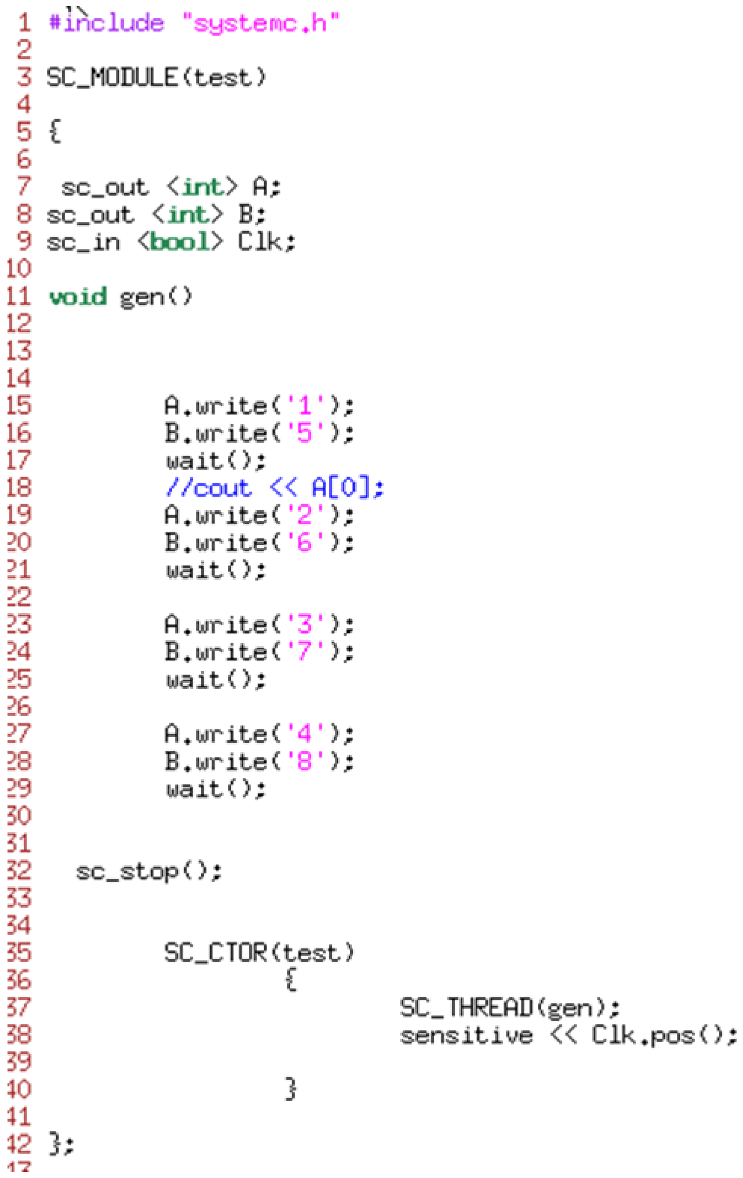
```
# include "systemc.h"
SC_MODULE(mac)
{

        sc_in<int> A;
        sc_in<int> B;
        sc_out<int> F;



 void do_fun()

{
    int acc;
     acc=0;
    int i;


F.write(A.read() && B.read());
acc = acc + F;
}


SC_CTOR(mac)

{
        int j;
        SC_THREAD(do_fun);

                sensitive << A << B << F;

}

};
```

Figure 3.1: mac.h(The header file)



```
#include "systemc.h"
#include "test.h"
#include"mac.h"


int sc_main(int argc, char* argv[])
{
        sc_clock clk;
        sc_signal<int>ASig;
        sc_signal<int>BSig;
        sc_signal<int> FSig;


        int i;
        test Test1("Tests");
         mac Mac1("macs");
        Test1.Clk(clk);

  // for (i=0; i<4; i++)

        {
            Test1.A(ASig) ;
            Test1.B(BSig) ;


            Mac1.A(ASig) ;
            Mac1.B(BSig) ;
            Mac1.F(FSig);


        }
    sc_start();

    return 0;


}
```

Figure 3.2: Mac.cpp (The cpp file)



```
#include "systemc.h"

SC_MODULE(test)

{

 sc_out <int> A;
sc_out <int> B;
sc_in <bool> Clk;

void gen()



        A.write('1');
        B.write('5');
        wait();
        //cout << A[0];
        A.write('2');
        B.write('6');
        wait();

        A.write('3');
        B.write('7');
        wait();

        A.write('4');
        B.write('8');
        wait();


  sc_stop();


        SC_CTOR(test)
                {
                        SC_THREAD(gen);
                        sensitive << Clk.pos();

                }

};
```

Figure 3.3: Test.h (Test bench file)

Code explanation:

a. **In (.h) file:**The header file systemc.h has all the systemC components i.e.; ports etc.

b. Declaration of the MODULE.

c. Declaration of input and output ports.

d. Function definition.

e. Declaring a constructor.

f. **In (.cpp) file:**Binding of ports and signals.

g. **In (test) file:**Passing values through test bench.

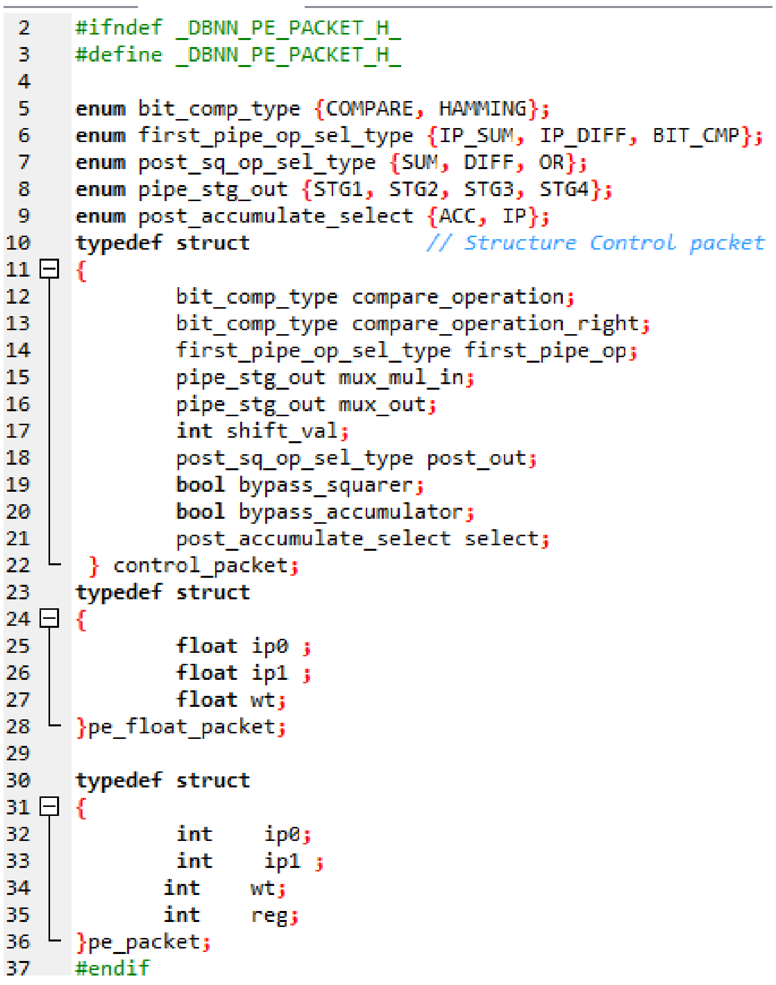
So, inshort the summary is that the code explains itself. It has the header file in which all all the omponents like ports are declared, modules are formed, functions and constructors are declared. In the cpp file, all the signals are declared and are bound to the modules. The test file is used to pass values to the ports. It is actually a testbench.

