

## MASTER

### TLM-based multi-core system level modeling and simulation (TM2S)

Siyoum, F.M.

*Award date:*  
2009

[Link to publication](#)


#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



# TLM-based Multi-core System Level Modeling and Simulation (TM2S)

Firew M. Siyoum



TECHNISCHE UNIVERSITEIT EINDHOVEN

# TLM-based Multi-core System Level Modeling and Simulation (TM2S)

by  
Firew M. Siyoum

**Supervisor TU/e:** prof. dr. Henk Corporaal  
**Supervisor Recore:** ir. Jordy Potman

A thesis submitted in partial fulfillment for the  
degree of Masters of Science

in  
Embedded Systems

August 2009



# *Abstract*

The integration scale of semiconductor chips has been continuously soaring high throughout the years. Today, it is possible to integrate a complete system of IPs and communication networks on a tiny die area and form what is called a System-on-Chip (SoC). To cope with this growing complexity and time-to-market pressure, the SoC industry has accepted raising the abstraction level of system designs above RTL as an effective approach. ESL (Electronic System Level) refers to different system modeling techniques at a level of abstraction above RTL.

As part of ESL, in recent years Transaction Level Modeling (TLM) is obtaining a huge attention in SoC design cycle, serving as a unique reference across different teams for three strategic activities: early software development, architecture analysis and functional verification. However, the name ‘*transaction level*’ is still a vague term as it does not denote a single level of detail. Rather, TLM refers a continuum of abstraction levels that each vary in the amount of functional or temporal detail they express depending on the use case they are modeled for. Different researches have been made in the last few years to define these abstraction layers from different point of views such as granularity of time, functional abstraction, communication abstraction and use-cases. However the dust has not yet settled down. Issues on models interoperability, flexibility, efficiency and implementation details are not yet fully addressed due to the vastness of the topic.

In this Master’s thesis, an integrated TLM-based system level modeling approach for multi-core systems is devised which is mainly targeting streaming applications. The work begins with the specification of communication and computation refinement levels as well as definition of abstraction layers in system level modeling. Then, a library of communication APIs and models of components is prepared by reusing and modifying an existing framework called OCCN. Finally, a digital radio receiver streaming application and a high-level AMBA AHB bus model, are carried out to show the usage of the devised methodology. The promising results obtained in simulation speed and model use-cases indicate that the approach lays down the foundation for an integrated system level modeling methodology which can be extended with new features and enhanced library into a complete tool.

# *Acknowledgements*

I would like to thank the management of Recore Systems for allowing me to conduct my graduation work in their company as well as for all of their friendly assistances during my stay.

My deepest gratitude goes to my supervisor at TU Eindhoven- prof.dr. Henk Corporaal and my supervisor at Recore Systems - ir. Jordy Potman, for their unparalleled guidance, encouragements and help throughout my thesis work. I am also deeply indebted to the executive director of Recore Systems - dr.ir. Gerard Rauwerda for his constant attention, sincere comments and supervision during my entire work.

I am profoundly thankful to Recore System's DAB receiver project team - Maciej, Reinier and Eelke. Without their help, I could not have able to undergo the digital radio receiver demonstration. In addition, I am truly grateful to them for spending their time in those several meetings and providing me a number of great ideas which were an invaluable input to my work.

Last but not least, I would like to thank my family - Emaye, Yenework, Selam, Nitsu and all my friends for their confidence on me, their priceless affectionate and encouragements which are always the driving force of my life.

*Firew M. Siyoun*

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Classical Design Flow . . . . .	2
1.2 Challenges of Modern SoC Development . . . . .	3
1.3 Thesis Objective . . . . .	5
1.4 Contributions . . . . .	5
1.5 Thesis overview . . . . .	6
<b>2 Transaction Level Modeling (TLM)</b>	<b>7</b>
2.1 Basics of TLM . . . . .	7
2.2 Fundamental ingredients of TLM-based system modeling . . . . .	10
2.3 TLM in the design flow . . . . .	11
2.4 Summary . . . . .	12
<b>3 Evaluation of Related Works</b>	<b>13</b>
3.1 Abstraction Layers . . . . .	13
3.2 SystemC-based Frameworks . . . . .	15
3.2.1 OCCN . . . . .	16
3.2.2 TLM 2.0 . . . . .	16
3.2.3 OCP SystemC Channels . . . . .	17
3.2.4 GreenBus . . . . .	17
3.2.5 CCATB AMBA Channels . . . . .	18
3.2.6 OSSS . . . . .	18
3.2.7 ARMn . . . . .	19
3.3 Evaluation . . . . .	20
3.4 Justification . . . . .	20
3.4.1 Languages . . . . .	21
3.4.2 Integrated Approach . . . . .	22
3.4.3 Cost . . . . .	22



3.4.4	Efficiency . . . . .	22
3.5	Summary . . . . .	23
<b>4</b>	<b>Methodology for TLM-based System Level Modeling</b>	<b>25</b>
4.1	Abstraction Levels . . . . .	25
4.1.1	Communication Refinement . . . . .	25
4.1.2	Computation Refinement . . . . .	27
4.1.3	Abstraction Levels in System Level Modeling . . . . .	29
4.1.4	Features and Use Cases of the Abstraction Levels . . . . .	30
4.1.4.1	Functional Model (FM) . . . . .	30
4.1.4.2	Process Model(PM) . . . . .	31
4.1.4.3	Transfer-based System Level Model (TSLM) . . . . .	31
4.1.4.4	Phase-based System Level Model(PSLM) . . . . .	32
4.1.4.5	Protocol-Specific System Level Model(PSSLM) . . . . .	33
4.1.4.6	Implementation Model (IM) . . . . .	33
4.2	Library . . . . .	33
4.2.1	OCCN's Communication Methodology . . . . .	34
4.2.2	Models of Other Components . . . . .	36
4.2.2.1	Processing Elements . . . . .	36
4.2.2.2	Transaction Interfaces . . . . .	37
4.2.2.3	Memory . . . . .	38
4.3	Building the Virtual Platform . . . . .	38
4.4	Modeling Software Applications . . . . .	38
4.5	System Level Design Flow . . . . .	40
4.6	Summary . . . . .	41
<b>5</b>	<b>Demonstrations- DAB Receiver and AMBA AHB Bus</b>	<b>45</b>
5.1	DAB Receiver . . . . .	45
5.1.1	Digital Audio Broadcasting(DAB) Standard . . . . .	45
5.1.2	DAB Transmission/Reception System . . . . .	46
5.1.3	DAB Receiver Architecture . . . . .	46
5.1.4	Software Architecture . . . . .	47
5.1.5	System Level Modeling of the DAB Receiver SoC . . . . .	50
5.1.5.1	Functional Model . . . . .	50
5.1.5.2	Process model . . . . .	51
5.1.5.3	Transfer-based System Level Model . . . . .	52
5.2	High-level AMBA AHB 2.0 Bus . . . . .	54
5.2.1	AMBA Bus Protocol . . . . .	54
5.2.2	AMBA AHB 2.0 Standard . . . . .	55
5.2.3	Implementation . . . . .	57

5.2.4	Experimentation . . . . .	60
5.3	Summary . . . . .	67
<b>6</b>	<b>Conclusions and Future works</b>	<b>69</b>
6.1	Conclusions . . . . .	69
6.2	Future works . . . . .	70
<b>A</b>	<b>Codes of Selected Library Components</b>	<b>73</b>
<b>B</b>	<b>Codes of Main Elements in AMBA AHB System Implementation</b>	<b>79</b>
	<b>Bibliography</b>	<b>81</b>
	<b>Abbreviations</b>	<b>86</b>
	<b>Glossary</b>	<b>89</b>



# Chapter 1

## Introduction

Nowadays, embedded systems are ubiquitously available around us. They are tightly linked to our daily life to the extent where it seems we can not live without them. They are now the core entities in consumer electronics products, home appliances, telecommunication, medical equipments, computing, automotive merchandises, multimedia, aerospace, industry and so forth. This was possible because today's advancements in semiconductor technology have enabled complex Systems-on-Chips (SoC) to be constructed in much compacted manner.

Since its first inception in the mid-20<sup>th</sup>-century, the integration scale of IC (Integrated Circuit) technology has been soaring high. In the early days, ICs comprised tens of transistors and were used to implement simpler logic gates. Today the technology has enabled to squeeze millions of transistors in a tiny silicon die and form powerful processors. For instance, the Cell processor, Figure 1.1, which is used in Sony's PlayStation3 comprises 234 million transistors in a die area of 221 mm<sup>2</sup> [1]. The tremendous shrinkage in size, reduction in cost and growth in capacity of semiconductor ICs have made them to be embedded everywhere in utilities we use in our daily life.

However, the growing complexity and application dimension of modern embedded systems is putting significant pressure on designers and manufactures. This is because the design of these systems involves contradictory constraints. On one hand, they often target broad market coverage and therefore should be cheap, small-sized, power efficient and be on market on time. On the other hand, they still need to satisfy performance requirements, and often support multiple applications of multimedia content. The breadth of the design requirements leads to designs of complex heterogeneous System-on-Chip (SoC) architectures consisting of multiple processors, configurable processing cores, customized hardware components, various memory units, peripheral interfaces and system interconnection network integrated on a single chip. These multi-core SoCs have now become the cornerstones in the development of today's embedded systems such as digital audio/video receivers, game consoles, navigation systems, 3G mobile phones and so many others.

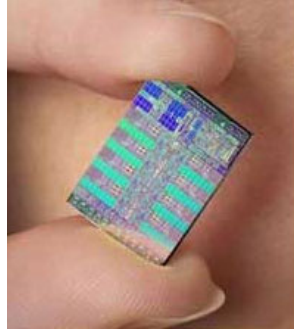


FIGURE 1.1: The Cell Processor: 234 million transistors in 221  $mm^2$  die area

With the growing complexity of these systems, the design space also gets extensively broad. This means there needs to be an efficient and systematic SoC development methodology that fits the needs and characteristics of these multi-application multi-core SoCs. In this chapter, the nature of the classical SoC design methodology and the related challenges are presented in section 1.1 and 1.2. The objective and the key contributions of this thesis are discussed in section 1.3 and 1.4. Finally the chapter concludes with an overview of this thesis in section 1.5.

## 1.1 Classical Design Flow

Traditionally, for digital electronic embedded systems designs, which are composed of discrete components such as microprocessors, memory chips and Application Specific Integrated Circuits (ASIC), the design process usually starts with one or two system design experts partitioning the functionality into hardware and software (Figure 1.2).

The hardware function is then further partitioned into standard parts and ASICs. It is possible to write a specification for an ASIC of a few thousands to few hundred thousands gates in a natural language. This specification is hand off to an ASIC designer or team, who start capturing the design at Register Transfer Level (RTL) for which Hardware Description Languages (HDLs) are a good match [2]. Once these HDL models pass the functional verification test, synthesis is performed to obtain a logic netlist. After the netlist is ready, the hardware design enters the back-end design steps. This basically includes layout drawing, floorplanning, place and routing, and so on, all the way down to physical verification. After this, the hardware design is essentially at a tape-out status ready to be sent to fabrication for building a prototype of the system [3].

The software function, on the other side, is developed on the other corner of the design floor with little or no communication with the hardware development. Mostly validating the software is delayed until an emulator or FPGA-based prototype is available. Once the system prototype is available, the software will be embedded into the prototype to conduct system integration and validation. If system specifications are not satisfied and/or faults are found in the hardware or software, the design process will be iterated through both the hardware and software paths. These loops might repeat until system specifications are satisfied [3].

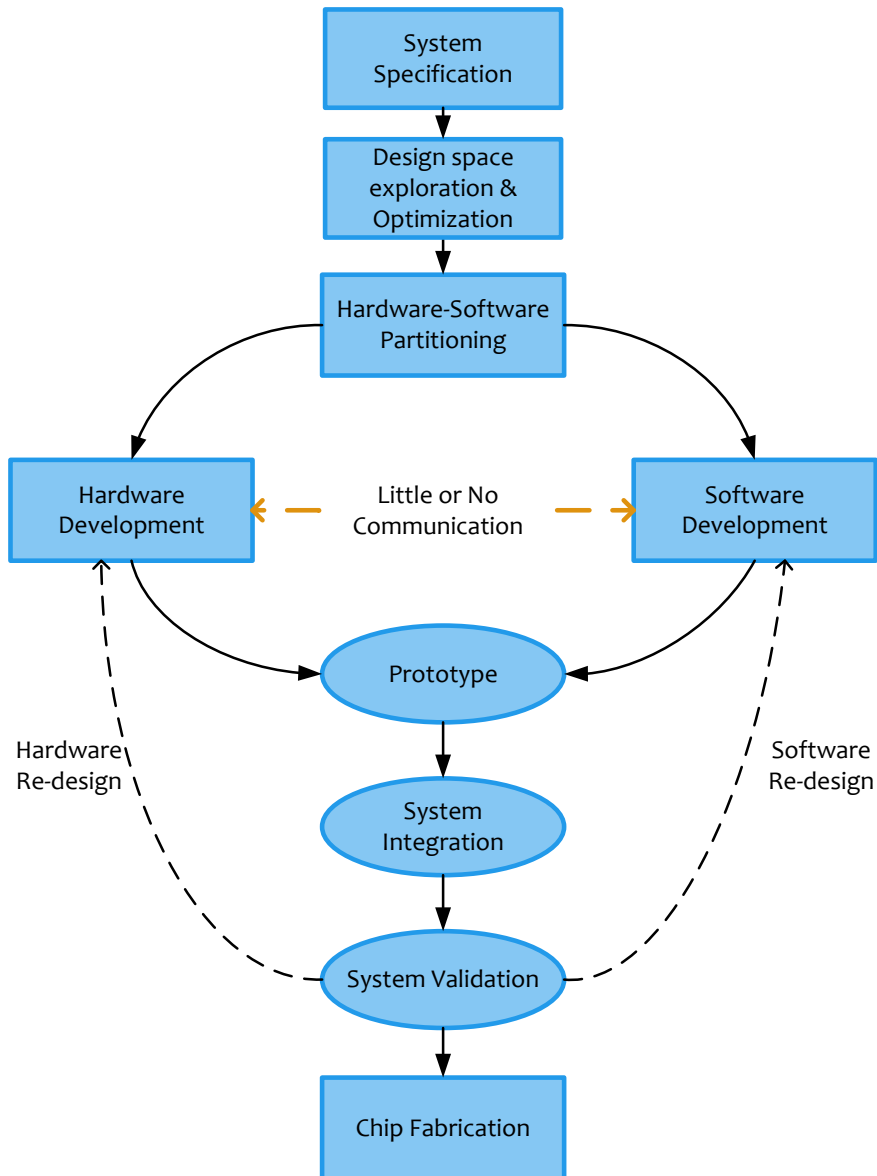


FIGURE 1.2: Classical SoC Design Flow [3]

As will be discussed in the next section, this classical design flow has a number of limitations that makes it fall short of fulfilling the needs and characteristics of today's multi-application multi-core SoC development.

## 1.2 Challenges of Modern SoC Development

Two of the main bottlenecks in modern SoC development are the explosively widening design space and the time-to-market pressure [3].

- **Explosive Design space**

As the embedded system design shifts gear from system-on-board to System-on-Chip paradigm, more applications are also being incorporated into the system to accomplish more sophisticated tasks. A typical SoC comprises various types of processors from fully programmable multi-processors to reconfigurable processors, Application Specific Integrated Circuits (ASICs), memory chips, peripheral interfaces and system-wide interconnection network. Such SoC architectures often run multiple software applications which vary in characteristics. Some of them might have strict real-time performance requirement which the design should fulfil. Some of them might have a dynamically changing runtime behaviour which makes decisions difficult at design time. The design space in such a SoC is broad as the designer has to investigate between various choices such as software partitioning, task-to-resource mapping, memory size, type of interconnect, arbitration strategy, communication protocol, bus width, burst size, topology and so many others. Using the RTL models discussed in section 1.1 for architectural analysis in the design flow is too slow to cover this broad design space given the limited project time available.

- **Time-to-Market Pressure**

The increasing complexity of current SoC products usually necessitates time-consuming development phases. Besides, the competition in the SoC industry is so fierce that the product development should proceed according to its intended timeline and be on market on time to achieve the desired market success. Otherwise, bankruptcy might be inevitable. The classical design approach discussed in section 1.1 is not handy with this respect. First, RTL modeling complex SoCs introduces too much detail which will be a bottleneck in project management and reduce designer's productivity. Secondly, system integration and validation is delayed until a prototype is available. Thirdly, faults or unsatisfied system specifications are mostly discovered during system integration making first time chip success difficult. This is mainly because the separate development path followed in the classical design flow does not have a common platform where hardware and software co-design could be done. The recursive nature of the fault correction approach then introduces more delays in the design flow.

Due to these reasons, in recent years the SoC industry has been in search for robust design methodologies for the design of heterogeneous multi-application multi-core SoCs. One solution that has gained popularity in recent years is raising the modeling abstraction layers above RTL to achieve fast simulation time. Transaction Level Modeling (TLM) is one such technique that enables modeling system communications at a higher abstraction level above cycle accurate RTL to achieve fast simulation with the price of accuracy for different use cases in the SoC design flow. However, the name '*transaction level*' is still a vague term as it does not denote a single level of detail. Rather, TLM refers a continuum of abstraction levels that each vary in the amount of functional or temporal detail they express depending on the use case they are modeled for. Different researches have been made in the last few years to define these abstraction layers from different point of views such as granularity of time, functional abstraction, communication abstraction and use cases. However the dust has not yet settled down. Issues on models interoperability, flexibility, efficiency and implementation details are not yet fully addressed due to the vastness of the topic.

This thesis, which adds to such similar efforts, tries to devise an integrated TLM-based system level modeling approach for multi-core systems, which are mainly targeting streaming applications. Objectives of this thesis are discussed in the upcoming section.

### 1.3 Thesis Objective

In the development process of streaming DSP applications it is useful to have a functional model of the application that accurately models the interaction between the different parts of the application running on the different components of the SoC. This makes it possible to develop and test the entire application at functional level before the complete SoC hardware is available. The goal of this Master assignment is to develop a system level modeling and simulation framework for multi-core systems. The framework should enable to integrate functional models of IP components with Network on Chip (NoC) or bus interconnection network. The functional models of the communication infrastructure in the multi-core simulation framework have to be developed as part of the project. The developed framework then needs to be demonstrated with a test application. Therefore the two main goals of this thesis work are the following.

1. Developing a system level modeling and simulation framework for multi-core systems.
  - The framework should be able to model/simulate streaming DSP applications running on multi-core SoCs, such as the CRISP [4] SoC.
  - The framework should provide the necessary application programming interface (API) for the different hardware components that can be used in the final application running on the SoC.
2. Demonstrating the multi-core system level simulation with some applications.
  - A digital radio receiver streaming application could be used as one of the demonstrations.
  - Simulation speed and use-case could be used to evaluate the demonstrations.

Additional requirements of this thesis work include the following.

1. The initial goal of the framework is to enable functional multi-core modeling/simulation but it should be possible to extend it to cycle accurate multi-core modeling/simulation in the future.
2. Investigating if the methodology can be the TLM 2.0 framework.

### 1.4 Contributions

This thesis aims at developing an integrated methodology for fast modeling and simulation of multi-core systems. The main contributions achieved in the course of this work are given below.



- The specification of refinement levels in communication and computation aspect of a system and the definition of abstraction layers in system level modeling of multi-core systems. For better interoperability and usage clarity, the distinctive features and use-cases of each layer are explicitly specified.
- A system level design flow 4.12 that includes approaches for modeling data-dependent software processes, building virtual platforms, integrating and simulating the system at different levels of abstractions.
- Preparation of a library of communication APIs and models of components. This is done by reusing and modifying an existing framework called OCCN [5].
- Demonstrating the use of the methodology with two applications: a digital radio receiver streaming application (DAB receiver) and an AMBA AHB bus based system with random data generators.

## 1.5 Thesis overview

This thesis is organized as follows. Chapter 2 discusses the basics of transaction level modeling (TLM). It presents the fundamentals of TLM, the various ingredients needed for TLM-based system modeling and how TLM fits in SoC design flows. Chapter 3 reviews several works in the domain of system level modeling and compares their features. This chapter also includes the brief description of transaction level modeling approaches such as TLM 2.0. Chapter 4 dives into the system level modeling methodology devised in this work. It discusses the definition of the different abstraction levels and the design flow through the abstraction levels that minimizes design effort (reuse of components between levels), balances the tradeoff between accuracy and simulation time and suits recursive design practices. The library of the communication APIs and models of components is also discussed in this chapter. The implementation of the two demonstration applications is covered in chapter 5. The first section covers the modeling of the DAB receiver at three different abstraction levels. The second section presents the AMBA AHB bus based system where the bus is modeled at two different communication refinement levels. Finally, the thesis concludes in chapter 6 with a summary of the work and recommendations for potential future extensions.

## Chapter 2

# Transaction Level Modeling (TLM)

The growing complexity in modern SoCs is forcing the industry to look for design methodologies above RTL that can be used for architectural analysis and embedded software development. In system level design of these SoCs, communication takes the central role. Therefore a methodology for modeling communication at a higher abstraction level has become important. TLM is one such methodology that promises to be used in system level SoC modeling for early software development, architectural analysis and functional verification. The main goal of this chapter is to discuss the fundamentals of TLM and its potential role in system level designs. Section 2.1 discusses the basics of TLM and how it differs from the conventional cycle accurate modeling. Section 2.2 presents the key ingredients needed to integrate TLM in system level modeling of multi-core systems. Finally the role of TLM in SoC design flows is pointed out in section 2.3.

### 2.1 Basics of TLM

It is quite common these days to use standard and well-known IP (Intellectual Property) components in SoC development. This means system design mainly focuses on architectural analysis. Hence system performance is significantly dictated by the communication between IPs. However, choosing the most appropriate system interconnection network and connection topology is not a simple task. The system designer needs to traverse a huge design space to get to his/her best-fit architecture. The choice may start between a bus, a NoC (Network-on-Chip) or a hybrid interconnect. For instance, design choices related to a bus interconnect may include the following [6].

- Standard buses (for example, AMBA 2.0, AMBA 3.0, CoreConnect, WishBone, STBus, etc.)
- Signal wires (shared or separate data, control and address bus)
- Bus width (data bus width and address bus width)

- Data transfer modes (for example, single non-pipelined, pipelined, burst, split transfer, etc.)
- Arbitration (for example, centralized versus distributed, round-robin, TDMA, dynamic priority, etc.)
- Topology (for example, single bus, hierarchical bus, split bus, etc.)
- Layers (single layer and multi-layer)

Selection for a NoC also involves numerous set of decisions [7]: topology (for example, mesh, torus, fat-tree, etc), router architecture (for example, worm-hole router, VCT router, circuit-switching router, etc), routing algorithms, network interface buffer size and so many others.

For each set of design decisions, the system designer needs to compute the cost with metrics such as area, power, latency and throughput. To make things worse in addition to the vastness of the design space, the dynamism of the multiple applications that run on the SoC affects the performance figures significantly. This forces the system designer to incorporate the software applications into the architecture analysis process, leading to the need for a common platform for hardware-software co-design.

Carrying out such complex architectural analysis at cycle accurate level gives a pretty accurate results but it has a number of disadvantages.

- Simulation is too slow for architectural analysis.
- It is available too late for architectural exploration.
- Modeling takes considerable design effort.

In addition, a library of cycle accurate model of components should exist to try out different architectures at CA level. Even in the availability of such a library, integrating components requires a significant design effort. Hence, making architectural changes and trying out different possibilities is costly.

This is why TLM is proposed by the SoC industry to circumvent the drawbacks of cycle accurate modeling in today's complex SoC development [3] [8]. Figure 2.1 shows the different modeling abstraction layers in SoC design flow and their distinctive features. All modeling abstraction layers above RTL are termed as Electronic System Level (ESL). Within ESL, modeling can be done at cycle accurate level, at TLM level or at algorithmic level. Each of these levels have varying levels of accuracy and simulation time. This makes each of them suitable for certain tasks in the SoC development process and unapt for others.

