Power estimation framework based on SystemC-TLM performance models of SoC interconnect and memory systems

Antonio Genov

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Estimation de la consommation basée sur les modèles de performance SystemC-TLM des systèmes d'interconnexion et de mémoire des SoC

Antonio Genov

Laboratoire d'Electronique, Antennes et Télécommunications (LEAT)

Présentée en vue de l'obtention du grade de **Docteur** en **Électronique d'Université Côte d'Azur (UCA)**

Dirigée par :
Prof. François Verdier, UCA
Co-encadrée par :
Loic Leconte, Ingénieur Arch., NXP

Soutenue le :
10 Décembre 2021

Devant le jury, composé de :

Dr. Arnaud Tisserand, Directeur de Recherche, CNRS
Prof. Frédéric Pétrot, Université de Grenoble
Prof. Ludovic Apvrille, Télécom Paris
Dr. Saif Ur Rehman, Ingénieur de conception, NXP
Manfred Thanner, Mgr. Analyze architecture, NXP
Dr. Tim Kogel, Pr. Ingénieur Prototypage, Synopsys

NXP

Estimation de la consommation basée sur les modèles de performance SystemC-TLM des systèmes d'interconnexion et de mémoire des SoC

Power estimation framework based on SystemC-TLM performance models of SoC interconnect and memory systems

Jury:

Rapporteurs

Arnaud Tisserand, Dr., Directeur de Recherche, CNRS
Frédéric Pétrot, Prof., Université de Grenoble

Examinateurs

Ludovic Apvrille, Prof., Télécom Paris
Saif Ur Rehman, Dr., Ingénieur de conception, NXP
Tim Kogel, Dr., Pr. Ingénieur Prototypage, Synopsys

Invités

Manfred Thanner, Mgr. Analyze architecture, NXP

Year: 2021
Title: Thesis at the University of Cote d'Azur
Author: Antonio GENOV

Glossary

API: Application Programming Interface - a software connecting computers or computer programs and offering a service to other pieces of software.

ARM: Advanced RISC Machines - providers of IPs, mainly processors.

CAD software: Computer-Aided Design - the use of computer and software to increase designer's productivity.
CD: Clock domain - the set of components that share the same clock source.

CG: Clock gating - dynamic power reduction technique used in synchronous circuits.

CMOS: Complementary Metal–Oxide–Semiconductor - a technology for manufacturing electronic components and the components manufactured using this technology.

CPU: Core Processing Unit - a hardware block that retrieves and executes instructions. It consists of an arithmetic and logic unit (ALU), a control unit, and various registers.

Design Flow: The explicit combination of design tools and steps to perform the design of an integrated circuit.

DFT: Design For Test - IC design techniques to add testability features to a hardware product design.

DRAM: Dynamic Random-Access Memory - volatile store memory, typically based on MOS technology.

DRC: DRAM Controller - component managing the movement of data in and out of DRAM devices

EDA: Electronic Design Automation - software tools for designing electronic systems.

ESL: Electronic System Level - a relatively high abstraction level used for SoC architecture definition.

FD-SOI: Fully Depleted Silicon on Insulator - planar process technology used for the engraving of transistors.

FinFet: Fin Field-Effect Transistor - a type of non-planar transistor, which is the basis for the fabrication of modern nanoelectronic semiconductor devices.

HDL: Hardware Description Language - a specialized computer language used to describe the structure and behavior of electronic circuits.

HW: Hardware - physical part of a computer; electronic device.

IC: Integrated Circuit - a set of electronic circuits integrated on one flat piece of semiconductor material, usually silicon.

InterConnect: Hardware module that connects the different elements of a SoC.

ISS: Instruction Set Simulator - a simulation model, which mimics the behavior of microprocessor.

IP: Intellectual Property - term used to designate ownership of a design or idea.

LPDDR: Low Power Double Data Rate - type of synchronous DRAM that consumes less power.

MMU: Memory Management Unit - element that translates the virtual addresses of queries into physical addresses using translation tables.

OS: Operating System - a software managing the computer hardware and software resources to provide common services for computer programs.

PD: Power domain - a sub-assembly of a SoC whose power can be switched off internally from the SoC to mainly reduces the static power consumption.

Power/Clock Intent: Power/Clock Intent - describes the partitioning of a design into power domains and clock domains. The Power/Clock Intent also sometimes describes the control signals used to control these power and clock domains.

QoS: Quality of Service - a concept that optimizes the use of resources during a process and ensures good performance of the targeted applications.

RTL: Register Transfer Level - a method for describing microelectronic architectures.

SoC: System on Chip - an abbreviation used to name an IC combining all the required computer components onto a single chip.

SW: Software - a program/set of instructions executed by one or more hardware components.

TBU / TCU: Translation Buffer Unit / Translation Control Unit - components that are part of the MMU and are used to store and control address translations.

Technology: Technology can be defined as a systematic study of techniques for making and doing things.

TLM: Transaction Level Modeling - a high-level approach to modeling digital systems by defining a sets of transactions transferred over a set of channels.

UVM: Universal Verification Methodology - a standardized methodology for verifying integrated circuit designs.

UPF: Unified Power Format - standardized power format specification to implement low power techniques in a design flow.

VD: Voltage domain - set of components or part(s) of components sharing a power source.

VLSI: Very-Large-Scale Integration - the process of creating an IC by combining millions of MOS transistors onto a single chip.

VP: Virtual Platform - a software based system mirroring the functionality of a hardware or SoC.

XML: eXtensible Markup Language - a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

uC: Microcontroller - an IC that combines several types of memories, a microprocessor, and communication peripherals in a single package.

Abstract

The rapid pace of development in microelectronics enables the semiconductor industry to constantly surpass itself and to offer ever more innovative and complex products and technologies. The most modern areas of development, such as 5G, the Internet of Things (IoT) and automotive, rely on complex, high-performance, low-power designs. Unfortunately, this increased complexity often leads to higher power consumption and more challenging designs.

In order to solve these problems and differentiate themselves in the market, System-on-Chip (SoC) manufacturers and engineers are putting tremendous effort into researching new development strategies. Numerous studies have shown that one of the key steps to take is to revisit the early stages of the design flow and, in particular, to integrate simulation-based modeling and verification at a higher level of abstraction. The early stages of product development are critical to avoiding costs, delays, and other unexpected problems. As a result, Hardware/Software (HW/SW) architectural exploration has become a key component of SoC modeling.

In this thesis, we address this gap and present a framework for mixed performance and power estimation/management of SoCs using high-level SystemC/TLM2.0 functional models. Our methodology allows us to dynamically extract performance and power, while considering functional model activity, power management and reduction strategies, and memory system consumption. In this way, we can observe the impact of power management on performance and optimize the trade-off between the two at the very beginning of the design flow. We address this shortcoming and present our first dynamic approach for mixed power/performance estimation applied to an NXP Intellectual Property (IP) interconnection subsystem used in i.MX8 SoC series. This modeling methodology uses the PwClkARCH library, which follows UPF semantics and enables power estimation and management. Its key point is that it maintains a strong separation between the functional code and the power intent description. There is no intrusive power-oriented code in the functional model, which simplifies architectural exploration, allows joint and separate reuse of behavioral and power models, and leads to more complete code and easier performance estimation.

Keywords*: System-on-Chip; Estimations; Power consumption; System-level modeling; Transaction Level Modeling (TLM) ; Abstraction ; SystemC*

Resumé

Le rythme rapide de développement de la microélectronique permet à l'industrie des semiconducteurs de se surpasser constamment et de proposer des produits et des technologies toujours plus innovants et complexes. Les domaines de développement les plus modernes, tels que la 5G, l'Internet des objets (IoT) et l'automobile, reposent sur des conceptions complexes, performantes et à faible consommation. Malheureusement, cette complexité accrue entraîne souvent une consommation d'énergie plus élevée et des conceptions plus difficiles.

Afin de résoudre ces problèmes et de se différencier sur le marché, les fabricants et les ingénieurs de Systèmes sur Puce (SoC) déploient des efforts considérables pour rechercher de nouvelles stratégies de développement. De nombreuses études ont montré que l'une des mesures clés à prendre consiste à revoir les premières étapes du flot de conception et, en particulier, à intégrer la modélisation et la vérification basées sur la simulation à un niveau d'abstraction plus élevé. Les premières étapes du développement d'un produit sont essentielles pour éviter les surcoûts, les retards et autres problèmes inattendus. Par conséquent, l'exploration architecturale matérielle/logicielle (HW/SW) est devenue un élément clé de la modélisation des SoC.

Dans cette thèse, nous comblons cette lacune et présentons un cadre pour l'estimation/gestion mixte des performances et de la puissance des SoCs en utilisant des modèles fonctionnels SystemC/TLM2.0 de haut niveau. Notre méthodologie nous permet d'extraire dynamiquement la performance et la puissance, tout en considérant l'activité du modèle fonctionnel, les stratégies de gestion et de réduction de la puissance, et la consommation du système de mémoire. De cette façon, nous pouvons observer l'impact de la gestion de la puissance sur la performance et optimiser le compromis entre les deux au tout début du flot de conception. Nous abordons cette lacune et présentons notre première approche dynamique pour l'estimation mixte de la puissance et de la performance appliquée à un sous-système d'interconnexion de la Propriété Intellectuelle (IP) de NXP utilisé dans la série de SoC i.MX8. Cette méthodologie de modélisation utilise la bibliothèque PwClkARCH, qui suit la sémantique de UPF et permet l'estimation et la gestion de la puissance. Son point clé est qu'elle maintient une forte séparation entre le code fonctionnel et la description de l'intention de puissance. Il n'y a pas de code intrusif orienté puissance dans le modèle fonctionnel, ce qui simplifie l'exploration architecturale, permet une réutilisation conjointe et séparée des modèles comportementaux et de puissance, et conduit à un code plus complet et à une estimation plus facile des performances.

***Mots-clés**: Système-sur-Puce; Estimations; Consommation de puissance; Modélisation au niveau système; Modélisation au niveau transactionnel; Abstraction; SystemC*

Preface

This is a CIFRE PhD project started with the collaboration between LEAT and NXP France. Due to some confidentiality and security measures, we will limit the detailed information on the NXP proprietary IPs designs and architectures. The chapter representing the behavioral modeling and the results will be synthesized, normalized and probably very generalized, thus it will represent a small fraction of the work actually done. In addition, all metrics (from measurements and simulation data) and scales in the figures and tables will be hidden as they are considered sensitive data.

Acknowledgements

There are so many people who have inspired and helped me during these last three years and I would like to say a big thank you to them!

First of all, I would like to thank all the structures of NXP Semiconductors and the LEAT laboratory, because they believed in me and gave me the opportunity to learn, build and pursue my research.

I would especially like to thank my thesis supervisor, Pr. François Verdier for his continuous support, his unwavering ambition to help me, and for all the time and effort he put into guiding me in my research.

This thesis would not have been possible without the considerable help and efforts of my industrial supervisor Mr. Loic Leconte. My words will not suffice to express my gratitude for all the knowledge, advice, and guidance he has given me. Thank you.

Sincere thanks also to my manager Mr. Moussa Belkhiter, who encouraged me with his dynamism, punctuality and positive energy.

I am very grateful to all the members of the jury Mr. Arnaud Tisserand, Mr. Frédéric Pétrot, Mr. Ludovic Apvrille, Mr. Saif Rehman, Mr. Manfred Thanner and Mr. Tim Kogel who accepted and took the time to read and listen to what I have to say. It is a great honor for me.

I would also like to thank all my colleagues who have made the last three years a real adventure for me. During this time I have met some amazing and inspiring people, but most importantly I have met new friends.

Last but not least, I would like to express my immense gratitude to my family. Thank you to my mother Mariana Genova, my father Evgeni Genov and my brother Nikolay Genov who, despite the territorial distance between us, are always by my side. Thank you to my wife Kristina Doncheva and my child Emanuel Genov for their patience, support and love. I want you to know that I owe you everything I am and have. I love you.

Contents

ListofFigures

Chapter 1

Introduction: Thesis overview

1.1 General introduction

We are more and more connected to all kinds of electronic devices. We use hundreds of them every day without thinking much about the technology behind them, their limits and the amount of human effort devoted to their production and optimization. The reason for this is that the semiconductor and electronic engineering industries are doing everything they can to improve, satisfy and simplify the customer experience. Over the past two decades, these industries have encountered many vicissitudes, such as approaching the limits of Moore's Law [9], chip shortages, rapidly changing technologies, and incredible growth in complexity, the result of which has predetermined the overall development of future technologies. The "computer era" (90s) was followed by the "mobile era" (2000), which changed and rearranged priorities in the production of semiconductor devices. Embedded systems have proven to be very effective in many application areas and have conquered the market. Since about 2007, they have led to a boom for smartphones and Internet-of-Things (IoT) devices. Their continuous optimization has led to their implementation in industrial equipments, cars and the design of all types of sensors and microcontrollers. These advances have led to the creation of new emerging markets, such as Artificial Intelligence (AI), quantum computers, advanced wireless networks (including 5G), electric and autonomous vehicles. We have entered a fast-moving circle in which advanced semiconductor devices have led to the creation of higher quality and more affordable products, which has led to greater profits, which in turn has led to faster development and the setting of higher goals.

Thanks to this fast pace, we have managed to optimize the already established cloud computing and complement it with more performant edge computing technology [10]. As a result, due to the high demand for efficient embedded systems, many benchmark technology processes have undergone general changes and innovations, and many technological limits have been reached and/or exceeded. The main reason for this is the change in human perceptions and the doors that have been opened by discoveries made before 2000, but buried at that time due to the inability to develop their maximum potential (such as artificial intelligence, wearable devices and electric cars). The increased computational power of today's computing technologies has allowed us to revive these technologies and push their capabilities further. Both the many challenges and current technological advances have led to the study of new methods, materials, and architectural strategies that may enable the implementation of several major projects in the future.

Motivated by increasingly innovative and complex ideas for electronic devices and embedded systems, semiconductor device manufacturers have been able to push technology even further and have managed to continue to reduce the size of transistors and to increase their density in a single chip following probably the last years of Moore's Law [9]. At the same time, designers of new electronic devices have innovated their methodologies and processes in a way that allowed them to accelerate the design procedure, improve circuit designs and increase chip performances.

There are many disruptive design approaches, technologies, algorithms, and paradigms that have made this possible, but this thesis is primarily related to two of them: increasing the level of abstraction in the design process and developing efficient interoperable devices. Increasing the level of abstraction in the design flow has played a huge role in increasing designer productivity. The transition from the gate/schematic level to the Register Transfer Level (RTL), initiated in the early 1980s, and its continued optimization to this day, has temporarily slowed the ever-increasing gap between designer productivity and the increasing density of transistors in a chip [11]. Hardware Description Languages (HDLs) have allowed designers to increase the level of abstraction at which they work, and as a result, design capability has increased from hundreds to thousands and millions of transistors. However, while designer productivity is increasing, chip capacity is increasing at a much higher rate (Figure 1.1).

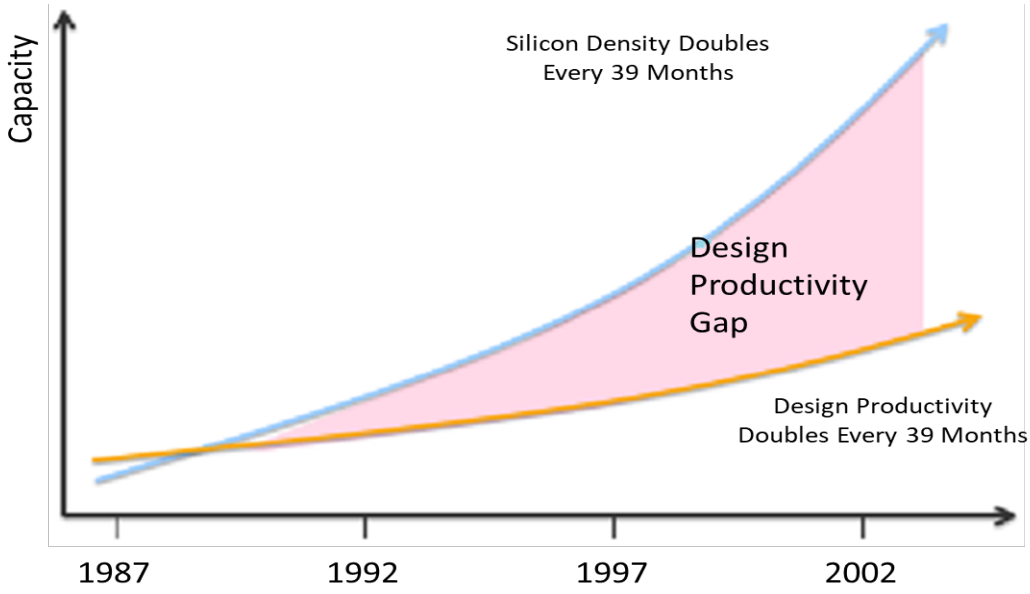


Figure 1.1: Design productivity gap [1]

It therefore became inevitable to start looking for possible approaches based on even higher levels of abstraction. Between 1997 and 2001, a new term, Electronic System Level (ESL), was coined, indirectly defining the (currently) highest level of abstraction - the system level [12]. Although the topic and research on moving to a higher level of abstraction than RTL had begun several years earlier, this is when the term ESL became more concrete. Initially, the idea of the ESL approach was to simplify the understanding of complex systems and to add another layer of design and verification that would allow for better implementation and cost control. Multiple high-level languages, tools, and Electronic Design Automation (EDAs) focused on ESL have enabled behavioral modeling of entire systems. As such, they

have greatly improved and accelerated the design definition and exploration processes. This approach has been widely adopted by System-on-Chip (SoC) designers and is gaining interest. It enables early functional modeling and performance estimation of new designs to eliminate bugs (or at least avoid their late discovery), choose the optimal partitioning of the system, and improve its performance. In addition, it has made possible the development of virtual platforms that are typically used to co-simulate hardware and software models earlier in the design flow and to initiate software development and verification in parallel with the hardware. This approach has addressed many flaws in the design process. The more abstract description and neglect of implementation-specific details accelerate both the design period and the simulation time. Numerous EDA tools have been created to facilitate this process and even enabled the synthesis of these hardware models. The ESL approach has proven again that increasing abstraction is an excellent way to speed up the design process and thus increase design productivity.

1.2 Context and objectives

The famous Dennard's scaling law [13], according to which the power density of chips remains constant as the size of transistors decreases, collapsed around 2006. The reason for this is that as transistor size decreases, current leakage and heat dissipation increase and thus power consumption becomes too large to manage. Dennard's downscaling (starting in 2004) shifted the focus from continuously increasing clock frequency to multicore architectures. Its collapse was closely related to the inability to significantly increase the clock frequency of Central Processing Units (CPUs). Multicore architectures were therefore the only solution to further increase performance and almost the entire industry focused on this. As a result, the integration of more cores increased the number of active switching elements, resulting in even higher static and dynamic power consumption. One way to reduce power consumption has been the wider adoption and improved use of existing SoC technology. The British semiconductor and software design company ARM [14] has been instrumental in this area. This company licensed the design of a "fabless" processor that proved to be highly optimized in terms of power consumption and offered advanced power management features, such as Sleep and Deep Sleep modes, different clock gating features and power domains. This breakthrough has completely revolutionized mobile devices, which are highly dependent on power consumption, a monumental constraint for them since forever. From 2006/2007 to today, the relentless adoption and production of ARM-based SoC devices has pushed the performance, complexity, and applications of SoCs ever further. As a result, more and more switching elements are added, and the continuous on-chip data circulation has become unscalable. Since the data conveyance is a major source of power consumption in modern processors [15] and surrounding nodes, the increase in chip size, wire length, and design complexity has made the interconnection between multiple nodes a bottleneck for low-power systems [16].

Many power optimization techniques have been introduced, allowing to activate certain parts of the SoC only when they are used, to distribute the computations between different blocks (CPU, GPU, DSP etc...) or to vary the voltage and frequency depending on the chip state. These techniques are described in parallel with the RTL models of the chip. However, the optimal application of power reduction techniques has become untenable due to the complexity of the chips and the exorbitant simulation time of the RTL models. Sim-

ulating relatively complex use cases to test these power optimizations ranges from several hours to several days. Given the needs, concerns, and constraints presented above, it was inevitable to think of a solution to combine them and find a reasonable and relatively accurate power estimation approach, applicable to separate blocks or/and entire SoCs, from the early stages of the design flow (system level). In addition, it was important to enable energy management modeling capabilities at this level and dynamically extract energy-related metrics during simulation at a higher speed.