3.3.2 MODELING AND VERIFIDCATION OF THE PROCESSING ENGINE MODEL

- **Modeling of DRIVER**

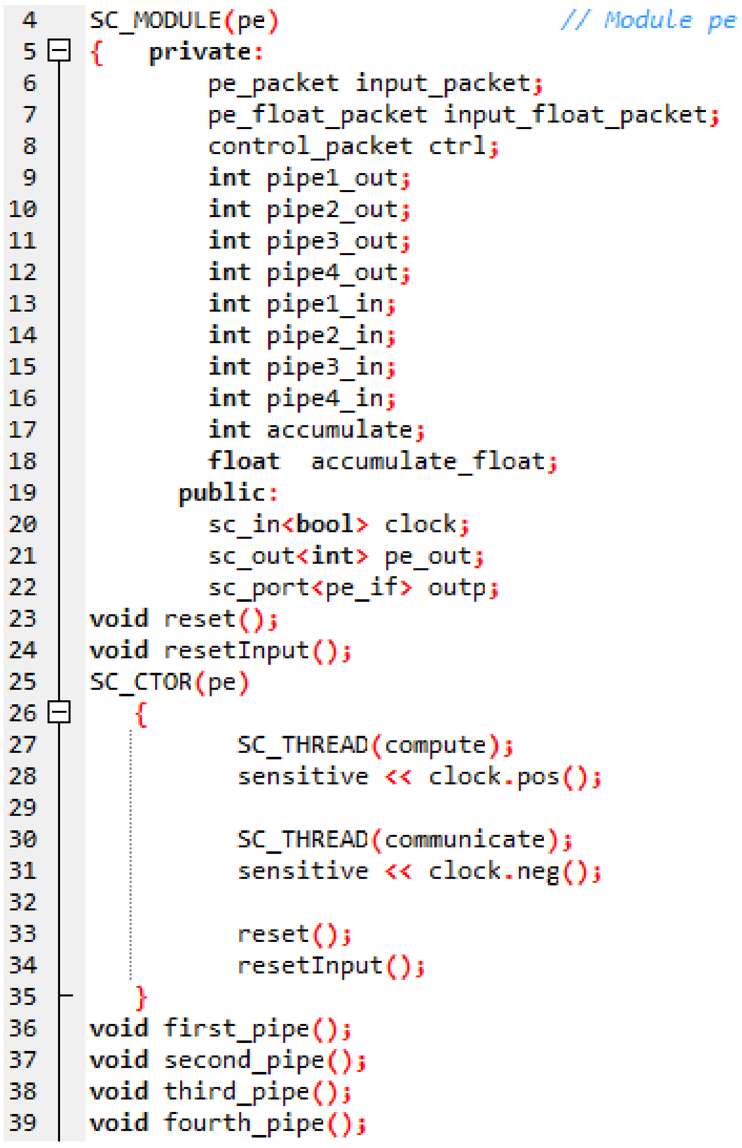
  Under this section, the actual modeling of the driver driving the PE was done.The driver drives the PE. As explained earlier, the PE has two input ports and four layers. The driver is responsible for supplying input and weights. So, the driver has the following files:

1. **pe_packet.h** : Wherein all the input and the weight packets are declared.
2. **pe.h** : Wherein all the layers and the functions according to those layers are declared.
3. **pe.cpp** : Wherein all the layers and the functions according to those layers are defined.
4. **main.cpp**: wherein all the ports are bound.
5. **driver.h** : A driver body is created.
6. **driver.cpp** : The driver defined.



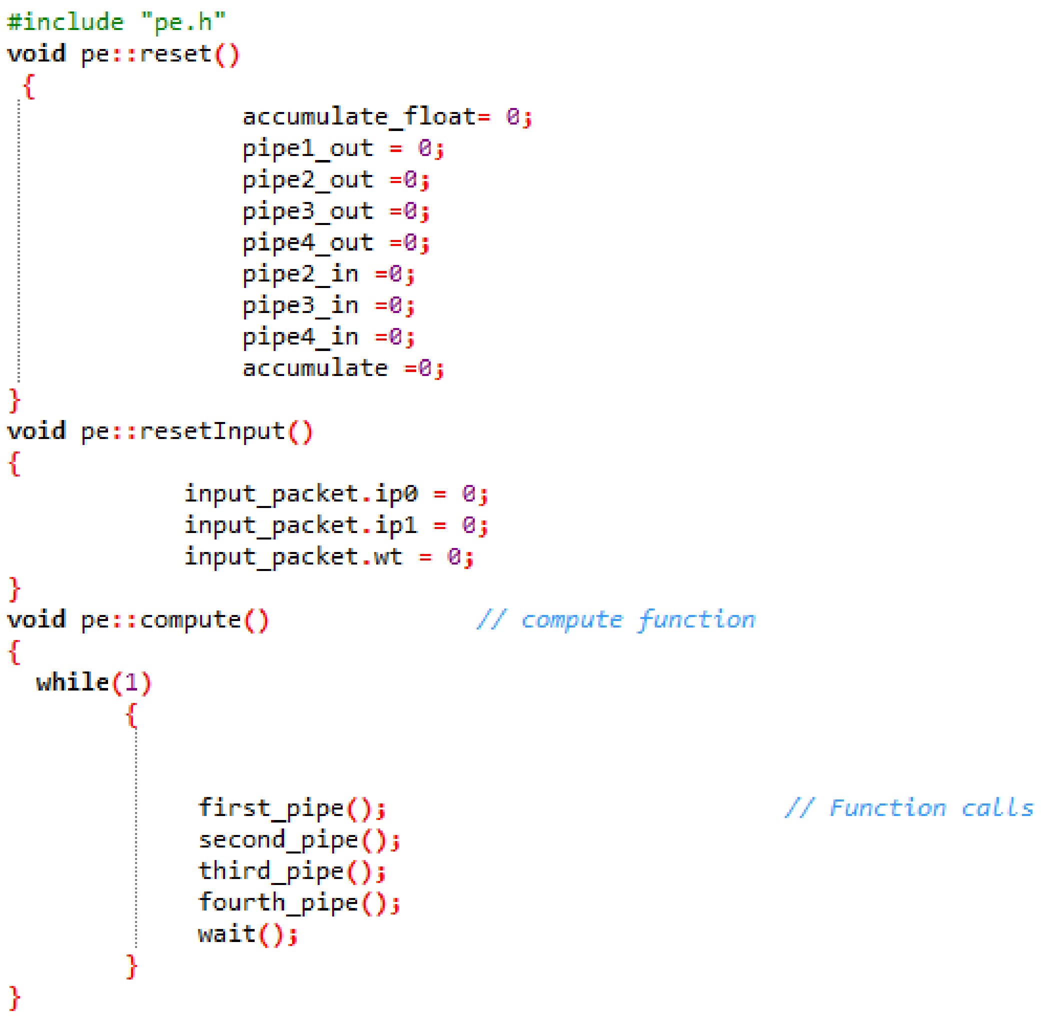
```
#ifndef _DBNN_PE_PACKET_H_
#define _DBNN_PE_PACKET_H_

enum bit_comp_type {COMPARE, HAMMING};
enum first_pipe_op_sel_type {IP_SUM, IP_DIFF, BIT_CMP};
enum post_sq_op_sel_type {SUM, DIFF, OR};
enum pipe_stg_out {STG1, STG2, STG3, STG4};
enum post_accumulate_select {ACC, IP};
typedef struct                     // Structure Control packet
{
        bit_comp_type compare_operation;
        bit_comp_type compare_operation_right;
        first_pipe_op_sel_type first_pipe_op;
        pipe_stg_out mux_mul_in;
        pipe_stg_out mux_out;
        int shift_val;
        post_sq_op_sel_type post_out;
        bool bypass_squarer;
        bool bypass_accumulator;
        post_accumulate_select select;
 } control_packet;
typedef struct
{
        float ip0 ;
        float ip1 ;
        float wt;
}pe_float_packet;

typedef struct
{
        int    ip0;
        int    ip1 ;
       int    wt;
       int    reg;
}pe_packet;
#endif
```

Figure 3.4: pe_packet.h



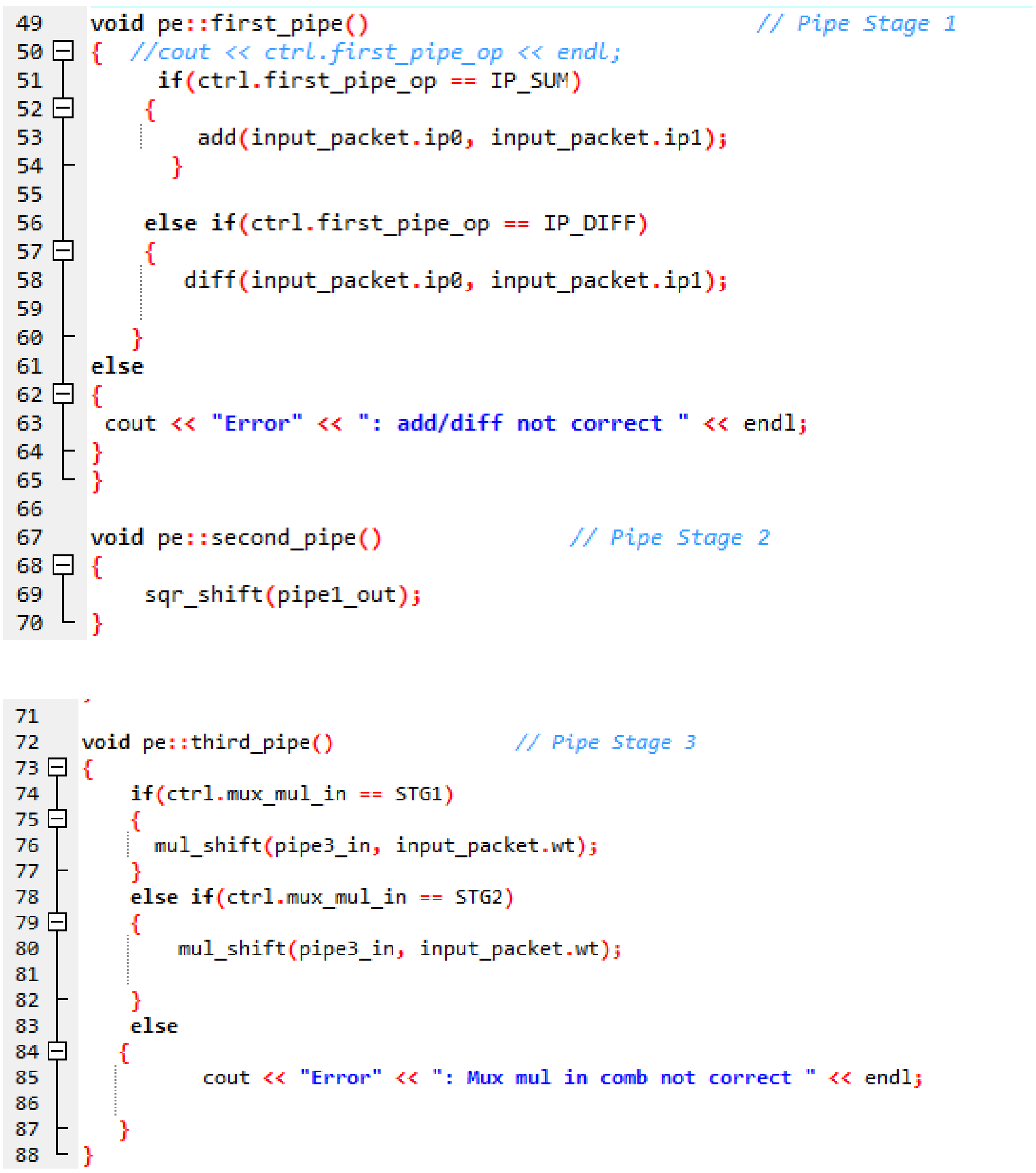
```
SC_MODULE(pe)                          // Module pe
{   private:
        pe_packet input_packet;
        pe_float_packet input_float_packet;
        control_packet ctrl;
        int pipe1_out;
        int pipe2_out;
        int pipe3_out;
        int pipe4_out;
        int pipe1_in;
        int pipe2_in;
        int pipe3_in;
        int pipe4_in;
        int accumulate;
        float  accumulate_float;
      public:
        sc_in<bool> clock;
        sc_out<int> pe_out;
        sc_port<pe_if> outp;
void reset();
void resetInput();
SC_CTOR(pe)
   {
          SC_THREAD(compute);
          sensitive << clock.pos();

          SC_THREAD(communicate);
          sensitive << clock.neg();

          reset();
          resetInput();
   }
void first_pipe();
void second_pipe();
void third_pipe();
void fourth_pipe();
```