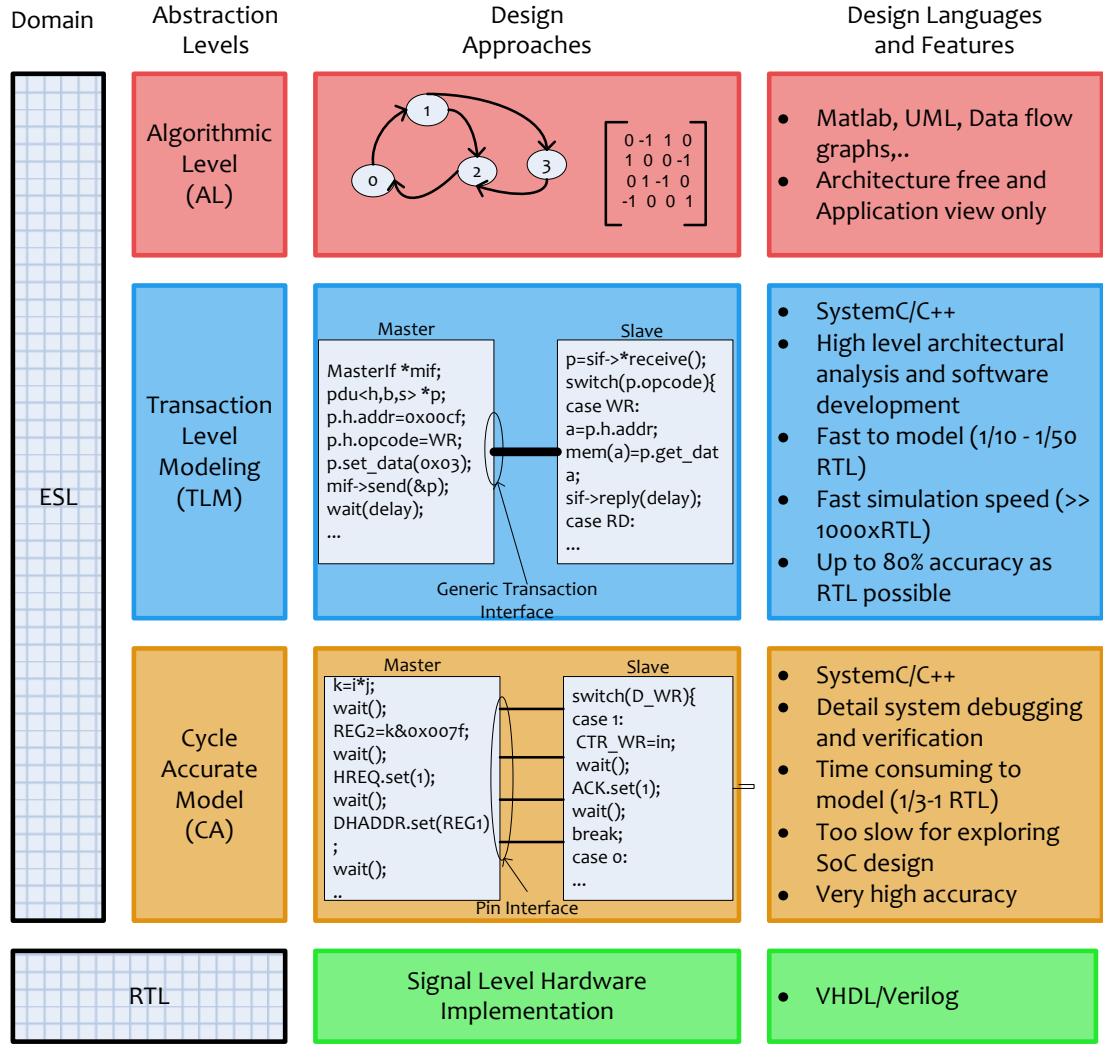


FIGURE 2.1: Different modeling abstraction layers in SoC design flow

In cycle accurate modeling, components have pin-interfaces and every signal on the interconnection link is simulated at every clock cycle. On the other hand, in TLM modeling components have generic interfaces and communication is done through function calls that takes as an argument a data structure which comprises signals in the communication link. This technique is handy in achieving separation of communication and computation, a key design target in TLM modeling frameworks.

A TLM-based system level modeling technique can be used for various use-cases [3] [9] [10]:

- Early platform for software development
- Aiding software-hardware integration
- Architectural analysis
- Functional verification

## 2.2 Fundamental ingredients of TLM-based system modeling

The central theme of TLM is modeling communication at different levels of abstraction. However, communication alone can not completely depict the behaviour and performance of the system being modeled. Computation should also be captured into the modeling picture. This triggers a need for a system level modeling approach that takes both computation and communication aspects of SoCs into account. In TLM-based system level modeling approach, communication is modeled with the techniques of TLM and computation by its own needs a convenient approach to go in harmony with the underlying TLM communication model. Developing a TLM-based system level modeling framework needs three fundamental entities: *Abstraction levels*, *Communication Interfaces* and *Models of components*.

### Abstraction Levels

The name *transaction level* in TLM is still a vague term as it does not denote a single level of detail. Rather, TLM refers a continuum of abstraction levels that each vary in the amount of functional or temporal detail they express depending on the use case they are modeled for [9]. A set of well-defined abstraction levels are key requirements for any design flow. The distinctive features of each level should be unambiguously described so that designers and tools can make systematic decisions and move between levels efficiently. The goal is to limit the number of objects to deal with at higher levels while providing enough detail for the desired exploration at each step. This is achieved by trading accuracy for efficiency, such as simulation speed and designer's effort. Furthermore, a clear and unambiguous definition of these levels is then needed to enable design automation for synthesis and verification [11].

### Communication Interfaces

The generic interfaces in TLM modeled components are the heart of TLM as they provide the necessary communication between IPs which is the central theme of TLM. Mostly these interfaces are implemented as APIs (Application Programmer's Interfaces) that can be used in various projects and help achieve interoperability between models. The two most common interfaces are *Master Interface* and *Slave Interface* which are used by *transaction initiators* and *targets*, respectively. Other interfaces include *DMI (Direct Memory Interface) Interface* and *Debug Interface* [8] which are used for direct memory accessing and system debugging, respectively.

### Models of Components

TLM models of components which are available off the shelf significantly enhances productivity in TLM modeling and help achieve crucial TLM use cases such as architectural analysis. Even though interconnection networks are the central theme of TLM, it is infeasible to have a system level model of SoCs without the necessary models of processing elements, memories and peripherals. Such a library of TLM models enormously improve productivity of TLM modeling as model reuse and reliability are significantly high.

Hence, in order to prepare a TLM-based modeling framework, first an API of the communication interfaces should be ready. Then use this API to prepare a library of SoC

component models of the desired extent of detail. With the availability such of a well-developed TLM library of components, a virtual platform of the SoC can be developed with in minutes where software processes can be mapped and very fast simulation can be done for software development and architectural analysis at the desired level of abstraction.

## 2.3 TLM in the design flow

Figure 2.2 shows how TLM can be used as a common platform for concurrent software-hardware design in modern SoC development. The hardware and software developments have different needs from the TLM model. The software development team needs fast simulation time. Hence detailed communication protocol implementation can be left out. Hardware development on the other hand needs detail timing points in the communication protocol but fast enough simulation for architectural analysis. Hence the TLM model should allow easy transition between abstraction levels to avoid the need for multiple TLM models if it is intended to be used as common platform.

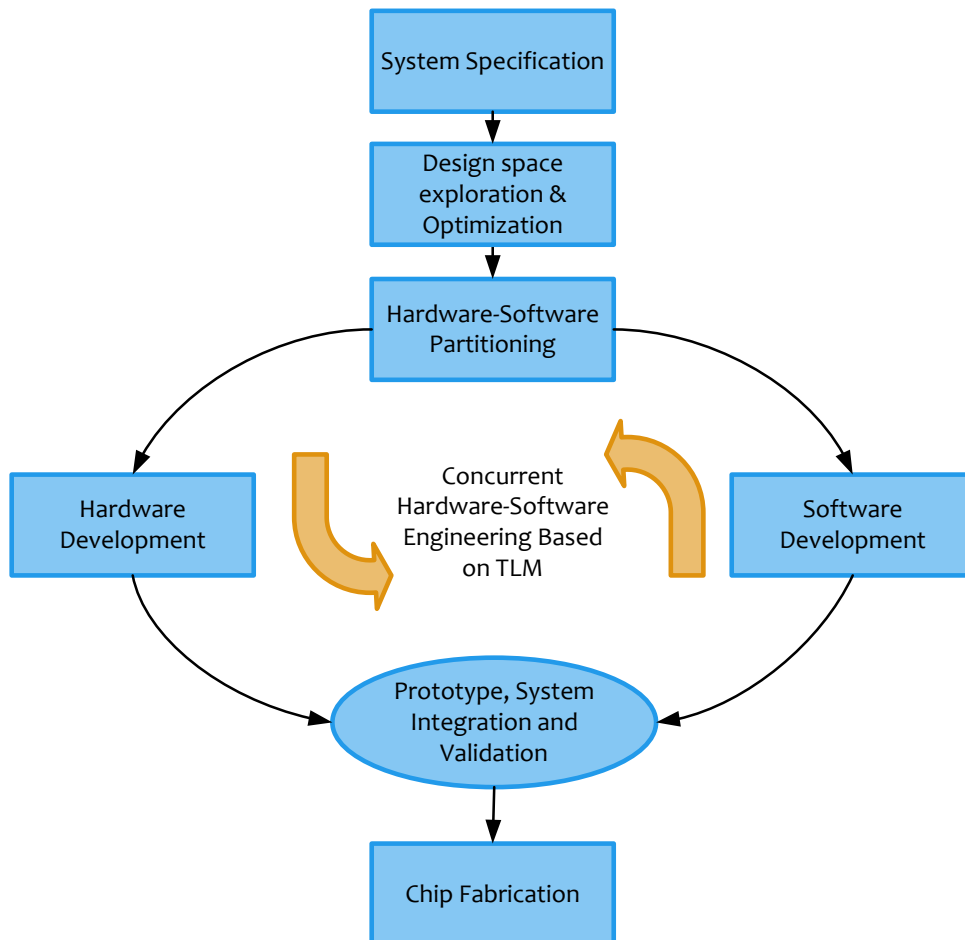


FIGURE 2.2: TLM in SoC Design Flow [3]

## 2.4 Summary

The growing complexity in modern SoCs is forcing the industry to look for design methodologies above RTL that can be used for architectural analysis and embedded software development. In system level design of these SoCs communication takes the central role. Therefore a methodology for modeling communication at a higher abstraction level has become important. TLM is one such methodology that promises to be used in system level SoC modeling for early software development, architectural analysis and functional verification. The three main TLM ingredients are the definition of the abstraction layers, the APIs and the library of modeled components. The abstraction layers help designers to deal with certain aspects in their exploration while abstracting out irrelevant details. The communication interfaces enable TLM achieve separation of communication from computation and interoperability between components. By incorporating TLM in a SoC design flow, it is possible to model systems at various abstraction levels each with different level of accuracy and simulation time.

## Chapter 3

# Evaluation of Related Works

In the past few years, various researches have been done related with TLM-based system level modeling. This chapter discusses these works with a brief description for each of them. Due to the breadth of the topic, the related works investigation is categorized into two sections. Section 3.1 summarizes a list of literatures that focus on defining abstraction layers in TLM. Different efforts on developing TLM frameworks are discussed in section 3.2. However, emphasis is mainly given to works that use non-proprietary programming languages such as SystemC [12]. Section 3.3 summarizes the features of these frameworks. Based on the assessments in the above sections, section 3.4 justifies with four reasons the need for the TLM-based system modeling methodology presented in this thesis.

### 3.1 Abstraction Layers

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels for transaction level modeling. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction and use cases. In this thesis, a survey is conducted on efforts made to specify the levels of abstraction layers and their features in TLM.

A. Gerstlauer et al. [11] presented a division of the system level design process into three models that support both computation and communication abstractions between requirement and implementation models: *specification*, *multiprocessing* and *architecture layers*. Table 3.1 shows the abstraction layers and their features in [11]. They showed the use of their abstraction layers on a JPEG decoder.

A. Donlin [9] presented a series of use models and flows for the exploitation of transaction level modeling in the development of complex system designs based on four abstraction layers between algorithmic (ALG) and cycle-accurate (CA) models: CP, CP+T, PV, PV+T. Five use models(UM1-UM5) are described according to the particular type of system-product being designed. For each of these use models, refinement flows are proposed selecting abstraction layers among the four.



TABLE 3.1: Abstraction layers in system level design as presented in [11]

Level	Computation	Communication	Structure	Order	Validate
Requirement	Concepts	Tokens	Attributes	Constraints	Properties
Specification	Behaviours	Messages	Behavioural	Causality	Functionality
Multiprocessing	Processes	Messages	Processors	Execution Delays	Performance
Architecture	Processes	Busses/Ports	Bus-functional	Timing-accurate	Protocols
Implementation	FSMDs	Signals	Micro-architecture	Cycle-accurate	Clock cycle

L.Cai et al. [13] defined six abstraction models in the system design process as shown in Table 3.2 and Figure 3.1. However use cases of the abstraction layers and details for implementation are not provided.

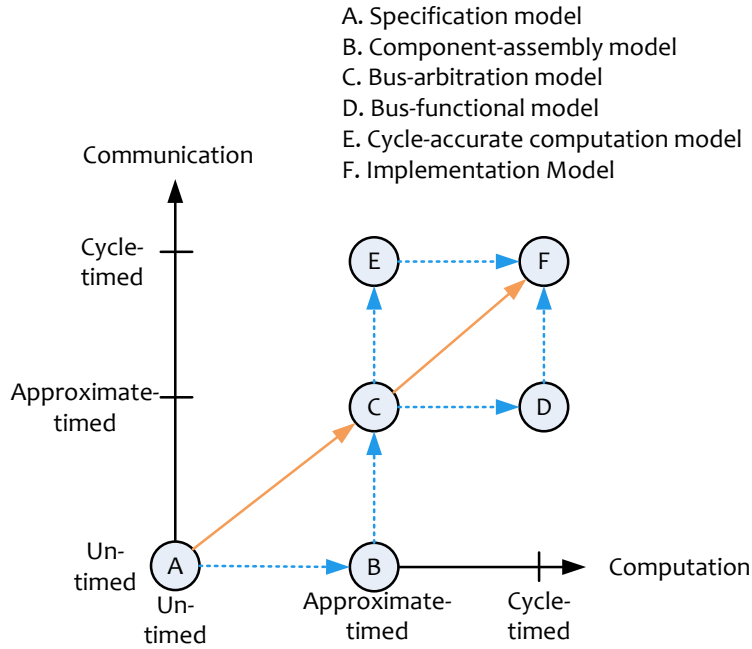


FIGURE 3.1: A 2D-view of abstraction layers in TLM as presented in [13]

The OCP TLM [14] defined four TLM use models: Functional View(FV), Architects View(AV), Programmers View(PV) and Verification View(VV). However, these use-models do not have a one-to-one correspondence with a particular abstraction layers. Instead, these use-models define the requirements imposed on a transaction level model to be adequate for a certain purpose.

A.Deb et al. [15] defined four abstraction layers above the implementation layer with the features and use cases shown in Figure 3.2 and Table 3.3.

TABLE 3.2: Abstraction layers in transaction level modeling as presented in [13]

Models	Communication Time	Computation Time	Communication Scheme	PE Interface
Specification Model	No	No	Variable/Channel	(No PE)
Component-assembly Model	No	Approximate	Message-passing channel	Abstract
Bus-transaction Model	Approximate	Approximate	Abstract bus channel	Abstract
Bus-functional Model	Time/Cycle-accurate	Approximate	Detailed bus channels	Abstract
Cycle-accurate Computation Model	Approximate	Cycle-accurate	Abstract bus channel	Pin-accurate
Implementation Model	Cycle-accurate	Cycle-accurate	Wire	Pin-accurate

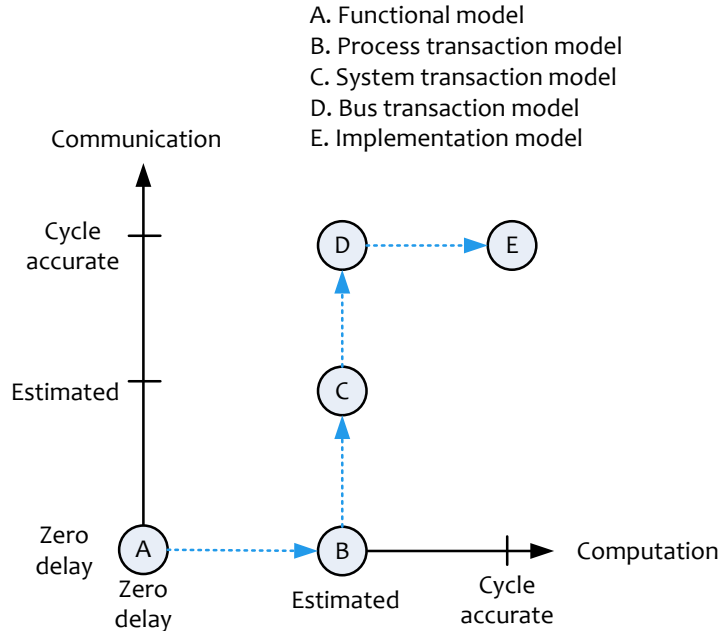


FIGURE 3.2: A 2D-view of abstraction layers in TLM as presented in [15]

### 3.2 SystemC-based Frameworks

Both in the academia and industry, there has been a significant effort in the development of system level modeling and simulation frameworks. Because of the reasons listed in section 3.4, our investigation does not include commercial products ([16–18]), frameworks which are based on proprietary languages or are non-open source. The following subsections present brief description of those open-source system level modeling methodologies which are based on SystemC.

TABLE 3.3: Abstraction layers in system level modeling as presented in [15]

Model	Communication	Computation	Transaction/Operation	Model Characteristics
Functional Model	Zero-delay	Zero-delay	Process execution (atomic execution consisting of read, computation, and write operations)	Process to resource mapping is not done, communication through infinite length FIFO
Process Transaction Model	Zero-delay	Estimated	Bulk read from FIFO, Computation (C function call, with estimated delay), Bulk write to FIFO	Process to HW/SW mapping is assumed, communication through finite length FIFO, read/write to FIFO using get/put procedures.
System Transaction Model	Estimated	Estimated	Bulk read from FIFO (with estimated delay), computation (C function call with estimated delay), bulk write to FIFO (with estimated delay)	Communication through get/put procedures from/to FIFO with estimated delay for memory access using shared medium
Bus Transaction Model	Cycle-accurate	Estimated	Memory read (Req, Ack, Address, Data, Split), computation (C function call with estimated delay), memory write (Req, Ack, Address, Data, Split)	Communication through cycle accurate component interface and shared medium, read/write to memory using physical address
Implementation Model	Cycle-accurate	Cycle-accurate	Memory read (Req, Ack, Address, Data, Split), cycle-accurate computation (RTL, ISS), memory write (Req, Ack, Address, Data, Split)	Cycle-accurate computation-RT level for HW implementation, or instruction level for SW implementation.

### 3.2.1 OCCN

OCCN [5], On-Chip Communication Network, is an open-source research and development framework for the specification, modeling and simulation of on-chip communication architectures. It provides an API and object oriented C++ library built on top of SystemC. The Master Interface (MasterIf), the Slave Interface (SlaveIf) and the Protocol Data Unit (PDU) in the API enable communication between components. The library can be extended with new components with guaranteed interoperability as long as the common API is used. An OCCN channel implements blocking send and receive methods for PDUs. Receive delays and timeouts can be specified as parameters, enabling timed-modeling. Despite the good API based framework, the library does not have sufficient models of components and lacks guideline for use cases and abstraction layers. It leaves these issues as an implementation detail upto the user. In addition, computation aspects are out of the scope of this framework as it is a tool for communication networks. Hence the library as it is, without additional extensions, can not be used for system level modeling.

### 3.2.2 TLM 2.0

Founded in 2003, the Open SystemC Initiative (OSCI) TLM Working Group (TLM-WG) pioneered the development of TLM frameworks for SystemC with the release of the OSCI TLM kit 1.0. A set of three interfaces is provided that form the heart of the kit,

enabling unidirectional blocking, unidirectional non-blocking, and bidirectional blocking transfers. The TLM 1.0 kit does not define standard data types nor abstraction levels. The intended use of the kit is to develop customized channels with it, using application-specific data structures and user-defined protocols. Thus, the OSCI kit does not help in achieving model interoperability. Having identified these issues, the OSCI TLM Working Group released a revised version of the kit, OSCI TLM 2.0, in June 2008 [8]. TLM-2.0 consists of a set of core interfaces, analysis interfaces, initiator and target sockets,. It also provides a generic payload and base protocol which enables modeling memory-mapped buses. TLM 2.0 aims to be an industry standard transaction level modeling framework that enables interoperability between models. Hence it kept itself away from specifying abstraction layers and use cases around them. Rather, it recommends coding styles which are appropriate for, but not locked to, various use cases. Though this increases its flexibility for wider industry use, it introduces vagueness on implementation. Just like OCCN, TLM 2.0 is all about communication. Therefore, a library of TLM 2.0 modeled components, including computation units, is needed to have a complete system level modeling framework. In addition, TLM 2.0 does not have the generic payload for Network-on-Chip interconnects.

### 3.2.3 OCP SystemC Channels

The OCP-IP (Open Core Protocol - Intellectual Property) [19] provides a comprehensive library of point-to-point channels for modeling of SoCs based on the Open Core Protocol with SystemC. Three levels of abstraction are supported, namely OCP-tl0 for cycle accurate, OCP-tl1 and OCP-tl2 for CCATB (Cycle-Count Accurate at Transaction Boundaries), and OCP-tl3 for PV (Programmer's View) communication modeling. A large set of interface methods for both blocking and non-blocking channel access is provided. OCP channels use predefined data types for transfer qualifiers such as address, command, thread identifier, etc. and provide basic instrumentation for transaction monitoring. However, they do not provide any means for communication architecture simulation, as they only support point-to-point communication.

### 3.2.4 GreenBus

GreenBus [20] is an open-source TLM fabric built on top of the SystemC-2.1 library. GreenBus supports simulation of both point-to-point communication modeling with abstract channels and communication architecture of different bus architectures. GreenBus follows a layered approach, as shown in Figure 3.3, that decouples the PE communication interfaces from the GreenBus interface, which is the underlying low-level transport API denoted as *connection layer*.

GreenBus has the TAQ(Transaction-Atom-Quark) approach for the refinement of communication transactions in TLM using the provided generic payload. Based on that, three abstraction layers are identified: Programmer's View(PV), Bus Accurate(BA) and Register Transfer Level(RTL). Timing and data refinements are well discussed along with implementation details. The authors claim that the concepts in the GreenBus framework are used in the development of the industry standard TLM 2.0 framework. However, abstraction layers in computation, Network-on-Chip interconnects and use cases are not within the scope of the GreenBus TLM fabric.

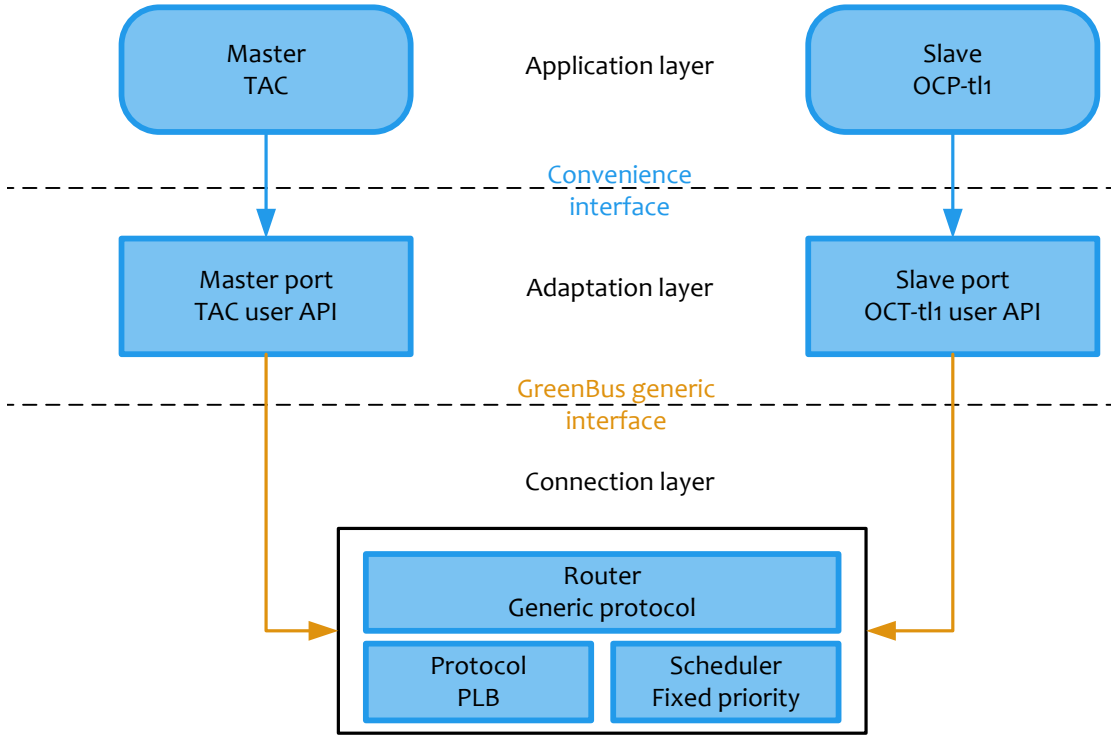


FIGURE 3.3: Layered Approach in GreenBus. (source: *GreenBus White Paper* [20])

### 3.2.5 CCATB AMBA Channels

In their paper on the CCATB simulation approach [21], Pasricha et al. present a CCATB simulation model for the AMBA AHB and AXI buses and compare them with ARM's cycle accurate bus functional models. Figure 3.4 shows where the CCATB modeling abstraction layer falls in the system design flow.

CCATB models aim at achieving fast simulation while encompassing sufficient timing points for architectural exploration. A speedup of 55% with respect to a cycle accurate model is achieved in their experiments. This limited result may be due to the utilized instruction set simulator.