This work is the result of a collaboration between NXP [17] and LEAT laboratory [18]. The objective is to define a simple and reusable system-level modeling framework for i.MX8 SoCs, to develop a model for a complex architecture and, most importantly, to test and evaluate a specific approach to estimate the power consumed by this architecture and the new generation of high complexity SoCs. The architecture chosen for this study is a complex and custom interconnect Intellectual Property (IP) from NXP, called Switch Matrix (SM), that is a central unit in several SoCs of the NXP i.MX8 family. Its purpose is to establish communication between all IPs on the chip, route data transfers, and schedule transactions using several multi-level algorithms. For modeling the Design Under Test (DUT) and the entire testbench from scratch, we used the C++ based SystemC and TLM2.0 libraries. The idea was to create a purely temporal (power-insensitive) behavior/performance model and wrap it, non-intrusively, in a power model description for power management and estimation. The motivation is to have both power and performance estimation capabilities with a single model without overloading it with non-behavioral code. Since there are many high-level, industrialized and academic performance estimation tools, libraries and approaches, our efforts remain focused on early power estimation, which is currently a major gap in the design process. In order to prove the possibility of such mixed power/performance estimation frameworks, we have developed simple and basic observation modules capable of extracting the most important performance metrics, but their evaluation is outside the scope of this study.

For the power estimation, we used an academic tool, called PwClkARCH developed by LEAT researchers. This tool covers all our concerns regarding the separation of power and behavioral models and the co-estimation of power and performance. In addition, it includes power and clock management features that allow us to test multiple power reduction techniques well before RTL. The power estimates are based on modules‘ activity (transactional level) and some area information (detailed in Section 3.4). It can be performed before the HW/SW implementation; thus, it allows us to consider different architecture options and power management strategies and optimize the exploration of the design space.

The strategy adopted in this thesis is to start with a kind of reverse engineering to prove the correctness of this methodology, then transfer it to new architectures and integrate it into the flow. What follows is a very generic overview of the phases and objectives of the thesis.

The first objective was to create the behavioral and power models of the Switch Matrix interconnect IP with an existing i.MX8 SoC configuration. The purpose of this model is to calibrate the power intent and power management related parameters, so that we can correlate our simulations with power measurements on silicon. Next, we needed to reuse the interconnect IP with its power definition and testbench to develop a model of another existing chip to test the interoperability of our functional and power models, perform additional power-on-silicon correlation, and evaluate the modeling effort. The final goal was to reuse this same power-sensitive IP model and carry it over to NXP’s next-generation i.MX8

SoC to extract power measurements early in the design definition. The flow of these generic objectives is visualized in Figure 1.2.

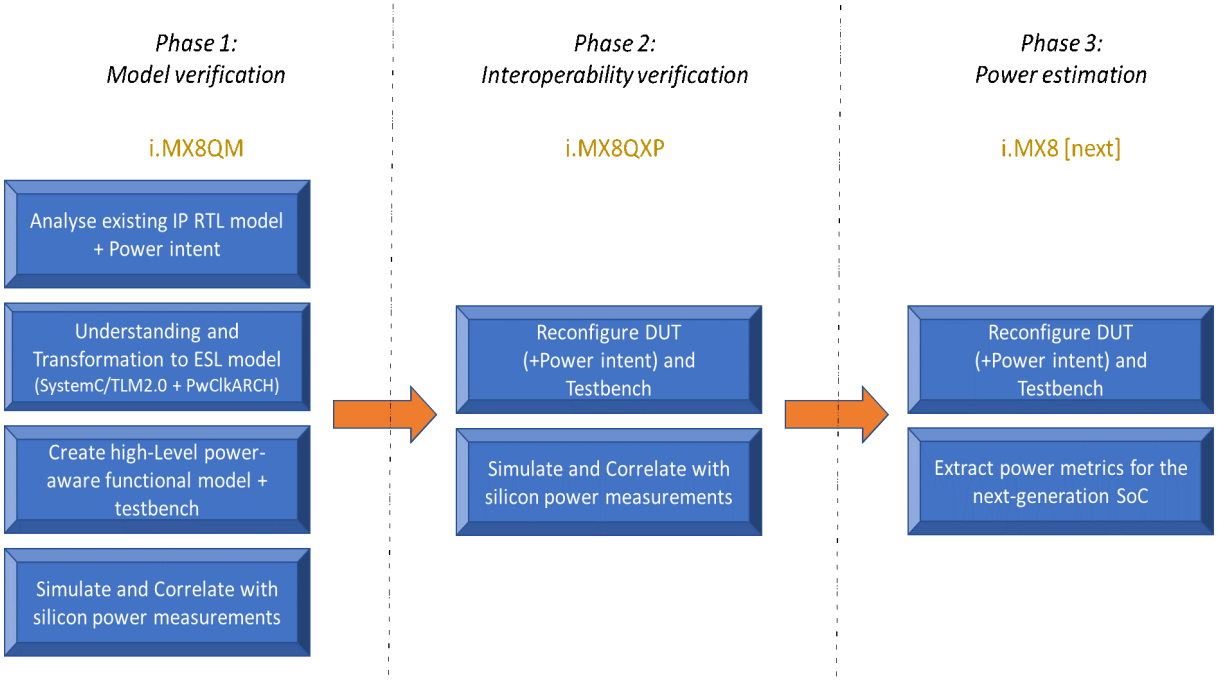


Figure 1.2: Thesis objectives flow

1.3 Summary of contributions

Widely used and well-established approach for early performances estimation of chip is to create functional models using languages like C, C++ and SystemC. At each functional model refinement, we can make the necessary performance analysis by some monitoring or using existing EDA tools. On the other hand, the power consumption is barely studied at this high abstraction level, which is due to the limited information at the beginning of the design flow and the difficulty to specify and test the power management strategies. A commonly used approach, at this level, is a static approach consisting of taking characteristics of the technology for static power consumption estimation, different IP power numbers (given by IP providers) for the worst/best case dynamic power consumption and components area information. Then, the total power consumption is calculated using Excel tables or automated tools and by applying power equations. This ad hoc approach works well for static or pseudo static use cases but cannot be applied on more complex dynamic ones, such as memory copy, display refresh, GPU traffic etc. Therefore, the need for a method to dynamically estimate the energy consumed by the SoC in complex use cases prompted us to investigate ESL-level power estimation methodologies.

With this thesis, we address this gap and present and evaluate the first industrial application of the dynamic PwClkARCH approach for mixed power/performance estimation applied on a complex custom interconnect system from NXP. The framework helps us to develop transaction level models described exclusively in C++/SystemC-TLM2.0 and to extract power consumption and performance metrics. It also enables the use of some well-

known power reduction techniques. The key point of this approach is that it maintains a strong separation between the functional code and the power intent description. There is no intrusive power-oriented code in the functional model, which simplifies architectural exploration, allows joint and separate reuse of behavioral and power models, and leads to more complete code and easier performance estimation. On NXP side, this is the first system level modelisation of a subsystem integrated in the i.MX8 platforms and a first evaluation of a dynamic system level power estimation approach. On the LEAT side, this is the first complex and large industrial proof-of-concept of the PwClkARCH tool performed on an interconnect module and supported by a silicon correlation. Overall, this is an interesting case study, as similar interconnect structures (custom or standard) are used in almost all modern high-performance SoCs, and their power and performance analysis is currently a global bottleneck in SoC development.

1.4 Dissertation structure

Chapter II provides a quick overview of some key insights related to the SoC design flow and analyzes some existing work that inspired and pushed our research further. We then answer the why, how and what of our research and present its significance. The main objective of this chapter is to simplify the understanding of current design flow problems and to identify where exactly this research can be placed and why it is important.

Chapter III describes some strategies, tools and flows for modeling power and performance at the electronic system level. Then, our framework for studying system-level power based on functional chip models at a high level of abstraction is presented. Finally, the technology on which the study was applied is outlined, starting with a brief introduction of the NXP i.MX family as a whole, and then focusing attention on the members modeled and analyzed during this thesis. The purpose of this chapter is to present other research and tools that have contributed to the development of this solution and to compare them to this one. The brief presentation of the technology used in the study provides a better understanding of its architecture, which is important especially when focusing on interoperability.

Chapter IV describes in detail our choices of abstraction level and granularity, as well as some of our strategies for functional modeling and description of the SystemC/TLM communication protocol, performance monitoring and power management. Its purpose is to better explain the structure of this framework and the power/performance extraction tools employed.

Chapter V is devoted to the analysis of the simulation results. In addition, our processes and the results of the power measurements on silicon are presented here. Finally, the correlation between the simulation results and the silicon measurements is performed. Considering that this is a reverse engineering methodology, the performance and accuracy of this framework is evaluated at this stage.

Chapter VI presents the first application of our approach on the new generation of NXP SoCs. Furthermore, some additional tests have been performed in order to integrate some

third-party tools into the framework and to allow their co-simulation. The objective of this chapter is to present how this framework can be integrated into the design flow and its flexibility in terms of co-simulation.

Finally, in **Chapter VII**, a summary of the results is presented, along with some limitations and possible future work. Finally, a general conclusion is drawn.

Chapter 2

Low-Power System-on-Chip (SoC): Background

2.1 System-on-Chip (SoC) definition

The market for SoCs is growing every day, and they are used in all kinds of electronic devices, from small home automation devices to smartphones, cars and spacecraft. By definition, System-on-Chip is an Integrated Circuit (IC) that incorporates a complete electronic system on a single die (Figure 2.1). It is based on interconnected processing units with multiple logic and analog functions. Primary objectives are reducing cost through system integration and improving power consumption. Fully integrated systems reduce the energy consumed when communicating between separate components and reduce the distance between them, thus increasing performance and reducing latency.

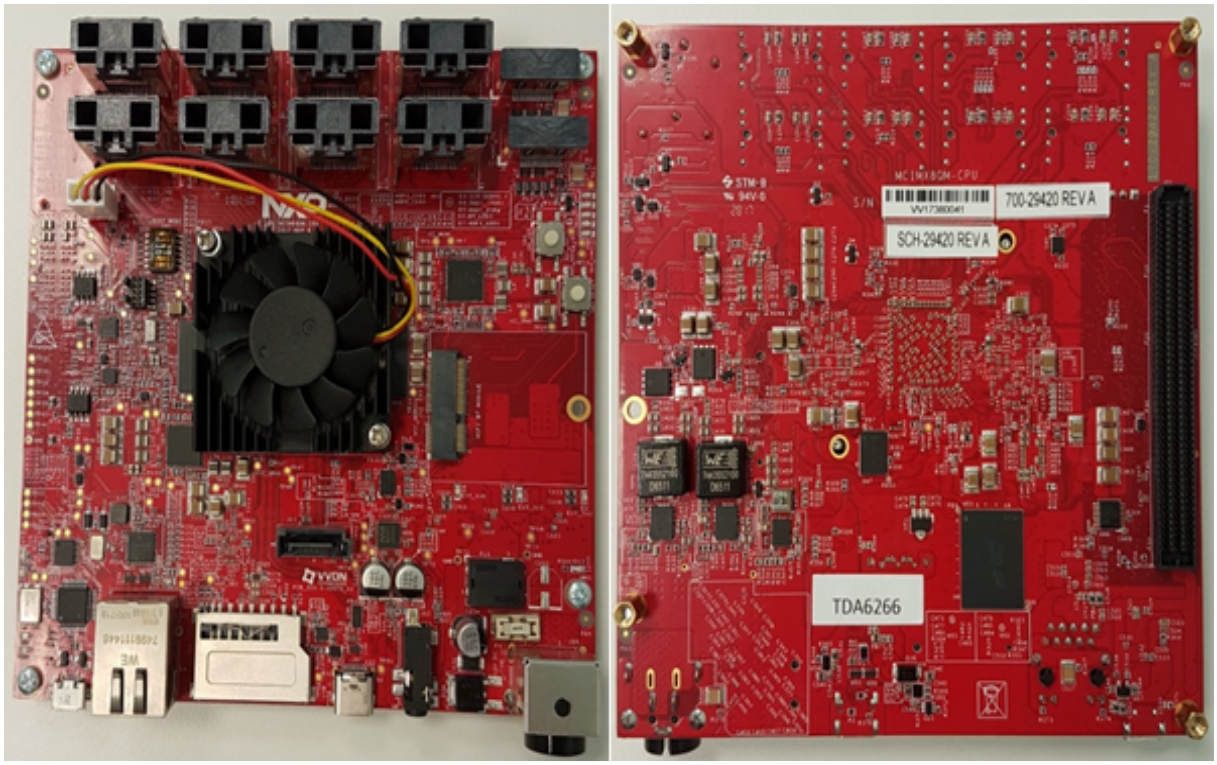


Figure 2.1: NXP SoC i.MX8QuadMax Evaluation Kit

As explained in Section 1.2, most SoCs are heterogeneous devices containing multiple processing elements. These devices are also referred to as multiprocessor SoCs (MPSoCs) and they may contain multi-core blocks or/and logically distinct processing modules. Increasing the number of these processing units also increases the overall number of active switching elements and the amount of data transfers. As a result, power consumption increases significantly. The design of such devices is a complex process and every error or bug found too late in the flow can lead to a significant increase in expenses. That is why it is important to strictly follow a well-defined methodology for their design, or what is called the design flow. In order to better situate our work in the design flow, it is necessary to present a quick overview of its general steps.

2.2 The modern power-aware SoC technologies and design flow shortfalls

The modern power-aware SoC designs must support the integration of advanced CMOS IPs in order to create digital processor-based systems. Its structure is primarily based on digital components, but in some cases may require a Mixed-Signal Design Environment (MSDE) for co-design and analysis of digital and analog IPs into Mixed-Signal subsystems (Figure 2.2).

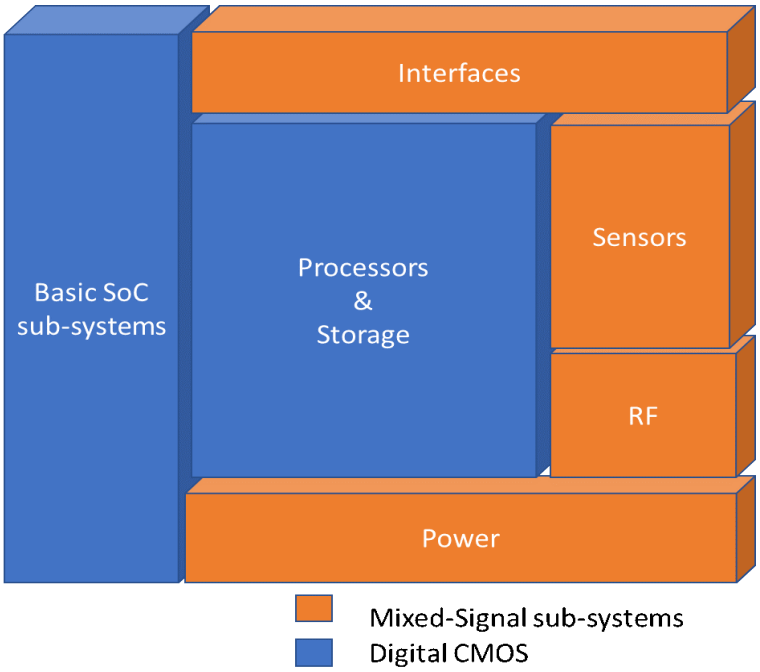


Figure 2.2: Basic SoC sub-systems

For SoC designs, it is essential to establish a well-structured and commonly accepted design methodology. A hierarchical structure is usually adopted for the design flow (see 2.2.1), as it allows it to be decomposed into design flow tasks that can be further broken down into design flow steps. This decomposition can be done separately at each level of the overall flow. To reduce the complexity and cover different types of technology products and

their development needs, the main SoC design flow is divided into 4 distinct design flows based on their technology and constraints. The 4 design flows are:

- **Analog Design Flow** – which is intended for analog and mixed-signals design architectures.
- **Digital Design Flow** – which is aimed at digital design architectures or mixed-signal designs dominated by digital components. Again, it is often extended with mixed-signal capabilities to integrate Analog Mixed-Signal (AMS) blocks into the digital design.
- **RF Design Flow** – based on the Analog design flow, it builds on it. The RF Design flow adds frequency-based simulations and EM analysis capabilities on top of the Analog design flow.
- **SoC Design Flow** – primarily composed of digital components and makes use of some AMS components. In addition to the digital design flow, it targets larger and more advanced multi-processor digital CMOS designs. Moreover, it includes HW/SW system level verification and is heavily based on IP and Platforms reuse and import of 3rd party IPs (memory, AMS blocks, GPUs) as hard-IP or digital **Register Transfer Level (RTL)** soft-IPs.

These flows can be enriched and readapted depending on the manufacturer's needs and internal processes. During the decomposition of the overall design, hierarchy layers can be introduced, resulting in a block-wise approach that allows the reuse of IPs composed of multiple sub-blocks and sub-systems. The interoperability of these Analog and Digital IPs between different design environments is crucial for such complex high-performance SoCs in order to share a common design database that can be reused in subsequent chip generations. **In this study, we are working with the SoC design flow, without directly using the analog part of the device. Thus, our work is closely related to the digital part of the SoC design flow and most of the following information will be based on it.**

2.2.1 Quick overview of SoC digital design methodology and flow

By definition, Design Flow is a mature and silicon-proven design process based on sequential design and verification tasks executed at multiple levels of design abstraction and completion [19]. The complete design flow includes various levels such as system level design, functional/electrical design and physical design. There are multiple steps and processes that need to be performed at each level or are level specific. By following these sequential tasks, the level of abstraction decreases with each further design step and the architecture become more exhaustive. In Figure 2.3, we can observe an abstract view of the main levels and steps present in the flow (detailed later in this section). At each level of design completion, the results from the verification tasks can trigger a feedback loop in the design flow and kick-start a top-down, bottom-up or meet-in-the-middle design flow depending on the available blocks or IPs.

- **Top-down approach** – (also known as stepwise design) is a high-level analysis. In this approach the design process begins with a top-level view of the system. Sub-systems

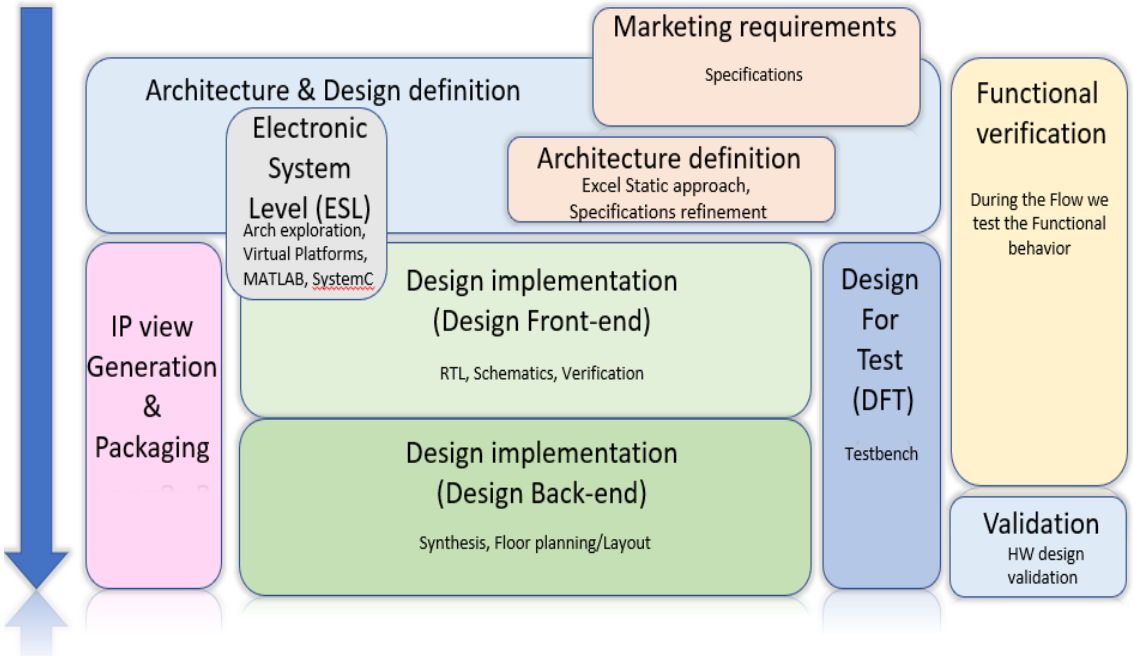


Figure 2.3: Hardware Design Flow

are first instantiated as "black boxes" and refined step by step until they are reduced to basic elements. It is very often associated with a V-model [20] design approach (detailed below).

- **Bottom-up approach** – is sort of the opposite of the top-down. It starts with the detailed definition of each individual sub-system which is then connected to build a more complex and complete design. It is often used for virtual prototyping based on instances of sub-system descriptions taken from IP models libraries and used to build an architecture.

- **Meet-in-the-middle flow approach** – is a successive refinement method interleaving between the top-down and the bottom-up approaches in order to converge on a physical solution. It starts with a top-down approach and uses the same refinement methodology.

A structured, top-down design and verification flow follows a serie of specific steps from product specification to design implementation. The application of the V-model approach (Figure 2.4) can easily match this methodology. It allows us to set a common way-of-working and maintain an easier and more efficient design verification process. There are standardized and well-structured design and verification workflows with common guidelines and rules and they are widely suitable for integrating top-down and V-model approaches.