Figure 3.5: pe.h



```
#include "pe.h"
void pe::reset()
 {
                accumulate_float= 0;
                pipe1_out = 0;
                pipe2_out =0;
                pipe3_out =0;
                pipe4_out =0;
                pipe2_in =0;
                pipe3_in =0;
                pipe4_in =0;
                accumulate =0;
}
void pe::resetInput()
{
            input_packet.ip0 = 0;
            input_packet.ip1 = 0;
            input_packet.wt = 0;
}
void pe::compute()              // compute function
{
  while(1)
        {

             first_pipe();                       // Function calls
             second_pipe();
             third_pipe();
             fourth_pipe();
             wait();
        }
}
```

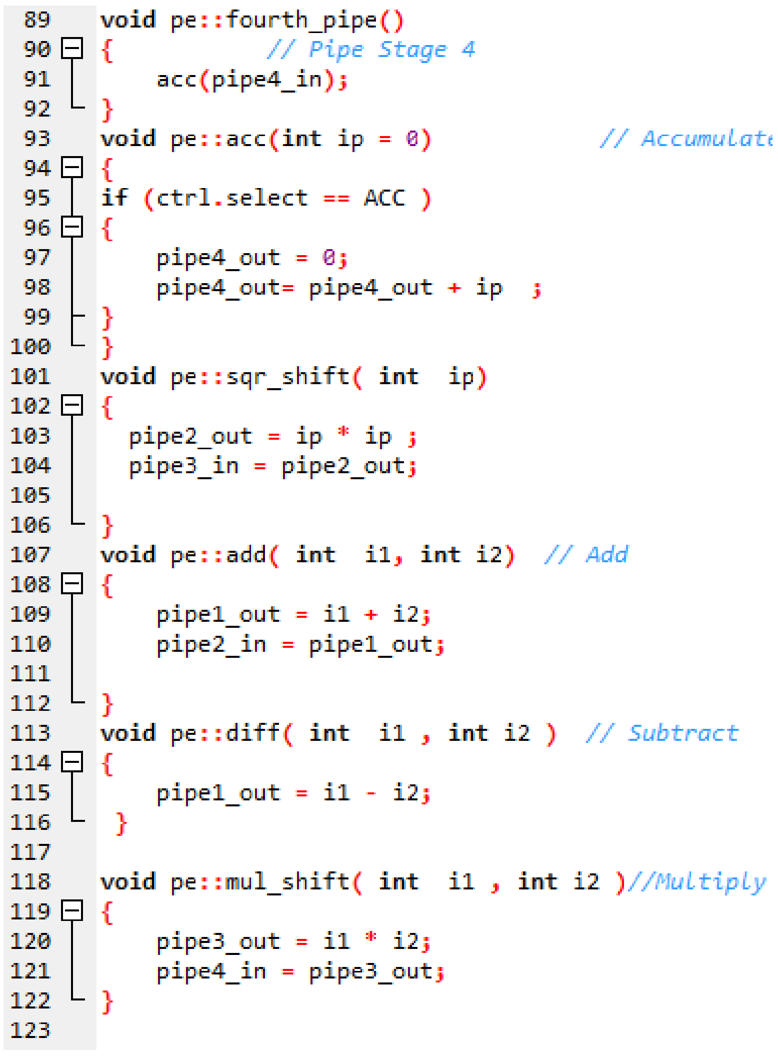
Figure 3.6: pe.cpp(reset all the pipe stages and inputs)



```
void pe::first_pipe()                                    // Pipe Stage 1
{  //cout << ctrl.first_pipe_op << endl;
     if(ctrl.first_pipe_op == IP_SUM)
    {
        add(input_packet.ip0, input_packet.ip1);
      }

    else if(ctrl.first_pipe_op == IP_DIFF)
    {
       diff(input_packet.ip0, input_packet.ip1);

    }
else
{
 cout << "Error" << ": add/diff not correct " << endl;
}
}

void pe::second_pipe()                        // Pipe Stage 2
{
    sqr_shift(pipe1_out);
}
```

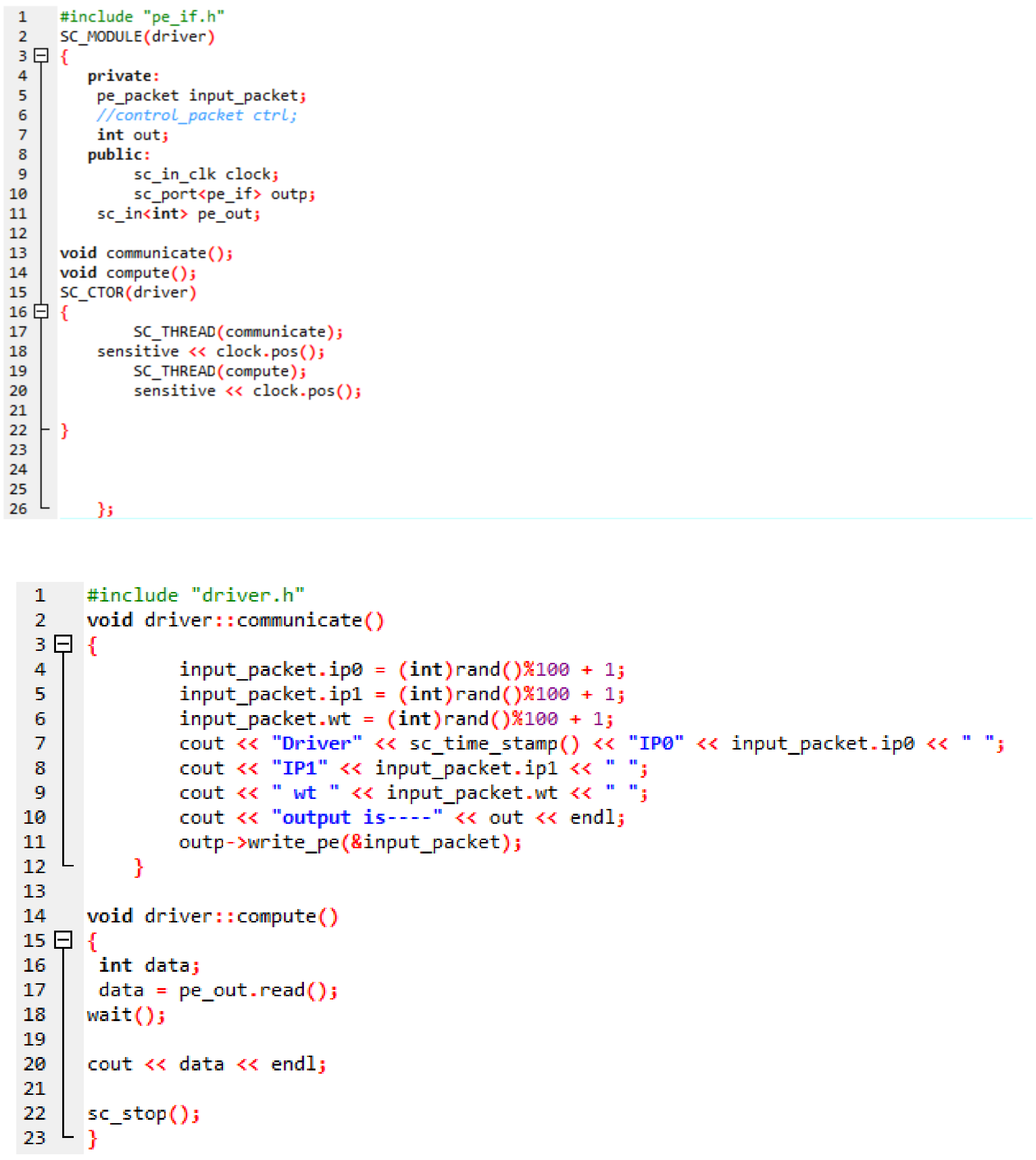
```

void pe::third_pipe()                    // Pipe Stage 3
{
    if(ctrl.mux_mul_in == STG1)
    {
      mul_shift(pipe3_in, input_packet.wt);
    }
    else if(ctrl.mux_mul_in == STG2)
    {
        mul_shift(pipe3_in, input_packet.wt);

    }
    else
   {
           cout << "Error" << ": Mux mul in comb not correct " << endl;

   }
}
```