### 3.2.6 OSSS

The goal of the OSSS library from OFFIS Oldenburg [22] is to enable HW/SW communication modeling with synthesizable channels. OSSS supports a two-layered channel model. Internally, a set of `sc_signals` is used, which is equivalent to RTL wires and therefore synthesizable. An application layer provides more abstract convenience methods. For bus modeling a predefined read/write interface is proposed. A protocol library contains different implementations of these interface methods: e.g. to simulate Core-Connect's OPB (On-chip Peripheral Bus). Data transport and arbitration in OSSS is performed at the RTL level of abstraction. Thus, bus simulation is cycle accurate. The

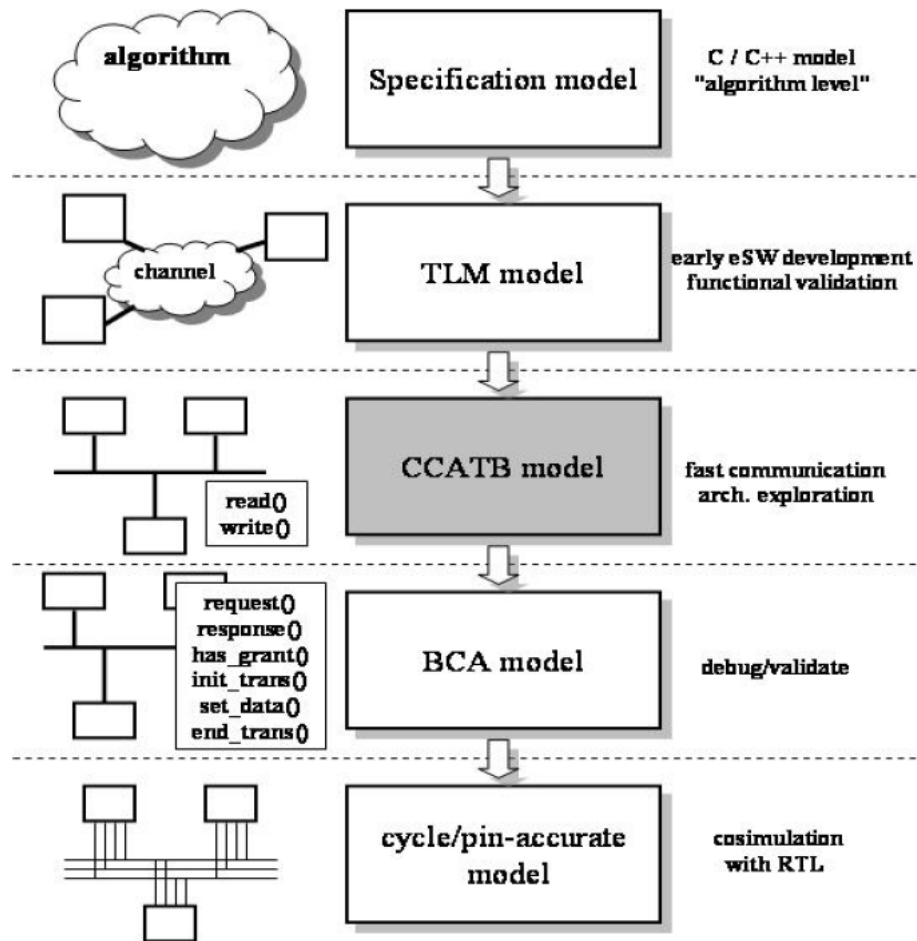


FIGURE 3.4: CCATB in System Design Flow (source: [21])

simulation performance achieved with this approach, hence, is hardly better than with pure RTL models. HW/SW communication synthesis is not discussed.

### 3.2.7 ARMn

ARMn [23] is an open-source multiprocessor cycle-accurate simulator which can simulate a cluster of ARM processor cores connected by custom communication schemes, such as wormhole based packet-switching, fully connected crossbar switches or regular standalone buses. The topology supported include mesh, torus and star shape. The PE model used by ARMn is based on the ARM simulator, SimIt-ARM [24]. The communication part is written in SystemC which wraps the PE model around and enables it to communicate with other SystemC modules. ARMn focuses on a very specific architecture and shows the possibility of integrating a cycle accurate ISS (Instruction Set Simulator) processing element model with high level SystemC interconnection networks.

### 3.3 Evaluation

From the investigation carried out on related multi-core modeling frameworks, it is found that, in one or the other way, each of them does not fit the objectives of this thesis (Chapter 2) in their current shape. This leaves us two alternative ways in the development of the multi-core modeling methodology. The first option is to build everything from the scratch as a new framework while the other alternative is to reuse existing works with the necessary additions and modifications to make them fit our needs. Taking into account the time and resource available for this thesis work, a preference is made for the second option. We figured out that TLM 2.0 [8], GreenBus [20] and OCCN [5] are the three top candidates for this purpose mainly because of their generic nature (3.4).

GreenBus and TLM 2.0 more or less follow the same approach and even GreenBus's communication refinement concept is reused in TLM 2.0. The GreenBus distribution package comes with a set of bus interconnect examples and complete communication APIs for bus based systems. However its scope is limited to the refinement of bus-based communication protocols and the necessary APIs for bus-based systems.

Though the TLM 2.0 is a powerful standard that provides all the communication APIs and the generic payload for memory-mapped bus interconnects, it does not have a similar off the shelf construct for NoC interconnects. The distribution package has some simple examples in it, but neither a library of components nor definition of abstraction levels. Even though all of these can be constructed from the provided package, it is dropped from the list of candidates because the creation of a new payload for NoC interconnects contradicts the notion of TLM 2.0's interoperability objective and the construction of a complete library of components from the scratch might take a considerable time. Nevertheless, as it will be discussed in *Chapter 6: Conclusions and Future works*, TLM 2.0 is a promising and powerful standard for the construction of a multi-core system level modeling methodology.

The scope of OCCN is limited to the construction and refinement of on-chip communication networks such as buses and NoCs. Even though the distribution package does not come with a complete library of model of components, the example generic bus, point-to-point channel and NoC models give directions on how more accurate and specific interconnects can be built up. Hence we decided to construct our system level modeling methodology around OCCN by adding the specification of computation refinement levels, definition of system level abstraction layers, models of computation and memory components as well as models of additional communication networks. The methodology also includes strategies on how to integrate models of various abstraction layers in the SoC design flow.

### 3.4 Justification

Both in the academia and industry, there has been a significant effort in the development of system level modeling and simulation frameworks. In the previous chapter, some of the main ones are presented and their key features are pointed out. In this section, four main reasons are discussed that triggered the need for the TLM-based system level modeling methodology devised in this thesis work.

TABLE 3.4: Summary of features of investigated frameworks

Features	OCCN	TLM 2.0	ARMn	OCP Channels	GreenBus	CCATB AMBA Channels
Abstraction levels	☒	☒	☒	☑	☑	☒
Communication interfaces	☑	☑	☑	☑	☑	☑
Computation models	☒	☒	☑	☒	☒	☒
NoC support	☑	☒	☑	☒	☒	☒
Models of components	☒	☒	☑	☒	☒	☒
Genericity	☑	☑	☒	☒	☑	☒

### 3.4.1 Languages

Since electronic system designers accepted system level modeling as a tool to cope with the ever increasing complexity of modern SoC development, different programming languages have been proposed to raise the level of abstraction. The first category of such languages are the hardware-oriented languages such as VHDL, Verilog, SystemVerilog, etc. These languages are found to be quite slow for complex system level modeling and unapt for software modeling. The other category of languages are general purpose programming languages such as C, C++, Java, etc. These languages are software-oriented and hence mostly do not have enough constructs for hardware descriptions. The third category are proprietary languages such as CowareC, SpecC, HardwareC, MyHDL, etc. Mostly languages in this category are designed to be appropriate for ESL modeling by balancing between the first two categories. However they also have a number of drawbacks. Some of the disadvantages of proprietary languages are [3]:

- Long learning curve for mastering each specific tool suite.
- Difficult to exchange models between different teams because of license issues and tool suite installations.
- High cost owing to license fees required for model development and simulation.
- Uncertainty of support for languages coming from research projects.

Due to these reasons, we found system level modeling approaches which are based on hardware-oriented, software-oriented or proprietary languages unfit to fulfil the objectives of this thesis work which are listed in Chapter 2. Hence, we focused on approaches which are based on non-proprietary languages that support hardware-software co-simulations. For this purpose SystemC [12] is found to be the best candidate due to the following reasons.

- It is a non-proprietary, free and open-source language.
- It is an industry standard language that assures interoperability.
- It supports both hardware and software primitives.
- A simulation kernel is available with it.



- It depends on standard C++ compilers.
- Easy learning curve as it is based on the C++ language.

Our related work investigation, thus, dropped all approaches which are not based on SystemC.

### 3.4.2 Integrated Approach

In Chapter 1, the basic ingredients needed in the development of a complete system level modeling and simulation framework are pointed out. Most of the related works investigated miss one or more of these ingredients. In this thesis work, it is tried to come up with an integrated modeling approach that covers the following.

- The definition of the levels of abstraction
- The preparation of the library which includes the communication APIs, models of interconnection networks, processing elements, memory and transaction interfaces
- The modeling of software applications
- Integration and simulation strategies

We believe that this integrated approach puts the foundation for multi-core modeling and simulation framework which can be extended and refined with future works.

### 3.4.3 Cost

There are a number of commercial products for TLM-based system level modeling based on SystemC [18], [16], [17]. These products may have sufficient library of components, fast simulation kernels and an easy-to-use graphical user interfaces that makes the life of the designer quite easy. However, these products are quite expensive and may be unaffordable for small-scale projects, academia researches and personal use.

### 3.4.4 Efficiency

Last but not least, all modeling approaches are not equally efficient. They may vary in their level of accuracy, simulation time, design effort, learning curve and fitness for the intended purpose. The continuously changing and growing nature of modern SoC development also brings new issues in the modeling arena. Hence, the vastness and dynamism of the topic dictate that there is always a space for improvement in the modeling strategies followed.

### 3.5 Summary

This chapter justifies with four main reasons the need for the TLM-based system level modeling and simulation framework developed in this thesis work. It is found that most of the investigated related works do not use a non-proprietary language, do not have all the necessary ingredients for system level modeling, are costly and/or not openly available. Therefore, to come up with an efficient framework for the modeling of multi-core systems, our methodology aimed at devising a SystemC-based integrated approach that focused particularly on streaming applications. From the investigation carried out on related multi-core modeling frameworks, it is found that, in one way or the other, each of them does not fit the objectives of this thesis (2) in their current shape. Hence in this work, it is preferred to reuse existing frameworks with the necessary additions and modifications to make them fit our needs rather than creating a new framework from scratch. And for that purpose, the OCCN [5] library is used.



## Chapter 4

# Methodology for TLM-based System Level Modeling

This chapter discusses the TLM-based system level modeling methodology proposed in this thesis work. The approach is categorized into five main sections. In the first section, definition of abstraction levels is given. Section 4.2 details the construction of the TLM-based library which is based on the open-source OCCN [5] framework. How this library can be used for building virtual SoC platforms is presented in section 4.3. Next, the software application development aspects are covered in section 4.4 and bringing the whole approach together for the system level modeling of multi-application multi-core SoCs is discussed in section 4.5.

### 4.1 Abstraction Levels

The abstraction levels defined in this thesis work are by large inspired by the concepts discussed in *Chapter 3: Related Works*. The 2D, communication versus computation, graphs used in [13] and [15] are nice figures to show the orthogonalization of concerns [25] and the refinements in computation and communication. Thus, the same approach is followed in this work.

#### 4.1.1 Communication Refinement

Communication abstraction comprises both timing and data abstraction. A bottom-up approach can help to define these levels of abstractions. Figure 4.1 shows an instance of a bus transaction.

During these transactions, signal values are changed by both the master and slave. The transactions are initiated by a clocked request signal. After the bus arbiter grants access, the master sets the target address, which later is acknowledged by the slave. Then data transmission is performed. The first transaction shows a single-beat transfer, whereas the second transaction is a burst transfer. The timing length of the transactions is defined by the idle/busy state changes of the protocol. Now we can consider different time points of the communication to model it at different levels of timing abstractions

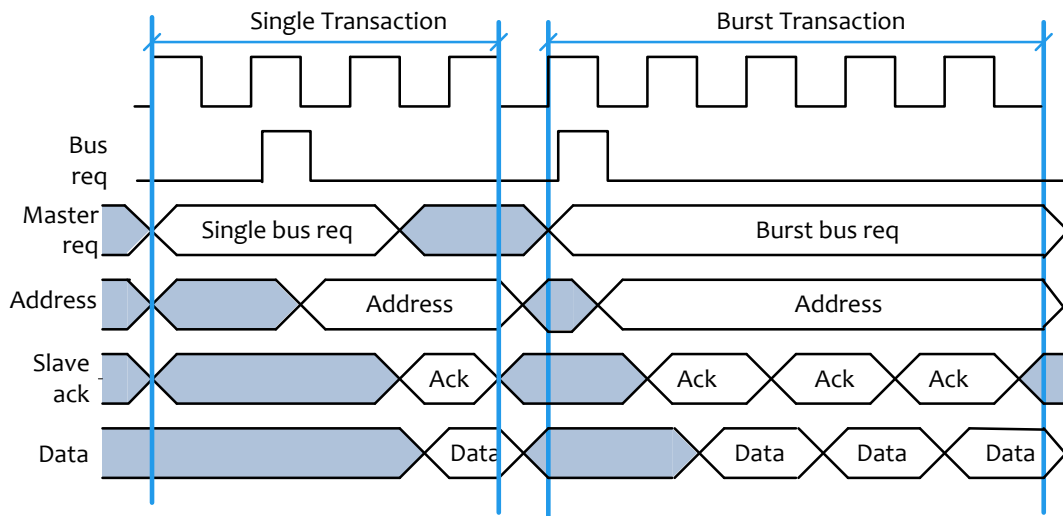


FIGURE 4.1: An instance of a bus transaction

as shown in Figure 4.2. A similar approach is originally used in GreenBus [20] and later by TLM 2.0 [8].

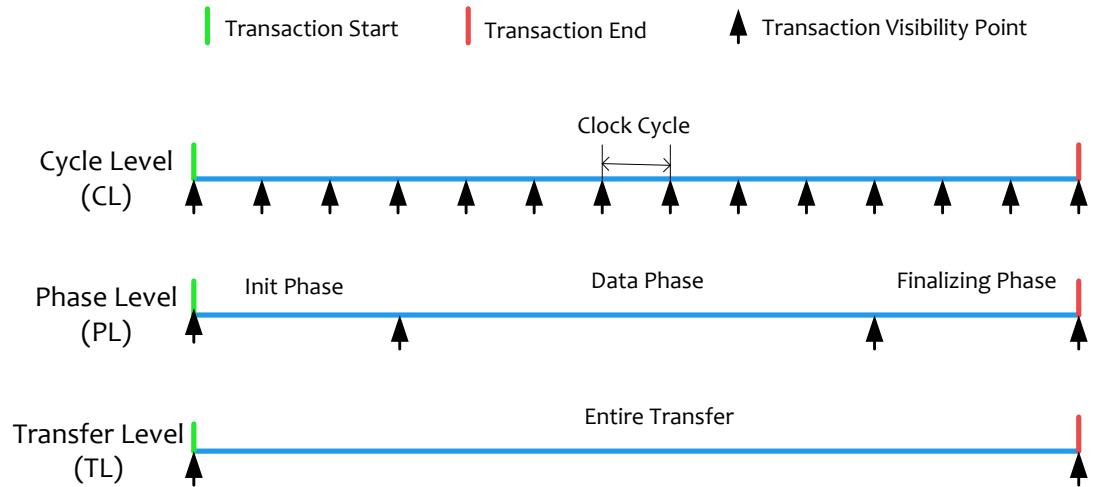


FIGURE 4.2: Communication Refinements

- **Cycle level:** Signals are simulated at each clock cycle and value changes are traced.
- **Phase level:** The entire transaction is split into phases that show distinct aspects of the protocol that the user is interested in and timing is limited to the start and end points of the phases. For instance, for the above bus transaction, we can have three phases.

- *Initialization phase*: the exchange of control signals from the master request upto the point where the master and the slave are ready to exchange data.
  - *Data transfer phase*: the transfer of the data and all accompanying qualifiers such as byte enables or error flags.
  - *Finalization phase*: the exchange of all control signals to release the bus.
- **Transfer level**: change of signals and timing only at the start and end of the entire transaction are simulated.

Simulation speed increases in the order *Cycle-Phase-Transfer* level but the accuracy of the simulation result decreases in that order as well. In addition, the visibility of the transaction drops as we go from cycle to transfer level. This means certain category of information can be extractable in one level but not in the others. For example on the transfer level, we have information on how long the entire transfer takes but no clue on how much of this time is spent due to arbitration delay, slave overhead or transmission delay. However this information can be obtained if the communication protocol is simulated on the phase level though at lower accuracy than the cycle level.

#### 4.1.2 Computation Refinement

Transaction level modeling mainly focuses on the communication aspect of the SoC design process. This is reasonable because using standard IPs is a very common phenomenon in today SoC designs and therefore the main focus of the design will be on performance optimization and functional verification of the communication architecture.

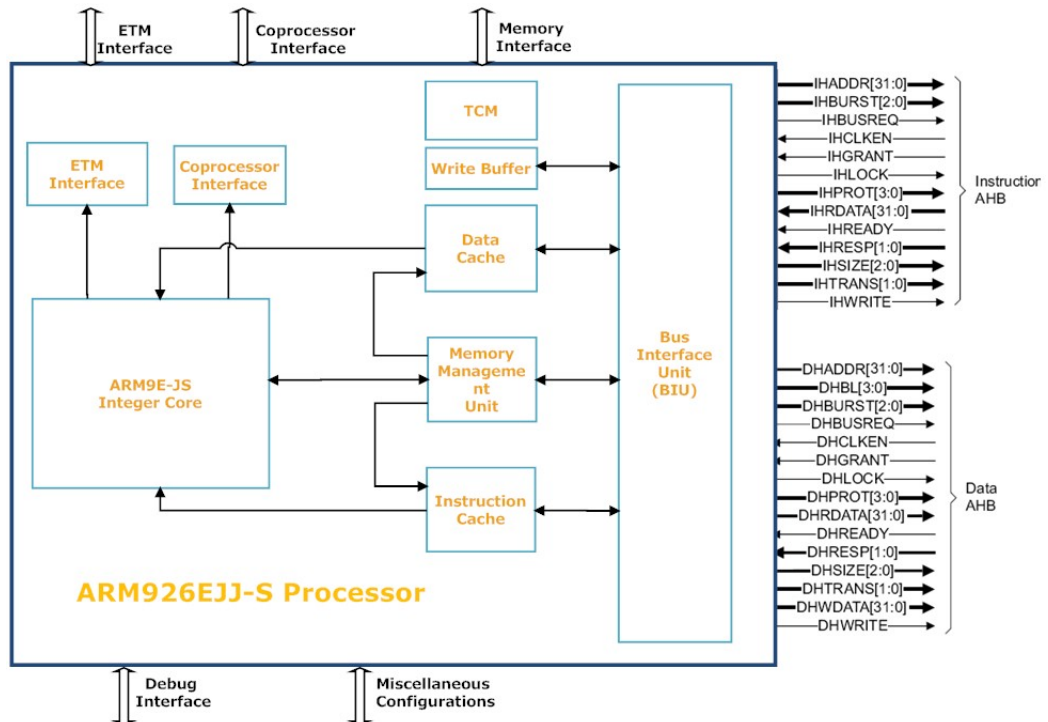


FIGURE 4.3: Example embedded processor: ARM926EJ-S

Non-functional models, where IPs are modeled as a black box and simulated with synthetic workload, are a suitable choice for design space exploration and high level architectural exploration. However for functional modeling where the virtual platform is going to be used for real software development and analysis, the computation models play a very critical role. Even architectural analysis of SoCs which are designed for multiple and/or dynamic applications can not provide dependable performance figures without simulating the actual workload using appropriate computation models.

Following a bottom-up approach as previous, let's consider a typical embedded processor (e.g. ARM processor shown in Figure 4.3), with internal cache memory, that runs some OS (Operating System) features and other software applications. When the system is started, a boot program loads the OS (and other applications) to the system memory. The processor fetches these programs from system memory and starts executing.

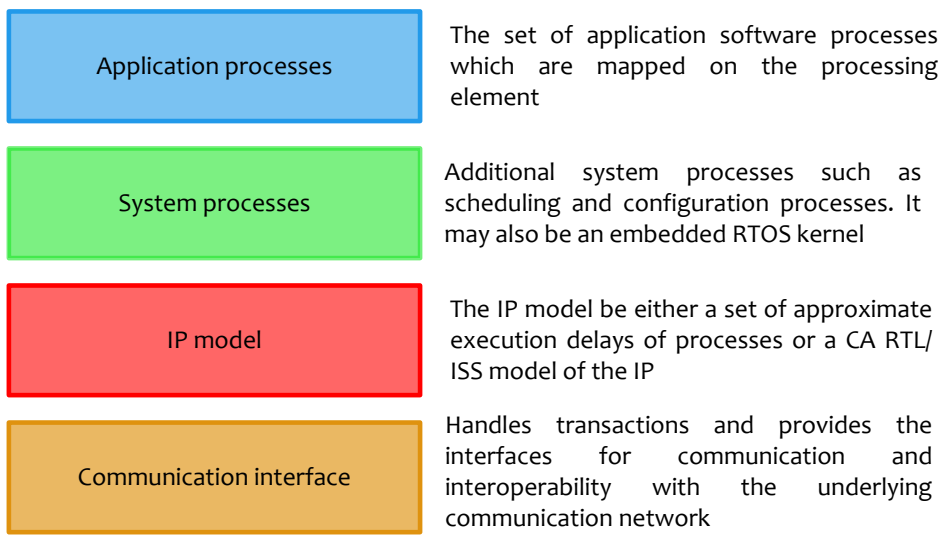


FIGURE 4.4: Separation of concerns in modeling computation units

Below here three levels of refinements are listed in modeling computation units. Each level is defined based on how it manages the four main concerns shown in Figure 4.4.

- **Computation Refinement Level 0 (CRL0):** CRL0 is a single-process computation model. Assuming the processor runs only one process, the effect of the OS kernel is very minimal and can be ignored. The software process runs on the simulating host machine but the IP model may have a static approximate timing to reflect the execution delay of the process when it runs on the actual IP. The communication interface serves to connect this processing element to the underlying interconnection network and have the necessary features for proper communication.
- **Computation Refinement Level 1 (CRL1):** CRL1 is a multi-process computation model. When multiple processes are mapped on the processing element, some OS features are needed for tasks such as scheduling and configuration. Approximate timing delays are used for the IP model to analyse the effect of the OS kernels and execution delays of the processes on the actual IP. The communication interface serves the same purpose as previous.

- **Computation Refinement Level 2 (CRL2)**: CRL2 is a multi-process and specific-processor computation model. On this abstraction level more accurate simulation is needed. The IP model is an ISS/RTL model of the specific IP being modeled which may have to be wrapped for appropriate communication with the external communication architecture. Additional OS features may be added as needed.

Table 4.1 shows the computation refinement levels and their features.

TABLE 4.1: Computation Refinement Levels and their Features

Concerns	CRL0	CRL1	CRL2
<b>Application processes</b>	Single process	Multiple processes	Multiple processes (binaries )
<b>System processes</b>	Not needed	Single or multiple process(es)	Single/Multiple process(es) (binaries)
<b>IP model</b>	None/Approximate execution delay	Approximate execution delay	CA-ISS
<b>Communication interface</b>	TLM interface	TLM interface	Pin-interface/ TLM interface

#### 4.1.3 Abstraction Levels in System Level Modeling

Combining the various communication and computation refinements discussed in the previous subsections, different abstraction levels in system level modeling can be defined. While defining the abstraction levels, a **use case** centric approach is followed. A use case refers to the particular purpose a model is being designed for. This approach is selected based on the practice and interest of the SoC industry. The advantage of use case based approach is that features of each level can be selected from the perspective of desired use case. The drawback on the other hand is that it limits the defined set of abstraction levels to the covered use cases only. However, it is always possible to define new intermediate abstraction levels with the inclusion or removal of a set of features between the already specified ones.

The main goal in the definition of each abstraction level is to obtain a system level model for the desired use case while trading accuracy for simulation time. The 2D graph shown in Figure 4.5 shows the different abstraction levels defined for system level modeling.



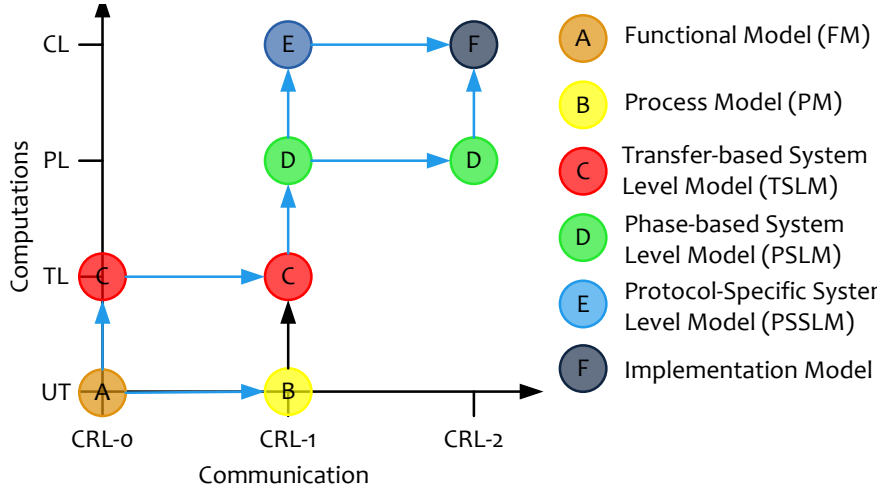


FIGURE 4.5: Abstraction layers in TLM-based System Level Modeling

It should be noted that other intermediate abstraction layers can be defined by combining various communication and computation refinement points per component level. Depending on the user's point of interest, a TLM model that emphasize certain aspects of the system while abstracting the rest can be modeled making use of the separate communication and computation refinements proposed above. For example, in PSLM it might be reasonable to model some processing nodes at cycle-accurate level (CRL2) while keeping the rest at CRL1 level.