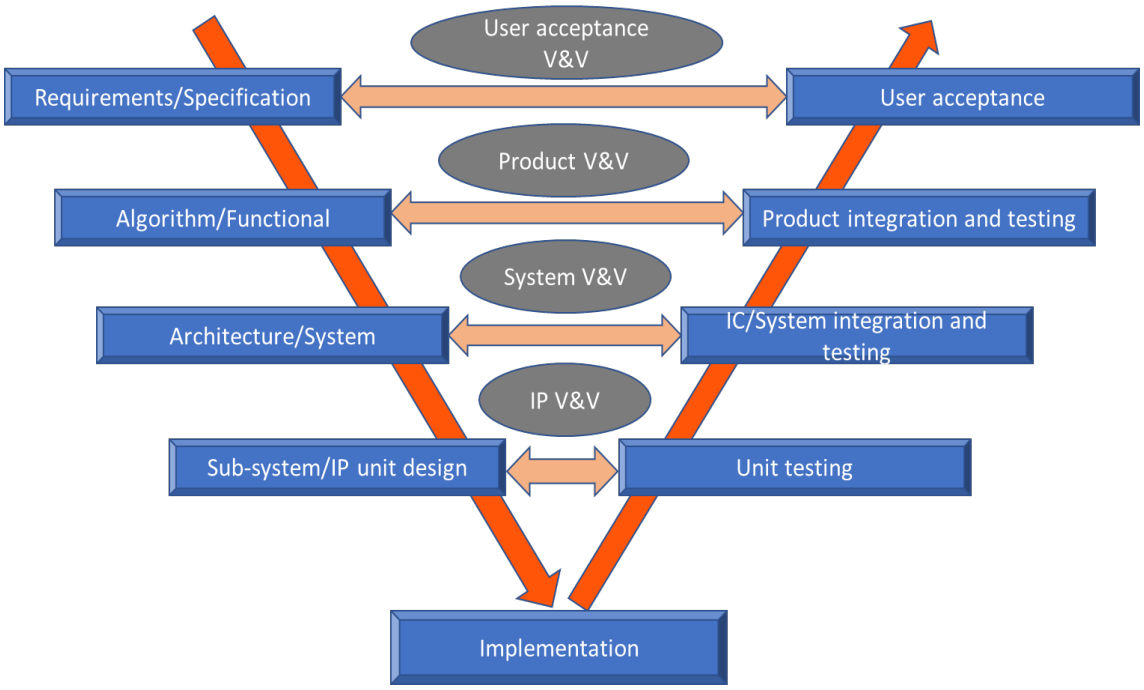


Figure 2.4: V-model design approach

The **left side** of the V-model describes the product specification and the architecture distribution. At this point, the product specification is translated into functions or/and algorithmic description, and then decomposed into a system architecture. After that, the architecture is further decomposed into sub-systems and blocks, which are sufficiently precise and ready for implementation. Each level of this part of the flow results in the creation of executable descriptions of the implementation at each level, allowing for continuous refinement of the product specification.

The **right side** of the V-model captures the verification and integration processes at each level and their specific test regressions. Testing is performed via the Verification & Validation (V&V) processes, where various tests are performed and their results are compared to the previously extracted executable descriptions in order to verify and validate whether the actual implementation meets the product specification.

Of course, this is a very simplified view of the whole process and, as mentioned earlier, each level has its own internal design flow, which can also be based on top-down, bottom-up or meet-in-the-middle methodologies.

By combining the information in Figure 2.3 and Figure 2.4, we can extract the following top-level design flow describing the main sequential and parallel tasks and steps (Figure 2.5). The relation and interaction between these tasks result in a refinement and reiteration of some of the steps (the tiny orange loop arrows).

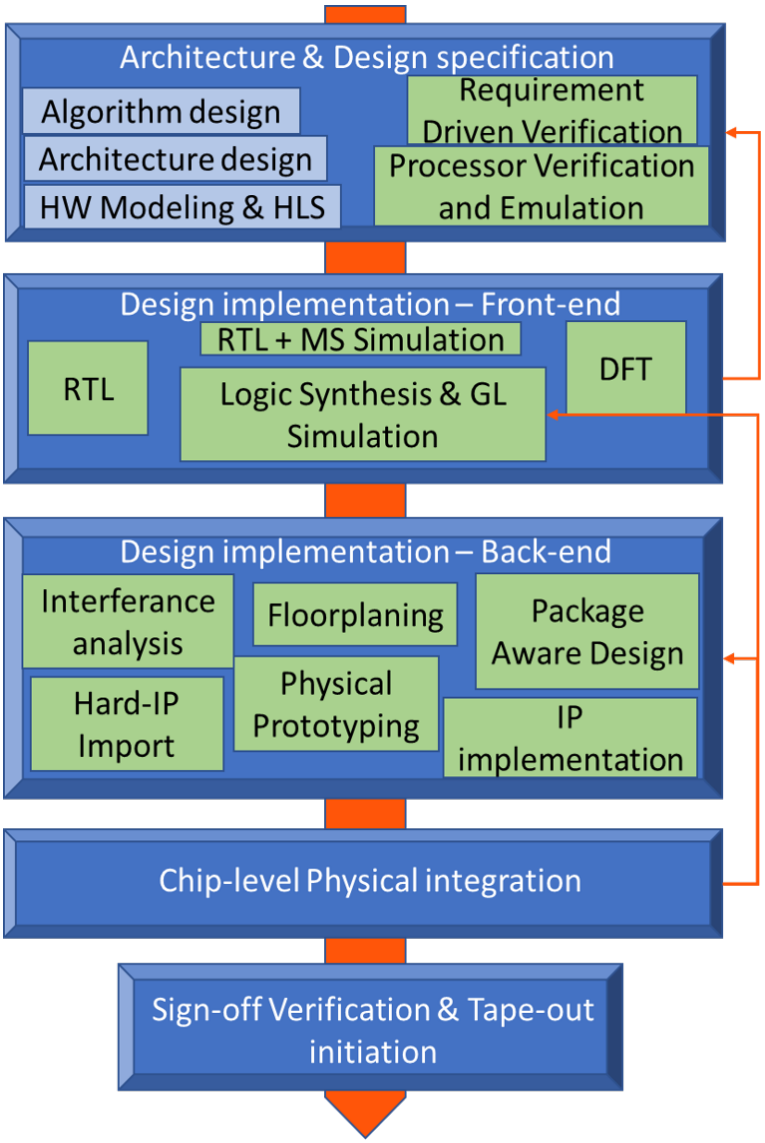


Figure 2.5: Top-down design flow

Architecture and Design specification

The flow begins with the **Requirements definition**, which leads to the specification of the device. At this level, architecture teams work on specifications and explore architectural options and available components. The intended application and key functionalities are captured in a high-level algorithm or functional model, typically described using approaches such as UML/SysML [21] or languages such as Matlab/Simulink [22] or C/C++ [23]. The objective of this model is to generate an executable specification that can be used and refined in later stages of the design. Then, these algorithms are mapped to a system architecture, where they are tested with different mapping options and evaluated in order to find the optimal solution.

The methodology used at this level is called **Electronic System Level**, or **ESL**, which refers to architectural design and exploration at a relatively high level of abstraction. This means that architecture team defines a **Virtual Prototype (VP)** of a system‘s architecture starting with only the key behavioral parts of the DUT and works with approximately timed

models. ESL can have different definitions, as the accuracy and level of abstraction of these behavioral models depends on the targeted IP, IP-reuse, the application, and even the designer‘s perspective.

A SoC combines analog, digital and software functionalities, so we need to use different domain-specific modeling techniques to describe each of these. For example, processor-based system models use **Instruction Set Simulators (ISS)** [24] for software execution, digital subsystems are based on **Transaction-level Models (TLM)** in SystemC [25], and analog/mixed-signal models are described with languages like SystemC-AMS [26]. Hardware models can be used in different ways, depending on their purpose. They can be used for architectural exploration and performance estimates, or they can be continuously refined until they become synthesizable and reusable for RTL code generation. This is possible with **High-Level Synthesis (HLS)** tools such as Verilator [27]. The design is entered in the high-level SystemC language and the HLS tool compiles it into RTL code, ensuring consistency in the design flow. The verification at this level is primarily requirements-driven and serves to maintain the traceability, test execution and back-annotation of the results. The initial functional verification is also covered.

Design Front-end and Design Back-end

The next layer refers to the RTL model definition made by the Front-end designers or resulting from HLS synthesis, and the testbench definitions made by the Design for Test (DFT) teams. The goal is to create a synthesizable RTL description and subject it to rigorous verification. At this level, logic designs are simulated at the RTL level and after logic synthesis (with RTL synthesis solutions like [28] and [29]) at the gate level using the netlist and technology logic library cells. Since the RTL model is pin and cycle accurate, it is more complex and much more time consuming than the ESL model, but can deliver very realistic performance metrics. **In the standard design flow, the power analysis and optimization task is initiated at this level.** During this stage of the flow, multiple verifications and checks, such as constraints, power intent, power implementation verifications and equivalence, and clock domains checks are performed to ensure that the timing and power intent descriptions are correct and complete. The SoC flow also needs to support mixed-abstraction level simulation allowing to co-simulate system level models with schematic, behavioral and/or RTL models.

The goal at back-end is to translate the gate-level netlist into a physical implementation [30] with respect to the floorplan. The floorplan describes the physical area, boundaries and pin positions of all digital components. The clock tree is inserted, and the logics are optimized in order to describe and complete the final gate-level netlist. At the end of this task, we have a fully placed and routed circuit stored in the design database and ready for electrical and physical verification.

Verification

The main objective of the verification process [31] is to determine whether a given product meets the requirements identified during its specification phase. It is an iterative process based on a verification plan (also called a Vplan) that should list all the functional and design features and requirements that need to be verified in the hardware. At each stage of the

development cycle, the product is tested to determine if it is consistent and complete and if the stage-specific requirements are met. This is the most critical and resource-intensive part of the flow. Without a complete design check, every mistake can cost the cancellation of the entire project. At each level of the complete flow, there are appropriate verification methodologies, generally based on two main typologies of verification techniques: Functional/Simulation-based techniques and Formal-based techniques.

- **Functional Verification** – includes different local verification workflows that are used depending on the level of accuracy required and the given simulation time constraints [32].

  - **Scenario generation** – enables the execution of the verification plan and ensures that the design implementation satisfies all elements of the verification plan derived from the design requirements.
  - **Event-based verification** – an event can be identified as a change in the input stimulus or lower level entities that can occur multiple times in a cycle. Event-based verification works by taking events and propagating them through the design until a stable condition is reached. These simulations take into account the temporal information in the design and are therefore very accurate, but time consuming. They are used for relatively small designs (e.g., at the block or IP level), because for large designs, the simulation time becomes completely unmanageable.
  - **Cycle-based verification** – evaluates the state of the logic and ports once per cycle. Accuracy is reduced compared to event-based verification and glitch detection is less effective. Time information from the design is not taken into account, but simulation time is reduced (by 5 to 100 times) compared to event-based verification. It is used for large simulation vectors (e.g. microprocessors, Application-Specific Integrated Circuits (ASICs), SoCs).
  - **Transaction-based verification** – allows for higher level of abstraction simulations compared to previous techniques and serves to increase productivity. It simplifies testbench creation and reuse, simulation debugging and coverage analysis. Signal-level protocol details are hidden by the Transaction Verification Model (TVM) at the RTL level or in hybrid simulations and neglected or abstracted at the ESL level (depending on the model-specific abstractions).
  - **Processors verification** - ensures that the implementation of a given core matches its architectural specification. In addition, it allows verification of multi-core systems interacting via a coherent shared memory interconnect. The input for this task is an ISS completed with a verification plan and the output generated is a coverage and closure report.
  - **Code coverage** – allows the quantification of the functional coverage obtained by the applied test suite. It can be used at each level of abstraction of the flow on the block, system or higher level. Using this analysis, we can also check which parts of the design are not sufficiently tested and therefore contain a higher probability of bugs.

  - **Emulation flow** – allows for much faster execution than HDL simulations and reduce time-to-market. It uses specially designed hardware and software systems to emulate the behavior of the target design. The latest generation hardware emulators (like [33] and [34]) provides considerably better hardware debugging than Field Programmable Gate Arrays (FPGAs) prototyping systems. For this purpose, we need the valid RTL files of the SoC and various external libraries such as verification IPs, memory models, and interface standards. Emulation is typically applied with longer and more complex runs in the final stages of the pre-silicon validation process. The resulting emulation builds are released and passed to various teams such as SoC verification, pre-silicon validation, early software, performance testing.

- **Formal Verification** – uses algorithms to mathematically verify that the design specification is preserved during implementation [35]. Formal verification is not benchmark-based, but property-based, which means that we check whether the design meets the property coverage of the design intent using the **Hardware Design Language (HDL)** code of the DUT. For this purpose, it does not use user-defined test vectors, but automatically analyzes all legal behaviors. With this technique, we analyze the space of possible behaviors of a design without executing all tests, but using mathematical models instead. It is applicable to several phases of the flow and allows us to discover inconsistencies and incompleteness in the specifications. Some of its subcategories are:

  - **Theorem proving** – consists of tests described as a theorem composed of a set of axioms and inference rules allowing the mathematical proof of this theorem. It can be used at all levels of abstraction and allows to compare the descriptions at these different levels.
  - **Model checking** – allows to check the properties of the design specifications through temporal logic formulas and has a relatively good automation. Its abstraction is user-defined. There is some risk that important properties are not covered and are not valid for a model, which may lead to undefined behaviors at a given stage.
  - **Equivalence checking** – as the name implies, it is a method to test and compare several design descriptions to check if they have the same behavior or to check that there are no inconsistencies between their behaviors (e.g. models with different abstraction levels).

Due to the high complexity and shorter development cycle of the design, the selection of the optimal verification process is a critical and challenging step that must be performed by design engineers. In general, the techniques presented earlier are combined and assembled into several verification methodologies with different advantages and disadvantages. Some of the widely adopted methodologies are:

- **Direct Test Method** – which is a straightforward method for evaluating test cases defined by the product specifications. It is typically used for digital designs built to perform specific well-defined tasks. This method allows us to test the behavior of the

product in the context in which it is intended to be used. It is a very good choice for precisely defined DUTs, but its coverage is low because it only tests essential and defined functionality. It can be used at every stage of the design cycle and is an essential part of ESL and RTL verification. As for the functional verification technique, it consists of a verification/simulation engine, a testbench (e.g. written in SystemC or SystemVerilog) and a DUT with well-defined states and applications. In this type of verification, it is essential to build an environment that is easy to reuse and update. Its structure can vary depending on the designer's intent and the level of abstraction. It can be built entirely with SystemC/TLM models, it can be a hybrid structure that allows analysis of SystemC/TLM and/or HDL models with command-stimuli files, or a complete RTL structure composed of HDL analyzers and emulators. For all these conditions, we use sequences of test cases generating commands and transactions that are processed by the different blocks in the architecture. During the simulation, different report files are generated, containing the messages issued by the DUT, the models and the emulators. The main advantage of this methodology is its modularity and its ease of use and reuse. Its main disadvantage is the low coverage of non-essential functionalities, which are not taken into account in the scenarios and which can hide critical bugs.

- **Coverage-Driven verification (CDV)** – which is based on the repetitive execution of the verification environment using different tests and allowing to confirm if the coverage objectives are reached. These tests can be aimed at verifying different constraints and/or verifying behavior with random stimuli generation. Each test is usually scored according to several criteria (functional coverage, bug detection, simulation time, etc.). The results of the coverage tests are merged, and the process is iterated until it reaches the required coverage grade. The main advantage of this methodology is the high level of automation (randomly generated stimulus) that allows to speed up the verification process. However, it requires the creation and debugging of a coverage model, which is time consuming.

- **Metric-Driven verification (MDV)** – which allows the verification of large digital designs. The huge state space of modern SoCs does not allow to simulate all possible transitions and state combinations. MDV allows defining guidelines to achieve good functional coverage. It is guided by a functional specification rather than a design implementation. This specification is decomposed into a hierarchy of smaller features and then correlated to a subset of the design space to generate a much smaller coverage space. At this level, a technique, called Constraint Random Testing, is applied to run a large number of simulations with random perturbations to optimize the process and increase coverage. Today, this methodology is widely adopted because it improves the quality, predictability and productivity of the entire verification process. It allows for a high degree of automation, through scripting, and enables the achievement of objectives and milestone reports in the verification plan.

As we can clearly notice, these methodologies are complementary and can be easily combined. These techniques and methodologies can be enhanced with assertion-based coverage, error injection, IP view generation, and other additional techniques. Environments such as UVM [36] are incredibly useful for creating reusable testbenches and new verification environments. The introduction of highly configurable Verification Intellectual Property (VIP)

modules allows for horizontal (between projects) and vertical (block to system level) reuse of these environments. However, these environments are out of the scope of this study, so we will not go into detail about them here.

Conclusion on the design flow overview

At the ESL level, we can use almost all these verification methods. Since this is not the main focus of this thesis, but it is perhaps the most important part of the design cycle (consuming almost 70% of the project resources for ASICs, reusable IPs and SoCs), we will at least mention some of the tests that have been performed. We work with SystemC/TLM models, so we favor verification based on **functional simulation**. All three methodologies are applied in our study in one way or another. The functional model is verified with the direct testing method combined with the Metric-Driven verification flow and our power-aware model includes assertion-based coverage. Error injection techniques allowed us to enforce the assertion mechanisms and to test and eliminate undefined behaviors. It is important to mention here that due to the increased complexity of the design, these techniques and methodologies are not only applied to the functional model, but they are also used to ensure the security and safety aspects and to verify the performance and power objectives.

This traditional design flow is becoming less suitable for complex systems, due to the large number of blocks and signals they contain. Their simulations and applications become time consuming and the verification process becomes less efficient. A major hardware design problem discovered at RTL stage or later can lead to significant production delays and unforeseen expenses. Furthermore, until recently, software testing and verification only began when a low-level RTL model existed. Fortunately, many efforts have been made over the past decade and the development of high-level system models and VPs [37] has enabled the early co-simulation of software and hardware. However, in its current form, the V-model alone is not flexible enough. One important problem is still relevant, and it is related to the power consumption. The only early stage power analysis performed at ESL level is a static one and it is not representative for complex use cases (only for worst/best case scenarios). Given the increasing importance of power consumption, it is totally unacceptable to initiate its advanced analysis so late in the design.

The methodology for power investigation and estimation, studied in this thesis, can be efficiently placed in the beginning of the design flow and can improve design predictability, early metrics extraction, and architecture exploration.

2.2.2 The weight of power in the Power/Performance/Area (PPA) trade-off

Nowadays, the power consumption is a major design constraint that can cause a number of flaws due to a limited power budget, glitches, or suboptimal power intentions taken too late in the design flow. The newly released Wilson Research Group and Mentor Functional Verification Study [38] shows that power consumption is still the third major flaw contributing to ASIC design respin, despite the positive trend compared to previous years (Figure 2.6).

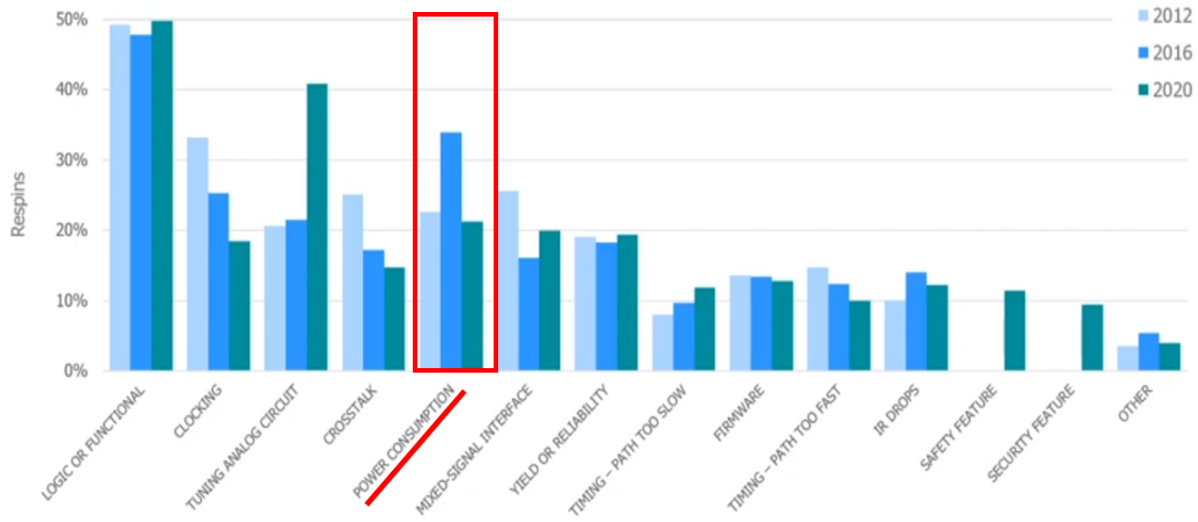


Figure 2.6: Type of ASIC Flaws contributing to Respin

This positive trend stems from the significant involvement of IC engineers in the design and development of energy-efficient systems and the improved understanding and application of existing power reduction technics [39]. Using the same Wilson Research Group studies [38] [40], we can compare the percentage of power-aware design projects in the two most recent releases and conclude on the current involvement in the low-power design. From 2014 to 2018, there is no major changes in this percentage, but only from 2018 to 2020 we have a 10% increase in favor of power-aware designs (Figure 2.7).