Figure 3.7: pe.cpp(Define stg1,stg2 and stg3)



```
void pe::fourth_pipe()
{           // Pipe Stage 4
    acc(pipe4_in);
}
void pe::acc(int ip = 0)            // Accumulat
{
if (ctrl.select == ACC )
{
    pipe4_out = 0;
    pipe4_out= pipe4_out + ip   ;
}
}
void pe::sqr_shift( int  ip)
{
  pipe2_out = ip * ip ;
  pipe3_in = pipe2_out;

}
void pe::add( int  i1, int i2)  // Add
{
    pipe1_out = i1 + i2;
    pipe2_in = pipe1_out;

}
void pe::diff( int  i1 , int i2 )  // Subtract
{
    pipe1_out = i1 - i2;
 }

void pe::mul_shift( int  i1 , int i2 )//Multiply
{
    pipe3_out = i1 * i2;
    pipe4_in = pipe3_out;
}

```

Figure 3.8: pe.cpp(Define all the pipelines)



```
#include "pe_if.h"
SC_MODULE(driver)
{
    private:
     pe_packet input_packet;
     //control_packet ctrl;
     int out;
    public:
         sc_in_clk clock;
         sc_port<pe_if> outp;
     sc_in<int> pe_out;

void communicate();
void compute();
SC_CTOR(driver)
{
         SC_THREAD(communicate);
     sensitive << clock.pos();
         SC_THREAD(compute);
         sensitive << clock.pos();

}



    };
```

```
#include "driver.h"
void driver::communicate()
{
        input_packet.ip0 = (int)rand()%100 + 1;
        input_packet.ip1 = (int)rand()%100 + 1;
        input_packet.wt = (int)rand()%100 + 1;
        cout << "Driver" << sc_time_stamp() << "IP0" << input_packet.ip0 << " ";
        cout << "IP1" << input_packet.ip1 << " ";
        cout << " wt " << input_packet.wt << " ";
        cout << "output is----" << out << endl;
        outp->write_pe(&input_packet);
    }

void driver::compute()
{
 int data;
 data = pe_out.read();
wait();

cout << data << endl;

sc_stop();
}
```

Figure 3.9: Driver

Code explaination: The first part of the code defines three structures namely control_packet,pe_float_packet and pe_packet. Each struct has different specific work or component.**control_packet** encapsulates all the control signals neccessary for driving PE.**pe_float_packet** defines two inputs and a weight variable of type FLOAT.**pe_packet** defines two inputs and a weight variable of type INTEGER(int).

The second part of the code is actually a header file which declares all the input and output ports, variables and functions. Next is the cpp file which encapsulates the defination of all the functions declared inthe header file. Next comes the "Driver".Driver is designed in order to provide values or inputs to the main file. And thus, verify the functioning of the PE.

- **Verification of PE**

  The verification processes is as:

  a. For the verification process a Python script is written.

  b. The Python script first reads an output from the SystemC file pe.cpp and writes the output in the file testout.txt.

  c. The Python script reads input from the SystemC file Driver.cpp.

  d. The Python script now has an algorithm which uses inputs and does operation on the inputs.

  e. Finally it compares the output from the algorithm and the output from SystemC file.

3.3.3 REASON BEHIND SYSTEMC

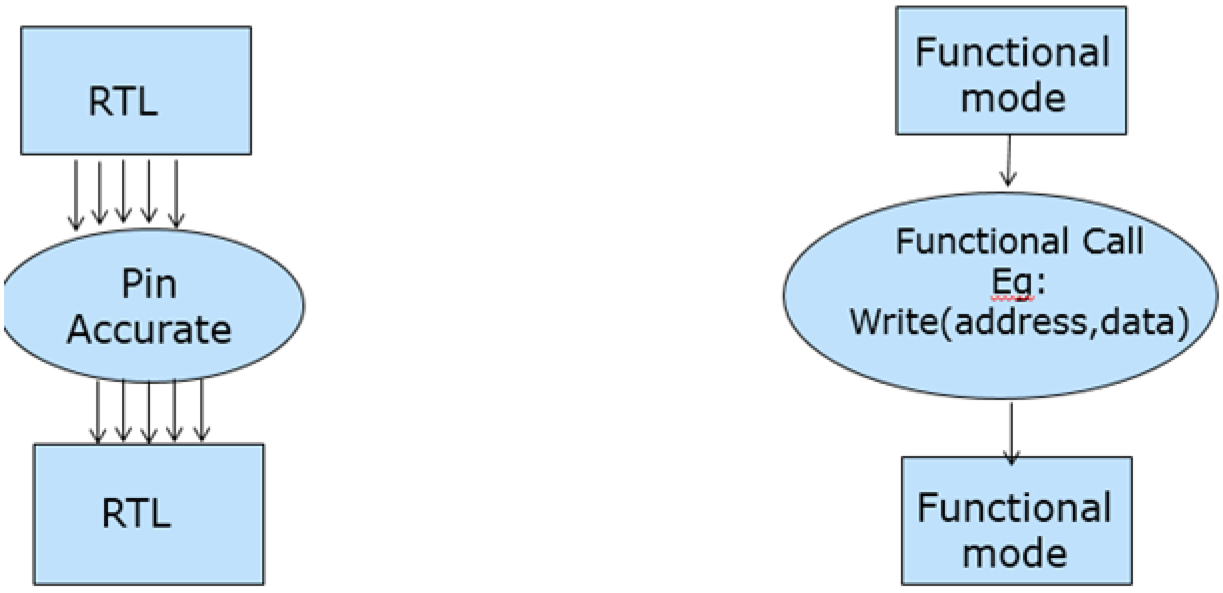
The question arises why only SystemC is used for designing of the IP? Why not any other HDL like Verilog or System Verilog?[9]
The answer to this question is quite simple:

SystemC:

- Improves simulation performance(Consumes less time).
- Increased Productivity.
- Is used for TLM modeling which allows higher level of abstraction than RTL.

Difference between TLM and RTL:

- RTL (Register Transfer level) is pin accurate which means all the communication is using pin.
- While TLM (Transaction level modeling) depends on functional calls.



3.4 Verif using PYTHON

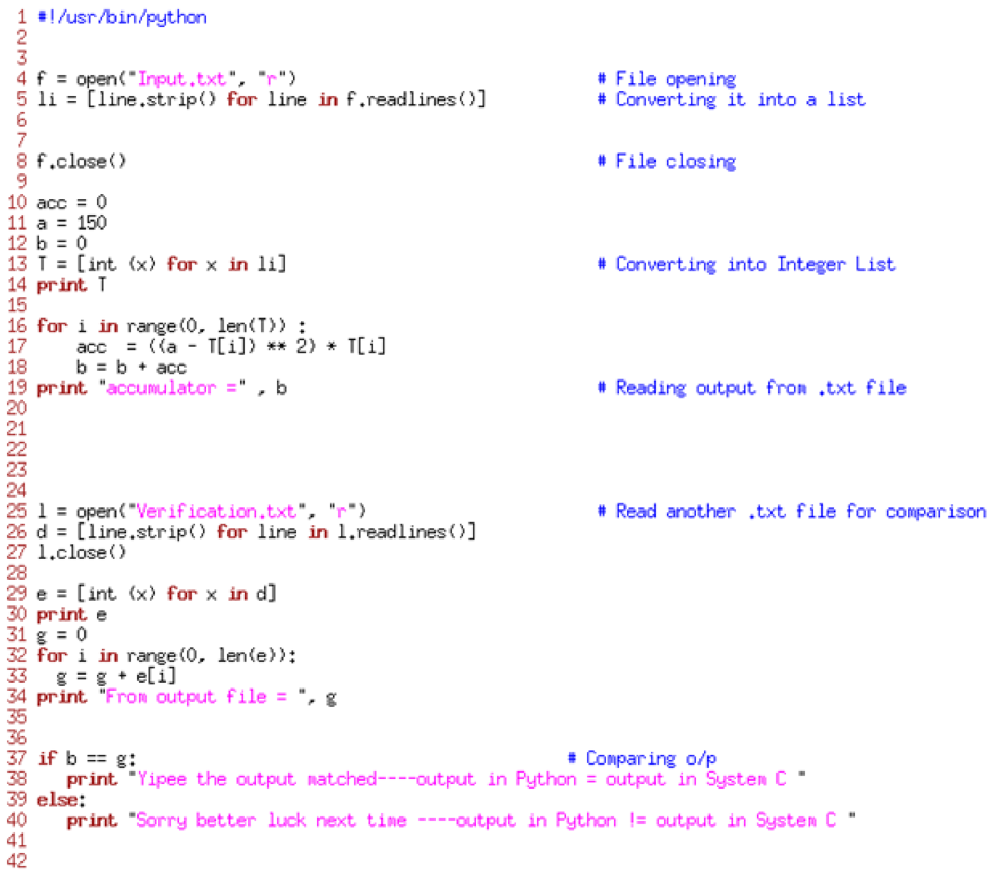
Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Pythons elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms. In my project up till now, Python has been used for three purposes:

a. For verifying algorithm in SystemC.