#### 4.1.4 Features and Use Cases of the Abstraction Levels

The distinctive features and use cases of the various abstraction layers is discussed in the following subsections.

##### 4.1.4.1 Functional Model (FM)

Parallel computer programs are more difficult to write than sequential programs; because concurrency introduces more software bugs, such as race conditions. Performance of the parallel program depends on the communication and synchronization points between the parallel processes. This implies the partitioning influences the amount of communication to be performed.

The main goal of FM is an architecture-free functional modeling of multiple concurrent processes at the very early stage of the design. Performance statistics extracted from this model can be used to compare various software partitioning options and improve parallelism of processes execution. This model is similar to the specification models of [11] and [13], the CP model of [9], FV model of [14] and the FM model of [25]. Typical features of this model are the following:

- A set of processes running in parallel and connected with point-to-point dedicated links based on their data flow dependency.

- This is an architecture-free implementation: No decision is made on the number of processing elements, interconnection network, task-resource mapping and memory size.
- Processes have infinite input and output buffer sizes.

Some uses of models at this abstraction layer includes an architecture-free software development at functional level, assisting decision on software partitioning (partitioning points and data sizes) and starting point buffer sizes for models at a lower abstraction.

#### 4.1.4.2 Process Model(PM)

When task to resource mapping is done, system resources such as processors and communication interfaces may be shared between multiple tasks resulting in resource contention. Hence it is very important in the design flow to analyse the schedulability and implementability of the system over the limited system resources available. Certain data flow networks such as SDF have efficient techniques for verifying schedulability and implementability over fixed FIFO sizes without actually running the process executable [26]. However, complex real world streaming applications possess very dynamic behaviour that makes such data flow models unfit for this purpose. Thus, the process model aims at achieving this goal by refining the functional model by grouping processes and mapping them onto a processing node while largely preserving the original processes' communication. In addition, this model can be used as a bridge to the other lower abstraction layers easing the modeling effort.

PM is a partially architecture dependent model that introduces task mapping and scheduling. Models at this layer can be used to check implementability of the streaming applications with fixed buffer sizes and number of processing elements (PEs). This model is similar to the multiprocessing model of [11] and the PTM model of [15]. The interconnection network is the same point-to-point links used in the FM. The sets of processes are taken from the FM and additional design decisions are made. The design decisions at this level include process mapping, process execution delay, task scheduling and input-output buffer sizes.

Models at this level of abstraction can be used to study impacts of design decisions on mapping, scheduling and buffer sizes. OS kernels can also be modeled as additional processes (for instance as a non-functional task model of a given execution delay) and their impact on the system can also be considered. The developer may iteratively switch between FM and PM to investigate between different software partitioning, mapping and scheduling options.

#### 4.1.4.3 Transfer-based System Level Model (TSLM)

Software development and its performance evaluation without considering the SoC platform interconnect architecture does not lead to realistic performance results. In most real SoCs where a shared bus or a NoC is used as communication network, contention occurs as multiple transactions may compete to access the network and transactions

have to be arbitrated. In addition, the communication network has fixed bandwidth which mandates the variable size transactions of the processes in FM and PM models to be fragmented into multiple transactions of a given fixed size. The implementation of the FIFOs of the FM and PM models is also important information for software development which should be brought into the modeling picture at this stage. For example the FIFOs could be implemented using message passing or shared memory; the single address space of the shared memory could have centralized or distributed physical memory. There is a wide design space related to the implementation of the communication architecture as well. On bus interconnects, for instance, decisions have to be taken on bus width, arbitration priorities, number of layers, the standard protocol, etc. Instead of a bus, a NoC-based architecture can be used. However, modeling the communication protocol in these communication networks with its full timing points makes the simulation time too slow for software development that simulates applications and possibly operating systems.

TSLM aims at providing an architecture-dependent virtual SoC platform model for software development and analysis. Some features of this abstraction level are given below.

- This layer is inspired by the LT coding style of the TLM 2.0 standard [8] and has similarities with the bus-arbitration model of [13], the STM model of [15], and the AV of [14].
- The same nodes with mapped processes which are used in the PM model are reused here.
- The point to point communication links of the FM and PM levels are replaced by a high level interconnect model such as bus or NoC with appropriate arbitration.
- Transactions use a blocking interface with only two timing points to achieve high simulation speed. The first timing point is the transport call from the initiator to the target and the second timing point is the return of the transport function from the target back to the initiator. These timing points are typically associated with the beginning of the request and response phases of the transaction.
- All transactions are done using a *protocol data unit* which is a data structure that models the signalling in the intercommunication network.

#### 4.1.4.4 Phase-based System Level Model(PSLM)

Though the TSLM level, which only has two timing points, is suitable for software development and analysis, the limited amount of timing details makes it unsuitable for the communication architecture exploration. A more detailed transaction-level model need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. The approximately timed (AT) coding style in TLM 2.0 [8] standard and the Bus-accurate (BA) layer of GreenBus [20] are examples of such a technique. Even though more timing points increase the simulation time, this model is significantly

faster than cycle level simulations. With PSLM, HW-SW co-simulation can be done by trying out various design options such as bus transfer modes (burst types and beat sizes), single-layer versus multi-layer bus, router arbitration techniques, bus width, bus protocol, multi-port versus single port DMA and so many others.

The main goal of PSLM is an architecture-dependent virtual SoC platform model for communication architecture analysis. It offers a faster simulation time and less design effort modeling option than cycle accurate network modeling. It can use the same nodes as used in TSLM as long as the necessary modifications are done on the communication interfaces. Process execution can be modeled with an approximate timing delay. Transactions use non-blocking interfaces with more timing points including the two timing points of TSLM.

#### 4.1.4.5 Protocol-Specific System Level Model(PSSLM)

Once design exploration is made on the communication architecture using PSLM abstraction layer, verification of the specific communication protocol needs to be done. In PSSLM, the limited timing points of PSLM are extended to full scale that evaluate signal changes at every clock cycle. The generic protocol data unit should also be replaced by the specific data unit of the communication protocol planned to be used. The processing nodes can be modeled either fully at multi-process level (CRL1) or a combination of multi-process(CRL1) and cycle accurate level (CRL2). This model can be used for functional verification of the selected communication protocol and to validate if certain system specifications are satisfied such as throughput and latency.

#### 4.1.4.6 Implementation Model (IM)

The Implementation Model (IM) is a complete but non-synthesizable representation of the SoC in a high level programming language. The model can be used for functional verification, system specification validation and debugging. It can also be used as the reference model for RTL modeling of the SoC. Simulation is considerably slow as compared to the PSSLM and PSLM models. However, accurate results are guaranteed.

## 4.2 Library

As discussed in section 2.2, communication interfaces and models of components are among the essential ingredients needed for TLM-based system level modeling. A well designed library of components and communication interfaces, therefore, assures efficient modeling strategy as model reuse, reliability and extendibility are quite high. For the demonstration of the system level modeling methodology devised in this thesis work, we preferred to reuse an existing library rather than creating a new one from the scratch. The library is based on the open-source OCCN [5] package. Even though the authors claimed that they have used their library to experiment a bunch of specific communication protocols such as AMBA and STBus as well as some real SoC models, the open-source distribution they provided on their website has only the core communication interfaces, a point-to-point channel and a generic bus interconnect. In this section,

it is clearly shown what modifications have been done on the package, which components are reused and what new components are added.

#### 4.2.1 OCCN's Communication Methodology

The OCCN methodology focuses on modeling complex on-chip communication network by providing a flexible, object-oriented C++-based library built on top of SystemC as shown in Figure 4.6.

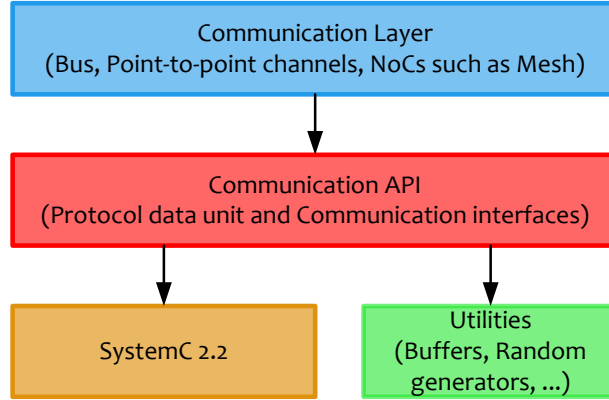


FIGURE 4.6: OCCN's library construction

The OCCN methodology defines three distinct OCCN layers [5]: Communication layer, Adaptation layer and Application layer from bottom to top. Communication between these three layers is made possible by the two OCCN APIs: the communication API and the application API. The OCCN communication API is based on a message-passing paradigm providing a small, powerful set of methods for inter-module data exchange and synchronization of module execution. This paradigm forms the basis of the OCCN methodology, enhancing portability and reusability of all models using this API. The application API enables communication between the application and adaptation layers. Since the adaptation layer is left out in the experimentation of this thesis, the application API is avoided as well.

When applying the OCCN conceptual model to SystemC, the following mappings can be identified.

##### 1. Communication API

It is implemented as a specialization of the *sc\_port* SystemC object. This API provides the required buffers for inter-module communication and synchronization and supports an extended message passing (or even shared memory) paradigm for mapping to any NoC. The fundamental components of the OCCN's communication API are the *Protocol Data Unit (Pdu)*, the *MasterPort interface* and the *SlavePort interface*. This API is entirely used in the modified library. However, a simpler one-directional MasterPort and SlavePort interfaces are added which are directly derived from SystemC's *sc\_interface*, instead of using the *Message-Box* class provided in OCCN. In addition, a protocol data unit for router-based interconnects is added.

The class declaration of the *Pdu* class template is shown in Listing A.1 in Appendix A. The class declarations of the *MasterPort* interface is shown in Listing A.2 in Appendix A. This class specializes the *MasterInterface* class which in turn inherits from the *MessageBox* class. The class declaration of the *SlaveInterface* is more or less the same.

## 2. Communication layer

This layer is implemented as a set of C++ classes derived from SystemC's *sc\_channel*. The communication channel then handles transactions between its different ports according to the communication protocol supported by a specific NoC. The OCCN distribution package provides a bidirectional point-to-point channel and a generic bus interconnect on this layer. In the modified library of this thesis the following components are added: a uni-directional point-to-point channel, router-based NoC interconnect (configurable mesh network) and a specialized bus interconnect (AMBA AHB 2.0 Bus).

The class declarations of a generic router and Mesh NoC are shown below in Listing 4.1 and Listing 4.2.

```
template <typename MDataUnit, typename SDataUnit>
class Router : public sc_module {
public:
    Router(sc_module_name name, unsigned int _id);
    Router(sc_module_name name, unsigned int n_master,
           unsigned int n_slave, unsigned char alg_type, unsigned int _id);
    UMasterPort<MDataUnit, SDataUnit> *master_port;
    USlavePort<SDataUnit, MDataUnit> *slave_port;
    SendingUnit<MDataUnit> **sending_unit;
    ReceivingUnit<SDataUnit> **receiving_unit;

    SoCInfo socinfo;
    int east, west, south, north;
    SC_HAS_PROCESS(Router);
    sc_in<bool> clock;

private:
    unsigned char algorithm_type;
    unsigned int num_of_masters;
    unsigned int num_of_slaves;
    unsigned int id;
    unsigned int buffer_size;
    unsigned int xDimension;
    unsigned int yDimension;
    void routingProcess();
    void createConnections();
    void setDirections();
};
```

LISTING 4.1: A Generic Router Model

```

template <typename MDataUnit, typename SDataUnit>
class Mesh : public sc_module {
public:
    Mesh();
    ~Mesh();
    Mesh(sc_module_name name, unsigned int xDim, unsigned int yDim,
        unsigned int in_buffer_size, unsigned int out_buffer_size);
    Router<MDataUnit, SDataUnit> **router;
    PPLink<MDataUnit, SDataUnit> **pplink;
    PPLink<MDataUnit, SDataUnit> **pplink_ti;
    USlavePort<SDataUnit, MDataUnit> *slave_port;
    UMasterPort<MDataUnit, SDataUnit> *master_port;
    sc_in<bool> clock;

private:
    unsigned int xDimension;
    unsigned int yDimension;
    unsigned int inBufferSize;
    unsigned int outBufferSize;
};

```

LISTING 4.2: 2D Mesh NoC Model

Figure 4.7 shows the hierarchical class library of the communication API and communication layer components. Components in darker yellow are from SystemC library. Whereas components in red color are part of the Communication API and components in blue are part of the Communication Layer.

## 4.2.2 Models of Other Components

Even though the communication API and communication layer form the core of the library, system level modeling can not be brought to life without models of additional components that form the entire SoC such as processing elements, transaction interfaces, memory and other peripherals. The OCCN package does not provide such items. Hence, for the demonstration applications (Chapter 6) all of them are created from scratch.

### 4.2.2.1 Processing Elements

Processing elements represent IPs with software program execution capabilities. It includes general purpose processors, reconfigurable processing cores as well as DSP cores. Figure 4.8 shows the model of a single processor along with the set of software processes mapped to it.

The model comprises a set of buffers, dedicated sending and receiving SystemC processes for each port, a SystemC computation process and a set of software processes which are mapped on the processing element. The software processes themselves comprise a set of tasks as discussed in section 4.4. Each task should be written in such a manner that it does not need any additional input in the middle of execution (data-independent input behaviour). An example of a process written as a set of tasks is shown in Appendix A in Listing A.4. The computation SystemC process is responsible for scheduling these tasks, checking their firing conditions and then firing them. Each software process has a set of input and output buffers dedicated to it. Listing A.3 in Appendix A shows the framework of the class declaration of processing elements. This class declaration is used in the modeling of all processing elements in the demonstration application. This

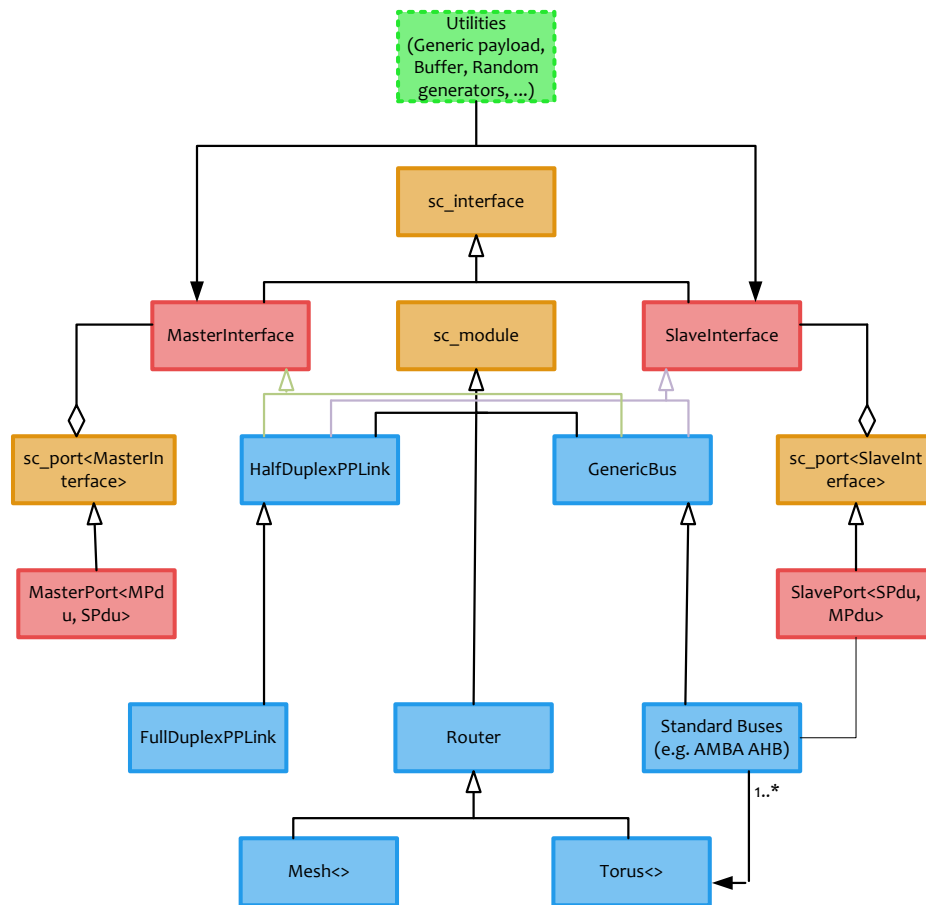


FIGURE 4.7: Hierarchical class library of the communication layer and API components

provides great help for design automation as instances of different processing elements can be instantiated on the fly at runtime by reading parameters from a configuration file.

#### 4.2.2.2 Transaction Interfaces

Transaction interfaces provide the necessary transaction adaptation between processing elements and the underlying communication infrastructure. The transaction interface model shown in Figure 4.9 has two interface sides: one to the processing elements and another one to the interconnection network. This means a change in the interconnection network can be made during architectural analysis without affecting the already constructed processing elements by just changing the interfaces on the network side of the transaction interfaces. The transaction interfaces can be mapped to communication interfaces such as Bus Interface Units (BIU) and Network Interfaces (NI) in the actual SoC hardware.



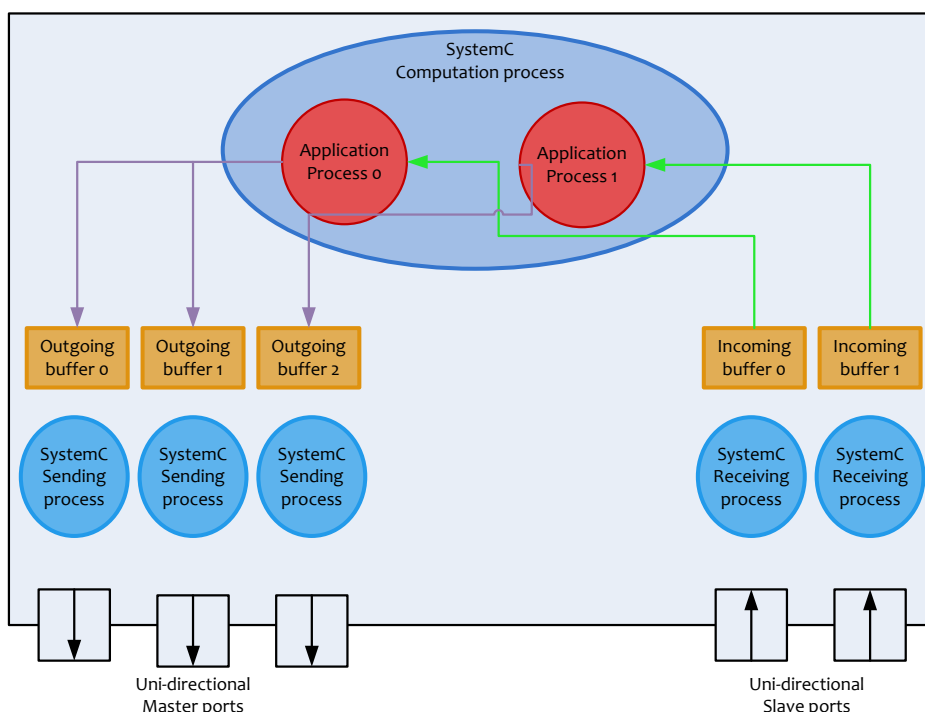


FIGURE 4.8: Model of a processing element

#### 4.2.2.3 Memory

Memory is modeled as an array of bytes with given read/write delays. Listing B.2 shows the class declaration of a simple memory model used for the demonstration applications.

### 4.3 Building the Virtual Platform

Now we have enough components in our library to build the virtual platform of the SoC. By properly instantiating and connecting all the needed SoC components such as processing elements, transaction interfaces and interconnection links the desired SoC platform can be built. An example of such a platform is shown in Figure 4.10.

In this platform the processing elements are connected by a point-to-point unidirectional links to a network box, which is introduced for the sake of an efficient hierarchical design. The transaction interfaces go between every processing element and the interconnection network which is a simple hierarchical bus network. What is left now for full system simulation is the modeling of the software processes which will be discussed in next section.

### 4.4 Modeling Software Applications

Software applications which are going to run on the SoC platform should be partitioned and mapped on processing elements before full system simulation is possible. Each

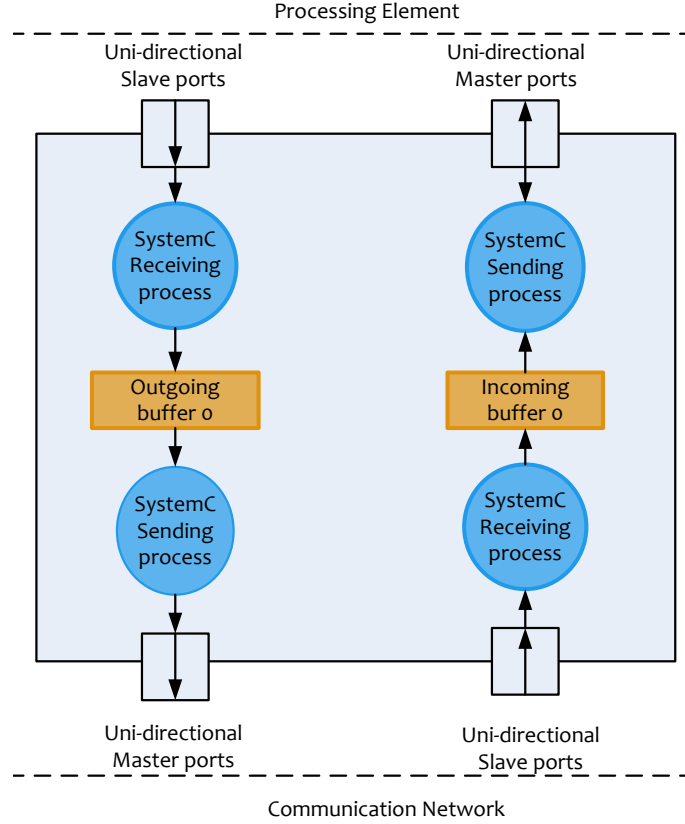


FIGURE 4.9: Model of a transaction interface

partition of a software application is termed as a *process*. Multiple processes might be mapped on a given processing element. Therefore, scheduling should be done to arbitrate the execution of processes. The processing element model discussed in section 4.2.2.1 has the computation SystemC process for this purpose. This is a SystemC *SC\_METHOD* process that implements the scheduling and firing of processes based on preset conditions. Processes are fired only if they have sufficient input data to process. Scheduling the firing of processes can be difficult if they have variable input data lengths. In this thesis work an approach is devised that enables easier scheduling of such processes by splitting a given process of variable input length and re-modeling it as a set of tasks where each of these tasks are of fixed input length. In this approach, each software process is a class instance with a set of states where each state represents a given task. The class definition comprises state variables, the main computation function and other supportive functions. The skeleton of the implementation of such a state machine is shown in Listing 4.3. In this example listing, the state of the process are stored in five state variables: `int state`, `cvec fineSyncBuffer`, `cvec ofdmFrameBuffer`, `uchar mode` and `std::complex<double> tempSample`. The main function called to fire the process is `synchronize`. In addition three support functions are included in it: `estimateFineOffsets`, `estimateCoarseFrequencyOffset` and `correctFrequencyOffset`.

During each firing, a process depending on its state consumes some input data from its input buffers (which are the arguments of the main function), manipulates its state variables, writes output data and gives control back to the computation SystemC process. This execution process is shown in Figure 4.11.

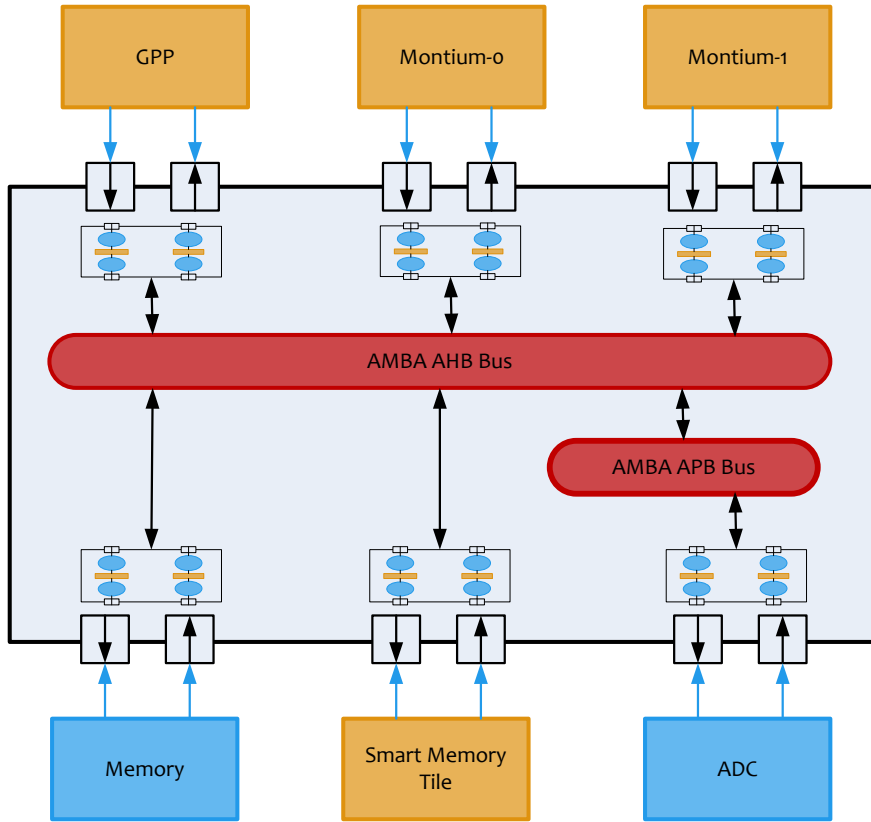


FIGURE 4.10: An instance of a SoC platform

## 4.5 System Level Design Flow

In the previous four sections, we have covered all the necessary ingredients we need for system level modeling of multi-core systems. Now we can see how these different ingredients come together and form a TLM-based system level modeling and simulation framework. Figure 4.12 shows the design process to follow to incorporate system level modeling in the SoC design flow for three main use cases: virtual platform for early software development, fast architectural analysis and functional verification.