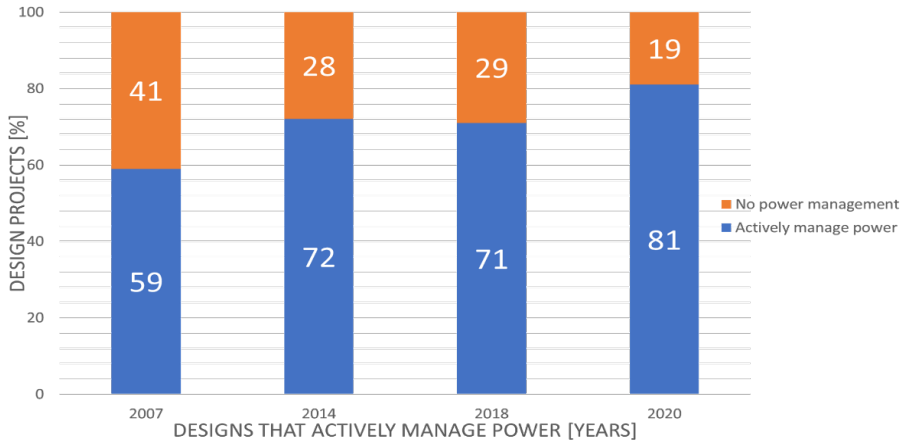


Figure 2.7: Designs that actively manage power

Unfortunately, this effort is not sufficient, because the heterogeneous designs of ASIC and SoC are too complex and it is difficult to select and describe an optimal power intent without penalizing the performance.

2.2.3 Power estimation predictability issues at architecture specification stage

Early stage power consumption estimates applied in the standard SoC design flow are limited to worst-case and best-case power consumptions and are not suitable for complex use cases. In general, the idea of this analytical approach [41] is to use static technique based on equations and Excel tables or automated tools that may work well for simple use cases but are hardly applicable to complex ones. The activity factor is estimated for dynamic energy consumption using available information and leakage is approximated using some area estimates. The basis for the calculations is data from IP data sheets and IP suppliers. Furthermore, it does not take into account the actual functional top-level platform, but the statically calculated sum of the energy consumption of all IPs in the design. We can deduce some simple data about the maximum and minimum power consumption at different temperatures and the simplest IP states. With this approach, it is not possible to estimate the power consumption of given test cases or the dynamic activity/behavior of the design (functional nor power). Theoretically, we can start thinking about the implementation of the power management strategy, but it is very difficult to optimize it and it is impossible to observe it. Figure 2.8 illustrate an example of input/output information used for this kind of power estimation. Due to confidentiality reasons, only the titles are presented, and the values are masked. We use information such as IP area estimations/data, clock frequency, supply voltage, number of cores, transistor technology related data and others.

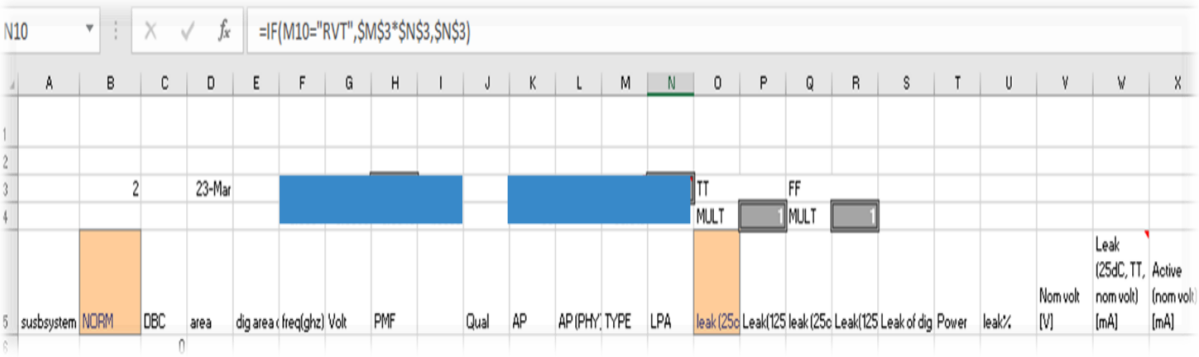


Figure 2.8: Power estimation - Static approach (Excel)

The approach studied in this thesis aims at improving the IP qualification and architecture definition steps in the design of complex heterogeneous systems. This is achieved by adding power modeling capabilities and allowing their reuse throughout the flow. Metrics are not calculated manually for different states, but rather they are calculated and visualized dynamically during the simulation. To this end, we reuse some of the inputs used in the first static approach and pass them to the automatic power estimation tool. In addition, we associate a power management strategy with the simulation, and observe its influence on power consumption and functional behavior. The activity, clock and power metrics extracted for the whole simulation are plotted and multiple reports are generated (assertion violations, errors, inconsistency problems between power/functional models etc.). Our methodology is implemented at the very beginning of the design flow, where system modeling is initiated and ESL investigations are performed. In order to clearly express our main motivations and the added value of this approach, we consider necessary to give a more in-depth overview

of the system modeling design flow and the most significant reasons for the integration of energy management/estimation approaches at the beginning of the design specification and design. This is done in the next section.

2.3 Shift-left urgency: Kick-off the design flow with high-level power/performance investigations

2.3.1 System modeling

System modeling is the process of specifying, structuring, and designing a given IC. In this phase, we specify the functionality/behavior, concept, and architecture. Following the top-down design process, we start the modeling process with the most abstract concept model, followed by a functional/behavioral model. The result is an architecture model describing both the structural decomposition and the behavior of the system.

System modeling flow and tools

The starting points for system modeling are the market and application requirements for the product, where the functional and non-functional requirements and its application are found. The system model consists of three main modeling tasks and each task delivers an executable specification or requirements description (Figure 2.9). Thanks to the strong link between these deliverables and the system model, the accuracy and design are continuously verified, compared, and validated against the requirements. Executable descriptions are also stored in a design database and serve as reference models during the flow.

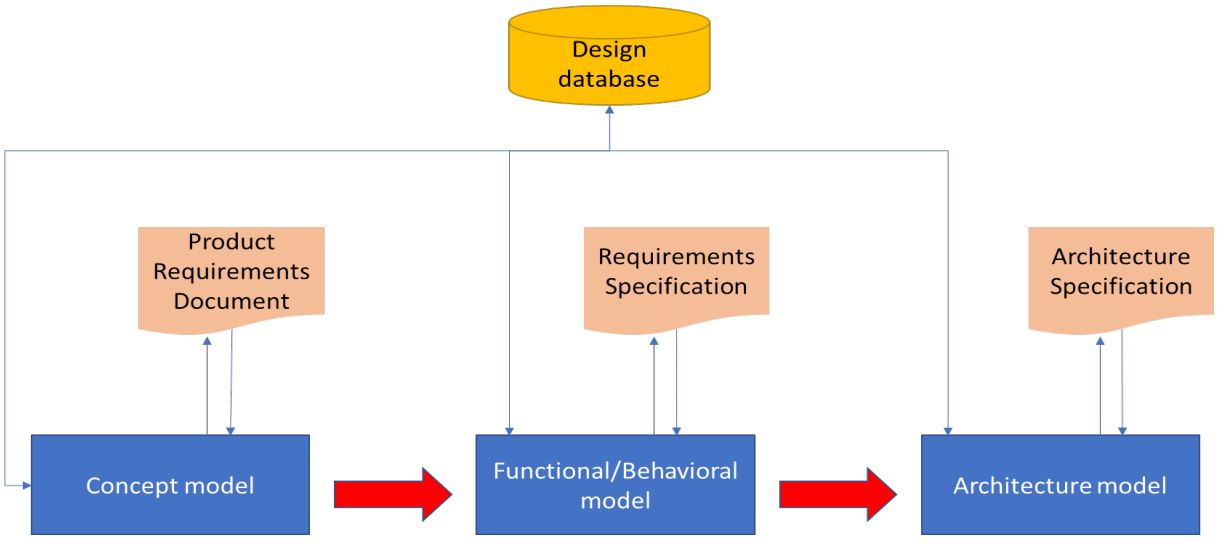


Figure 2.9: System modeling flow

- **Concept model task** – this is the first task, where we capture the customer‘s view of the product. It should contain the definition of the intended functionality, programmable features and targeted/supported interfaces and protocols. The model captures the abstract behavior and structure, and it is represented by a high-level block diagram (for example, using UML/SysML).

- **Functional/Behavioral model** – in this task, we capture the behavior of the system. This is done by modeling the algorithms and functionalities using languages or/and tools, such as SystemC/C++ or Platform Architect [42].

- **Architecture model** – at this stage, we capture the architectural decomposition of the system. The goal is to conceptualize the structure with bus interfaces, interconnects, registers, and certain behaviors. At first, the intrinsic behavior of each IP is not modeled. When the concept is clear, we start refining the IP's description by adding intrinsic structures and system level models describing the functionalities. For this purpose, we create a framework combining the functional/behavioral SystemC models, TLM interfaces and eventually the available more refined RTL-described blocks.

At the end of this stage of the main SoC design flow, we should theoretically be able to synthesize the system model using HLS. Unfortunately, there is still a lack of good translation, due to poor optimization. A lot of work is put in place in order to improve the standardized synthesizable subset of SystemC and to increase the optimization capabilities of HLS tools [43]. Due to these limitations, these system models are typically used for functional validation and reference models in order to enhance the RTL designing and verification. Since we are using this kind of framework which is based on SystemC, TLM2.0 and VPs, it is important to understand their semantics. Thus, the next few subsections are devoted to their presentation.

Overview of SystemC

1. ***Introduction***

   SystemC is an open source event-driven system design language. Specifically, it is a library built with the use of C++ classes that allows the hardware and software design and verification using high-level language syntax.

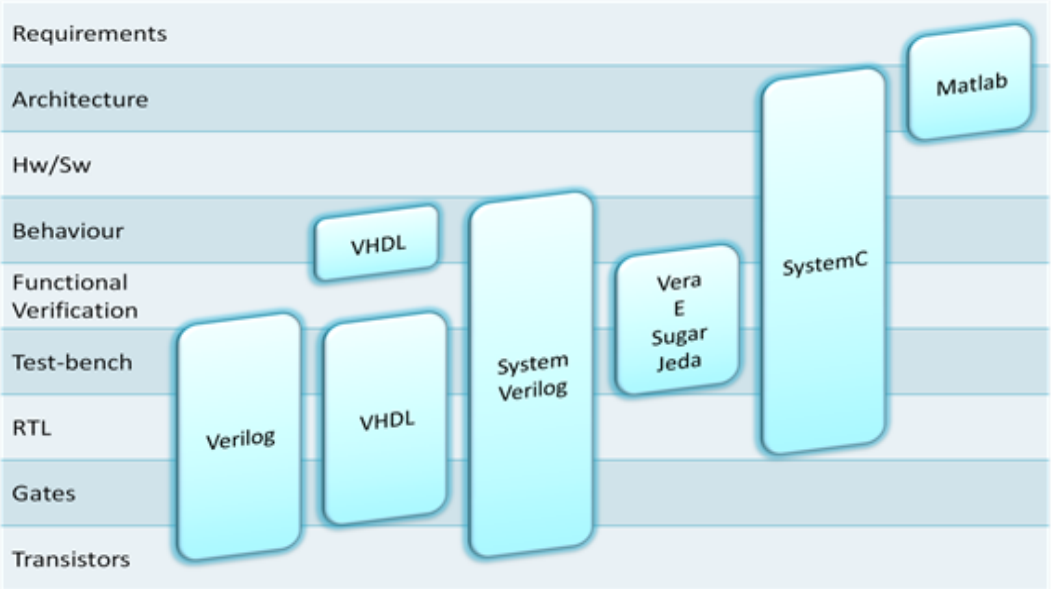


Figure 2.10: Languages comparison [2]

SystemC deliberately mimics the hardware description languages VHDL and Verilog but is better suited for system level modeling, system‘s partitioning and architectural exploration. It is a ratified IEEE 1666 standard defined by the Open SystemC Initiative (OSCI) which later merged with the Accellera Systems initiative [44]. The main purpose of this language is to offer the possibility to model hardware/software blocks at various abstraction levels from RTL to transactional (Figure 2.10).

The most remarkable features and capabilities of SystemC are:

- **Hierarchical decomposition** – It provides processes and modules allowing to implement complex hierarchical structures. Processes are C++ functions serving to support the execution and the uncoupling of small concurrent/parallel pieces of code which describe the behavior/logic of the design. They are generally encapsulated in modules that serves to better define large systems granularity and to separate the large designs into smaller pieces. Modules are generally composed of processes, interfaces, events, member data/functions and instances of other modules.
- **Structural connectivity** – it provides communication mechanisms like interfaces, ports and channels enabling the communication between different modules.
- **Scheduling and synchronization mechanisms** - SystemC library provides synchronization mechanisms like events and sensitivity lists for processes scheduling.
- **Concurrency** - execution of several processes in parallel. A specific simulation kernel is put in place to ensure that parallel activities are modeled correctly. SystemC is a mono-threaded language (sequential execution) but emulates the parallelism that takes place at the hardware level. For this purpose, it uses a scheduler based on discrete event simulator principle [45]. This notion of concurrency facilitates the parallel modeling of hardware and software.
- **Bit accuracy** - SystemC provides all C++ variable types, but it is also enhanced with other types close to these used in an RTL description. This way we can execute bit level operations. For example, we have the types sc_logic, sc_bv, sc_uint<size>.
- **Simulation time progress** - the notion of time present in SystemC allows to model hardware components and communication mechanisms with their latencies and to manage the execution order of methods. Finally, we can derive metrics related to power consumption and performance by using other additional libraries.

As previously mentioned, the discrete-event simulator of SystemC supports timed simulations. This feature is based on a specific SystemC guest clock. This clock is different from the host system wall-clock and it measures the simulation time from the inside of the simulated system. As it is based on discrete events, it is not continuously evolving, and it is known only for some instants of the continuous system wall-clock.

2. ***Execution***

Underneath the SystemC public shell defining all classes, functions, and macros is the core functionality implementation private kernel. The execution of SystemC application consists of two phases: elaboration and simulation (illustrated in Figure 2.11).

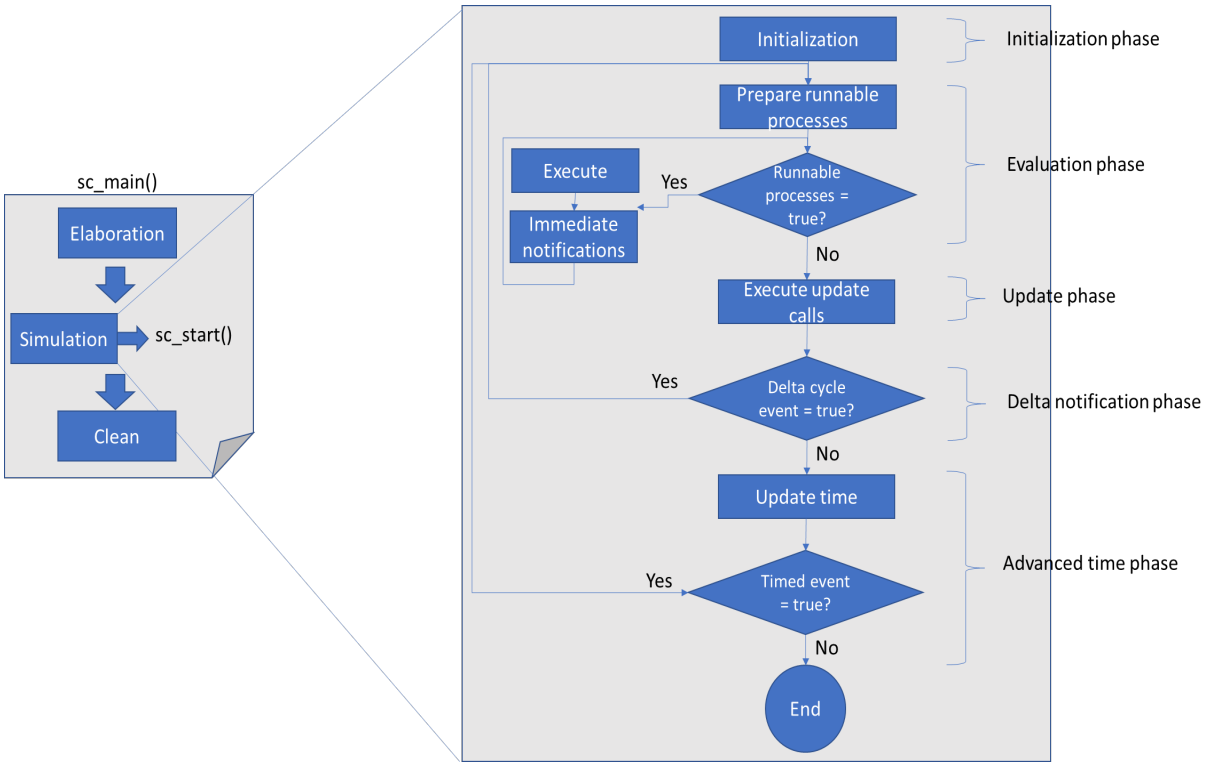


Figure 2.11: SystemC execution (simulation kernel)

The **elaboration phase** builds the modules hierarchy and prepare the execution with some kernel support functions. At this level, the design structure is created and fixed (cannot be modified during simulation), all ports and channels are bound, and all static processes are declared.

The **simulation phase** is closely related to the kernel‘s scheduler responsible to plan and choose a coherent processes execution. It can be seen as a closed loop with several steps. The first step is the initialization step, where the first runnable processes are prepared for simulation and the update and delta notification steps are executed once before passing to the evaluation step. During the evaluation step a runnable process is selected and executed. It can result in another immediate notification creating new ready to run process. This loop is repeated until an update call is made or until all runnable processes are finished. An update call brings the simulation to the update phase step, where any pending call to update function are executed. If there are no remaining pending calls, the simulation moves to the delta notification step. There, all processes instances sensitive to the delta notification event are added in the runnable processes set. Consequently, if the set is not empty, the scheduler returns to the evaluation phase and restarts the loop. If it is empty, the scheduler checks for timed notifications, advances the simulation time to the time of the earliest timed notification and adds the given event processes to the runnable processes set and executes. Finally, if there are no more timed notifications, the simulation is finished.

3. ***Processes***

   Processes play a key role in SystemC. As we previously mentioned, they allow us to describe concurrent pieces of code and to simplify and structure our design. There are several ways to describe and classify them. They can be automatically activated thanks to events and sensitivity lists, but they also can be explicitly activated as normal C++ methods. Processes can be statically declared at elaboration time or dynamically declared at simulation time. Generally, for the ***static declaration*** we use three unspawned process macros *SC_METHOD*, *SC_THREAD* providing two different execution models.

   - **SC_METHOD** – is a method that can be run multiple times during simulation but cannot be suspended during execution. The *SC_METHOD* is activated thanks to its sensitivity list.
   - **SC_THREAD** – is a versatile thread which is automatically executed only once in the beginning of the simulation. One of the threads characteristics is that they can be halted multiple times at any point of execution (using "wait()" function), and this can be very useful during simulation. Thus, enrobing the thread in an infinite loop is a great trick to prevent the thread reaching its sequential end, and allowing us to have access to haltable processes during the entire simulation.

   The ***dynamic declaration*** is done using the spawned process *sc_spawn()*. It allows us to declare processes during the simulation time. This can be useful when the design needs to respond to some conditions which are not known at the elaboration time.

Overview of TLM2.0

OSCI TLM2.0 standard [46] is an overlay of SystemC that allows us to simulate the communication between the different modules/components in an abstract manner. HDL codes require sending a large number of signals and connections between different inputs/outputs, whereas with TLM2.0 a large part of these connections is abstracted and replaced by function calls. TLM is useful to hide complexity in more complex heterogeneous systems. Thus, it increases the level of abstraction and it can speed up the simulation by a factor of x1000 (or more) compared to a classical RTL simulation. TLM2.0 is often used for architecture exploration, building virtual simulation platforms and performance analysis.