b. For comparing C and SystemC outputs of the Processing Engine.

c. For non-linearity in Processing engine (PE).

Code Snippet (1):



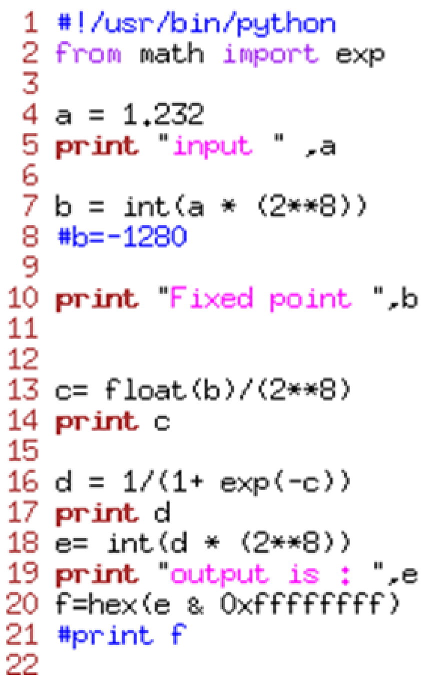
```
#!/usr/bin/python


f = open("Input.txt", "r")                          # File opening
li = [line.strip() for line in f.readlines()]       # Converting it into a list


f.close()                                           # File closing

acc = 0
a = 150
b = 0
T = [int (x) for x in li]                           # Converting into Integer List
print T

for i in range(0, len(T)) :
    acc  = ((a - T[i]) ** 2) * T[i]
    b = b + acc
print "accumulator =" , b                           # Reading output from .txt file





l = open("Verification.txt", "r")                   # Read another .txt file for comparison
d = [line.strip() for line in l.readlines()]
l.close()

e = [int (x) for x in d]
print e
g = 0
for i in range(0, len(e)):
  g = g + e[i]
print "From output file = ", g


if b == g:                                        # Comparing o/p
   print "Yipee the output matched----output in Python = output in System C "
else:
   print "Sorry better luck next time ----output in Python != output in System C "


```

Code Explanation:

- Here **input.txt** file is read.
- Then these inputs are given to algorithm and output from algorithm is saved.
- Another file **Verification.txt** is opened for comparison.
- Now the Python output is compared with the output in file **Verification.txt** which is SystemC output.

Code Snippet (2): This code cannot be provided as it is **INTEL CONFIDENTIAL.**

Code Snippet (3): The non- linearity in processing engine is checked in the below code (using sigmoid function)



```
#!/usr/bin/python
from math import exp

a = 1.232
print "input " ,a

b = int(a * (2**8))
#b=-1280

print "Fixed point ",b


c= float(b)/(2**8)
print c

d = 1/(1+ exp(-c))
print d
e= int(d * (2**8))
print "output is : ",e
f=hex(e & 0xffffffff)
#print f

```

3.4.1 WRITING A MAKEFILE

a. What is a makefile?[1]

Make files are simple way to organize code compilation. Compiling your source code files can be tedious, especially when you want to include several source files and have to type the compiling command every time you want to do it. Make files are special format files that together with the make utility will help you to automagically build and manage your projects.

b. The make utility

If you run **make** this program will look for a file named makefile in your directory, and then execute it.If you have several makefiles, then you can execute them with the command: **make -f MyMakefile** There are several other switches to the make utility.

c. Build Process

1. Compiler takes the source files and outputs object files
2. Linker takes the object files and creates an executable

d. Compiling by hand

The trivial way to compile the files and obtain an executable, is by running the command:

g++ main.cpp hello.cpp factorial.cpp -o hello

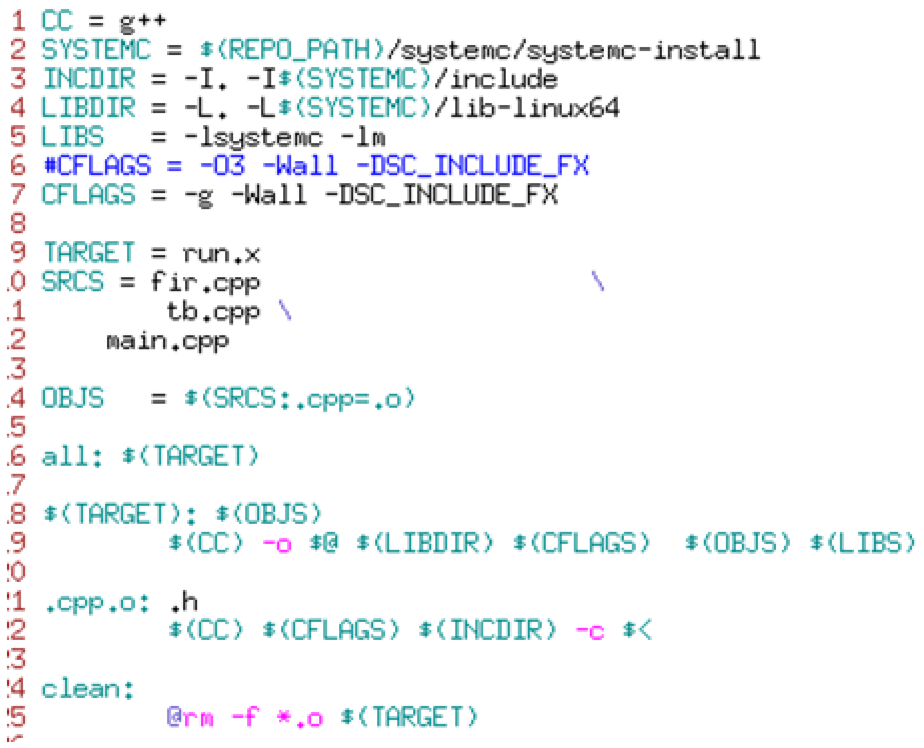
e. Build Process

The basic makefile is composed of:

target: dependencies

[tab] system command

f. Makefile example



```
CC = g++
SYSTEMC = $(REPO_PATH)/systemc/systemc-install
INCDIR = -I. -I$(SYSTEMC)/include
LIBDIR = -L. -L$(SYSTEMC)/lib-linux64
LIBS   = -lsystemc -lm
#CFLAGS = -O3 -Wall -DSC_INCLUDE_FX
CFLAGS = -g -Wall -DSC_INCLUDE_FX

TARGET = run.x
SRCS = fir.cpp                       \
        tb.cpp \
     main.cpp

OBJS   = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
        $(CC) -o $@ $(LIBDIR) $(CFLAGS)  $(OBJS) $(LIBS)

.cpp.o: .h
        $(CC) $(CFLAGS) $(INCDIR) -c $<

clean:
        @rm -f *.o $(TARGET)
```

3.5 Conclusion

From this chapter it is concluded that the PE was modeled successfully using SystemC as the language.The driver which drives the PE was modeled without flaws.From the chapter, it can also be concluded the reason behind using SystemC over other HDLs as it consumes less time and improves simulation performance .This chapter also concludes proper verification of PE . For the verification purposes,Python has been used as a scripting language .It also shows methods and steps to write a MAKEFILE. Hence, the modeling and the verification of PE was successfully carried out.