The design flow begins with two separate processes: building the virtual platform as discussed in section 4.3 and modeling the software applications as set of processes with the necessary data dependency between them as discussed in section 4.4. Based on an externally developed mapping rule, the integration of the software applications and the virtual platform proceeds. There are two different possibilities for the modeling and simulation of the system at different levels of abstractions. The first alternative is to model the virtual platform so that it can support the different abstraction levels all together. In this case a configuration file is needed to change parameters so that the system can be simulated at different abstraction levels. The second option is to prepare separate virtual platforms for each abstraction level and select the desired platform during integration. The advantage of the first approach is that it enables multiple abstraction level simulations at reduced design effort. However, it introduces certain level of complication during implementation. On the other hand, the second alternative has the opposite effect in terms of design effort and implementation complexity. Automating

```

class Process0 {
private:
    //state variables
    int state;
    cvec fineSyncBuffer;
    cvec ofdmFrameBuffer;
    uchar mode;
    std::complex<double> tempSample;
public:
    Process0();
    //these are additional support functions
    void estimateFineOffsets(cvec &samples,
        uchar mode, uint &timingOffset, double &frequencyOffset){
        ...
    }
    int estimateCoarseFrequencyOffset(cvec &samples, uchar mode){
        ...
    }
    cvec correctFrequencyOffset(cvec &samples,
        uchar mode, double &frequencyOffset){
        ...
    }
    // this is the main computation function
    //the input arguments are the input and output buffers
    void synchronize(QueueObject<double> &fine_frequency_offset,
        QueueObject<uchar> &mode_received,
        QueueObject<cvec> &sampleBuffer,
        CQueueObject<std::complex<double>> &adc_samples,
        QueueObject<cvec> &to_ofdm)
    {
        switch(state){
            case 0:
                ...
                break;
            case 1:
                ...
                break;
            ...
            default:
                ...
        }
    }
};

```

LISTING 4.3: A skeletal implementation of processes as a state machine

the design flow can be an effective solution for these two contradictory concerns. The virtual platform building phase and the integration of the application with the platform can be automated and achieve significant design effort reduction.

## 4.6 Summary

In this chapter, a methodology for TLM-based system level modeling and simulation is presented. The core components of the methodology are the defined abstraction levels, the developed library and the integration strategies. The defined abstraction levels are Functional Model, Process Model, Transfer-based System Level Model, Phase-based System Level Model, Protocol-specific System Level Model and Implementation Model. Each of them has a set of distinct features and use cases. The library used for the experimentation of the design methodology is the OCCN's open-source package. Different modifications and additions were made to the library to make it fit for the intended

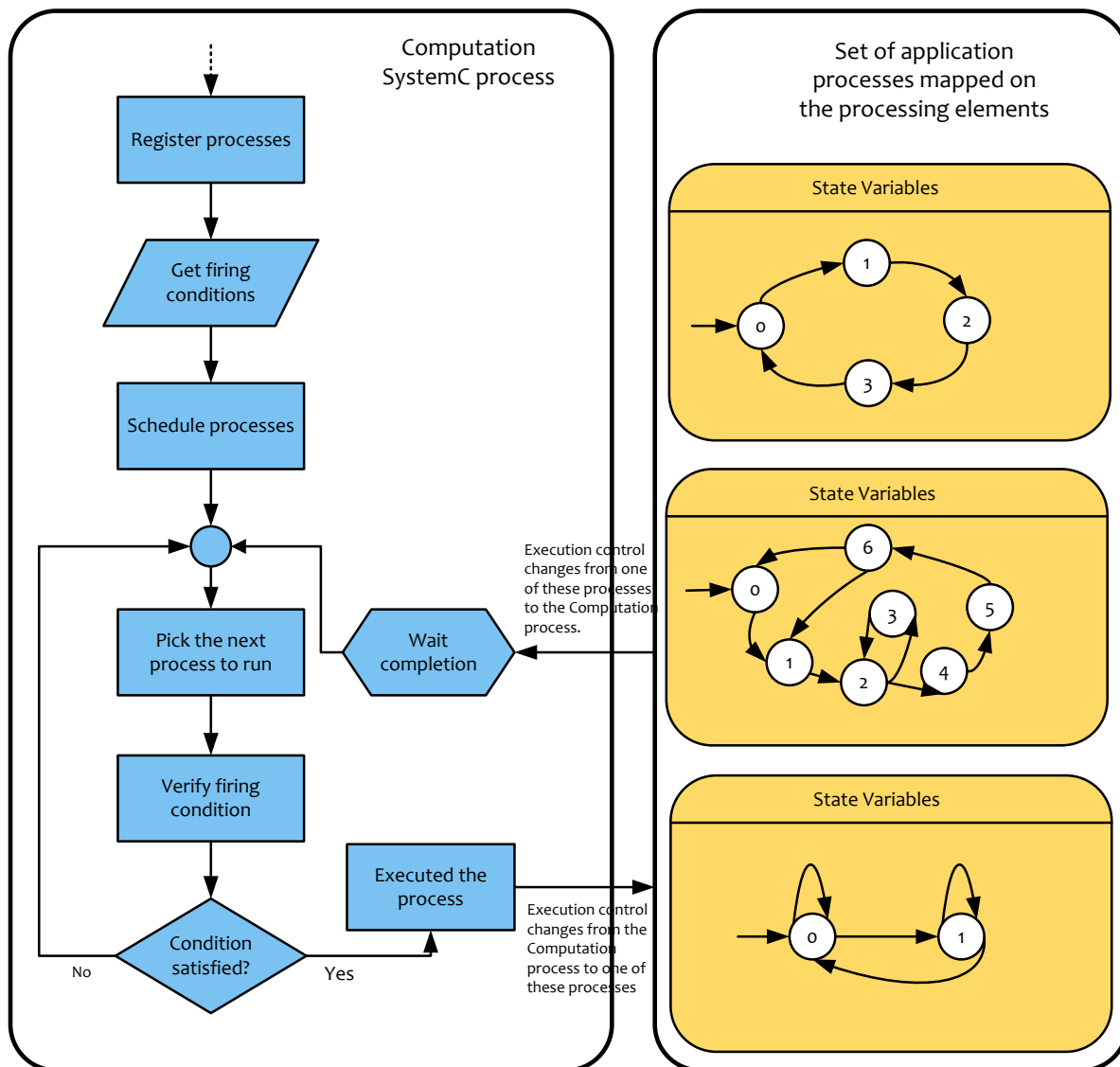


FIGURE 4.11: The scheduling and firing of processes

purpose. In addition to communication APIs and channels, models of processing elements, transaction interfaces and memory are added to the library to enable complete SoC modeling. The system level design flow using the discussed methodology starts by building a virtual platform using components from the developed library. The software applications are then partitioned into processes and the data dependency between the processes is figured out. These two entities are finally integrated based on a given mapping and simulated at the desired level of abstraction.

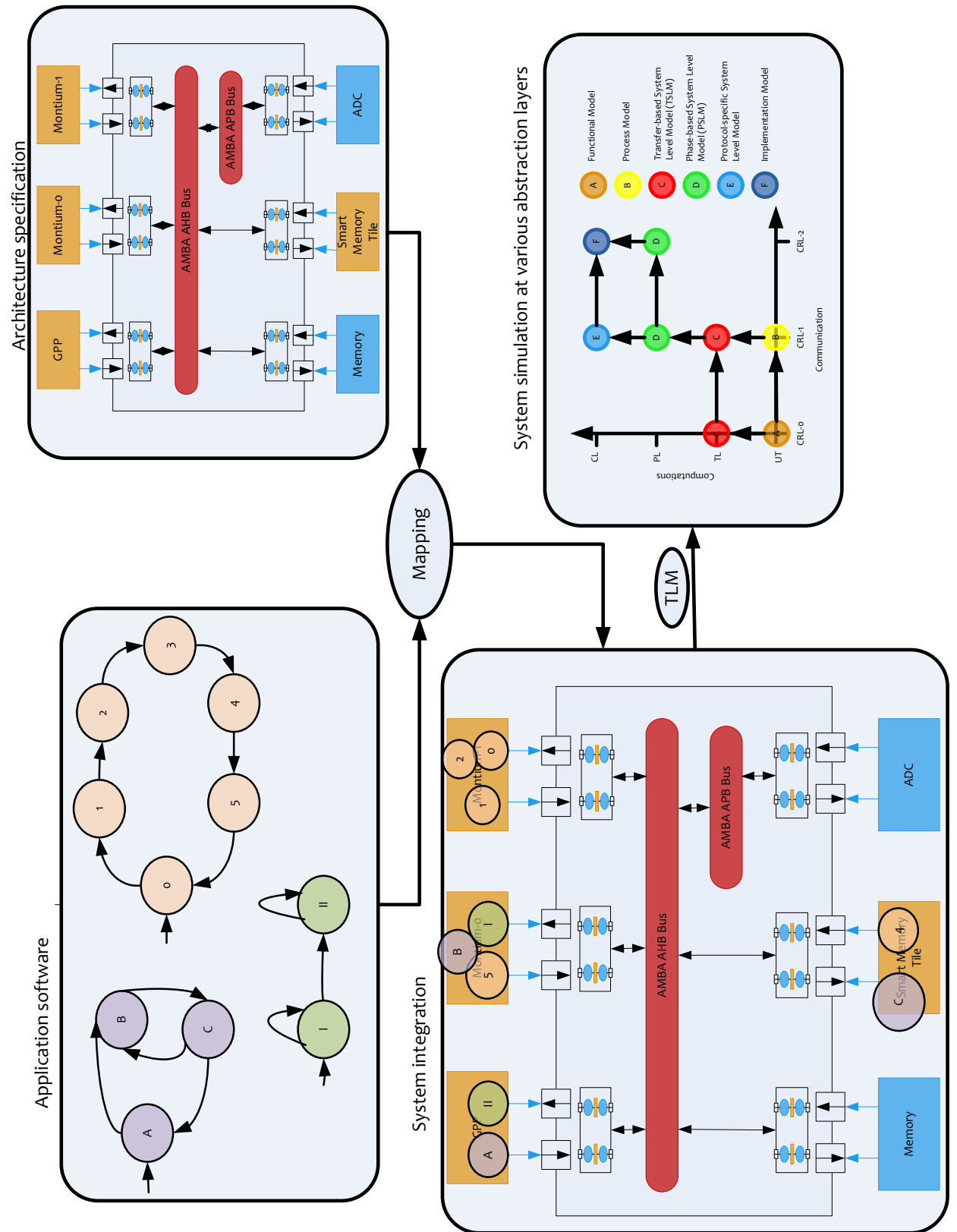


FIGURE 4.12: TLM-based System Level Design Flow



## Chapter 5

# Demonstrations- DAB Receiver and AMBA AHB Bus

In the previous chapter, a system level modeling approach is presented that uses TLM as a central communication approach. In this chapter, two demonstration applications are discussed to show the usage of the methodology. The first demonstration is a digital radio receiver streaming application (also called the DAB receiver) and the second one is an AMBA AHB bus system with random traffic generators. Section 5.1 and section 5.2 discuss the modelings of the DAB receiver and the AHB bus-based system, respectively. The chapter concludes finally with a summary of the main issues covered in the chapter.

### 5.1 DAB Receiver

Before discussing the implementation of the DAB receiver demonstration application, the following subsection introduces the DAB standard and the DAB transmission-reception system.

#### 5.1.1 Digital Audio Broadcasting(DAB) Standard

Digital Audio Broadcasting (DAB), is a digital radio technology for broadcasting radio stations, used in several countries, particularly in Europe. There are now over 1000 different DAB receivers commercially available. 30 countries have regular DAB services on air, and more than 12 million DAB receivers have been sold worldwide [27]. DAB allows for a much more efficient use of frequency spectrum than traditional analogue radio. Instead of just one service per frequency as is the case on FM (Frequency Modulation), DAB permits up to nine (or more) services on a single frequency. Proponents also claim the standard, which has gone through different revisions since its first draft in the 1980s, offers several benefits over existing analogue FM radio, such as increased resistance to noise, multipath fading, and co-channel interference.

An upgraded version of the system was released in February 2007, which is called DAB+. DAB+ is designed to provide the same functionality as the original DAB radio, though not backward compatible with it. DAB+ is more efficient than DAB mainly because of



the AAC+ (MP4) audio codec it uses. This allows equivalent or better subjective audio quality to be broadcast at lower bit rates than DAB.

DAB is also used as physical layer for DMB (Digital Multimedia Broadcasting), which is a video and multimedia broadcasting technology [28]. DMB offers a wide range of new innovative services, such as mobile TV, traffic and safety information, interactive programmes, data information and many other applications. Since DMB is based on the globally used digital audio broadcasting (DAB) core standard, DMB devices are always backwards compatible and can receive not only DMB multimedia services but also DAB audio services [28].

### 5.1.2 DAB Transmission/Reception System

Figure 5.2 shows the generation of the DAB signal from the transmission side. The transmission system has three main units: multiplexer, channel encoder and OFDM signal generator. The multiplexer merges multiple service signals, which are individually coded and error protected at source level, and multiplex control and service information which travel in the Fast Information Channel (FIC). The output of the multiplexer is then block partitioned, QPSK symbol mapped and frequency interleaved in the channel encoding process. Finally, Orthogonal Frequency Division Multiplexing (OFDM) is applied to shape the DAB signal, which consists of a large number of carriers. The signal is then transposed to the appropriate radio frequency band, amplified and transmitted. Figure 5.1 shows the transmission mode independent description of DAB transmission frame. In time domain the DAB signal is made up of such transmission frames which are of 24 ms duration. The frame is composed of 154 symbols. Each frame begins with the *null symbol* for coarse synchronisation of receivers and the *phase reference symbol* for fine synchronisation. These two symbols form what is called the *Synchronization Channel*. The Synchronization Channel is followed by *Fast Information Channel (FIC)* which has 8 symbols to mainly carry the Multiplex Configuration Information (MCI). Following is the *Main Service Channel (MSC)* that comprises 144 symbols. The MSC is further divided into sub-channels and each sub-channel carries audio data of a particular service, coded using the MPEG Layer I standard.

Figure 5.3 demonstrates a conceptual DAB receiver. It performs more or less the reverse of the transmission process to extract the multiple audio and data services encoded in the received DAB frames.

Details of DAB can be obtained from [29], [30] and [31].

### 5.1.3 DAB Receiver Architecture

The hardware architecture of the DAB receiver used in the experimentation of this thesis is taken from the DAB/DAB+/DMB receiver project at *Recore Systems* [32]. This architecture is shown in Figure 5.4. The main components of the SoC are a general purpose processor (e.g. ARM processor), two Montium processors, SRAM memory and different GPIO (General Purpose Input/Output) units.

The architecture modeling focuses on the ARM processor, the two Montium processors, the smart memory tile, the SRAM memory and the ADC (analog-to-digital converter).

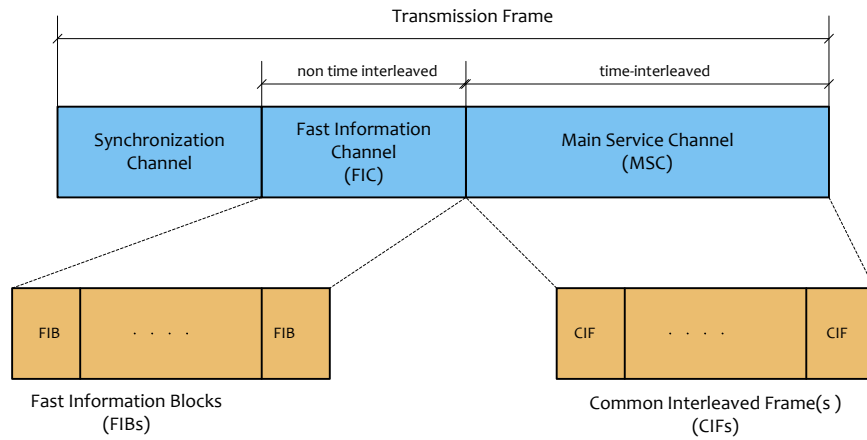


FIGURE 5.1: Transmission mode independent description of the FIC and MSC (*source: DAB standard [30]*)

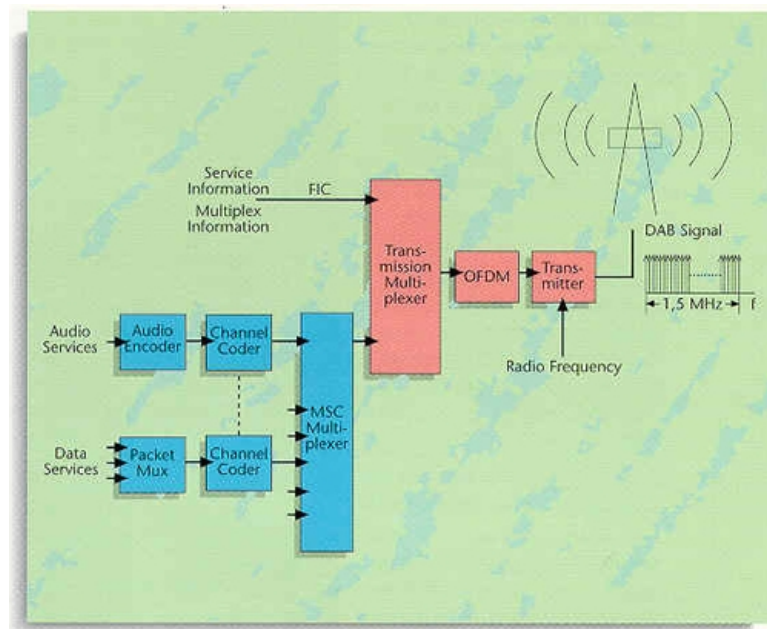


FIGURE 5.2: Generation of DAB Signal (*source: WorldDMB forum [29]*)

The other I/O interfaces and components are not included. Different interconnection networks are modeled other than the hierarchical bus interconnect shown in the figure.

#### 5.1.4 Software Architecture

The functional blocks of the DAB receiver application software is shown in Figure 5.5. The application reads a received DAB stream file and produces an audio output. The first three blocks: *Mode detection*, *Synchronization* and *OFDM* correspond to the *OFDM Demodulator* block shown in Figure 5.3. Whereas the remaining three blocks: *De-multiplexer*, *Channel Decoding* and *Source Decoding* correspond to the *Packet Demux*, *Channel Decoder* and *Audio Decoder* blocks respectively.

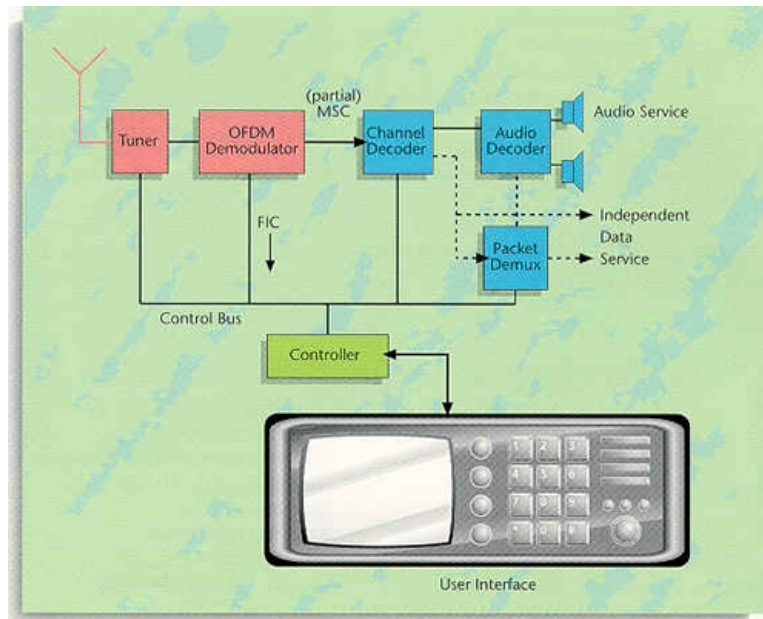


FIGURE 5.3: Generation of DAB Signal (*source: WorldDMB forum [29]*)

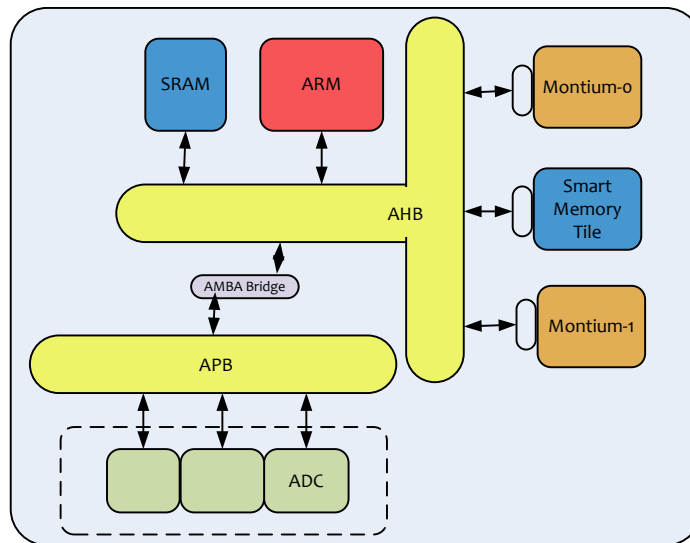


FIGURE 5.4: DAB/DAB+/DMB receiver architecture: (*source: Recore Systems DAB receiver project [32]*)

In the implementation, the DAB receiver application is partitioned into eight separate processes. The data dependency graph of these eight processes is shown in 5.6.

Here is a brief description of each of these processes.

- **ADC** : This process reads data from the input stream file and sends them either to the *Mode detection* or *Synchronization* process depending on its state.

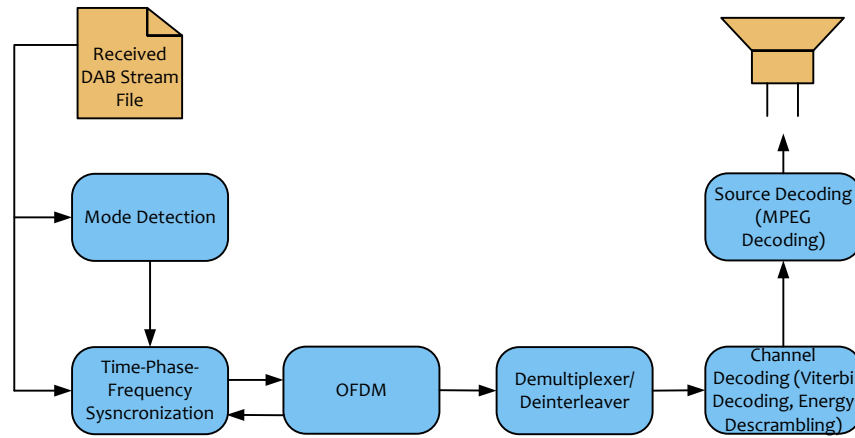


FIGURE 5.5: DAB receiver application functional view

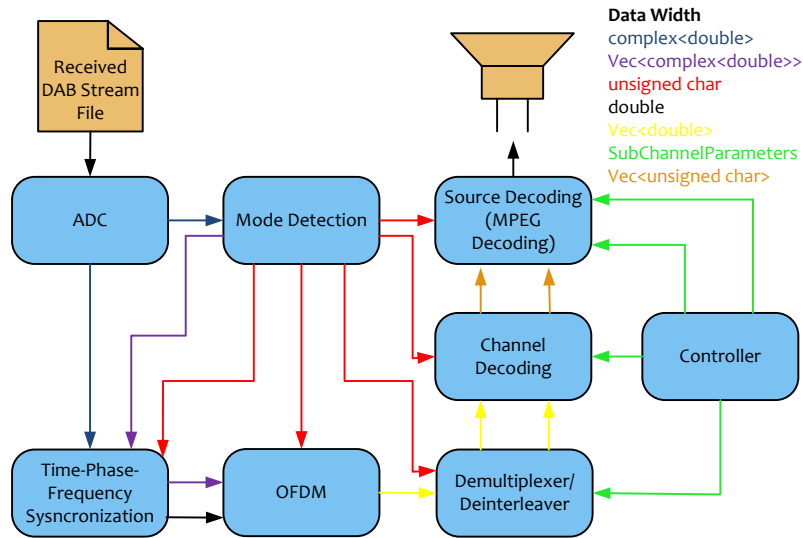


FIGURE 5.6: data dependency graph of the software functional blocks

- **Mode detection** DAB transmission can be carried out in four different modes. Each mode of transmission has its own frame structure. The *Mode detection* process recognizes the mode of transmission of the received DAB stream by comparing the length of the null symbol that separates consecutive frames.
- **Controller:** This process controls the *Channel decoding* and *Source decoding* processes by supplying the sub-channel parameters based on the user's program selection.
- **Synchronization:** DAB transmission frames have a synchronization channel (see Figure 5.1) which contains the Null Symbol and Phase Reference Symbol (PRS). The Null Symbol contains nothing and has a much lower energy. Its length is used by the *Mode detection* process to identify the mode of transmission of the DAB signal. On the other hand the PRS is used for phase, time and frequency

synchronization of the receiver. Frequency synchronization is needed to balance out local oscillator offsets and time synchronization is needed to avoid incorrect position of FFT window.