Its predecessor, TLM-1, defined a set of core interfaces for transporting transactions, but it was not well suited for the principles of today's SoC design methodologies, which are mainly based on blocks assembling, 3rd party IP reuse and the choice between various communication protocols. Moreover, simulations were still not fast enough, due to the realistic model of data transfer. For this purpose, the TLM2.0 standard was developed in a way to increase models' interoperability and to define a common rule for protocols modeling. It also replaced the data type used for inter-modules communication from data copy to a reference to the data value. In this way, the simulation time is largely decreased. There are two main sections in the TLM2.0 description allowing to increase the model's interoperability and to give larger use case testing possibilities.

1. ***Interfaces and communication tools***

   The first section defines a set of Application Programming Interfaces (APIs) fixing a common transaction transport methods and formats that ensure the interoperability. The master/slave (also called initiator/target) semantics are followed and a common data packet structures, called payloads, are used. These payloads are extensible and/or replaceable, but their base provided with the standard is called **generic payload**. The generic payload is closely related to the TLM2.0 base protocol defining the rules of communication to ensure interoperability when using the generic payload. The generic payload is designed to model memory-mapped buses. It includes the most important attributes for communication between modules. These are the attributes found in most protocols: command, address, data, byte enables, simple word transfers, bursts, streaming and response status. The generic payload also allows the modeling of other types of protocols, thanks to an extension mechanism with which it is possible to add all attributes necessary to the specific protocol. The extensions mechanism does not impose any restrictions on the number and type of extensions added. In fact, the extensions are not stored in the generic payload, but in a memory external to the transaction. The generic payload contains an array with pointers to the extensions associated with the transaction in question. This allows to have fewer heavy transactions, because we do not send the data, but only a pointer to this data. In addition, you can choose and retrieve only the extensions that the transaction needs. The addition of extensions extends the basic protocol and makes it possible to model a specific protocol. However, instantiating a large number of extensions enlarges the table and slows down the simulation. There are other ways to model specific protocols, but we will not go into the details of these methods.

   An initiator module is the one creating new transactions and sending them to the target (or intermediate module) by calling a transport interface on its forward path (ex. Traffic generator). The target module is the one accepting the transactions, and in some cases, returning an acknowledgement on backward path (ex. Memory module). These paths are created using the so-called **sockets**. The socket is the combination of a port (*sc_port*) and an export (*sc_export*). The *sc_port* handles outgoing transactions, and the *sc_export* handles incoming transactions. Thus, there are two types of sockets - **Initiator socket** and **Target socket**. The connections between the *sc_port* of the Initiator and the *sc_export* of the Target are called forward path and the connections between the *sc_export* of the Initiator and the *sc_port* of the Target are called backward path (Figure 2.12).

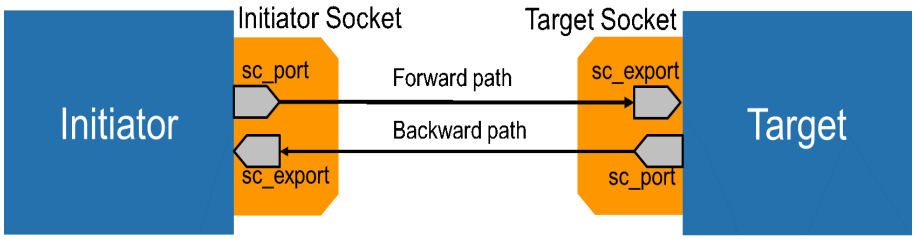


Figure 2.12: Sockets and connections

Initiator sockets provide the interface method calls on the forward path, and an export for interface method calls on the backward path. Target sockets provide the opposite. There are also more elaborate sockets, which offer additional features. These are called **convenience sockets**. Each type of convenience socket implements additional functionality to make the component models easier to write. They derive from the classes *tlm_initiator_socket and tlm_target_socket* which are the basic sockets. For our application, we are interested in two types:

- *simple_initiator_socket / simple_target_socket*
- *simple_initiator_socket_tagged / simple_target_socket_tagged*

Simple sockets (including *simple_socket_tagged*) are so called because they are intended to be simple to use. They are derived from the interoperability layer sockets *tlm_initiator_socket and tlm_target_socket* and can therefore be directly linked to sockets of those types. Instead of having to bind a socket to an object that implements the corresponding interface, each *simple_socket* provides methods that register callback methods, called register callbacks. These callbacks are in turn called whenever an incoming interface method call arrives. Callback methods can be registered for each of the interfaces supported by the socket.

A core feature in TLM2.0 are the transport mechanisms, and more precisely, the transport interfaces. The standard sockets provide all transport interfaces and enable their parallel execution. There is a **blocking transport interface** transferring fast untimed transactions with higher level of abstraction. It is largely used for software modeling in combination with **Loosely-timed (LT)** coding style (detailed later in this section). There is also a **non-blocking transport interface** giving more control on the transaction and its timing. This is done by separating the interface in two parts – forward and backward interfaces. These interfaces enable us to cut the transactions into multiple phases and associate a timing points to each part of the transfer. It is generally used for hardware modeling and architectural exploration in combination with the **Approximately-timed (AT)** coding style (detailed later in this section). There are other interfaces, such as DMI interface and Debug interface, but they are out of the scope of this thesis.

2. ***Coding style***

   In TLM2.0 there are two predefined coding styles. These styles are not mandatory, but they bring simplicity and clarity to the code writing. They are **Loosely-timed (LT)** and **Approximately-timed (AT)**.

   - **Loosely-timed** coding style is often associated with the blocking transport interface. This interface allows only two synchronization points to be associated with each transaction, corresponding to the call and return of the blocking transport function. In the case of the basic protocol, the first time point marks the beginning of the request and the second marks the beginning of the response. These two timing points can occur at the same time in the simulation or at different times. This coding style is suitable for the use case of software development using a SoC virtual platform model, where the software content may include one

or more operating systems. We will not go into detail about this coding style, because we do not use it in our model.

- **Approximately-timed** is supported by the non-blocking transport interface, which is suitable for use cases in architectural exploration and performance analysis. The non-blocking transport interface allows synchronization between transmission phases throughout the lifetime of a transaction. For approximate modeling, a transaction is decomposed into multiple phases, with an explicit timing point marking the transition between phases. In the case of the base protocol, there are exactly four timing points marking the start and end of the request, and the start and end of the response. Specific protocols may add additional synchronization points, which may result in the loss of direct compatibility with the Generic payload. Each process typically executes in a lock phase with the SystemC scheduler. Process interactions are annotated with specific delays that are associated with different types of transactions (read or write).

In our study we use the non-blocking transport interface and the Approximately-timed coding style. An approximately-timed transaction consists of 4 phases:

- **BEGIN_REQ** and **END_REQ (Start/End of the request)**
- **BEGIN_RESPONSE** and **END_RESP (Start/End of the response)**

A timing annotation can delay the timing point for a phase transition. This is done by using a *sc_time* argument, which is sent with the non-blocking transport interface and is called delay. Since the communication is bidirectional, with a forward path and a backward path, we need two non-blocking transport methods - *nb_transport_fw* and *nb_transport_bw*. The execution of these methods is done by function calls.

Note: For example, the initiator creates the payload and passes a reference to this payload on the forward path. The initiator calls nb_transport_fw interface of the initiator socket, which calls the method registered as callback in the target. Similarly, the target answers using the nb_transport_bw interface of the target socket.

On reception of each phase, the calling module responds with a return signal which validates the correct reception. This signal is called return value and can take three values:

- **TLM_ACCEPTED** - this value indicates that the calling module does not change the phase, time or transaction object during the call. The called module can neglect the values of the nb_transport arguments that were returned, because they are unchanged. Thus, the return path can be considered unused.
- **TLM_UPDATED** - The calling module changes one or more arguments in the transaction object. This means that the return path is used, and the calling module moves to the next state in the protocol state machine.
- **TLM_COMPLETED** - The calling module modifies the transaction object and the transaction is completed. There are no more transfers related to this object and the called module can test the response status to make sure that the transaction has been completed.

3. ***Memory management***

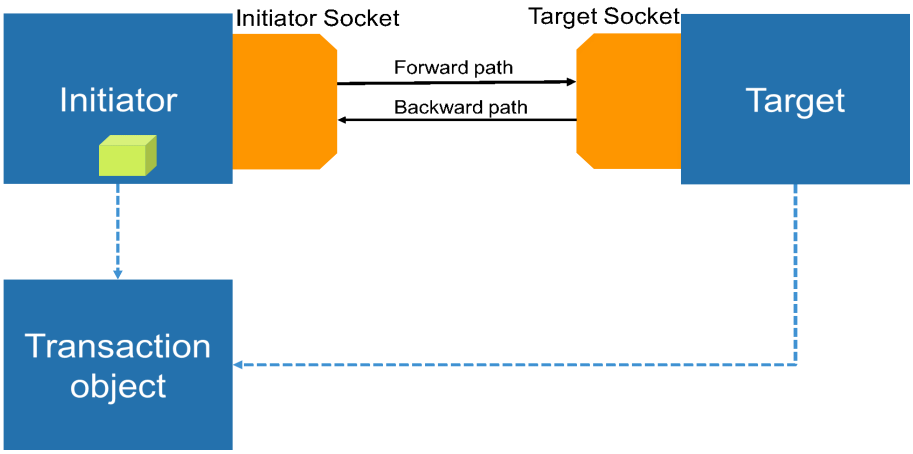


Figure 2.13: Memory management

In Figure 2.13 we notice that the main blocks point to a *Transaction object*. This is also a feature of TLM2.0 that helps to optimize the code. The transaction object is located in a storage space called *Transaction pool* which is part of a TLM2.0 tool called *generic payload Memory Manager*. The Memory Manager is a user-defined class that inherits from the *tlm_mm_interface* and *tlm_generic_payload* classes. It contains all transactions and takes care of, at least, their allocation and release. The memory manager provides an *acquire()* method to allocate a generic payload transaction object from the transaction pool and implements a *free()* method to return a transaction object to the same pool.

The initiator is responsible for setting the attributes of the object that was retrieved from the existing storage (transaction pool). He must not delete this storage before the end of the transaction lifetime. Each block calls the *acquire()* method when the transaction is received and *release()* when it is released. These two methods increment and decrement the reference counter. The *free()* method is called by the *release()* method of the *tlm_generic_payload* class when the reference count of a transaction object reaches 0.The construction and destruction of objects of type *tlm_generic_payload* are costly in terms of execution time of the simulation, because of the implementation of the array of extensions contained in the generic payload, hence the Memory Manager.

To summarize and conclude on this tool, we can say that the memory manager helps us to manage the allocation of transactions, in order not to build and destroy several objects of type *tlm_generic_payload*. To do this, we use the transaction pool containing a limited number of objects, which we reuse for each transaction.

The same principle is used for each block. In Figure 2.14 we can see an example of how a transaction is executed in a simple model. The Traffic Generator creates the Transaction pool, retrieves the pointer to the transaction object, associates the address, order, size and all other attributes with it and sends the *BEGIN_REQ* request to the next block on the forward path. On reception, the following block accepts the transaction (by a *TLM_ACCEPTED* signal sent on the return path of the forward path) and increments the reference counter (by the *acquire()* method). Then it performs its processing on

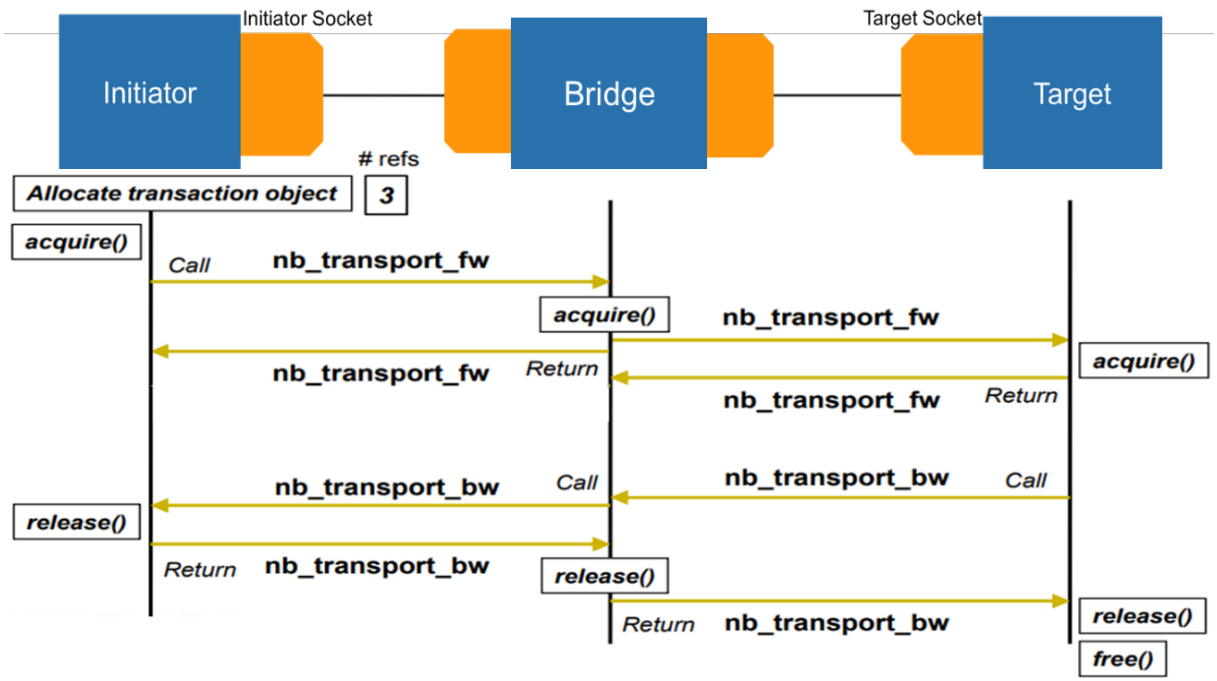


Figure 2.14: Memory manager and non-blocking interfaces

the transaction, returns *END_REQ* to the previous block and *BEGIN_REQ* to the next block. The Target receives this request, accepts it and responds on the backward path with an *END_REQ* to inform the previous module that the request has been accepted. Once the *END_REQ* phase has arrived and been accepted by the Initiator (Traffic Generator), the Target (Memory) sends the next phase *BEGIN_RESP*. During this phase the request to write or read to the memory is executed and when the execution is complete, the Initiator sends the new phase *END_RESP* which ends the transaction. On receiving *END_RESP*, each block releases the object using the *release()* method.

Thanks to these SystemC and TLM2.0 functionalities and semantics, we are able to create early timing approximative Virtual Prototypes of the design.

Virtual Platforms (VPs) using SystemC/TLM

SystemC/TLM-based Virtual Platform‘s (VP) purpose is to model and simulate the SoC behavior at a sufficiently high level of abstraction and high execution speed. However, the hardware model must be accurate enough to enable the parallel development and testing of software. VPs aim to improve the design flow and start the software development in parallel with the hardware. They can be used for multiple purposes like early software development, as a golden reference for hardware verification, design functional verification, architecture exploration, soft IP vendors early models etc. The major advantages of VPs are their controllability (being a software program), visibility (thanks to simple observation and analysis tools), portability (can be easily shared across distributed teams) and determinism (reproducing fixed test cases). A VP contains multiple hardware blocks, that can be modeled at different abstraction levels. It enables the co-simulation between functional, TLM and RTL models. In this way, we can directly include reused IPs, with already existent RTL, and simulate them with

our new IP models, at different stages of their development. However, the simulation speed is kind of proportional to the level of abstraction. The RTL models slow down the simulation and it is better to use accurate SystemC and TLM2.0 models instead. Many industrial solutions, such as [42] [47] [48] and some academic solutions, like [49] [50] present frameworks or/and several emulators allowing to accelerate the process of prototyping and to use libraries of already tested models.

2.3.2 The necessity of early stage power and performance dynamic estimation

With the increasing use of battery-driven devices, the importance of power consumption increases drastically. As mentioned earlier, in the current flow the device power modeling is initiated at RTL level, which is already an advanced stage of the flow. Therefore, it is risky to "neglect" power in its preceding steps. At the RTL level, in order to reduce complexity, power modeling is applied individually to each IP and can be effective at this level. However, the complexity and the fact to apply IP-level power optimization leads to extremely high simulation time for the top-level and difficult verification. We believe that there is a lot of room for simplification and extension in the study of power consumption during the design flow. Thus, following the shift-left tendency in its evolution, we have studied a few more hypothetical steps by raising the level of abstraction. The biggest gap here comes at the very beginning of the flow. In the early stages of architecture definition, we have only a rough idea of what energy consumption can be. As explained page 21, at this stage, architects are swapping between calculations, specifications, manufacturers' documentation, tables, and Excel sheets. Hence, there are not many tools or methodologies to help architects define an optimal architecture and power intent. The lack of automation in the architectural exploration process is one of the well-known flaws in the design flow. At this point, the power consumption can be optimized through careful components and power management strategy selection, but this is not obvious without **Simulation-Based Dynamic Analysis (SBDA)**, verifications and exploration.

2.3.3 Power-aware IP-reuse and Platform-Based Design (PBD) extension to ESL VPs

Over the years many well-established design methodologies and tools have been adopted to simplify the advanced designs handling and reduce expenses. All these methodologies are based on the idea of raising the level of design abstraction and automating complex and time-consuming workflow processes. A well-known and widely used methodology to reduce the cost, effort and time-to-market of designs is the IP-reuse or Block-Based Design reuse (BBD-reuse). Today, it is unthinkable to write a complete RTL HW design from scratch for complex chips [51]. This approach completely revolutionized the SoC design flow and pushed us to a block-based 'lego' approach, combining soft, firm and hard macros. However, this introduced other new challenges and constraints [52]. Following the same principles, the Platform-based design added

another layer of abstraction and made possible the reuse of entire platforms composed of multiple IPs. Since developing designs or platforms for reuse takes much more time and effort than developing standard non-reusable designs, it may become more expensive to make them than to purchase them. Therefore, the ”plug and play” design approaches with high-quality third-party IPs is often preferred, but the development of reusable internal macros are inevitable for final product differentiation.

In this line of though, VPs and languages like SystemC and TLM2.0 allowed us to extend these methodologies to the system level and to simplify the functional modeling and performance analysis. Thanks to this approach we found a way to efficiently evaluate hardware components early in the design process and extract some important metrics. However, the early power analysis still suffers from the lack of a good methodology for dynamically evaluating the impact of different use-cases on power consumption and a methodology for power intent reuse. The early power analysis can be extremely beneficial for the entire design flow because it can help us avoid errors and unnecessary design respins, present early metrics to our customers and serve as a golden reference for the rest of the flow (details on benefits from early power analysis will be presented in the next chapter). In the same time, the insertion of portable and reusable power description to each IP (creating power-aware IP) can increase our productivity and decrease the time-to-market period.

Chapter 3

Put it all together: Early Power/Performance estimations – State of the Art

3.1 Design Space Exploration (DSE) - challenges and benefits

Design space exploration (DSE) is a term referring to the exploration of different design options and their comparison based on design specific parameters of interest. These parameters depend on the systems that we want to model, but the most common ones are the **Power, Performance, Area and Cost (PPAC)**.

The sum of the challenges presented in Chapter 2 has led to many researches on the industrialization of higher-level SystemC/TLM DSE and more precisely on power and performance estimation methodologies. At system architecture definition one main task is to balance between PPAC of the chip by investigating different architecture options and possible bottlenecks. Adding abstraction layers to the design flow is an already proven and widely adopted technique for this. In fact, modern digital design flow is entirely based on an approach combining abstraction, decomposition, and refinement techniques, masking the complexity of the schematic connections of electronic components. By increasing the level of abstraction, implementation details can be ignored with justified and limited timing and accuracy loss. Omitting these implementation details and using high-level languages for the first functional models allows us to increase simulation speed and reduce coding effort. Thus, the system level exploration of design scenarios is way more manageable and faster than at RTL level. Multiple architecture options can be tested with lower effort and cost than at RTL level. The DSE can be divided in two main areas: Formal analytical method and Simulation-based method.

Formal analytical methods are often used as the first step in the flow, as they can be very accurate for best-/worst- case estimations. There are many variations and mathematical solutions based on different holistic and combinatorial approaches and tools [53] [54] [55], and for many years this method has been the golden standard. The strategy was mainly to use individual components analysis and later extrapolate them to the complete system. With the increase in design complexity and the insertion of shared resources, the designs behavior became more interdependent and difficult to predict and analyze.

Therefore, the simulation-based methods took the privilege for a while with the trend for full system simulation. The most commonly used simulation-based methods for early stage design space exploration are based on execution- or trace- driven models, because these methods are more flexible and can be applied for specific use cases [56] [57] [58]. However, the number of IPs in a chip keeps increasing from one generation to another, making simulation-based methodologies too slow and complex. Using SystemC/TLM2.0, we are able to apply a combination of several important techniques related to the execution of these tasks. One of them is the ”divide and conquer” [59] technique, which consists in decomposing a module into sub-modules and the sub-modules into blocks to obtain a manageable complexity. This technique allows for the natural creation of reusable and similar blocks; it thus reduces the verification time of the functional model. In addition, it enforces block locality and independence, which facilitates the portability of our power and performance estimation techniques (described later). Combined with a top-down design approach it allows for better control over granularity and complexity.