Chapter 4

Conversion of Percsense Arch into TLM2.0

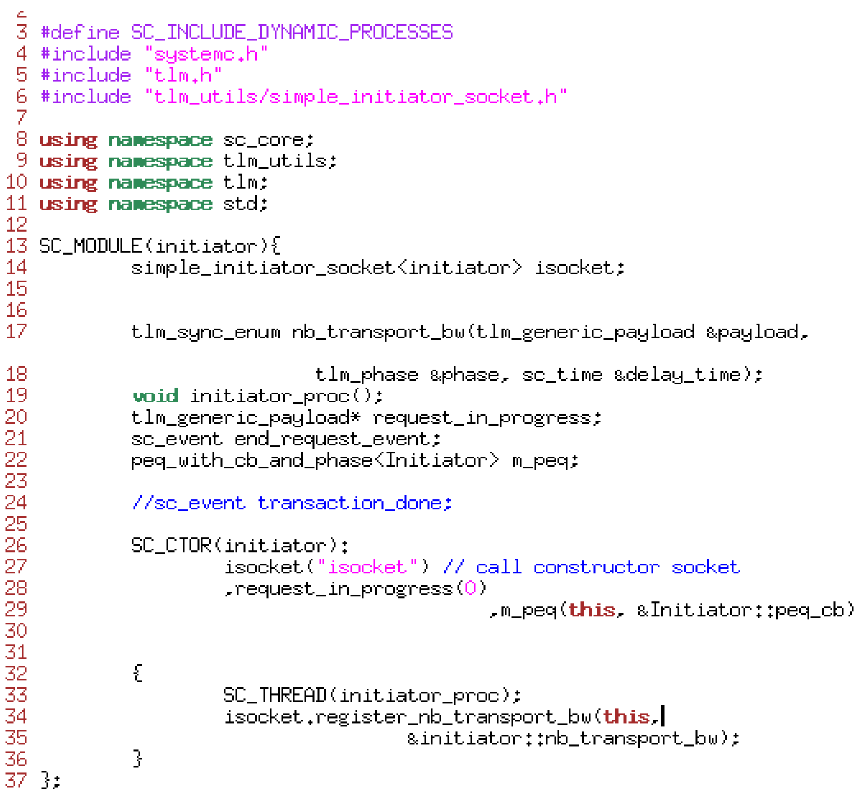
In TLM2.0 task given was to convert the complete PercSense code into TLM2.0 based interfaces. The next task lined up was to convert all the interfaces in the PercSense System level architechture into TLM 2.0 compliant.The main reason to do this was as follows:

a. Interoperability

b. Faster

1. **Interoperability** : The TLM2.0 gives interoperability among different modules[9]. For example, different vendors having different codes in different languages can play together if they are TLM 2.0 compiant.

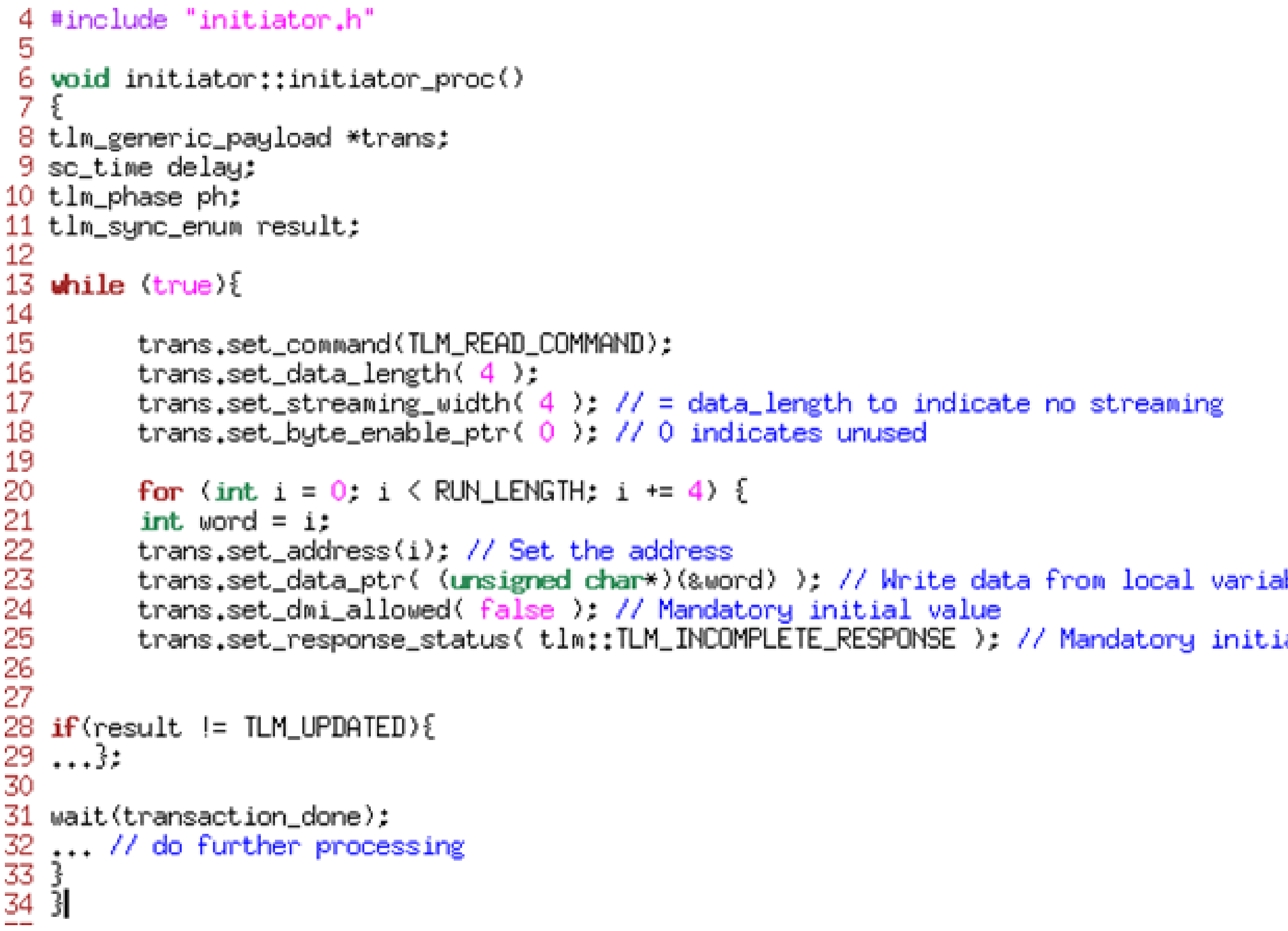
2. **Fast**: It is also observed that TLM is faster than RTL. [9] For having a glimpse of what TLM2.0 is and how does it work. Below is one of the code written while the course of work:



```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"

using namespace sc_core;
using namespace tlm_utils;
using namespace tlm;
using namespace std;

SC_MODULE(initiator){
        simple_initiator_socket<initiator> isocket;


        tlm_sync_enum nb_transport_bw(tlm_generic_payload &payload,

                        tlm_phase &phase, sc_time &delay_time);
        void initiator_proc();
        tlm_generic_payload* request_in_progress;
        sc_event end_request_event;
        peq_with_cb_and_phase<Initiator> m_peq;

        //sc_event transaction_done;

        SC_CTOR(initiator):
                isocket("isocket") // call constructor socket
                ,request_in_progress(0)
                                       ,m_peq(this, &Initiator::peq_cb)


        {
                SC_THREAD(initiator_proc);
                isocket.register_nb_transport_bw(this,
                                &initiator::nb_transport_bw);
        }
};
```

Figure 4.1: initiator.h

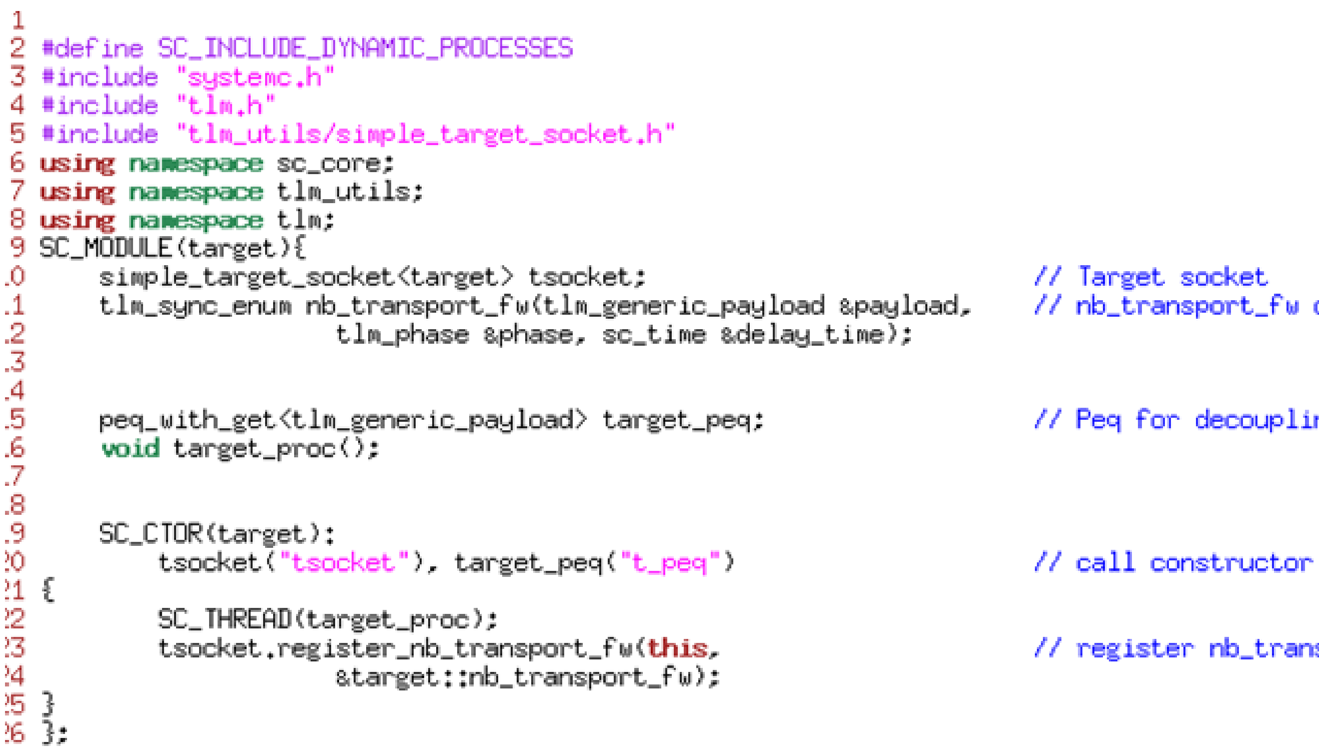
Code explaination: The above code(figure 4.1) declares a TLM2.0 module consisting of an initiator socket, TLM2.0 generic payload,TLM2.0 event,constructor and thread.All the TLM header files are also included.



```
#include "initiator.h"

void initiator::initiator_proc()
{
tlm_generic_payload *trans;
sc_time delay;
tlm_phase ph;
tlm_sync_enum result;

while (true){

        trans.set_command(TLM_READ_COMMAND);
        trans.set_data_length( 4 );
        trans.set_streaming_width( 4 ); // = data_length to indicate no streaming
        trans.set_byte_enable_ptr( 0 ); // 0 indicates unused

        for (int i = 0; i < RUN_LENGTH; i += 4) {
        int word = i;
        trans.set_address(i); // Set the address
        trans.set_data_ptr( (unsigned char*)(&word) ); // Write data from local varia
        trans.set_dmi_allowed( false ); // Mandatory initial value
        trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE ); // Mandatory initi


if(result != TLM_UPDATED){
...};

wait(transaction_done);
... // do further processing
}
}
```