- **OFDM:** The *OFDM* process carries out frequency offset correction based on the offset information obtained from the *Synchronization* process, performs the QPSK and OFDM demodulations.
- **Demultiplexing/Deinterleaving:** This process is responsible for the time deinterleaving and demultiplexing of subchannels.
- **Channel decoding:** carries out the Viterbi decoding process.
- **Source decoding:** performs the MPEG/AAC audio decoding process and produces the output audio file.

### 5.1.5 System Level Modeling of the DAB Receiver SoC

This section discusses the implementation of the DAB receiver system at three different abstraction layers; FM, PM and TSLM. The discussions include the features of the models and the achieve simulation speeds.

#### 5.1.5.1 Functional Model

The Functional model (FM) (section 4.1.4.1) is the concurrent simulation of multiple processes for simple functionality check. Functional correctness of various partitioning options can be tried out. It is also the entry point to the lower abstraction layers. Below here are the main features of this model.

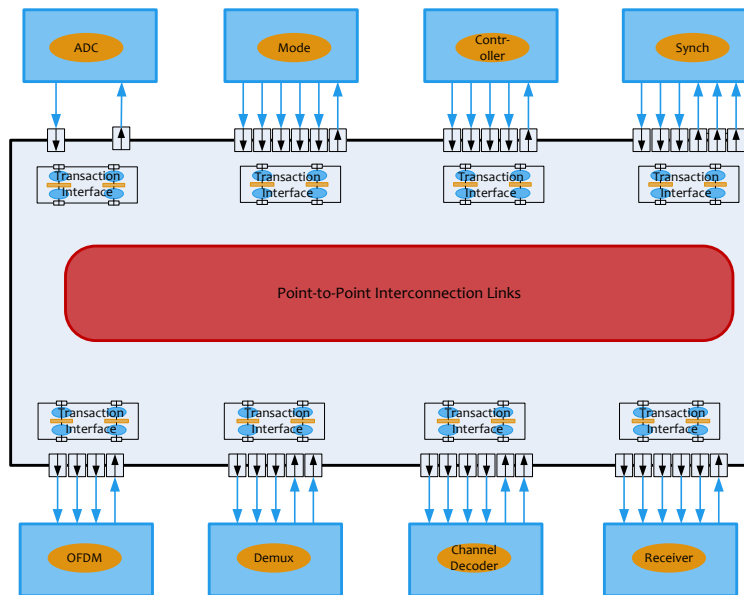


FIGURE 5.7: Functional Model

- Eight processes, each mapped on a separate SystemC module (processing element).

- Modules are connected by point-to-point links of different data widths as shown on the data dependency graph in Figure 5.6.
- Processes have infinite input/output buffers.
- Processes are modeled as a set of tasks (states) each with specific firing conditions.
- No execution or transmission delays are introduced in the simulation.

The simulation time of the DAB receiver at the FM level is shown in Table 5.1 for an input DAB stream of size 102MB.

TABLE 5.1: Simulation time at FM level

Simulated time	491,417,950 ns
Real time	201 sec
Simulation speed	2,443,042 ns/sec

#### 5.1.5.2 Process model

The Process model(PM) (Section 4.1.4.2) evolves from the Functional model by the introduction of mapping, scheduling and buffer size decisions. The designer can probe latency, throughput and size statistics at different points in the model to analyse various design decisions. Main features include:

- Eight processes mapped on five processing elements and processing elements with multiple processes are scheduled in a static round-robin manner.
- Processes mapped on the same processing element communicate through a shared memory.
- Processes mapped on different processing elements communicate through point-to-point links like FM.
- Processes are provided with fixed input/output buffer sizes.
- Estimated process execution delays are introduced in the processing elements.

The simulation time for the system at the PM model is shown in Table 5.2 for an input file size of 102MB. The system has a clock period of 10ns. Hence the simulation speed is in the order  $10^6$  cycles/sec.

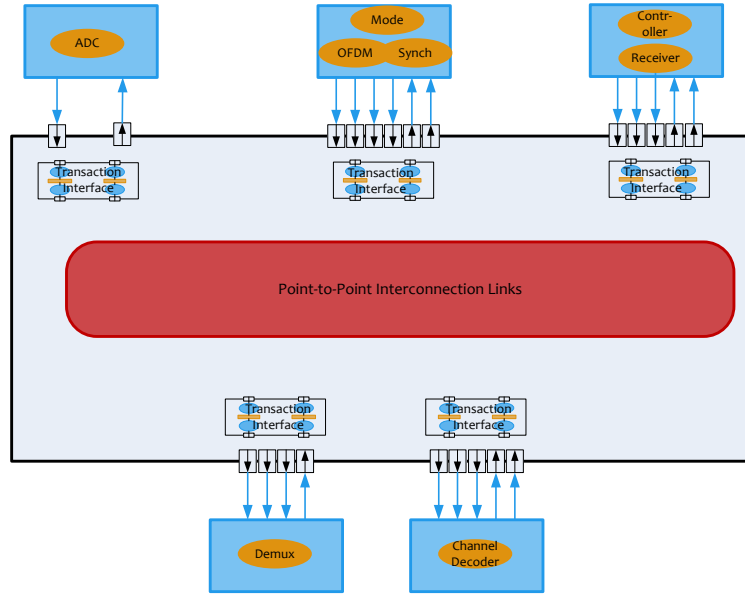


FIGURE 5.8: Process Model

TABLE 5.2: Simulation time at PM level

Simulated time	188,686,320 ns
Real time	106 sec
Simulation speed	1,771,037 ns/sec

### 5.1.5.3 Transfer-based System Level Model

The Transfer-based system level model (Section 4.1.4.3) gives a more realistic virtual platform because in addition to the mapping, scheduling and memory size decisions taken in the Process model, decisions are taken on the topology of the interconnection network and implementation of memory.

This is an appropriate abstraction level to model a virtual platform for software development since it incorporates all basic architectural details and still maintains fast simulation speed. Main features of the above models at TSLM level include:

- It has the same features as PM in terms of mapping, scheduling, memory size and process execution delay.
- Decision on the topology of the interconnection network is taken as shown in the figures. A flat single layer bus and 3x2 mesh NoC are used in the experimentation.
- Transactions over the interconnection network are not visible except at the start and end of the transaction.

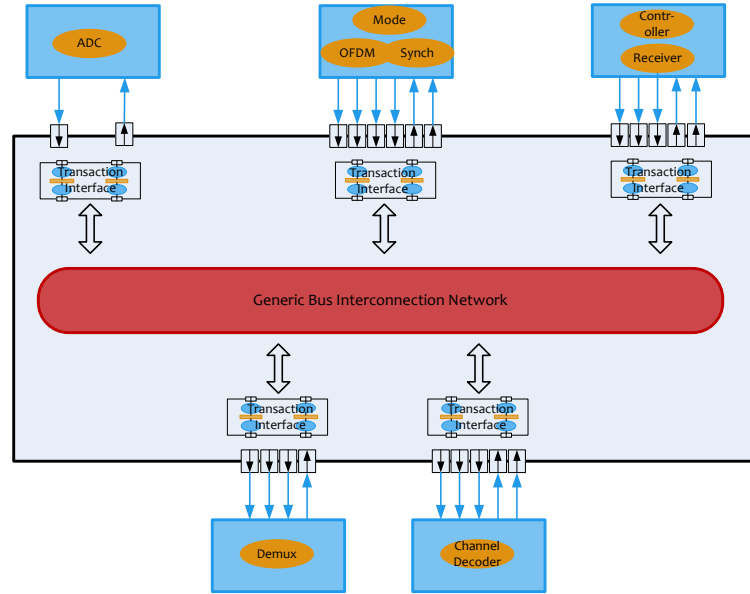


FIGURE 5.9: Transfer-based system level model with bus interconnect

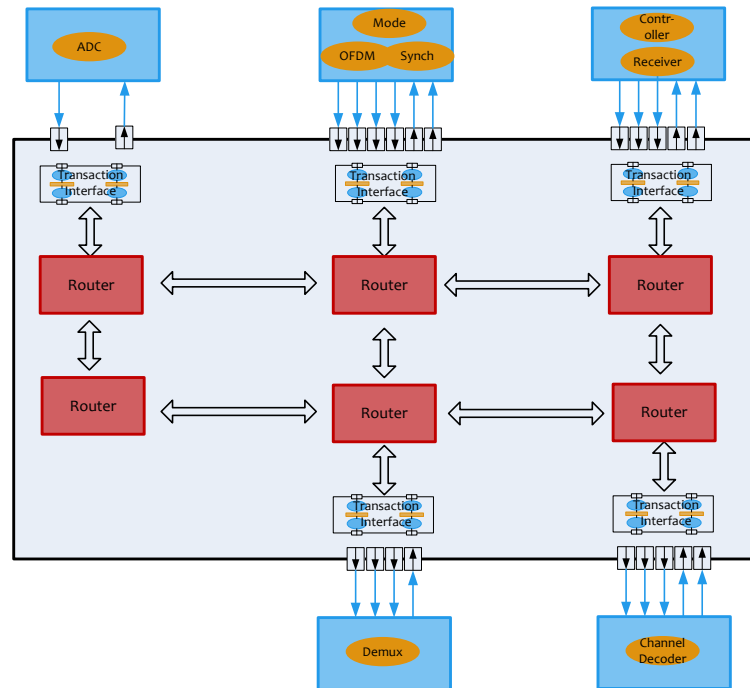


FIGURE 5.10: Transfer-based system level model with mesh interconnect

The simulation time for the DAB receiver system at the TSLM level is given in Table 5.3 for bus and mesh interconnection networks for an input DAB stream size of 102MB.



TABLE 5.3: Simulation time at TSLM level

	<b>Bus</b>	<b>Mesh</b>
Simulated time	491,417,950 ns	880,122,250 ns
Real time	211 sec	686 sec
Simulation speed	2,327,340 ns/sec	1,282,977 ns/sec

## 5.2 High-level AMBA AHB 2.0 Bus

The second demonstration application is the implementation of a high level AMBA AHB bus at two refinement levels: Transfer level (TL) and Phase Level (PL). This section presents the implementation details of the high level AHB bus models as well as the experimental setup used to analyse their use.

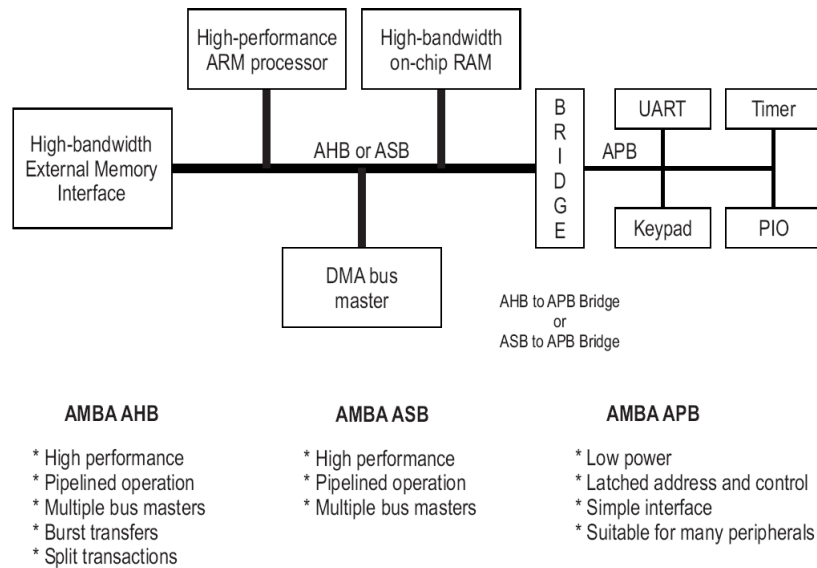
### 5.2.1 AMBA Bus Protocol

The AMBA bus protocol [33] is one of the most popular bus communication networks used in today's embedded system designs. Since its first introduction in 1995, the AMBA Specification has been refined and extended with additional protocol support to provide the capabilities required for SoC design. The two key versions are the AMBA 2.0 and AMBA 3.0 standards.

The goal of AMBA version 2.0 is to provide a flexible high performance bus architecture specification, that is technology independent, takes up minimal silicon area, and encourages IP reuse across designs. This version defines three distinct bus standards: AHB, ASB and APB. Advanced high performance bus (AHB) is a high performance bus meant to connect high bandwidth, high clock frequency components such as processors, DMA controllers, off-chip memory interfaces, and high bandwidth on-chip memory blocks. Advanced system bus (ASB) is a scaled-down alternative to the AHB bus which targets to connect high clock frequency components that do not need the advanced protocol features of AHB. The Advanced peripheral bus (APB) is a low complexity bus optimized for low power operation which is intended for high latency, low bandwidth peripheral components such as timers, UARTs, user interface (e.g. keyboard) controllers, etc. Figure 5.11 shows what a typical AMBA-based system looks like.

The AMBA 3.0 bus architecture specification introduces the Advanced eXensible Interface (AXI) bus that extends the AHB bus with advanced features to support high performance MPSoC designs. AMBA AXI is backward compatible with AHB and APB, and has key features such as separate address/control and data phases, support for unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost Direct Memory Access (DMA), ability to issue multiple outstanding addresses, out-of-order transaction completion and easy addition of register stages to provide timing closure.

Other AMBA specifications intended for high bandwidth operations include AHB-lite which supports a single-bus master and multi-layer AHB that allows for parallel access paths between multiple masters and slaves in a system.

FIGURE 5.11: Typical AMBA-based System (*source: AMBA 2.0 Specification*)

We selected the AMBA AHB 2.0 standard to demonstrate the use of the modeling methodology presented in the previous chapter. The next three sections respectively discuss the AHB 2.0 standard, the implementation of the high level bus models and the experimentation set-up.

### 5.2.2 AMBA AHB 2.0 Standard

AMBA AHB 2.0 has the features required for high-clock frequency and high performance systems. The features include burst transfers, split transactions, single-cycle bus master handover and wider data bus configurations.

A typical AMBA AHB system has AHB masters, AHB slaves and the bus arbiter. Each bus signal is controlled by either of these components. Table 5.4 and 5.5 show the complete list of AHB bus signals.

TABLE 5.4: AMBA AHB Signals

	Name	Source	Remark
<b>HCLK</b>	Bus clock	clock source	
<b>HRESETn</b>	Reset	Reset Controller	active LOW and is used to reset the system and the bus
<b>HADDR[31:0]</b>	Address Bus	Master	the 32-bit system address bus
<b>HTRANS[1:0]</b>	Transfer Type	Master	can be BUSY, IDLE, SEQ or NONSEQ
<b>HWRITE</b>	Transfer Direction	Master	LOW for read, HIGH for write
<b>HSIZE[2:0]</b>	Transfer Size	Master	beat size(8-1024 bits)
<b>HBURST[2:0]</b>	Burst Type	Master	eight types including four, eight, sixteen incremental or wrapping bursts
<b>HPROT[3:0]</b>	Protection Control	Master	tells if transfer is opcode or data fetch, cacheable, bufferable, user mode or privileged mode
<b>HWDATA[31:0]</b>	Write Data Bus	Master	extendable upto 1024 bits
<b>HSELx</b>	Slave Select	Decoder	
<b>HRDATA[31:0]</b>	Read Data Bus	Slave	extendable upto 1024 bits
<b>HREADY</b>	Transfer Done	Slave	When HIGH, HREADY signal indicates that a transfer has finished on the bus
<b>HRESP[1:0]</b>	Transfer sponse	Re-Slave	OKAY, ERROR, RETRY or SPLIT

An AHB transfer consists of two phases: an address phase which lasts only one cycle and the data phase which may require one or more cycles. Since the address and the data phase of a transaction occur at separate clock cycles, pipelining is usually done by driving the address of the next transaction in parallel with the data phase of the current transaction. The number of cycles for the data phase depends on the burst type and the slave's waiting cycles. The HTRANS signal of the first beat of every transaction has the NON-SEQUENCE value and the remaining sequence of beats have the SEQUENCE value. For beats whose HTRANS has the value of SEQUENCE, the slave computes the transaction address based on the address in the first beat and whether the burst is incremental or wrapping. The burst transaction can be terminated early if the slave received a beat with HTRANS value of NON-SEQUENCE.

TABLE 5.5: AHB Arbitration Signals

	Name	Source	Remark
<b>HBUSREQx</b>	Bus Request	Master	indicates that the bus master requires the bus
<b>HLOCKx</b>	Locked Transfers	Master	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW.
<b>HGRANTx</b>	Bus Grant	Arbiter	indicates that bus master x is currently the highest priority masters
<b>HMASTER[3:0]</b>	Master Number	Arbiter	indicate which bus master is currently performing a transfer and is used by the slaves which support SPLIT transfers to determine which master is attempting an access.
<b>HMASTLOCK</b>	Locked Sequence	Arbiter	Indicates that the current master is performing a locked sequence of transfers
<b>HSPLITx[15:0]</b>	Split Completion Request	Slave (SPLIT capable)	used by a slave to indicate to the arbiter which bus masters should be allowed to re-attempt a split transaction

Readers who are interested for the complete specification of the AMBA bus can freely download the official documentations from ARM's website [33].

### 5.2.3 Implementation

This section discusses how the high level model of the AMBA AHB 2.0 bus is implemented using the methods discussed in the previous chapter. The implementation has two main components: the protocol data units and the bus process.

1. **Protocol data units:** As discussed in Section 4.2.1, all transactions between masters and slaves are carried out using the protocol data units. The implementation of the protocol data units for the AHB bus is based on OCCN's *Pdu* class template. By defining the data structures for the header field (also called Protocol Control Information -PCI) and data field (also called Service Data Unit -SDU), the *Pdu* class template can be specialized as protocol data unit of a particular protocol. For the AMBA AHB bus, two PCI fields are defined: *AHB\_MasterCtrl* and *AHB\_SlaveCtrl*, for the master and slave controlled bus signals respectively. Listing 5.1 shows the definition of these header fields.

The SDU field which holds the actual data can be of any size by specifying the *BH* and *size* template parameters of the *Pdu* class template. Instantiation of protocol data units for AHB transactions looks like as shown in Listings 5.2.

2. **Bus Process:** The AMBA AHB bus is implemented by adding the necessary modifications to the generic bus in the OCCN's library. During the elaboration

```

enum Transfer_type{IDLE, BUSY, NONSEQ, SEQ};
enum TransactionDirection{READ, WRITE};
enum SlaveResponse{OKAY, ERROR, RETRY, SPLIT};

enum BurstType{SINGLE=0, INCR=1, WRAP4=4, INCR4=5,
               WRAP8=8, INCR8=9, WRAP16=16, INCR16=17};

enum BeatSize{ERROR_BS=0, BYTE=1, HALFWORD=2,
              WORD=4, _2WORDLINE=8, _4WORDLINE=16,
              _8WORDLINE=32, _16WORDLINE=64, _32WORDLINE=128};

class AMBA_AHB_MasterCtrl
{
public:
    N_uint32 HADDR;
    TransferType HTRANS;
    TransactionDirection HWRITE;
    BeatSize HSIZE;
    BurstType HBURST;
    N_uint8 HPROT;
    bool HBUSREQ;
    bool HLOCK;

    //non-real signals
    N_uint8 priority; //for arbitration
};

class AMBA_AHB_SlaveCtrl
{
public:
    bool HREADY;
    SlaveResponse HRESP;
    N_uint16 HSPLIT;
};

```

LISTING 5.1: Master and Slave PCIs for AMBA AHB

```

//Pdu instances for masters
typedef unsigned int N_uint32;
Pdu<AMBA_AHB_MasterCtrl, N_uint32, 16> pdu_to_send;
Pdu<AMBA_AHB_SlaveCtrl, N_uint32, 16> pdu_to_receive;

//Pdu instances for slaves
Pdu<AMBA_AHB_SlaveCtrl, N_uint32, 16> pdu_to_send;
Pdu<AMBA_AHB_MasterCtrl, N_uint32, 16> pdu_to_receive;

```

LISTING 5.2: Master and Slave pdu instances

phase of the SystemC simulation, all the slaves and masters are connected and address ranges are registered. During the execution phase, the bus process runs until the end of simulation. The implementation of the process depends on the communication refinement level of the bus. For TL and PL models, the bus process is a SystemC SC\_THREAD process and for SL model it is a SystemC SC\_METHOD process. The bus process for the TL model is the same as the one in the OCCN's generic bus. The flow chart in Figure 5.12 shows the SystemC SC\_THREAD process for the PL model. The complete source code for of the AHB PL bus process is available in Listing B.1 in Appendix B.

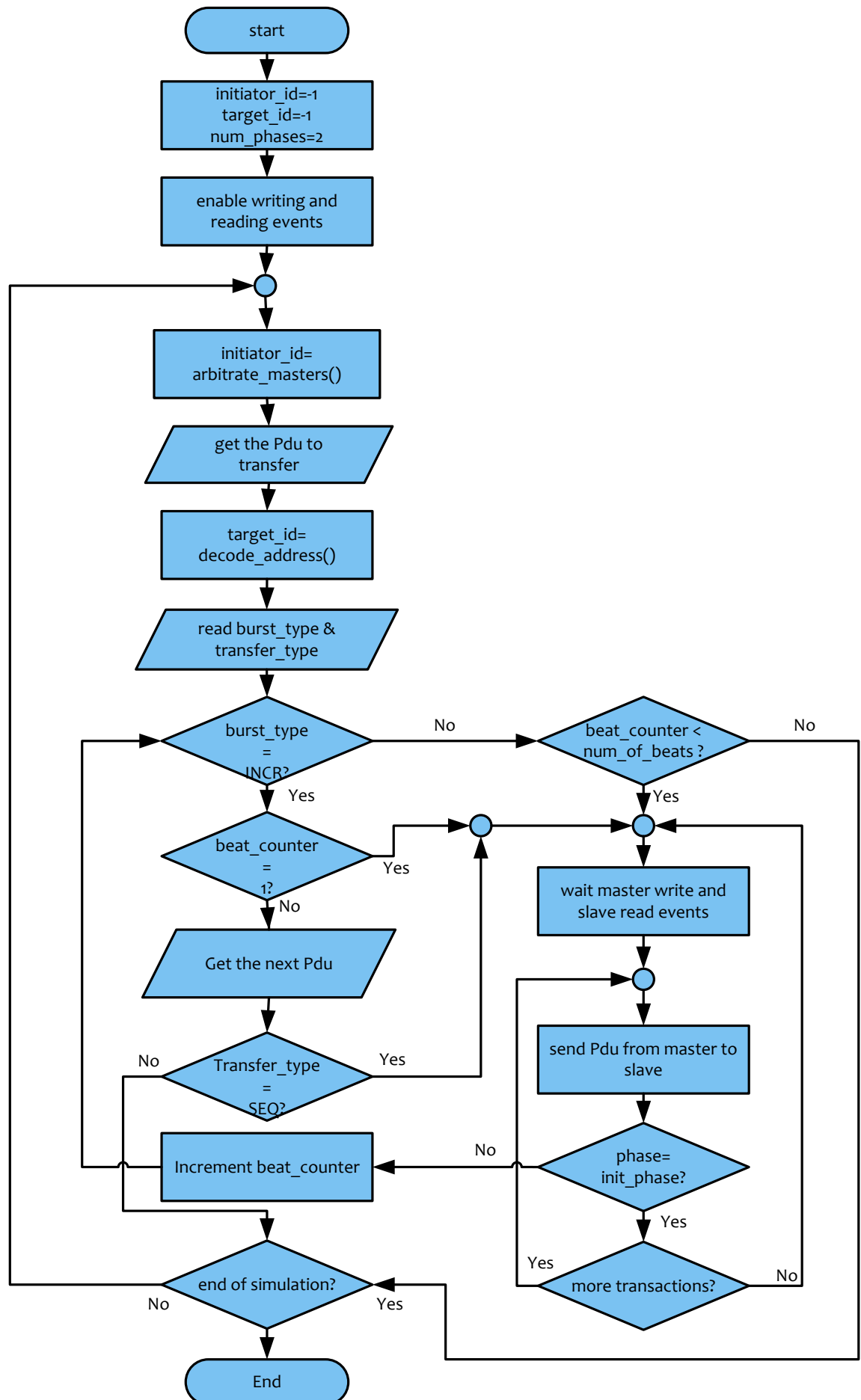


FIGURE 5.12: Flow chart of the bus process for the PL AHB model

### 5.2.4 Experimentation

To investigate the usage of the TL and PL AHB bus models, an experimental set up is built that comprises traffic generators as initiators, memories as targets and a configuration file to specify system parameters.