However, one major, still relevant, challenge is that there is no single standard level of abstraction that provides maximum accuracy, fast simulation, and minimal coding effort. Often, greater abstraction means lower precision and lower abstraction means slower simulation speed. Thus, we must always keep in mind that in order to obtain optimal results for estimating power and performance metrics, we must implement sufficiently accurate SystemC/TLM models, where “sufficiently accurate” is not a constant, but a variable term and this remains an open issue. We should efficiently model the hardware functionality and in the same time, we should keep the high level of abstraction in order to maintain acceptable simulation speed.

Different studies try to evaluate and balance the trade-off between accuracy and simulation speed/performance. In [60] [61], a comparison between latency error measurements at different modeling detail levels is performed. In [62] we can find some techniques and simple rules allowing to simulate relatively accurate SystemC/TLM-based VPs with OS and software/firmware (SW/FW), while optimizing the simulation speed. Another approach is taken in [63] [50] [64], trying to benefit from the host system computational power, parallelize the simulation and distribute it between multiple cores. However, as the nature of our study is not closely related to simulation speed and parallelism, we target this issue by concentrating our development effort on the reduction of context switching and following some well-known C++ coding guidelines [65] [66]. The meaning of “sufficiently accurate” can be also different depending on our target. The power estimation methodology used in this approach is based on the activity/transactions observation and some timing information. The performance estimation methodology takes additional information on transactions ordering, buffering, pipelining and treatment, so it demands more functionally accurate model. Fortunately, using SystemC/TLM-based structured development, the model refinement is easily manageable and estimates extraction can be done at each refinement.

Generally, the energy consumption can be considered as part of the performance metrics of the IC. However, we will consider it as a separate measure for reasons of separation of concerns and because its introduction to the ESL is the main contribution of our new approach. In this chapter, we will:

1. Review the existing methodologies for early-stage performance and power estimates,
2. Present the library we use for power intent description and power estimates,

3. Provide a brief overview of the SoC family used for demonstrations in this study.

3.2 ESL Performance modeling & estimation methodologies

3.2.1 Introduction

Chip performance analysis is a critical point for the design of advanced heterogeneous systems; therefore, it is inevitable to incorporate early performance estimates into the flow. Using the Simulation-Based Dynamic Analysis (SBDA) approach, the accuracy of the performance estimation depends largely on the accuracy of the functional model. The more the functional model respects the actual behavior of the hardware and its timing, the more accurate the performance estimation will be. This section is devoted to some key points of performance estimation of a memory-based system. In order to perform this estimation, we need to be able to configure several limits for each IP. In memory-based systems, some of the most important configurable parameters are *READ/WRITE* access delays, *MAX/MIN* arbitration delays, port width, and burst length. The key metrics for performance evaluation are bandwidth and latency, deadline respect for real-time systems, the influence of configurations on performance. Of course, the existence of application-specific performance metrics requires the ability to add user-defined performance observation and estimation models in a simplified way. In order to accurately configure our models, we need to take into account some functional and physical information such as communication delays, communication protocol, and initial floorplan ideas.

Some of the metrics used for accuracy quantification are:

- **Transaction duration (*[sec] or [cycles]*)** - Time between the start and the end of each transaction.

$$T_{duration} = EndTime - StartTime \tag{3.1}$$

- **Throughput** - Number of transactions during the total simulation time or during a given time frame. It can be represented in *[Bytes/sec]* or in a more abstract way *[transactions/sec]*. The term *Data* in the following equation can represent the size of each transaction or a count of the number of transactions.

$$TP = \frac{\sum_{n=0}^{Transnumber} Data}{TotalTime} \tag{3.2}$$

- **Transaction reordering** - Important for modules receiving/transmitting multiple simultaneous transactions. The accurate modeling of transactions order can have a significant impact on performance estimation accuracy. There are two main types:
  - *Port Transaction ordering* - ordering on each port
  - *Global Transaction ordering* - global view on system

If we have the RTL model, we can also perform the correlation between RTL and ESL results for each metric by running the same use cases on both models.

3.2.2 Existing approaches and related works

There are many ways to approach this problem. Solutions for performance estimation can be more generic and reusable for several types of components, they can be solely processor or memory oriented, or specific to another type of component.

In [67], the authors propose a framework based on SystemC and the older version of TLM (TLM-1). They provide a library of reusable components for executing and observing computations and specific channels for delay observation. Their strategy is based on cycle-accurate (not fully functional) performance models propagating predefined delays. This approach has many limitations, as TLM-1 does not target interoperability issues and is therefore not suitable for heterogeneous SoCs and their complex communication models.

A performance estimation of a NoC, based on an internal traffic analyzer (SystemC/TLM target module) is performed in [68]. They use Loosely-timed transactional level models and study latency and throughput for different mesh and traffic configurations using this traffic analyzer. The LT coding style allows for very fast simulations, but it is difficult to model the specific semantics of communication protocols. The loss of accuracy will therefore be significant when modeling complex SoCs.

Similarly, the authors of [69] propose another framework containing a parameterizable SystemC/TLM router base structure for recording a set of measurements during simulation. They also provide utility components and correctness verification. The main performance metrics evaluated in this approach are overall average latency, fifo occupancy, total data transfers, and average distance (hop count).

A processor-oriented approach is presented in [70] where the synchronization model is attached to processor core models (RISC-V HiFive1 case study). Three main components track runtime dependencies related to synchronization (pipeline), branch and jump prediction, and cache hit/miss. The authors indicate that this approach can be extended by integrating a complementary synchronization model of the bus system.

There are also open-source EDAs and methodologies available to model, debug, and explore architecture options and their performance at the ESL level.

For example, in [8] the authors propose a flexible DRAM subsystem design space exploration framework based on SystemC/TLM2.0. It can be used with a standalone simulator or it can also be coupled with gem5 simulator and TLM AT-complient libraries. A special Trace Analyzer (not open-source) is used to explore the usual performance-related outputs, such as requests/responses/commands trace. These traces can be evaluated using the Python interface of the Analyzer. Similar approach is taken in [71] solution.

Another framework allowing the modeling and analysing of ESL virtual platforms is presented in [72] [73]. It contains a library of many useful models and utilities simplifying the construction of virtual platforms. It also contains analysis tools allowing to add various counters for throughput and activity monitoring.

Industrial solutions and frameworks like [42] [47] [48] [50] are very complete and widely adopted when talking about design conception and performance analysis. These frameworks and tools present large model libraries and enable the in-depth transactions monitoring, timing tracking and performance estimations.

3.2.3 Performance summary

In general, performance analysis is relatively easy to perform when we have an accurate functional model of the platform. Since SystemC and TLM2.0 allow virtual prototyping at an early stage, we can have functional models relatively early in the design flow. Thus, many modeling and performance estimation solutions already exist and are widely adopted at the system level. For this reason, the goal of this study is not to compete with and innovate existing performance estimation approaches. Our study focuses on ESL-level power modeling and estimation problems, as they remain an open topic and there is no widely accepted solution. Our performance estimation technique is presented only to prove that our framework can be used for the joint study of power and performance.

3.3 ESL Power modeling & estimation methodologies

3.3.1 Introduction

Earlier, we explained the design flow, discussed the importance of power estimation, and mentioned the fact that power estimation methods are usually applied at low level. The main reason for this is the low-level information needed to effectively estimate power consumption.

However, using generic *CMOS* power equations and IP area estimates or data extracted from IP reuse reverse engineering methodologies, we can obtain fairly accurate estimates even at the system level with SystemC/TLM models. Which estimation and approach to take often depends on the information available and the flexibility targeted. For example, a top-down approach based on experimental characterization of the power of existing devices can give very good accuracy, but its flexibility and subsequent availability may be limited. This lack of flexibility may make them unsuitable for initial power studies, but making them more application specific can reduce their complexity and speed up simulations. This type of model, once validated, can be used in broader platforms aimed at developing/integrating new device models based on a bottom-up approach. The bottom-up approach is better suited to study power management, due to its flexibility and adaptability to different specifications. However, for power estimation and modeling, we can use already available functional models (top-down approach) and wrap them with reconfigurable power models (bottom-up approach). In this way, we can start the power study with a less precise configuration of the power intent and readjust it as the flow progresses (reducing the abstraction). This methodology allows us to have a semantically correct exploration of *"what-if"* power management strategies with less precise estimates of power values at the beginning, and to increase the precision with progress. In all cases, we can base our approach on standard power equations.

The energy *E [J]* is calculated as the time integral of the instantaneous power:

$$E = \int_0^t P(t)dt \tag{3.3}$$

The classical power equation *P(t) [W]* is the product of the voltage *V(t) [V]* and the current *I(t) [A]* flowing in the circuit (3.4).

$$P(t) = V(t) \times I(t) \tag{3.4}$$

In many cases, *V(t)* remains nearly constant, so the only variable that needs to be calculated is the current drawn *I(t)*.

The power dissipation of an SoC can be divided into two general parts:

- *Static Power* - often due to transistor leakage current.
- *Dynamic Power* - due to the switching activity of the transistors.

The total power consumption is simply the sum of these two parts.

$$P_{total} = P_{static} + P_{dynamic} \tag{3.5}$$

Static Power [W]

As mentioned above, static power consumption is due to several current flows and one of them is the transistor leakage current. In *CMOS* architectures, the leakage current is responsible for almost all of the static power consumption. In more recent technologies, there are more contributors, but in our methodology, we only consider the leakage current, because at this level of abstraction, it is difficult to predict the behavior of the other contributors. Therefore, it is easy to deduce that in modern SoCs, this leakage part becomes more and more important, due to the higher density of transistors. Static power is the product of power supply *(V)* and leakage current, which depend on the technology. It is observed when a device is powered, but no information/signal is transmitted. In other words, it is the so-called *"sub-threshold"* or **"leakage"** carrier diffusion current, appearing between the source and the drain of the transistors when the gate voltage is lower than the threshold voltage.

$$P_{static} = V \times I_{leakage} = \frac{V^2}{R_{leakage}} \tag{3.6}$$

Dynamic Power [W]

Dynamic power is the result of charging and discharging the *CMOS* transistors' load capacitance. It is therefore only observed when the device in question is active and carrying signals. It is related to two important factors, namely the short-circuit currents and the switching currents.

$$P_{dynamic} = P_{short-circuit} + P_{switching} \tag{3.7}$$

The short-circuit power is the power dissipated by an instantaneous short-circuit connection when the gate changes state.

If we observe a simple *CMOS* inverter structure (Figure 3.1), the upward movement of the voltage (V_{in}) leads to a simultaneous switching of the two transistors, which creates, for a short time, a direct path between the power supply and the ground, giving rise to a short-circuit current. The expression of the short-circuit dissipated power is:

$$P_{short-circuit} = Vdd \times I_{sc} \tag{3.8}$$

In general, during an increasing voltage operation (V_{in}), the output load capacitance *(C)* alternately charges and discharges, resulting in power dissipation. In theory, if all the transistors switch at the same time and on every cycle of a clock, then the dynamic power is the

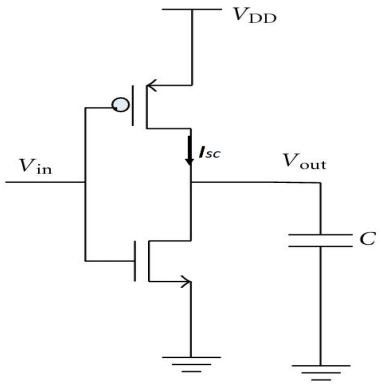


Figure 3.1: CMOS inverter

product of the transistor's load capacitance *(C)*, the transistor's supply voltage (V_{dd}) squared and the frequency *(F)*. In reality, we will never have all the transistors switching at the same time, so for each IP we can add an activity factor (α) with a value between 0 and 1, which will be in charge of modeling the average activity according to the number of transistors switching.

$$P_{dynamic} = \alpha \times C \times V_{dd}^2 \times F \tag{3.9}$$

In the literature it is considered that the short-circuit power is negligible compared to the switching power, so it is often ignored.

Total Power [W]

(3.5), (3.6) and (3.9), results in the total power equation:

$$P_{total} = (\frac{V^2}{R_{leakage}}) + (\alpha \times C \times V_{dd}^2 \times F) \tag{3.10}$$

The parameters in this equation are obviously technology dependent. Some of them can be easily found in device datasheets, others can be inferred from previous simulations or physical measurements (in the case of IP reuse), and still others can be estimated for the first time to compare different architecture/power intent strategies and refined during the development cycle. The strategy is explained in Sections 3.4 and 4.6.

Existing power reduction techniques play with some of these parameters, such as clock frequency and supply voltage, to reduce the power consumption of the device. The idea is to adapt the power consumption to the needs of the device, in order to provide the optimal power for the execution of the given task. Some of them target static power consumption, some target dynamic power consumption and still others can influence both.

Static power reduction techniques

The successful reduction of the *CMOS* supply voltage V_{dd} in modern technologies has led to the reduction of the transistor threshold voltage V_t. This has introduced some negative effects, such as increasing the leakage current and thus the static power consumption. Considering the positive effects of the decrease in supply voltage, it was inevitable to implement more mechanisms to maintain this trend and limit these leakage currents.

- *Multi-V_{dd}* - a multi-voltage supply that provides a high voltage V_{dd} to the high performance transistors and a low voltage V_{dd} to the low performance transistors. Thus, the cells of the circuit belonging to a critical path would be supplied by relatively high voltages, in order not to degrade the overall performance, while the other cells could be supplied by lower voltages. In this way, the static power can be largely reduced.

- *Multi-V_t and Body Biasing* - Multi-V_t and Body Biasing are used to adapt the threshold voltage to the system requirements. The principle is that transistors on critical paths operate at a low threshold voltage V_t while others can have a high V_t with a low leakage current. In the Adaptive Body Biasing (ABB) an adaptive system controls the threshold voltage V_t, by changing the substrate voltage, and the V_{dd} according to the performance requirements of the application so as to have an optimal leakage current. This system continuously checks and adjusts the substrate voltage to achieve a constant leakage current.

- *Power Gating* - Power gating is performed by cutting off the supply voltage to inactive blocks in the circuit. The supply voltage to these blocks is temporarily cut off to eliminate leakage currents. When a block in the circuit needs to return to the active state (perform an operation), the supply voltage can be reapplied. For this purpose, properly sized *pMOS* and *nMOS* transistors are used. In order to stabilize V_{dd}, additional delay and energy are required.

Dynamic power reduction techniques

Since dynamic power consumption is strongly related to the switching activity of the transistors, we can reduce it by optimizing the terms of the equation (3.9). We can reduce the activity factor (α) by reducing the number of switching transistors. We can also reduce the load capacity *(C)* by reducing the length of the connections. The terms over which we have a little more control are the frequency *(F)* and the supply voltage V_{dd}. We can, for example, decrease or cut clock frequencies for separate inactive logic cells. Here are some widely adopted techniques:

- *Pipelining and parallel processing* - These techniques are twofold and can be used to replace, for example, a single-core architecture with an architecture containing multiple cores running in parallel at a lower frequency. If a computation can be pipelined, it can also be executed in parallel. In this approach, we trade area for energy dissipation. The frequency is decreased, which means that the computation time of a given function is increased. Thus, the supply voltage can be reduced (readjusted to the propagation delay). This approach does not fall within the spectrum of our study as it is more oriented towards processing units, but a more detailed description can be found in [74] and [75].

- *Clock Gating* - Clock gating is the best technique for reducing the dynamic power. The reason is that clock signals are mandatory in every synchronous circuit. The switching rate of this clock signal is very important and is a major consumer in the circuit. Its functionality is somehow similar to that of power gating. We cut the clock signals of the unused components in the circuit. Cutting the clock signals of idle components can lead to significant power gains (up to 70%). To apply this technique, we need to divide the

circuit into areas belonging to different clock domains (groups of components receiving clock signals from the same source clock). In this approach, when a set of flip-flops is clocked by the same clock signal and the same enable signal, we can mix these two signals by a simple *AND gate*. This leads to a clock signal being active only when its enable signal is also active. However, this type of operation modifies the clocks themselves and must be implemented very carefully. A more detailed description can be found in [76].

- *Dynamic voltage and frequency scaling (DVFS)* - DVFS is an active power reduction technique based on supply voltage and frequency scaling [eq. (3.9)]. The frequency is directly related to the supply voltage, so we can think of a reduction technique using this relationship. A decrease in the value of the supply voltage results in a quadratic reduction in power, and a decrease in the value of the frequency results in a linear reduction in power (this can be seen in the equations (3.10) and (3.9)). An additional benefit of the voltage reduction is a collateral reduction in static power. However, if we reduce the value of the transistor frequency, we immediately increase the response time. Thus, in order to use this technique safely, we must consider the time constraints and ensure that they are met. In this case, it may be beneficial to use an **Operational Performance Point (OPP)** table containing a frequency and its associated supply voltage. When the frequency changes, the OPP table is consulted and the corresponding new voltage is applied.

To illustrate the importance of these power management strategies, we consider a very theoretical and schematic example. Figure 3.2 shows an example architecture where we run a periodic computational traffic load with a large idle cycle between activations. Here we can observe the difference between the power behavior of a simple (non-power-aware) architecture and a power-aware architecture. In Figure 3.2a the static power remains constant and the dynamic power varies with activity. In Figure 3.2b we apply clock gating whenever the activity is zero and we apply power gating only when the *IDLE mode* is longer (to avoid too much states switching resulting in higher power consumption). In this schematic example, only power gating and clock gating mechanisms are applied and the power consumption is already significantly reduced.

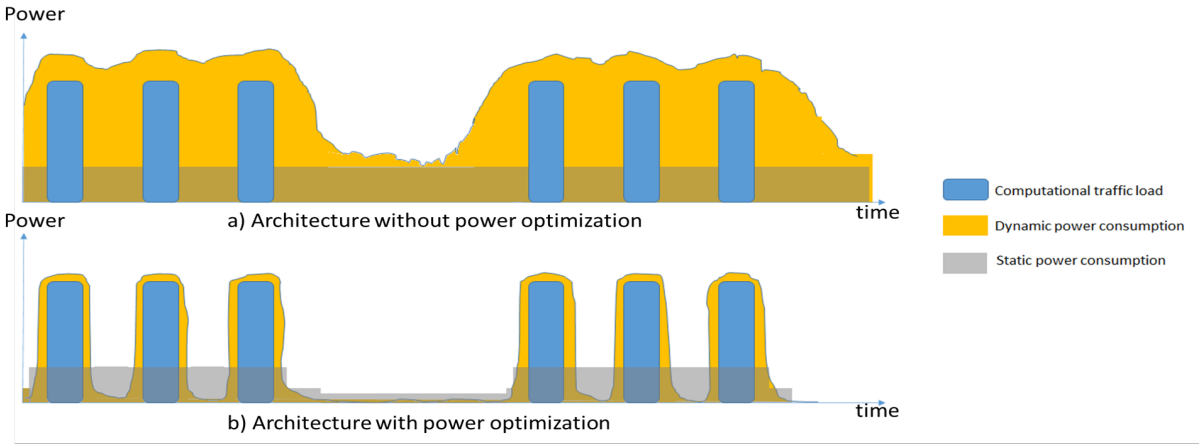


Figure 3.2: Architecture with and without power optimization

3.3.2 Voltage, Power and Clock domains

Voltage domain

The voltage domain is a single power domain or a group of power domains supplied with the same voltage value defined for that domain. The supply voltage is supplied to the voltage domain by an internal or external voltage regulator and is distributed in the circuit by supply rails. This distribution is achieved by the external voltage regulator method, based on a specific circuit called **Power Management Integrated Circuit (PMIC)** or on integrated voltage regulators in the case of the internal method. In some cases, the voltages can be adjusted (or cut) dynamically using operations internal to the circuit. As mentioned earlier in this section, reducing the supply voltage of some components can have a significant and direct impact on the total power consumption. High-performance blocks (such as the CPU and GPU) can often require a higher supply voltage than low-performance components (such as some interface controllers). Therefore, it may make sense to include them in two different voltage domains. Figure 3.3 is an example that shows the distribution of voltage domains applied in the NXP i.MX8M Quad [77].