Figure 4.2: initiator.cpp

Code explaination: In the above code(figure 4.2) TLM2.0 generic Payload is declared which includes command, data length,address,data pointer and so on.

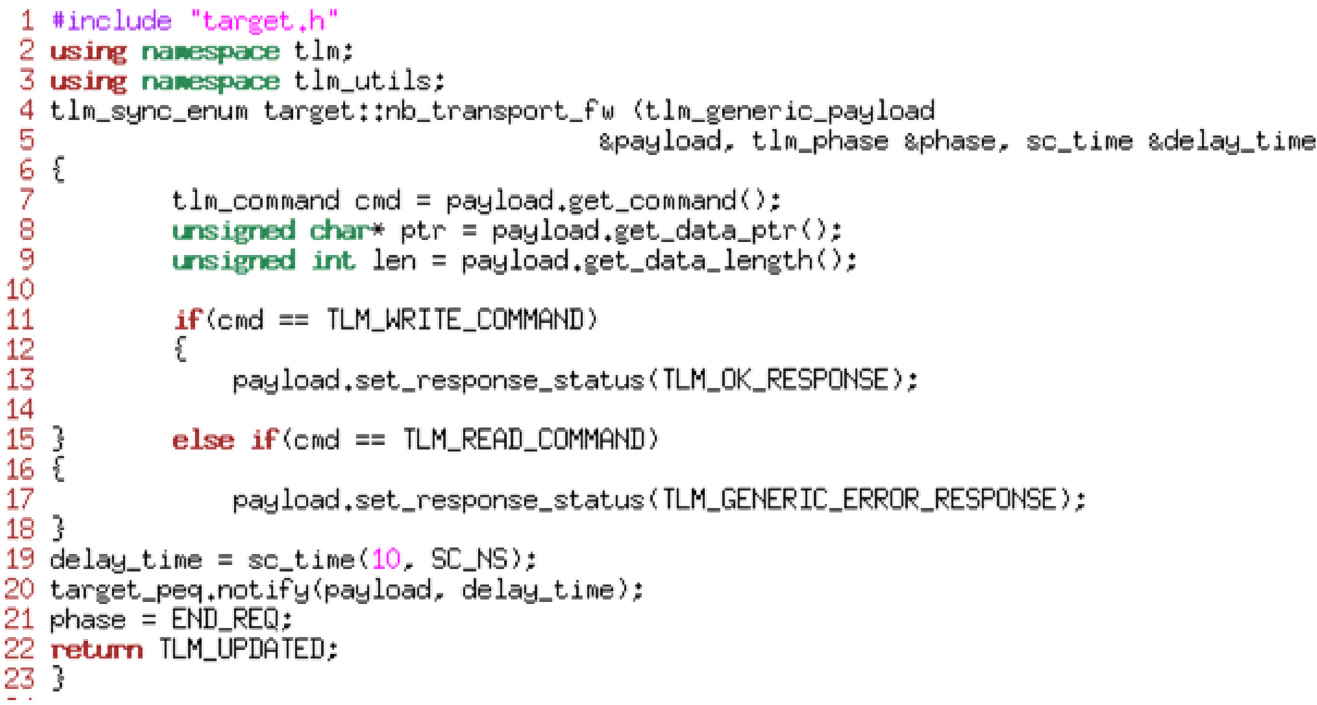


```

#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_target_socket.h"
using namespace sc_core;
using namespace tlm_utils;
using namespace tlm;
SC_MODULE(target){
    simple_target_socket<target> tsocket;                               // Target socket
    tlm_sync_enum nb_transport_fw(tlm_generic_payload &payload,         // nb_transport_fw c
                    tlm_phase &phase, sc_time &delay_time);


    peq_with_get<tlm_generic_payload> target_peq;                       // Peq for decoupli
    void target_proc();


    SC_CTOR(target):
        tsocket("tsocket"), target_peq("t_peq")                         // call constructor
{
        SC_THREAD(target_proc);
        tsocket.register_nb_transport_fw(this,                          // register nb_trans
                    &target::nb_transport_fw);
}
};
```

Figure 4.3: target.h

Code explaination: In the figure(figure4.3) a target similar to the initiator is designed and target sockets are created.

target.cpp Code explaination: Here in the above figure(figure 4.4) all the

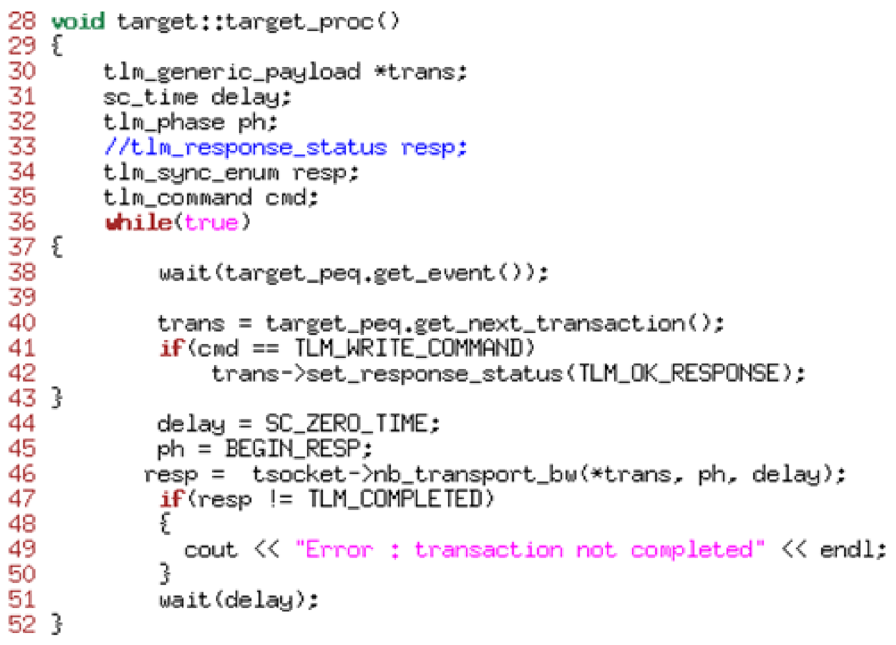


```
#include "target.h"
using namespace tlm;
using namespace tlm_utils;
tlm_sync_enum target::nb_transport_fw (tlm_generic_payload
                                    &payload, tlm_phase &phase, sc_time &delay_time
{
        tlm_command cmd = payload.get_command();
        unsigned char* ptr = payload.get_data_ptr();
        unsigned int len = payload.get_data_length();

        if(cmd == TLM_WRITE_COMMAND)
        {
            payload.set_response_status(TLM_OK_RESPONSE);

}       else if(cmd == TLM_READ_COMMAND)
{
            payload.set_response_status(TLM_GENERIC_ERROR_RESPONSE);
}
delay_time = sc_time(10, SC_NS);
target_peq.notify(payload, delay_time);
phase = END_REQ;
return TLM_UPDATED;
}
```