1. **Configuration file:** The configuration file is used to specify system parameters such as the abstraction layer, number of initiators, number of targets and parameters for each initiator and target. Listings 5.3 shows an example of a configuration file that specifies a system of two initiators and one target that uses the TL AHB bus.
2. **Traffic Generator:** The traffic generator randomly initiates transactions of different types based on parameters specified in the configuration file. Handling of transactions varies between TL and PL models. For TL model, every beat of the burst is packed into a single protocol data unit and sent in one transfer. In addition, there is no separate initialization and data phases. This makes simulation in the TL model faster. For the PL model, however, separate data unit is prepared for each beat in the burst and the transfer of each beat is completed in two separate phases. However, separate timing information can be extracted for each beat of the burst and for each phase of the beat transfer. For the AHB 2.0 bus with possible data width range between 32 and 256 bits, a total of 96 different transaction types are identified: two transaction directions (WRITE or READ), eight burst types (SINGLE, INCR, INCR4, WRAP4, INCR8, WRAP8, INCR16 and WRAP16) and six beat sizes (BYTE, HALF\_WORD, WORD, 2WORD\_LINE, 4WORD\_LINE and 8WORD\_LINE). Figure 5.13 and 5.14 show the message sequences for write and read transactions in the TL model. As shown in the figures, the timing information that can be obtained from the TL model is limited to the duration of the whole transaction.

```

# Abstraction layer

abs_layer = PL          # abstraction layer (options: TL, PL, SL)

# System Parameters

num_initiators= 2      # number of initiators (traffic generators)
num_targets    = 1      # number of targets (memories)
comm_network   = bus    # communication network type (eg: bus, mesh,)

# Communication network parameters

bus_protocol   = AMBA AHB 2.0 # the specific bus protocol to use
bus_width      = 32          # the bus width in bits
bus_lite       = false       # the bus type- normal vs lite versions
num_layers     = 1          # single layer vs multi-layer options
arbitration    = SP         # arbitration (options: SP, RR, TDMA)
split_support  = false       # the bus support split transactions

# Target parameters
target0_parameters= 4000     # address space size: 16 kb
                    0000     # target0 starting address
                    3fff     # target0 ending address
                    3        # byte read latency- clock cycles
                    2        # byte write latency- clock cycles

# Initiator parameters
initiator0_traffic_type =
    60 40          # write percentage & read percentage
    20 20 60 0 0 0 # byte, half_word,...,8wordline percentages
    30 0 40 20 10  # single, incr,..., burst16 percentages
    10000 115fff   # number of transactions, maximum address

initiator1_traffic_type =
    15 85          # write percentage & read percentages
    25 35 40 0 0 0 # byte, half_word,...,8wordline percentages
    20 0 10 30 40  # single, incr,..., burst16 percentages
    48000 115fff   # number of transactions, maximum address

#Screen display
print_config = true    # to show the configuration parameters
print_screen = false   # to show simulation data on screen

```

LISTING 5.3: Example Configuration File

Figure 5.15 shows the message sequence for both write and read transactions in the PL model. As shown in the figure, timing information can be obtained for each phase of every beat in the burst. The duration of each phase emphasizes a certain aspect of the communication protocol. For instance, the initialization phase spans the duration from the point where the master forwards a request to the bus until the slave replies its readiness for the transaction. This duration is mainly occupied by the arbitration delay of the master and the slave overhead (the waiting time before a slave is ready for the transaction). The data phase is also determined by the propagation delay, slave's transaction processing delay as well as the slave overhead.



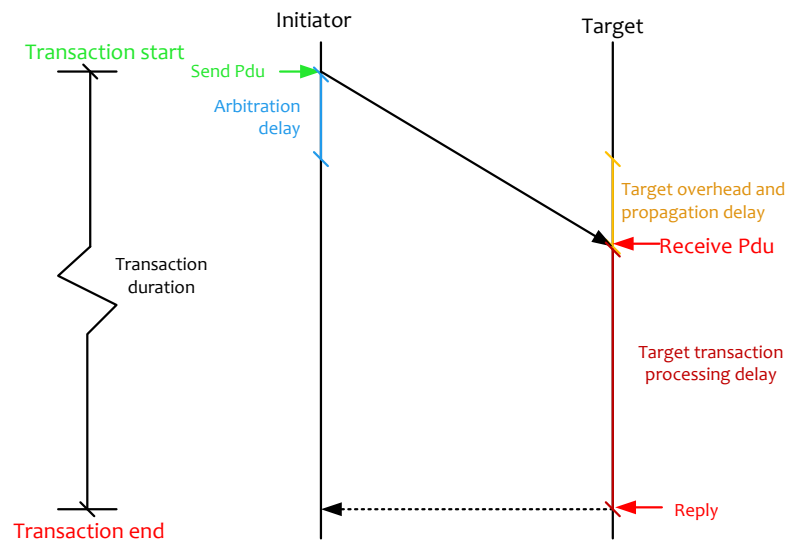


FIGURE 5.13: Message sequence of TL model write transaction

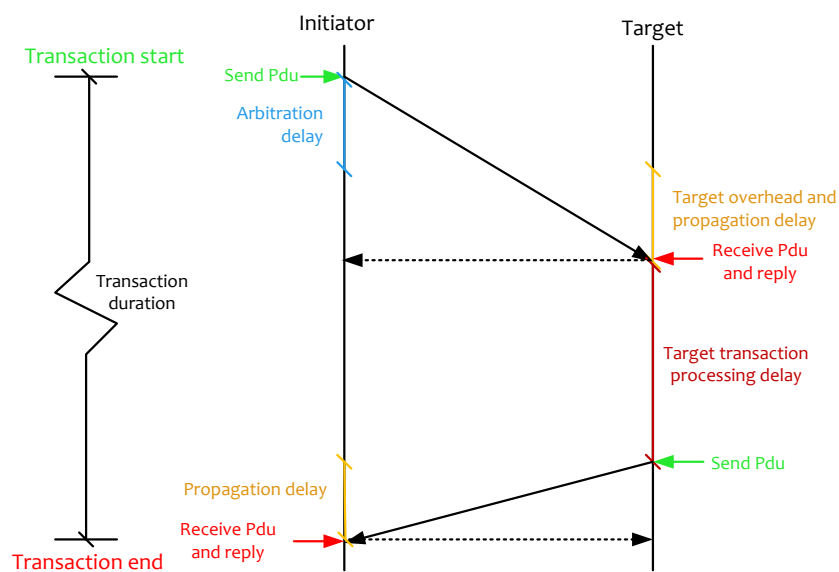


FIGURE 5.14: Message sequence of TL model read transaction



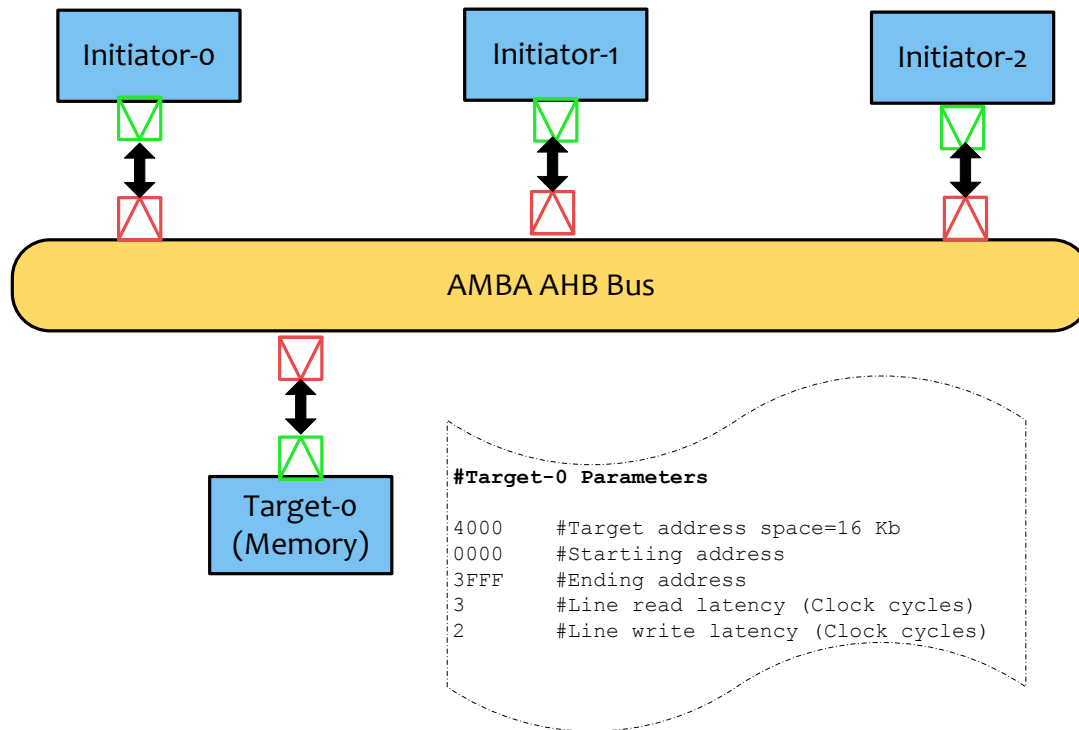


FIGURE 5.16: Example AMBA AHB System

For this system, issues that may be of interest to the designer include the transaction latency of Initiator-0 (ADC) and the read-write throughput of the memory. In order to avoid sample misses from the ADC, the maximum transaction latency should be less than the transaction gap which is the sampling period of the ADC (1000 nsec). Figure 5.17 shows the obtained latency for the first 100 transactions for the AMBA bus in the TL level which uses a random arbitration scheme (called RD). As shown in the figure, most transactions have a latency beyond the maximum tolerable latency which is the ADC's sampling period. Running the same system with a bus in the 2-phase PL level discovers that most of this transaction latency is spent during arbitration (Figure 5.18). By replacing the random arbitration (RD) with a static priority arbitration (SP) that gives the highest priority to Initiator-0, the transaction latency for the majority of the transactions can be cut down below the maximum tolerable latency as shown in Figure 5.19.

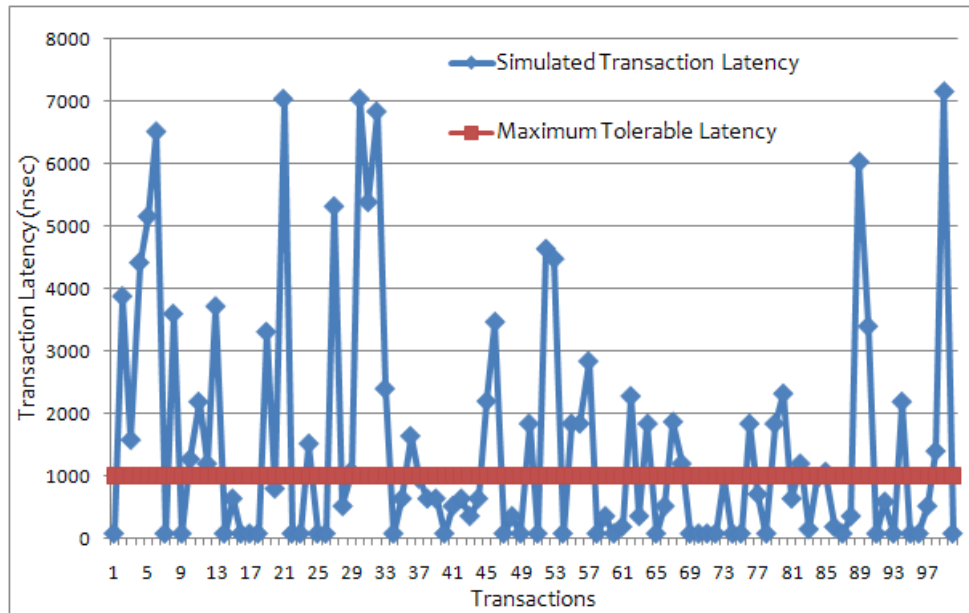


FIGURE 5.17: Initiator-0's transaction latency of the system shown in Figure 5.16 with TL level AHB bus and arbitration RD

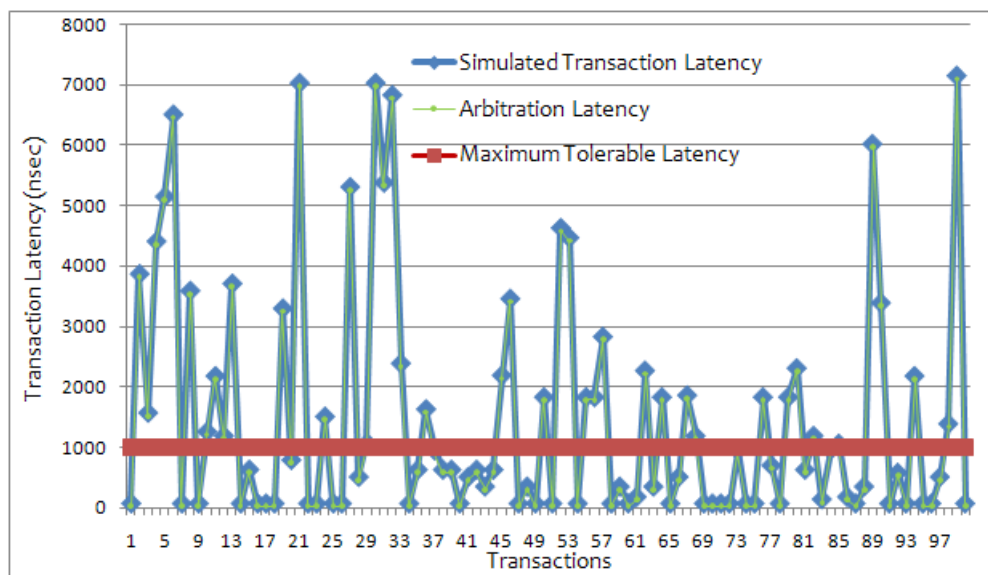


FIGURE 5.18: Initiator-0's transaction latency of the system shown in Figure 5.16 with PL level AHB bus and arbitration RD

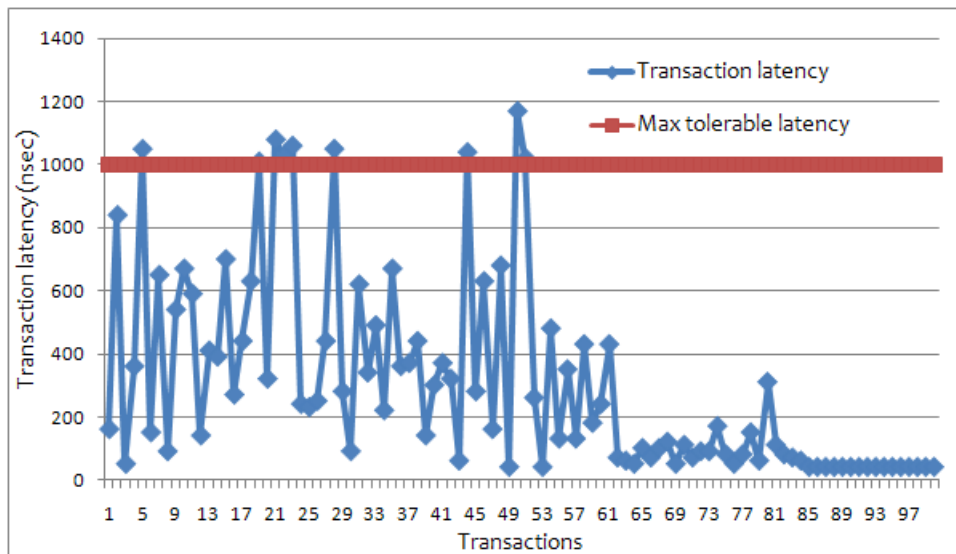


FIGURE 5.19: Initiator-0's transaction latency of the system shown in Figure 5.16 with TL level AHB bus and arbitration SP

Statistics regarding the write-read throughput of the memory can be obtained using OCCN's statistics package that targets the Grace 2D graphing tool. The average read and write throughputs of the memory are shown in Figures 5.20 and 5.21.

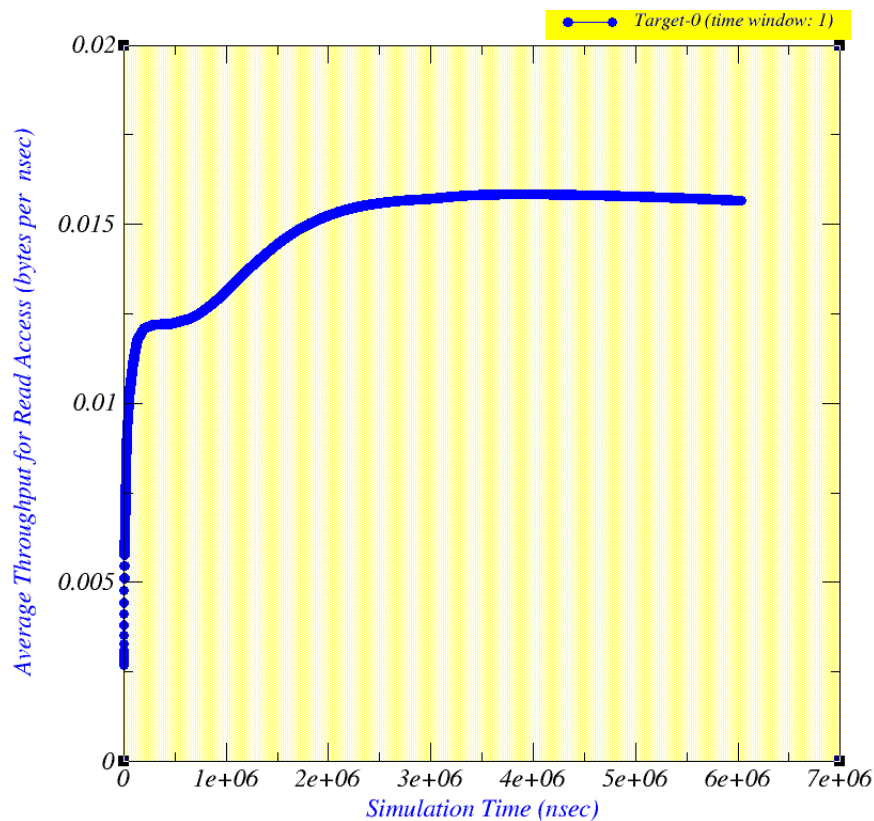


FIGURE 5.20: Target-0's average read throughput of the system shown in Figure 5.16 with PL level AHB bus and arbitration SP

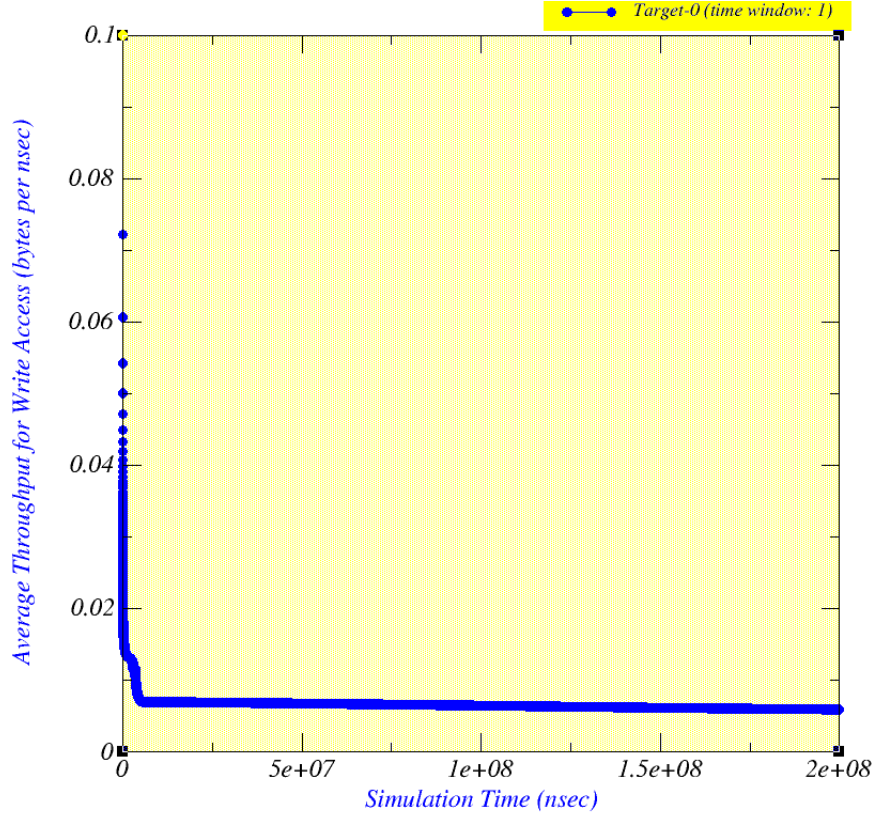


FIGURE 5.21: Target-0's average write throughput of the system shown in Figure 5.16 with PL level AHB bus and arbitration SP

### 5.3 Summary

In this chapter, two demonstrations are discussed to show the usage of the devised system level modeling methodology. The first one is a digital radio receiver streaming application (the DAB receiver). The hardware architecture of the DAB receiver comprises one general purpose processor, two reconfigurable processors, a smart memory tile and one ADC unit. The software application is partitioned into eight processes. Since the software processes have data-dependent behaviour, they are modeled as a set of tasks (states) where each task has a data-independent behaviour in terms of the amount of input data it needs for execution. The system is modeled at three different abstraction levels: FM, PM and TSLM. The second demonstration is the modeling of an AMBA bus based system that comprises random traffic generators, high level AHB 2.0 bus and target memories. For each model in both demonstrations, implementation aspects are discussed, example use cases are shown and the differences from other models are contrasted. The next chapter discusses the investigation results of the demonstration models and recommends possible extensions to this thesis work.



## Chapter 6

# Conclusions and Future works

In this chapter, the key achievements and drawbacks of the work are pointed out and possible extensions to this work are recommended.

### 6.1 Conclusions

Developing an efficient system level modeling and simulation framework for multi-core systems demands a substantial effort. Crafting the design flow, devising the abstraction concepts and developing the necessary tools and libraries are not sufficient alone. The method should be tested with a significant set of real applications and results need to be contrasted with their RTL counterparts to assess the effectiveness of the methodology and appreciate its benefits. It is not the intention of this work to come up with a complete set of libraries, tools and undertake an intensive testing. Nevertheless, this thesis offers a promising groundwork for a SystemC-based modeling and simulation framework for multi-core systems.

One strong aspect of this work is that it offers an integrated approach. All the ingredients needed for the system level modeling of multi-core systems are identified and unlike other works, discussed in *Chapter 3: Related Works*, our approach embraces each of these ingredients. Defining the abstraction layers without providing any directions on how that could be implemented leaves the designer on the desert with no clue. Even offering the communication APIs alone without clear definition of abstraction layers or use cases may lead to confusions on their usage and raise interoperability issues between models. Having a complete toolset on the table is the solution for these problems and that is what this work brings to the system modeling arena.

The approach presented in this work is also able to achieve the main objectives of system level modeling: reduced complexity and faster simulation. The FM, PM and TSLM layers abstract away detailed architectural aspects and provide a simpler but faster virtual platform for early software development. Faster architectural analysis can also be carried out at the PSLM and PSSLM layers without being irritated by the mind-blowing protocol and device nitty-gritty details. Despite the considerable effort needed, the IM layer can be the option to go for functional verification and debugging of the entire system without indulging oneself into HDL implementations. In addition, with IM upto a three times faster simulation can be obtained as compared to their HDL



counterparts. The achieved simulation speed in both demonstration applications, in the order of  $10^6 \text{ cycles/sec}$  is within the range of the TLM simulation technique. The AHB bus system demonstration application also showed the use-cases of the different communication refinement levels. In summary, the main strong aspects of this work include the provided integrated approach, the achieved promising simulation speed, the achieved promising use-cases and its genericity (not tailored for a specific application).

One shortcoming of this work that needs mentioning is that the simulation accuracy of the demonstration applications is not analysed. This is mainly due to the unavailability of models of their cycle-accurate counterparts during the thesis work. In addition, more investigations and test applications needs to be carried out to verify the scalability of the methodology for complex systems which are modeled at the PSSLM and IM layers. This is because the SystemC simulator might be a bottleneck for models with a significant number of CRL-2 (CA-ISS) components.

## 6.2 Future works

This work lays down the foundation for an integrated system level modeling framework for multi-core systems. The methodology as it is now has sufficient ingredients to model small to medium scale systems. Nevertheless, it still has a large space for improvements and additional works. Below here are some potential future works that can be done to extend this work.

1. **Enhancing the library:** The communication API (the communication interfaces and the protocol data units), the communication layer (the channels and the interconnection networks such as bus and NoC) and the models of other components are the corner stones of the modeling methodology. A well designed and tested library of these components fortifies the methodology and ensures designer productivity. Because the significant effort of modeling new components can be avoided as the designer can use off-the-shelf components. In addition, such a library offers well tested and reliable models which can be reused, extended to actual system units and, most importantly, alleviate interoperability problems. Furthermore, the library should be extended with more cycle accurate (CA) interconnection networks as well as CA-ISS/RTL processing elements. These are needed to model systems at the PSSLM and IM abstraction layers and benefit their respective use cases.
2. **Using TLM 2.0:** OSCI's TLM 2.0 provides a communication API which has various communication interfaces and the generic payload for memory-mapped bus interconnection. As discussed in Chapter 3, TLM 2.0 mainly aims to be the industry standard for interoperable TLM modeling. It has all the communication interfaces available in OCCN and adds additional ones such as DMI and Debug interfaces. This means all bus related components in the modified OCCN library can be constructed using TLM 2.0's interfaces without compromising its interoperability objective. Therefore, it might be interesting to reconstruct all bus-related components of the library using TLM 2.0's interfaces and model bus-based multi-core systems using the same design flow presented in this thesis. However, the current TLM 2.0 standard does not have a generic payload for NoC interconnects and hence library components related to NoC interconnects can not be constructed without compromising the interoperability goal.