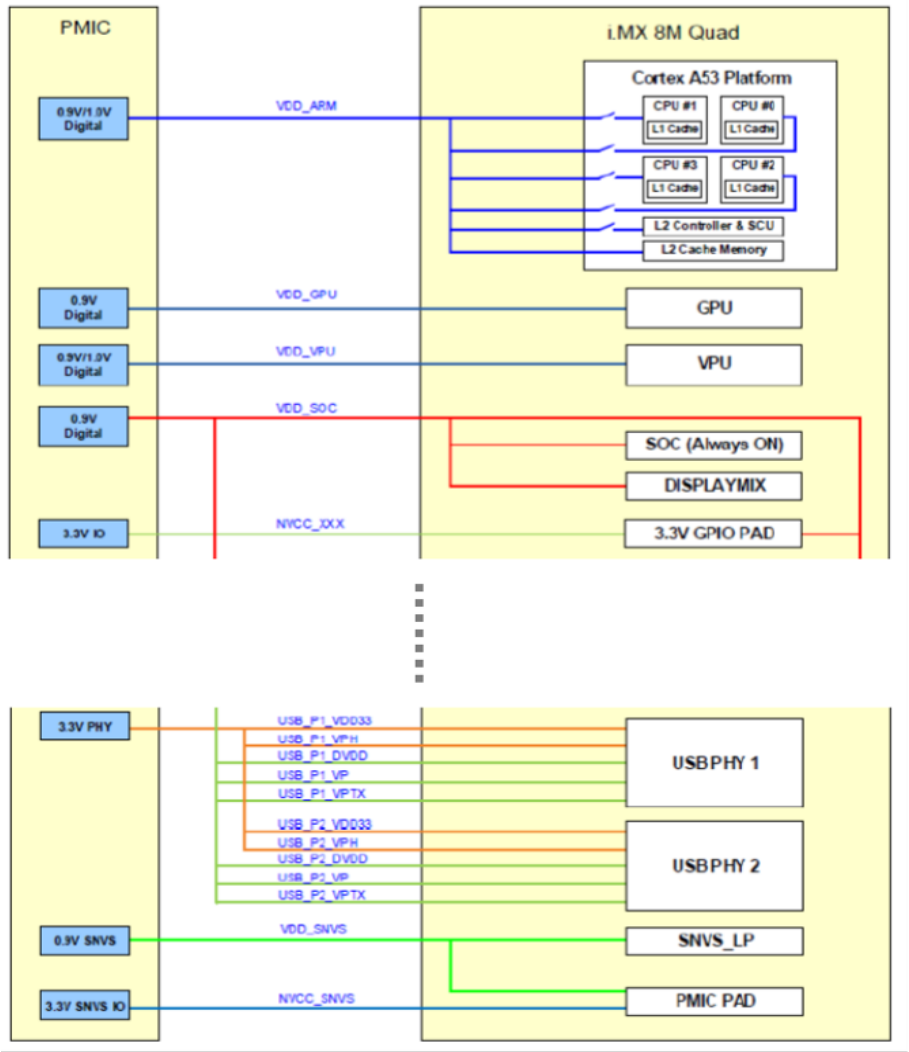


Figure 3.3: NXP i.MX8M Quad power rails

Power domain

The power domain is a group of blocks powered by the same voltage regulator and an array of power switches distributed over the area of the power domain. It can have a hierarchical structure, so a power domain can contain one or more other power domains. In the example shown in Figure 3.4 there are five power domains: *PD_TOP*, *PD_INTERC*, *PD_DRAMCTRL*, *PD_DRAMCTRL0* and *PD_DRAMCTRL1*.
These power domains are supplied by two different voltages. The top power domain containing all the others is called *PD_TOP*. *PD_INTERC* is a non-switchable power domain and its power supply cannot be switched off. *PD_DRAMCTRL* contains two power domains - *PD_DRAMCTRL0* and *PD_DRAMCTRL1*. The *PD_DRAMCTRL* power domain can have its power cut off via switch *SW0*, thus *PD_DRAMCTRL0* and *PD_DRAMCTRL1* receive no power. If *SW0* is closed, we can cut the power supply of *PD_DRAMCTRL0* or/and *PD_DRAMCTRL1* separately, if one of them is inactive, via the switches *SW00* or/and *SW01* respectively.

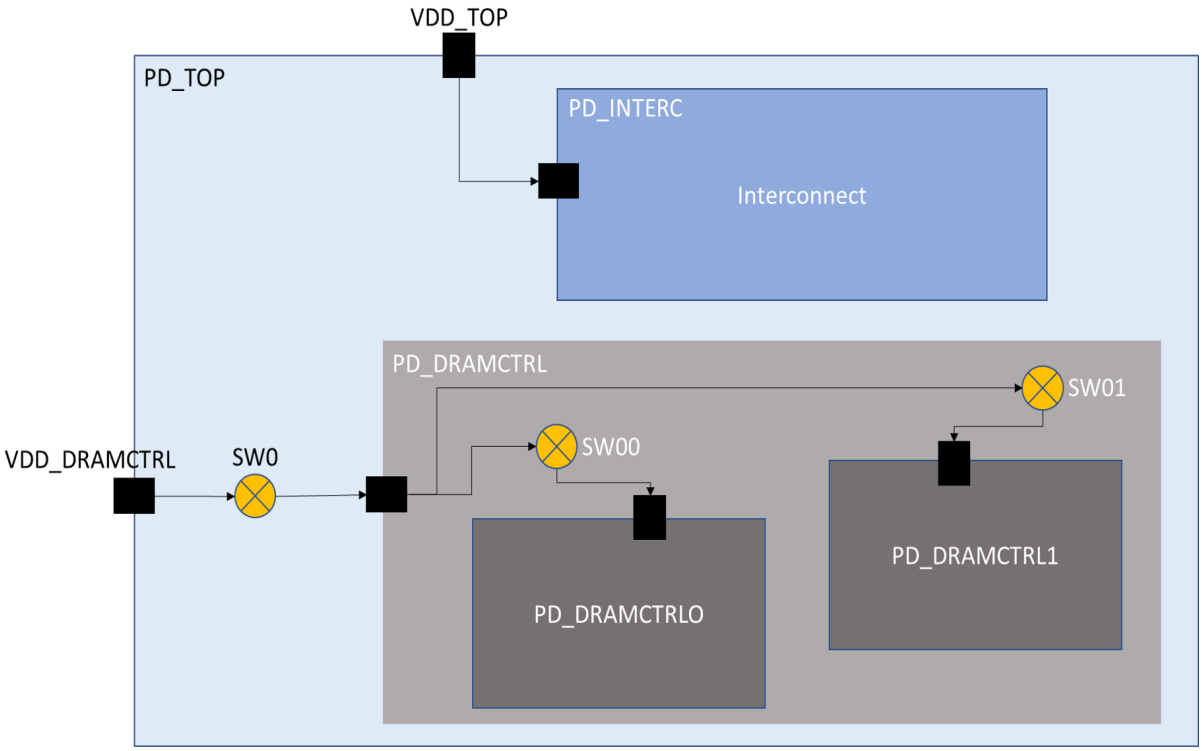


Figure 3.4: Example of Power domains structure

This example is a simplified version of the power domain distribution. In general, in order to synchronize and optimize the communication, power consumption and efficiency of the low-power design, specific components have been developed. For example, there are level shifters allowing to normalize signals transferred from one power domain to another, there are isolation cells that set a logic level to 1 or 0 when the power domain from which the signals originate is power gated (not supplied), and there are retention registers allowing to save the state/data of inactive (power gated) power domains. These components are mandatory for low-power design to eliminate functional bugs and remove the possibility of undefined behavior. The disadvantage of these specific components is that they add delays, increase

complexity, and some of them can result in significant area overhead. For this reason, some trade-offs must be made in defining the power domains and their granularity.

These principles of structuring the division of circuits into power domains are well covered by the UPF standard (detailed later in the section).

Clock domain

A block in the hardware architecture may require one or more clocks to operate. A clock domain is a set of blocks clocked by synchronous or constant phase clock signals coming from the same source. Unlike the case of power domains, there is no hierarchy possible in a clock domain. The clock source of a clock domain can be an external signal or a signal generated internally from an external signal. Often, a **Digital Phase Locked Loop (DPLL)** is associated with each clock domain, allowing a multiplication and division factor to be set at initialization or dynamically. This DPLL represents the primary source clock of the given clock domain and its outputs are connected to the different blocks that make up the clock domain. The logic for this clock distribution and control of the clock tree is usually integrated into **Clock Management blocks (CM)** located between the clock generator and the function blocks. These CM blocks typically consists of multiplexers, dividers and gates to enable or disable the clock. The logic can be controlled by HW or SW and allows us to control the clock more finely, which can allow us to optimize the trade-off between performance and power consumption. An example for clock domains partitioning is shown in Figure 3.5.

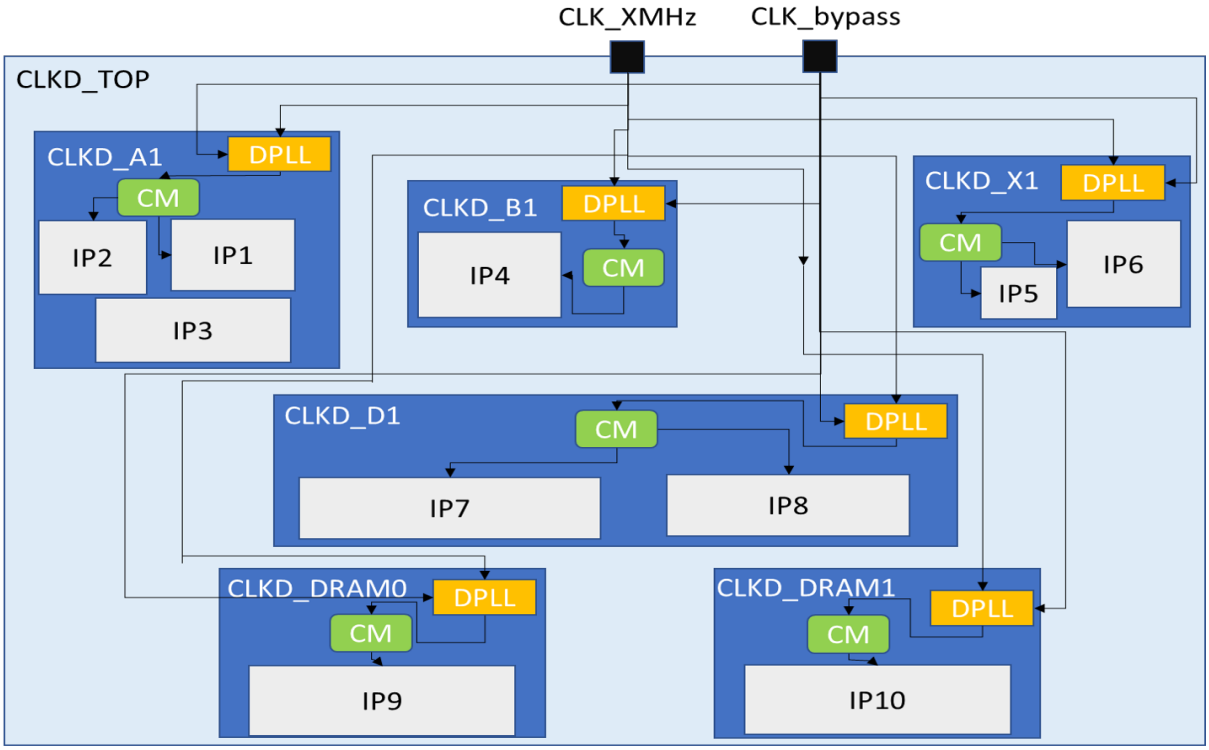


Figure 3.5: Example of clock domains structure

Dividing a circuit into voltage domains, power domains, and clock domains define the knobs a power management strategy can play with. Defining the right configuration of those knobs can significantly reduce the power consumption of the device without affecting the behavior of the application being executed.

The implementation of power reduction techniques, such as those presented above, requires the use of several tools and standards that are generally available at the RTL level and below. These techniques are difficult to test at the ESL level due to the lack of tools allowing to define power sensitive system level models and due to the presence of many unknown parameters. In the following section, we will review some of the standards that can be used as a reference and extended to the ESL level and discuss some of the currently missing standards.

3.3.3 Presentation of existing standards

There are a some widely used standards at the RTL level to initiate the description of power management at this level. The description of power intent cannot be performed in the RTL model, so different notations have been deployed and several standards have been defined. Figure 3.6 illustrates the observations of Wilson Research Group and Mentor Graphics, a Siemens company, in their 2020 Functional Verification Study [38] on the different standards used to describe power intent. The trend in recent years has been to use the **Unified Power Format (UPF)** standard and to slowly adapt to its latest versions.

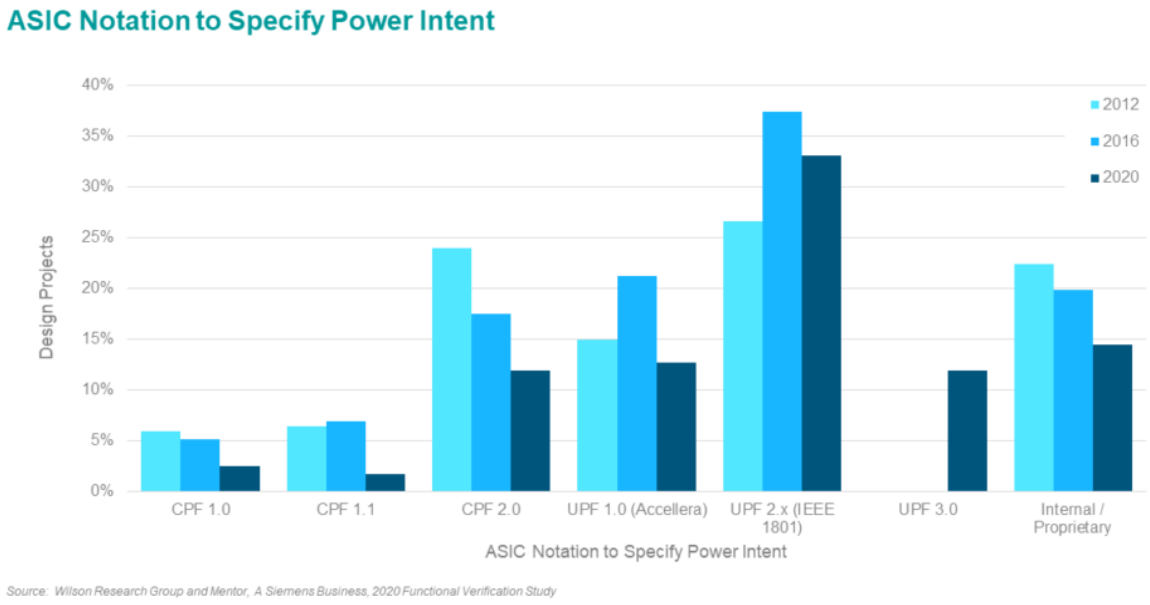


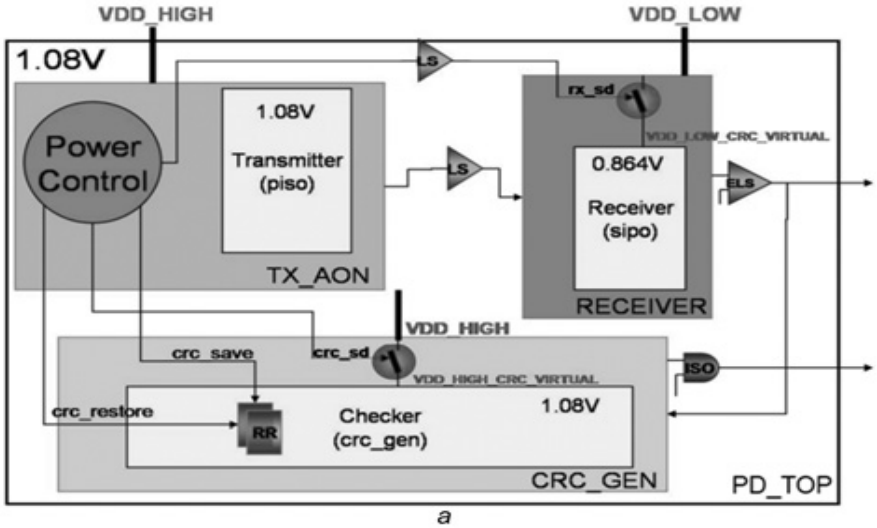
Figure 3.6: Notation used to describe power intent

IEEE 1801/UPF3.0 - System-level semantics

IEEE UPF [78] is an evolving standard for specifying power intent that has been and continues to be a major game changer for power optimization.

It describes the elements involved in the control of power consumption, independently of the functional specification. This concept is based on an approach where the functional and performance aspects of a model are studied and validated before optimizing the power consumption. UPF is implemented using the Tool Control Language (TCL) syntax and is typically used at the RTL level and below. Its semantics define power domains as a set of design elements sharing a common primary supply set (e.g., power and ground supply

nets). **Power management strategies** are implemented using **Power State Tables (PSTs)** containing valid combinations of power states from each power domain. These combinations are referred to as power modes. The implementation of this semantics in a system is called **Power Intent**. An example of defining power intent and creating PST is given in Figure 3.7.



a

	VDD_HIGH	VDD_LOW	VDD_HIGH_ CRC_VIRTUAL	VDD_LOW_ CRC_VIRTUAL
PRE_BOOT	High_Voltage	Low_Voltage	CRC_OFF	RX_OFF
CRC_ON	High_Voltage	Low_Voltage	High_Voltage	RX_OFF
RX_ON	High_Voltage	Low_Voltage	CRC_OFF	Low_Voltage
ALL_ON	High_Voltage	Low_Voltage	High_Voltage	Low_Voltage

b

Figure 3.7: Example of UPF concepts [3]

In this example, in Figure 3.7a *TX_AON* is a power domain containing the function block *Transmitter* powered by the supply net *VDD_HIGH*. The power domain *RECEIVER* contains a Power Switch that turns on or off the power to the receiver function block based on the power control signal (*crc_sd*). In addition, retention registers (RR), level shifters (LS), and isolation elements can be instantiated. Figure 3.7b shows a valid PST composed of the power modes (lines) defining the power network states of the power domains corresponding to each mode (columns).

Coupled with a hardware description, the UPF standard enables portability of low-power designs and eliminates the problem of choosing and relying on a single electronic design automation (EDA) tool. Its growing use and integration within EDAs and IPs makes it a prime candidate for further enhancements and expanded use.

The latest version of UPF, also called UPF3.0 [79], suggests an increase in the level of abstraction to begin the power strategy definition at system level. The strategy consists of using IP models that control their corresponding power models to enable or toggle between different power states and recalculate power consumption. Each power model is attached to an IP model and can only communicate with it and potentially with its power data characterization layer. The three key elements to be defined in the system-level power models are the enumeration of power states, the power state consumption data, and the power state

legal transitions. All possible target power states must be defined, regardless of whether they are triggered by the supply network or by a mode change. The power consumption is recalculated each time the parameters in the IP sensitivity list are changed or each time a power state is activated. Power state consumption data should be provided separately for static and dynamic consumption using floating values or power functions (*-power_expr* - suggested solution). Using the power function, we can define static build time parameters, dynamic run-time parameters and dynamic parameters based on the event-controlled rate. All parameters related to the specific power model must be instantiated in its body.

The UPF is not intended to describe a standard method for the representation and description of system-level power data. A new standard, called IEEE 2416 or **Unified Power Modeling (UPM)**, has therefore addressed this topic. It aims to define the standardization of power data representation to enable early power and thermal analysis.

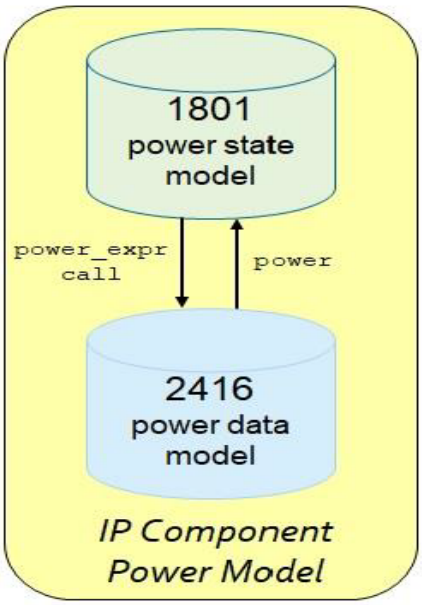


Figure 3.8: UPF/UPM relationship [4]

IEEE 2416/UPM - System-level semantics

The IEEE 2416 standard [4] enables the system, software and hardware power analysis and optimization with a parameterized and abstracted power model. It describes an approach that includes several extensions to the IEEE 1801/UPF power expressions, such as a power and thermal management interface and architecture parameterization.

The standard is based on Process, Voltage, Temperature (PVT) independence, which enables cross-platform interoperability and portability at all levels of the development flow. The primary considerations and concerns addressed in this standard are the modeling standards interoperability (interfaces), data representation, and model evaluation capabilities. The goal is to allow the reuse of a model throughout the development cycle and to facilitate the design development and use of other power, workload, and function modeling standards (UPF, SystemC, SystemVerilog). UPM semantics are based on XML elements and attributes with extensions to represent expression using Verilog-AMS.