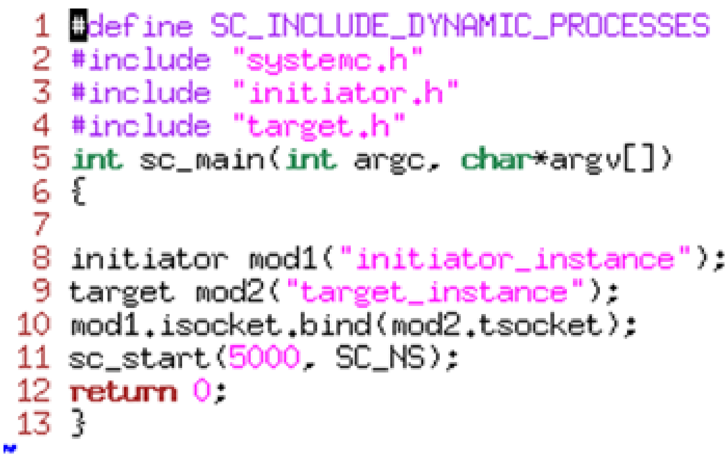
Figure 4.4: target.cpp

generic payload attributes are fetched and response are set according to the command in payload.

contd..



```
void target::target_proc()
{
    tlm_generic_payload *trans;
    sc_time delay;
    tlm_phase ph;
    //tlm_response_status resp;
    tlm_sync_enum resp;
    tlm_command cmd;
    while(true)
{
        wait(target_peq.get_event());

        trans = target_peq.get_next_transaction();
        if(cmd == TLM_WRITE_COMMAND)
            trans->set_response_status(TLM_OK_RESPONSE);
}
        delay = SC_ZERO_TIME;
        ph = BEGIN_RESP;
       resp =  tsocket->nb_transport_bw(*trans, ph, delay);
        if(resp != TLM_COMPLETED)
        {
          cout << "Error : transaction not completed" << endl;
        }
        wait(delay);
}
```



```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "systemc.h"
#include "initiator.h"
#include "target.h"
int sc_main(int argc, char*argv[])
{

initiator mod1("initiator_instance");
target mod2("target_instance");
mod1.isocket.bind(mod2.tsocket);
sc_start(5000, SC_NS);
return 0;
}
~
```

Figure 4.5: main.cpp

Code explaination: In the main.cpp file(figure 4.5) all the initiator and target sockets are bound to modules.

After all the ramp up task, a base for TLM2.0 was created which made it easy to make the actual PercSense architechture TLM2.0 compliant. The figure below :
In the figure above(figure 4.6), which is the actual architecture, the red colored

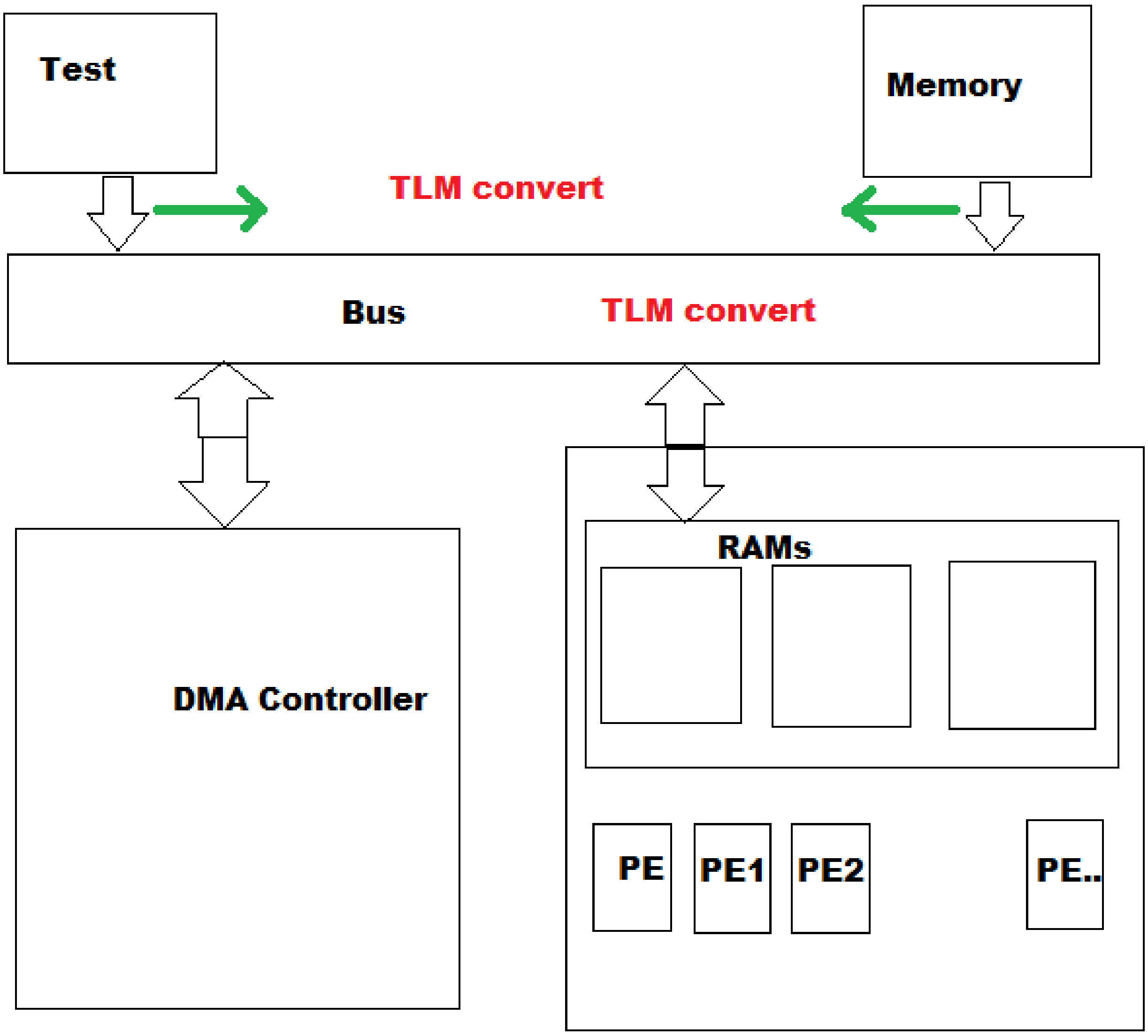


Figure 4.6: PercSense architecture

TLM convert means TLM2.0 compliant sections.The architecture has a test block, a memory block,a Bus, a DMA controller, RAMs and piles of PE. All the interfaces can be converted into TLM2.0.

4.1 Conclusion:

So, at the end of this chapter it is concluded that some of the interfaces of the Percsense system level architecture were made TLM2.0 compliant successfully.

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

In this dissertation, SystemC and TLM2.0 models of PE, a part of PercSense Architecture of Machine learning IPs were studied and modeled and verified for the same.During the course of modeling the PE, various ramp up tasks were done, for example,Ex-or gate using NAND gates and testbench for the same in SystemC (IEEE- 1666) and code for MAC (multiply and accumulate) testbench for the same in SystemC(IEEE- 1666).The modeling of processing engine was first done without using any of the control signals like sum,difference,compare,accumulate and so on.Next, PE was modeled and verified using all the control signals. The PE parameters like First interger input (ip0),Second integer input(ip1) and Weight (wt) are passed as arguments through a driver module.These parameters were varied and output of PE was observed for its functionality.
Comparison between SystemC, C and HDLs was also carried out.From the comparison it was concluded that SystemC performs best over other HDLs and C as it improves simulation performance and takes less time comparitively.For the verification of PE, Python scripting was used. As the modeling and verification of PE had lots of files i.e, the header files, the cpp files, there was a need to combine all the files and then

compile them rather than compiling and running on the command line on UNIX. So, this problem was solved using what is called as a MAKEFILE. All the PE files were combined and compiled by designing just one Makefile.
Various use cases and advantages of TLM2.0 were studied.Ramp up tasks were also done wherein Initiator and target model framework for communication and synchronisation were designed as TLM2.0 compliant which made a base for the converting the Percsense architechture into TLM2.0.So, with the help of these,some of the parts of Percsense architechture were made TLM2.0 compliant.

5.2 Future Scope

The future scope of the project is vast. For instance,the main scope of the work is further converting all the remaining blocks of the Percsense architecture into TLM2.0 compliant.Furthermore, TLM2.0 is a very useful tool in industries nowadays.Every IP should be designed in such a way that it can play with IPs from other vendors.Suppose,one has a code written in System verilog and other person has it in SystemC and both the codes are to be combined, here TLM2.0 comes handy.

References

[1] Intel India Tech Pvt. Ltd Internal Modules.”PercSense Architecture Specification”,2013

[2] 1666-2011 - ”IEEE Standard for Standard SystemC Language Reference Manual”, Jan. 9 2012

[3] David Black, Jack Donavan, ” SystemC from the Ground Up”

[4] `www.asic-world.com/systemc/tutorial.html`

[5] `www.accellera.org`

[6] `www.forteds.com/SystemC`

[7] `https://www.coursera.org/course/ml`

[8] `https://www.python.org/`

[9] `http://www.doulos.com/knowhow/systemc/tlm2/`