3. **Automating the design flow:** Section 4.5 showed the design process to follow for the system modeling technique presented in this thesis. Depending on the complexity of the system, manually creating these models may sometimes take considerable effort and time. It may be an interesting task to automate the design flow so as to save significant design effort and to ultimately use the modeling framework for iterative tasks such as design space exploration. During this thesis work, a simple implementation is tried out to show how this can be done using object-oriented programming techniques. However, such an approach can be extended to full scale in order to automate the construction of a multi-core system at a particular abstraction layer given the specifications for architecture and the software applications.



## Appendix A

# Codes of Selected Library Components

### Listing A.1: Declaration of OCCN's Pdu class template

The following code shows the declaration of OCCN's Pdu class template. This class provides the data structure which is exchanged between initiators and targets in all transactions.

```
template <class W, class R>
class BMasterPort;

template <class W, class R>
class BSlavePort;

template <typename H, typename BU=H, int size=1>
class Pdu
{
public:
    Pdu();
    // Assignments modify & return lvalue.
    Pdu& operator=(const BU& right);
    Pdu& operator=(const BU* right);
    BU& operator[](unsigned int x);
    operator const BU();

    // Conditional operators return true/false:
    int operator==(const Pdu& right) const;
    int operator!=(const Pdu& right) const;

    // std streams
    friend ostream& operator<< (ostream& os, const Pdu& ia);
    //... more std streams go here

    template <typename H2, typename BU2, int size2>
    void* operator new(unsigned int sz,
                      BMasterPort<Pdu,Pdu<H2,BU2,size2> > *port);
    template <typename H2, typename BU2, int size2>
    void* operator new(unsigned int sz,
                      BSlavePort<Pdu,Pdu<H2,BU2,size2> > *port);
    void* operator new(unsigned int sz);

    //for channels implementation like AHB (need for BU OR Hdr transfer only)
    void copy_sdu(Pdu& src);
    void copy_pci(Pdu& src);
    int get_pdu_size();

public:
    enum {pci_size = sizeof(H)};
    enum {sdu_size = size * sizeof(BU)};
```

```
enum {pdu_size = sizeof(H) + size * sizeof(BU)};

union
{
    struct
    {
        H hdr;
        BU body[size];
    }
    pdu;
    char stream[pdu_size];
} view_as;

unsigned int stream_tail;
unsigned int stream_head;
};
```

LISTING A.1: OCCN's Pdu class declaration

**Listing A.2: Bidirectional MasterPort Class Declaration**

The following code shows the declaration of the bi-directional master port class. This class provides the port which initiators can use for both sending and receiving transactions.

```
template<class WPdu, class RPdu=WPdu>
class BMasterPort :public sc_port<BMasterInterface<WPdu,RPdu>,MAX_IF >
{
public:
    BMasterPort();

    void bind(BMasterInterface<WPdu,RPdu>& interface);
    void operator () (BMasterInterface<WPdu,RPdu>& interface);
    void operator () (sc_port<BMasterInterface<WPdu,RPdu>, MAX_IF> &port);

    // communication API
    //synchronous blocking call (emission + propagation + reception delays)
    void send(WPdu* pk);
    // asynchronous blocking call (emission delay)
    void asend(WPdu* pk);
    RPdu* receive();
    void reply();
    void reply(N_uint nb_cycles);
    void reply(sc_time& delay);

    // same with time-out feature
    void send(WPdu* pk, sc_time& time_out, bool& sent);
    void asend(WPdu* pk, sc_time& time_out, bool& sent);
    RPdu* receive(sc_time& time_out, bool& received);

protected:

private:

};
```

LISTING A.2: OCCN's MasterPort class declaration

**Listing A.3: ProcessingElement Class Declaration- Framework only**

```

/*
 * Copyright (c) 2009, Recore Systems B.V., The Netherlands,
 * web: www.recoresystems.com, email: info@recoresystems.com
 *
 * Any reproduction in whole or in parts is prohibited
 * without the written consent of the copyright owner.
 *
 * All Rights Reserved.
 */

class ProcessingElement0 : public sc_module {
public:
    ProcessingElement0(sc_module_name name);
    //The set of MasterPorts it has
    UMasterPort<std::complex<double>,
                std::complex<double> > complexd_master_port0;
    //...
    //The set of SlavePorts it has
    USlavePort<uchar, uchar> uchar_slave_port0;
    //...

    //Buffers for each software process ports
    QueueObject<uchar> uchar_incoming_fifo0 ;
    QueueObject<std::complex<double> > complexd_outgoing_fifo0;
    //...

    SC_HAS_PROCESS(ProcessingElement0);
    sc_in<bool> clock;

private:
    //Events to decide transaction sendings
    sc_event sending_event_complexd_outgoing_fifo0;

    //Instances of software processes mapped on this PE
    Actor0 adc;
    Actor3 synch;
    //...

    //SystemC processes for computation, reception and sending
    void computationProcess();
    void ucharFifo0ReceivingProcess();
    void complexdFifo0SendingProcess();
    //...
};

```

LISTING A.3: The skeleton of class declarations of processing elements

**Listing A.4: Example software process written as a set of data-independent tasks**

To enable multiple process execution on a processing element

```

/*
 * Copyright (c) 2009, Recore Systems B.V., The Netherlands,
 * web: www.recoresystems.com, email: info@recoresystems.com
 *
 * Any reproduction in whole or in parts is prohibited
 * without the written consent of the copyright owner.
 *
 * All Rights Reserved.
 */

class Actor3 {
public:
    Actor3();
    void estimateFineOffsets(cvec &samples, uchar mode,
        uint &timingOffset, double &frequencyOffset);
    int estimateCoarseFrequencyOffset(cvec &samples, uchar mode);
    cvec correctFrequencyOffset(cvec &samples, uchar mode,
        double &frequencyOffset);
    void synchronize( QueueObject<double> &fine_frequency_offset,
        QueueObject<uchar> &mode_received,
        QueueObject<cvec> &sampleBuffer,
        QueueObject<std::complex<double>> &adc_samples,
        QueueObject<cvec> &to_ofdm
        );
    int state;
private:
    cvec fineSyncBuffer;
    cvec ofdmFrameBuffer;
    int index;
    uchar mode;
    std::complex<double> tempSample;
};

Actor3::Actor3(){
    state=0;
}

void Actor3::synchronize( QueueObject<double> &fine_frequency_offset,
    QueueObject<uchar> &mode_received,
    QueueObject<cvec> &sampleBuffer,
    QueueObject<std::complex<double>> &adc_samples,
    QueueObject<cvec> &to_ofdm
    ){
    switch(state){
    case 0:
        mode=mode_received.remove();
        fineSyncBuffer= sampleBuffer.remove();
        state=1;
        break;
    case 1:
        if (mode > 0 && mode < 5){
            double fineFrequencyOffset; uint fineTiming;
            estimateFineOffsets(fineSyncBuffer, mode, fineTiming, fineFrequencyOffset);
            cvec ofdmSymbol =
                fineSyncBuffer.mid(fineTiming, ModeParameters::TU[mode-1]);
            ofdmSymbol = correctFrequencyOffset(ofdmSymbol, mode, fineFrequencyOffset);
            cvec freqDomainSymbol = fft(ofdmSymbol);
            int coarseFrequencyOffset =
                estimateCoarseFrequencyOffset(freqDomainSymbol, mode);
            fine_frequency_offset.add(fineFrequencyOffset + 2*pi*coarseFrequencyOffset);
            ofdmFrameBuffer = fineSyncBuffer.get(fineTiming, fineSyncBuffer.size()-1);
            ofdmFrameBuffer.set_size(
                ModeParameters::TS[mode-1]*ModeParameters::L[mode-1], true);
            state=2;
            index=(fineSyncBuffer.size() - fineTiming);
        }
        else state=4;
        break;
    }
}

```



```

    case 2:
        if(index < ofdmFrameBuffer.size()){
            ofdmFrameBuffer[index] = adc_samples.remove();
            index++;
        }
        else{
            to_ofdm.add(ofdmFrameBuffer);
            state=3; index=0;
            cout<<"Actor3: vector sent to ofdm!"<<endl;
        }
        break;
    case 3:
        if(index < ModeParameters::TNULL[mode-1] -
            ModeParameters::TCP[mode-1]){
            tempSample=adc_samples.remove();
            index++;
        }
        else{
            state=4; index=0;
            cout<<"Actor3: null symbols skipped!"<<endl;
        }
        break;
    case 4:
        if(index < fineSyncBuffer.size()){
            fineSyncBuffer[index]= adc_samples.remove();
            index++;
        }
        else
            state=1;
        break;
    default:
        cout<<"Actor3: unsupported state!"<<endl;        break;
    }
}

```

LISTING A.4: Example software process written as a set of data-independent tasks

## Appendix B

# Codes of Main Elements in AMBA AHB System Implementation

### Listing B.1: AMBA AHB Bus SystemC Process at Phase-Level Abstraction

The following code listing shows the implementation of the AMBA AHB bus process at phase-level abstractions with two phases.

```
template <typename MasterPdu, typename SlavePdu>
void AMBA_AHB_Bus<MasterPdu, SlavePdu>::AMBA_AHB_Bus_PL_Process(){
    N_int id_initiator=-1;
    N_int id_target=-1;
    int number_of_phases=2;
    BurstType burst_type;
    Transfer_type transfer_type;

    // enable events
    for (N_uint i=0; i<masters.get_length(); i++)
    {
        masters[i]->enable_writing_event();
        masters[i]->enable_reading_event();
    }
    for (N_uint i=0; i<slaves.get_length(); i++)
    {
        slaves[i]->enable_writing_event();
        slaves[i]->enable_reading_event();
    }

    // infinite loop
    do{
        // get or wait for the next request
        id_initiator=get_next_request_initiator_id();
        if (id_initiator == -1)
        {
            wait(*masters_write_ev);
            id_initiator=get_next_request_initiator_id();
        }
        id_target = get_slave_id_according_address(
                    occn_hdr(*(masters[id_initiator]->
                    get_write_pdu_ptr()),HADDR));
        burst_type= (BurstType)occn_hdr(*(
                    masters[id_initiator]->
                    get_write_pdu_ptr()),HBURST);
        transfer_type = (Transfer_type)occn_hdr(
                    *(masters[id_initiator]->
                    get_write_pdu_ptr()),HTRANS);
    } while (1);
}
```

```

    //if burst type is not unspecified length incremental type
    if(burst_type!=INCR){
        //for each beat in the burst
        for(int i=0; ((i < (burst_type/4)*4) ||
            (burst_type==SINGLE && i==0)); i++ ){
            //for each phase in each beat transfer
            for(int j=0; j < number_of_phases; j++){
                //don't wait the master for data in the data
                //phase of read transactions
                if( !(*(slaves[id_target]->
                    get_read_pdu_ptr()).view_as.pdu.hdr.HWRITE==READ && j==1)){
                    //if the master is not ready for a new transaction
                    if (!masters[id_initiator]->is_writing_completed())
                        wait(*masters_write_ev);
                    //if the slave is not ready for a new transaction
                    if (!slaves[id_target]->is_reading_completed())
                        wait(*slaves_read_ev);
                    swap_master_pdu(id_initiator,id_target);
                    masters[id_initiator]->authorize_writing();
                    slaves[id_target]->authorize_reading();
                    wait(*slaves_read_ev);
                    wait(clk.posedge_event());
                    masters[id_initiator]->notify_sending_completion();
                    slaves[id_target]->notify_receiving_completion();
                }

                // waiting for response from the slave

                if (!slaves[id_target]->is_writing_completed())
                {
                    wait(*slaves_write_ev);
                }
                if (!masters[id_initiator]->is_reading_completed())
                {
                    wait(*masters_read_ev);
                }

                swap_slave_pdu(id_initiator,id_target);
                slaves[id_target]->authorize_writing();
                masters[id_initiator]->authorize_reading();

                wait(*masters_read_ev);
                wait(clk.posedge_event());
                slaves[id_target]->notify_sending_completion();
                masters[id_initiator]->notify_receiving_completion();
            }//End: for each phase
        }//End: for each beat
    }//End: if not INCR

    //If burst type is unspecified length incremental type,
    else
    {
        //process the first beat
        //get the transfer type of the next
        while(transfer_type==SEQ){
            //for each phase in each beat transfer
            for(int j=0; j < number_of_phases; j++){
                //repeat the same code as the previous if clause
            }
            //get the transfer type of the next
        }
    }
}while(1);
}

```

LISTING B.1: AMBA AHB Bus Process at Phase-Level Abstraction

**Listing B.2: SimpleMemory Class Declaration** The following listing shows the implementation of a simple flat memory which is used in the experimental setup of the AMBA AHB bus system demonstration.

```

/*
 * Copyright (c) 2009, Recore Systems B.V., The Netherlands,
 * web: www.recoresystems.com, email: info@recoresystems.com
 *
 * Any reproduction in whole or in parts is prohibited
 * without the written consent of the copyright owner.
 *
 * All Rights Reserved.
 */
template <typename MasterPdu, typename SlavePdu>
class Simple_memory : public sc_module {
public:
    Simple_memory(sc_module_name name);
    ~Simple_memory();
    Simple_memory(sc_module_name name, N_uint64 r_latency,
                  N_uint64 w_latency, N_uint32 mem_size,
                  N_uint32 s_address, std::string abs_layer,
                  int b_width, bool p_screen);
    BSlavePort<SlavePdu, MasterPdu> slave_port0;
    sc_in<bool> clk;
    SC_HAS_PROCESS(Simple_memory);

private:
    StatDelay write_read_latency;
    StatThroughput throuput_read;
    StatThroughput throuput_write;
    N_uint64 read_latency; //byte data read latency
    N_uint64 write_latency; //a single byte data write latency
    N_uint32 memory_size; //the maximum size of the memory in byte
    N_uint32 start_address; //an offset for mapping
    N_uint8 *mem; //memory array
    int bus_width;
    bool print_screen;
    std::string abstraction_layer;

private:
    void memory_Process(); //SystemC process
    bool validate_address(N_uint32 addr, BurstType bt, BeatSize bs);
    void process_idle_transfer_type();
    void process_busy_transfer_type();
    void process_nonseq_transfer_type(
        BeatSize bs, BurstType bt, N_uint32& addr,
        TransactionDirection td, MasterPdu received_pdu);
    void process_seq_transfer_type(
        BeatSize bs, BurstType bt, N_uint32& addr,
        TransactionDirection td, MasterPdu received_pdu);
    bool process_init_phase(
        BeatSize bs, BurstType bt, N_uint32 addr);
    void process_write_data_phase(
        BeatSize bs, BurstType bt,
        N_uint32& addr, MasterPdu received_pdu);
    void process_read_data_phase(
        BeatSize bs, BurstType bt, N_uint32& addr);
    void write_data(
        BeatSize bs, BurstType bt, N_uint32& addr,
        SlaveResponse& slave_resp, N_uint32 data);
    N_uint32 read_data(
        BeatSize bs, BurstType bt, N_uint32& addr,
        SlaveResponse& slave_response);
};

```

LISTING B.2: Class Declaration of Simple Memory



# Bibliography

- [1] The Cell Architecture. URL <http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html>.
- [2] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, The Netherlands, 2002.
- [3] F. Ghenassia. *Transaction Level Modeling with SystemC*. Springer, The Netherlands, 2005.
- [4] CRISP: Cutting edge Reconfigurable ICs for Stream Processing. URL <http://www.crisp-project.eu/>.
- [5] On-chip Communication Architecture, OCCN. URL <http://occn.sourceforge.net/>.
- [6] S. Pasricha and N. Dutt. *On-Chip Communication Architectures, System on Chip Interconnect*. Morgan Kaufmann Publishers, USA, 2008.
- [7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks, An Engineering Approach*. Morgan Kaufmann Publishers, USA, 2003.
- [8] Open SystemC Initiative, OSCI, TLM 2.0 Standard, . URL <http://www.systemc.org/downloads/standards/tlm20/>.
- [9] A. Donlin. Transaction level modeling: flows and use models. In *International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004*, September 2004. URL [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1360484](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1360484).
- [10] TLM 2.0 User Manual (included in the tlm 2.0 library kit), . URL <http://www.systemc.org/downloads/standards/tlm20/>.
- [11] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *15th International Symposium on System Synthesis (ISSS-2002)*., kyoto, Japan, 2002. URL <http://www.cecs.uci.edu/~cad/publications/conferences/2000-04/iss02.semantics.ps.gz>.
- [12] Open SystemC Initiative, OSCI. URL <http://www.systemc.org/home>.
- [13] L. Cai and D. Gajski. Transaction level modeling: An overview. In *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003.*, October 2003. URL [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1275250](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1275250).

- [14] T. Kogel, Haverinen A., and Aldis J. OCP TLM for architectural modeling. In *OCP white papers*. URL [http://www.ocpip.org/socket/whitepapers/OCP\\_TLM\\_for\\_Architectural\\_Modeling.pdf](http://www.ocpip.org/socket/whitepapers/OCP_TLM_for_Architectural_Modeling.pdf).
- [15] A.K. Deb, A. Jantsch, and J. Oberg. System design for dsp applications in transaction level modeling paradigm. In *Proceedings of the 41st annual Design Automation Conference*, California, USA, June 2004. URL <http://portal.acm.org/citation.cfm?id=996698&dl=>.
- [16] Coware's esl2.0. URL <http://www.coware.com/company/esl20.php>.
- [17] Synopsis's innovator, a system level development tool. URL <http://www.synopsys.com/Tools/SLD/VirtualPlatforms/Pages/Innovator.aspx>.
- [18] Mentor's tlm 2.0 based system level modeling tool. URL <http://www.mentor.com/products/esl/news/scalalbe-tlm-esl>.
- [19] OCP-IP: Organization for Common Standards for Intellectual Property (ip) Core Interfaces. URL <http://www.ocpip.org/home>.
- [20] GreenBus, Building Blocks for Tool Independent ESL SystemC. URL <http://greensocs.com/>.
- [21] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of the 41st Design Automation Conference*, California, USA, 2004. URL <http://portal.acm.org/citation.cfm?id=996566.996603>.
- [22] OSSS. URL <http://www.ecsi-association.org/ecsi/projects/odette/downloads.html>.
- [23] ARMn, A Multiprocessor Cycle-accurate Simulator, . URL <http://www.gigascale.org/mescal/forum/197.html>.
- [24] SimIt-ARM- Cycle-accurate ARM Processor Simulator For StrongArm architecture. URL <http://simit-arm.sourceforge.net/>.
- [25] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000. URL [http://www.embedded.eecs.berkeley.edu/asves/embedded/platform\\_based/papers/tcad-system-level.pdf](http://www.embedded.eecs.berkeley.edu/asves/embedded/platform_based/papers/tcad-system-level.pdf).
- [26] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. In *IEEE Transactions on Computers*, volume 36, pages 24 – 35, 1987. URL <http://www.ptolemy.eecs.berkeley.edu/publications/papers/87/staticscheduling/staticscheduling.pdf>.
- [27] World DMB Forum List of Benifits, . URL <http://www.worlddab.org/technology/dab>.
- [28] World DMB Forum DMB Technology Page. URL <http://www.worlddab.org/technology/dmb>.
- [29] World DMB Forum, . URL <http://www.worlddab.org>.

- 
- [30] DAB Standard (ETSI EN 300 401 V1.4.1). *Radio Broadcasting Systems: Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*. June 2006. URL <http://www.etsi.org>.
  - [31] W. Hoeg and T. Lauterbach. *Digital Audio Broadcasting, Principles and Applications of Digital Radio*. John Wiley and Sons Ltd, Germany, 2003.
  - [32] Recore Systems. URL <http://www.recoresystems.com/>.
  - [33] ARM AMBA Bus, . URL <http://www.arm.com/products/solutions/AMBAHomePage.html>.





# Abbreviations

<b>ASIC</b>	<b>A</b> pplication <b>S</b> pecific <b>I</b> ntegrated <b>C</b> ircuit
<b>CA</b>	<b>C</b> ycle <b>A</b> ccurate
<b>CCATB</b>	<b>C</b> ycle- <b>C</b> ount <b>A</b> ccurate at <b>T</b> ransaction <b>B</b> oundaries
<b>CRL-0</b>	<b>C</b> omputation <b>R</b> efinement <b>L</b> evel-0
<b>CRL-1</b>	<b>C</b> omputation <b>R</b> efinement <b>L</b> evel-1
<b>CRL-2</b>	<b>C</b> omputation <b>R</b> efinement <b>L</b> evel-2
<b>DAB</b>	<b>D</b> igital <b>A</b> udio <b>B</b> roadcasting
<b>DMA</b>	<b>D</b> irect <b>M</b> emory <b>A</b> ccess
<b>DMB</b>	<b>D</b> igital <b>M</b> ultimedia <b>B</b> roadcasting
<b>ESL</b>	<b>E</b> lectronic <b>S</b> ystem <b>L</b> evel
<b>FM</b>	<b>F</b> unctional <b>M</b> odel
<b>GPIO</b>	<b>G</b> eneral <b>P</b> urpose <b>I</b> nterface <b>O</b> utput
<b>HDL</b>	<b>H</b> ardware <b>D</b> escription <b>L</b> anguage
<b>IM</b>	<b>I</b> mplementation <b>M</b> odel
<b>IP</b>	<b>I</b> ntellectual <b>P</b> roperty
<b>ISS</b>	<b>I</b> nstruction <b>S</b> et <b>S</b> imulator
<b>PE</b>	<b>P</b> rocessing <b>E</b> lement
<b>PM</b>	<b>P</b> rocess <b>M</b> odel
<b>PSLM</b>	<b>P</b> hase-based <b>S</b> ystem <b>L</b> evel <b>M</b> odel
<b>PSSLM</b>	<b>P</b> rotocol <b>S</b> pecific <b>S</b> ystem <b>L</b> evel <b>M</b> odel
<b>PV</b>	<b>P</b> hase <b>V</b> iew
<b>RTL</b>	<b>R</b> egister <b>T</b> ransfer <b>L</b> evel
<b>RTOS</b>	<b>R</b> eal <b>T</b> ime <b>O</b> perating <b>S</b> ystem
<b>SLM</b>	<b>S</b> ystem <b>L</b> evel <b>M</b> odel

<b>SoC</b>	<b>S</b> ystem-on- <b>C</b> hip
<b>SV</b>	<b>S</b> ignal <b>V</b> iew
<b>TI</b>	<b>T</b> ransaction <b>I</b> nterface
<b>TLM</b>	<b>T</b> ransaction <b>L</b> evel <b>M</b> odeling
<b>TSLM</b>	<b>T</b> ransfer-based <b>S</b> ystem <b>L</b> evel <b>M</b> odel
<b>TV</b>	<b>T</b> ransfer <b>V</b> iew
<b>UT</b>	<b>U</b> n- <b>T</b> imed

# Glossary

<b>Initiator:</b>	Component of the modeled system that initiates a transaction. Eg: processor
<b>Target:</b>	Component of the modeled system that processes a transaction. Eg: memory
<b>Transaction:</b>	The exchange or synchronization of data and/or control information between two components of a modeled and simulated system. Transactions are either <i>read</i> or <i>write</i> transfers initiated by an initiator and processed by a target. The terms initiator and target are used to indicate the direction of the control flow for the transaction, which is from an initiator to target. The data, however, may flow in both directions
<b>Write Transaction:</b>	A transaction to transfer data from initiator to target.
<b>Read Transaction:</b>	A transaction to transfer data from target to initiator.
<b>Node:</b>	Component of the modeled system that is either an initiator, a target or both an initiator and a target. Eg: Processor, DMA Controller, GPIO etc.
<b>Processing Element:</b>	An initiator node on which software processes can be mapped and executed.
<b>Interconnection Network:</b>	A set of system components other than nodes connected in a certain manner for the sake of relaying transactions between nodes. Eg: Bus, point-to-point link, mesh network, etc.
<b>Transaction Interface:</b>	An abstract model of the physical communication interfaces of IPs.

---

<b>Process:</b>	A portion of a software application after partitioning. Each process is further divided into tasks. This does not refer to SystemC processes, which will be explicitly specified whenever mentioned.
<b>Task:</b>	A portion of a software process which can be fully executed without the need for further input data.
<b>Master and Slave:</b>	When used to refer a node, the term master is equivalent to an initiator and the term slave to a target.
<b>Master Interface:</b>	A SystemC module that inherits from <code>sc_interface</code> and declares virtual functions to be implemented by channels. It should have enough function constructs that enable it to initiate new transactions.
<b>Slave Interface:</b>	A SystemC module that inherits from <code>sc_interface</code> and declares virtual functions to be implemented by channels. It does not have constructs for initiating transactions but has functions for processing initiated transactions.
<b>Master Port:</b>	An abstract model of the physical I/O port of a master IP. A typical implementation is a SystemC module that inherits from <code>sc_port</code> and is binded to a Master Interface to be used by Initiators for carrying out transactions.
<b>Slave Port:</b>	An abstract model of the physical I/O port of a slave IP. A typical implementation is a SystemC module that inherits from <code>sc_port</code> and is binded to a Slave Interface to be used by Targets for carrying out transactions.
<b>Channel:</b>	A channel is an abstract model of a physical link in SoCs. A typical implementation is a SystemC module that implements transaction functions declared by Master and Slave Interfaces so as to relay transactions from initiators to targets.