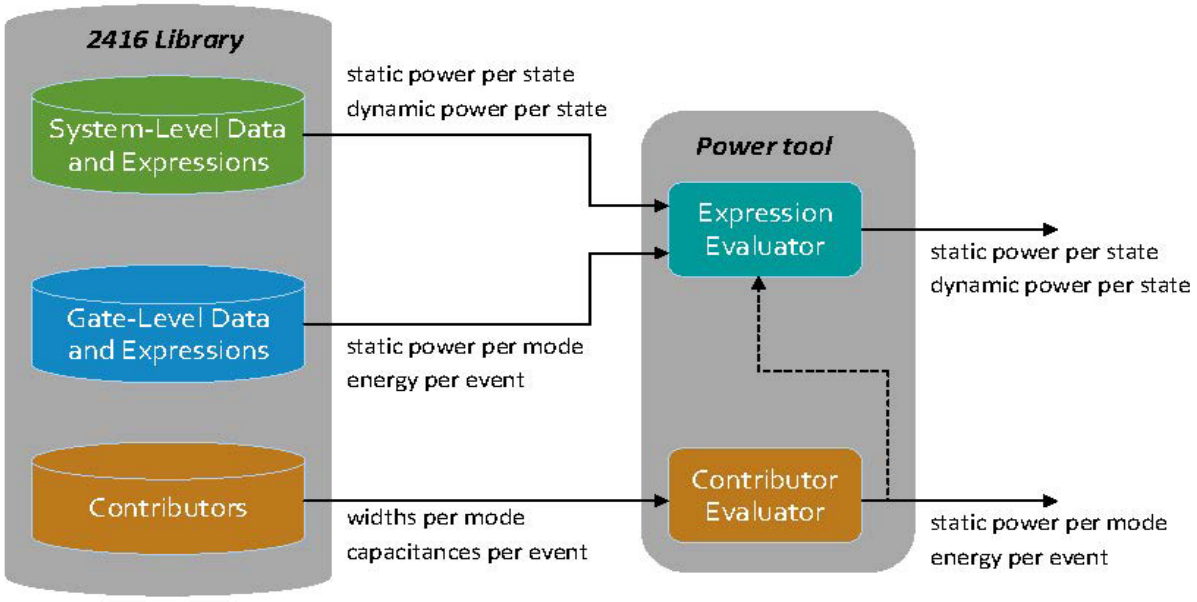


Figure 3.9: UPM: General model architecture [4]

Depending on the level of abstraction, mandatory and optional attributes, cells and sub-elements must be defined in the UPM model. There are two main modeling levels (Figure 3.9) - system level and gate level. Figure 3.10 illustrates the difference between the UPM models at the gate-level and at the system-level.

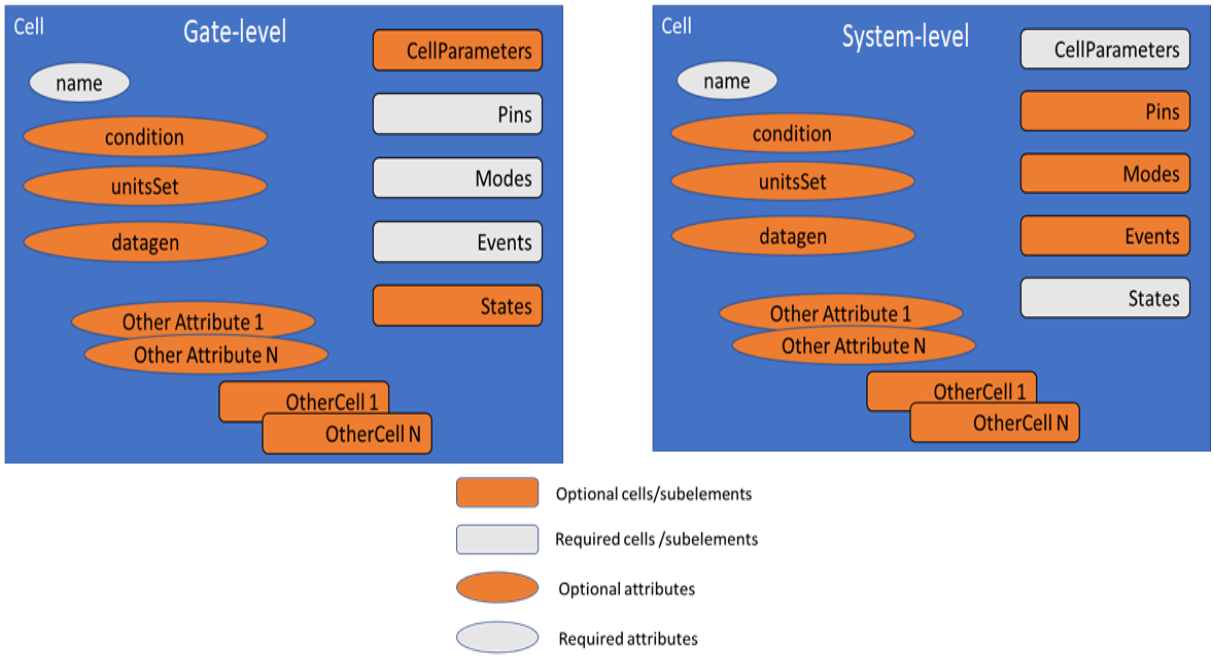


Figure 3.10: UPM: Gate-level vs System-level

- At the gate-level, we need to define cells and sub-elements such as cell pins, functional and power modes (where all signals are implemented), and energy consuming events.
- At the system-level we need to define cell parameters (set of parameters defining the

power model) and the functional and power states (where certain signals are implemented). The approach is designed to ensure interoperability with the UPF power(-state) models, which take as input a pair of static and dynamic power values/expressions for a given state.

These standards can be taken as a reference when talking about power modeling, as they are actively evolving and a very large community is contributing to their development. However, some points still need to be addressed in order to meet the needs of the industry.

Areas of improvements

These standards are more focused on defining power intent. In contrast to this case, to date there is no commonly established definition of the clock domain concept. Almost any block in the hardware architecture may need one or more clocks to operate. Clock routing is usually performed after placement and routing in the design and is done based on the results of synthesis. Since UPF contains very few clock-related concepts, there is no well-defined standard for specifying and designing clock trees in a circuit. Nevertheless, the best practices applied by leading companies converge on very similar clock domain definition concepts (presented above). Due to the lack of a clock domains unification standard, the majority of ESL power estimation tools and approaches do not implement a realistic clock management mechanism, which limits the testing of power management strategies and thus the modeling of complex low power designs.

In the following section, we will present some of the existing academic and industrial solutions for ESL power estimation and then compare them to the one used in this study.

3.3.4 Existing approaches and related works

Previously, many power estimation methodologies have been studied or developed, mainly based on static and/or purely mathematical approaches [41], [80] such as the one presented in 2.2.3.

With the increase in software and hardware complexity, the need for dynamic approaches to account for the impact of the application on energy consumption, has led to the search for several simulation-based power estimation methods.

In the following approach [81], the authors use the IP activity during TLM simulation and extract it into a VCD file. Then, they synthesize the TLM model and compile the design using the design constraints. The power consumption is estimated based on cell reports extracted from the design.

In another study [82], the approach is also based on IP activity. Different types of activity are defined, and counters are added for each of them. Each counter increments when the corresponding activity is executed during the simulation. The corresponding power consumption is applied to each type, which allows to calculate the activity of the entire module. Each IP type has its own modelling approach: i.e. memory, processor, crossbar etc.

In the article [83], the power-related information is added to the SystemC functional model. In this work, it is assumed that power-related transactions are developed before the functional transactions. Each corresponding IP transaction is identified, and its consumption is characterized. Macro models of IPs with multiple parameters are created and the TLM

functional models are enriched in order to dynamically call these macro models during the simulation.

The authors of [84] use power monitors adaptable to different levels of abstraction.

Some solutions are processor specific (in particular in articles [85], [86]) and present the power estimation using instruction consumption. In the first case, such a simulation is done using IASI (Instruction Accurate Simulator Imperas). In the second, SystemC, PERFidiX (to add OS tools to communicate with the SW) are used.

The SEAS approach [87], does not use functional models. The objective is to estimate performance, timing and power as early as possible in the flow. SEAS analyzes functionality in combination with analysis tools. It allows the user to create a typical block diagram description. The component descriptions, models and characterizations are contained in a library. For performance analysis, they use only FSMs and not functional models (high abstraction) in order to represent only the impact of the component on performance. Data processing is not modeled, they only model the number of cycles and latencies. FSM models communicate through tokens. The arrival of a token can trigger a state transition or an event that generates new tokens. For planning analysis, they need to give characteristics about the size and shape of the components. During the planning analysis and routing simulation, the size of the connections between the components is approximated. This data is used for timing analysis, but is not meant to be representative. They are only used to find critical delays on connections. The power analysis is performed by the familiar spreadsheet formulas, but expanded to include power modes. Each IP is associated with a **Power State Machine (PSM)** that defines its power state, transition, and delay (cycles to change the power state). The performance model captures the power states of the IPs and records the activities of the IPs (idle, sleep, on, off). By summing the power values for the states, we estimate the power consumption. There is a PMU that monitors the power information of the IPs. They use a tool called CORAL that recognizes the block diagram and automatically associates it with an RTL description that is then analyzed.

Some approaches, such as [88], [89], aim to track signal and register activity for power estimation. They use UVM to create an interoperable testbench between TLM and RTL and perform a direct comparison.

In this methodology [90], applied to chip devices, a power consumption is associated with different ”driver calls”. Driver calls are a set of assembly instructions corresponding to an API. Instead of giving a power consumption value to each instruction, they abstract a set of them. An ArchC CPU simulator is used for accurate cycle simulation. Their model is based on a platform that considers all devices and they use finite power state machines containing the power states for each device. Then power monitors are implemented to extract power estimates.

All of these methods require a significant amount of time and coding effort in order to extract power estimates. In addition to that, there is no information on the application of any power reduction technique.

Several UML-based model-driven approaches [91] [92] have been developed to enable the generation of power-sensitive models, but they are tightly bound to specific system-level power descriptions/tools.

The open-source TLM POWER3 library [93] provides a power estimation and management methodology based on the characterization and annotation of the power of individual TLM models. The advantage of this method is that it can be used with either loosely or

approximate timing TLM models. This approach can give significant accuracy, as it can contain low-level information, but it seems to be very difficult to use, especially for complex heterogeneous systems (we did not find any such examples available). Also, it does not follow the UPF standard, which means that the whole description of the power intent has to be rethought when moving to a lower level of abstraction. They allow the use of TLM generic payloads, but suggest the use of their own extended payloads (without using the generic payload extension mechanism), which can largely decrease the interoperability of the models and complicate their reuse. The authors mention that it is possible to create energy management strategies and enable modeling of energy reduction techniques, but from what we understand, this would add significant and highly intrusive energy-related code to the models (unmanageable for complex heterogeneous systems). One of our main goals, which is the separation of concerns between power and behavior models, is not sufficiently emphasized in this approach.

The following approach [94] provides a configurable power modeling methodology that uses power state machines taking state names, current and voltage as parameters. Their approach allows the creation of power components containing a mechanism for defining power states. In each functional model, possible power states must be added and a user-defined function *update()* must be created to drive transitions between power states. Their approach does not include a description of the clock domains (nor the real power domains) and no direct tools for applying power reduction techniques are provided. However, the authors present a fairly good correlation with real hardware measurements.

The authors of [95], propose an approach that respects the separation of concerns between application, hardware and power models. They use task graphs to recreate traffic and different strategies to model the IP (depending on whether it is a third party, custom IP, or software). They use power state machines, but dynamically extract information from functional models. Their approach allows the creation of power islands (f, V_{dd}) and the application of DVFS. For the software part, the power consumption per block is determined using an instruction-dependent power model generated by a simulation of the processor RTL model. For the hardware IP part (SystemC/TLM models), they use simple activity monitors extracting power data and an internal monitor-specific PSM is triggered (based on predefined power states).

In [96] there is an approximately timed SystemC/TLM approach applied to a NoC architecture with DVFS. The authors mention that they do not use pure functional models and describe IP specific power model for each IP. In their approach, the DVFS modeling is independent to the actual power modelling.

In [97] and [98], the authors present a well-structured approach, which is inspired by many studies. They use SystemC functional models augmented with their PMS library to create UPF-like power intent. Once the SystemC/PMS model is sufficiently accurate and detailed, they proceed to High-Level Synthesis (HLS) to generate RTL code and UPF file. The main Power intent verification is done with the generated RTL and UPF. Like their solution is provided for HLS, they use SystemC interfaces for the communication. Thus, it remains a highly time-consuming approach for modelling complex and protocol-based systems.

Besides, there are many industrialized methods and tools that are being used for power estimation.

We can cite tools like Synopsys Platform Architect [42] which uses a high-level state

machine description based on UPF. A power consumption is associated to each state and for each state triggered by given events, the virtual prototype detects these events and updates the total power consumption. This is a relatively common approach for power estimation. However, the lack of easy way to describe clock/power domains and to apply power reduction techniques make it unsuitable for power management investigation.

Another industrial solution is Intel Docea Power Simulator (IDPS) [99] used to create system-level SoC power and thermal models and apply post-processing power analysis [100]. Intel‘s simulator is mainly based on dynamic approach for power estimation. For each component, it needs a description of the power state transition events, translated into PSM. For each power state, the user has to provide the corresponding power profile. Extracting specific data from the SystemC-TLM functional model to VCD files allows us to co-simulate the power model with these VCD files. Thus, it is an efficient method for power and temperature estimation. However, the IDPS does not provide a tool for describing power optimization strategies. The only way to do this is to integrate the entire description into the functional model. There is not a direct co-simulation of the functional and power models, thus it may be more time consuming and the relationship between the functional and power models may become less stable.

Mentor Graphics presents another solution at the TLM level called Mentor Graphics Vista [48] but similar to the previously cited industrial solutions, it is not suitable for power management strategies investigation.

3.3.5 Conclusion

The goal of our work is to propose and study a methodology allowing the joint estimation of power and performance values for any chip and for any use case. This method must be easily adaptable and integrable to the design flow, and must therefore be compatible with the standards currently used. Furthermore, it must be based on high-level (system-level) models in order to start extracting metrics and analyzing the architecture and power intent early in the design flow. The first reason to do this is that we need to enrich the static datasheet/Excel approach with a dynamic approach increasing the architectural exploration capabilities. Simulation-based dynamic verification can allow us to explore different architectural options early in the design process and optimize the trade-off between power and performance. If we want to go even further, the metrics extracted at the beginning of the flow, can be used as a reference by the designers and developers who will follow our specifications. Moreover, we can avoid some bugs and reduce the number of design iterations. In our solution, we use the well-known SystemC library for functional modeling of IPs, TLM2.0 to abstract the communication between IPs, while considering timing and protocols, and we use a power management library called PwClkARCH. The solution we will present is based on the UPF standard and can be used as an extension and association to the IEEE UPF3.0 standard and the IEEE 2416 standard for system-level power modeling. The PwClkARCH library does not rely on state machines, but on relatively accurate surface information and IP activity reports. The power estimate is extracted from the sum of this information and is therefore more accurate than the one based on power states.

3.4 PWClkARCH library for power-aware VPs

3.4.1 Introduction

PwClkARCH [101] [102] is a C++/SystemC-TLM library developed for ESL power estimation. It has been developed for 10 years in the LEAT laboratory by M.Auguin. Most of the semantics included in this library are based on the UPF standard and abstracted at the TLM level (Figure 3.11). The details of the low-level power description are neglected in order to allow power modeling at a very early stage. An automated generic assertion verification mechanism is integrated in the library in order to ensure the consistency between the behavioral and power models.

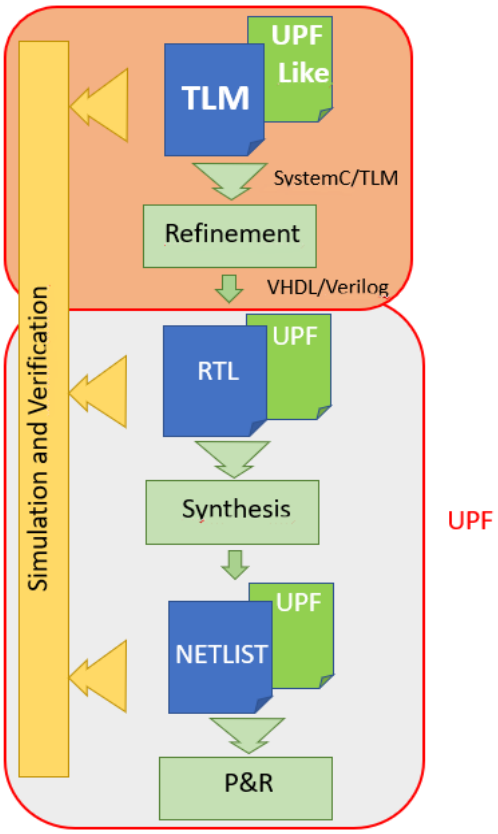


Figure 3.11: Extending UPF to System-level

The library includes components such as **Design Elements (DEs)**, **Power State Table (PST)**, **power switches** and **supply nets**. Since there is no current standard for the description of the clock domains, PwClkARCH introduces a methodology that allows for the insertion of **external clocks**, **generated clocks**, **DPLLs**, **Clock Managers (CMs)** and a **Clock State Table (CST)** similar to the UPF PST. This unified clock domains description approach is inspired by the current most widely used clock distributions (briefly described in 3.3.2).

Figure 3.12 illustrates the decomposition of the library. There are four main types of classes. One of them is dedicated to the verification mechanism. Two of them are composed of C++ classes containing the definition and description of the components necessary for the Power/Clock intent specification and their observers. And the last one contains the SystemC/TLM classes defining the control components and mechanisms.

These components allow us to describe a power and clock intent structures and divide the architecture into different power and clock domains. The power switches, DPLLs and Clock Managers allow us to control the supply voltage and clock frequency of each power and clock domain separately. The DPLL controllers, PST, CST and the use of an Operating Performance Point (OPP) table, based on the one used in the Linux kernel, a**llows us to define power management strategies such as Clock gating, Power gating, DVS, Frequency ramping/CPU throttling, DVFS etc...**

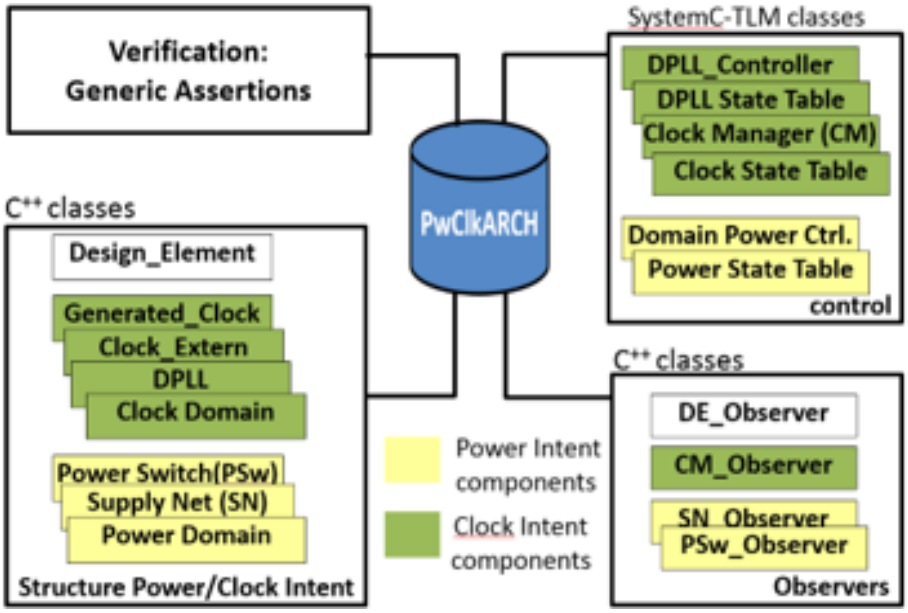


Figure 3.12: PwClkARCH structure [5]

3.4.2 Separation of concerns

Similarly to UPF, PwClkARCH allows us to maintain the **separation between the functional model and the power intent** (Figure 3.13). This means that there is no intrusive power-related code in the functional model, allowing it to be reused for performance estimation alone and making it more readable and efficient. This also allows parallel simulation between the functional/performance model and the power model and to easily study multiple power/clock intent structures and power management strategies for the same functional model. Thus ”what if” analyses are largely facilitated with this approach.

We can observe the influence of each power management strategy on the performance of the functional model and easily detect bugs, such as poor power management and possible data loss. The PwClkARCH library is able to continuously monitor the power consumption of all IPs in any architecture running any application.

The relevance of this approach can only be validated if the power consumption models are sufficiently accurate with respect to the real system and if the performance/power consumption relationship is also sufficiently accurate.

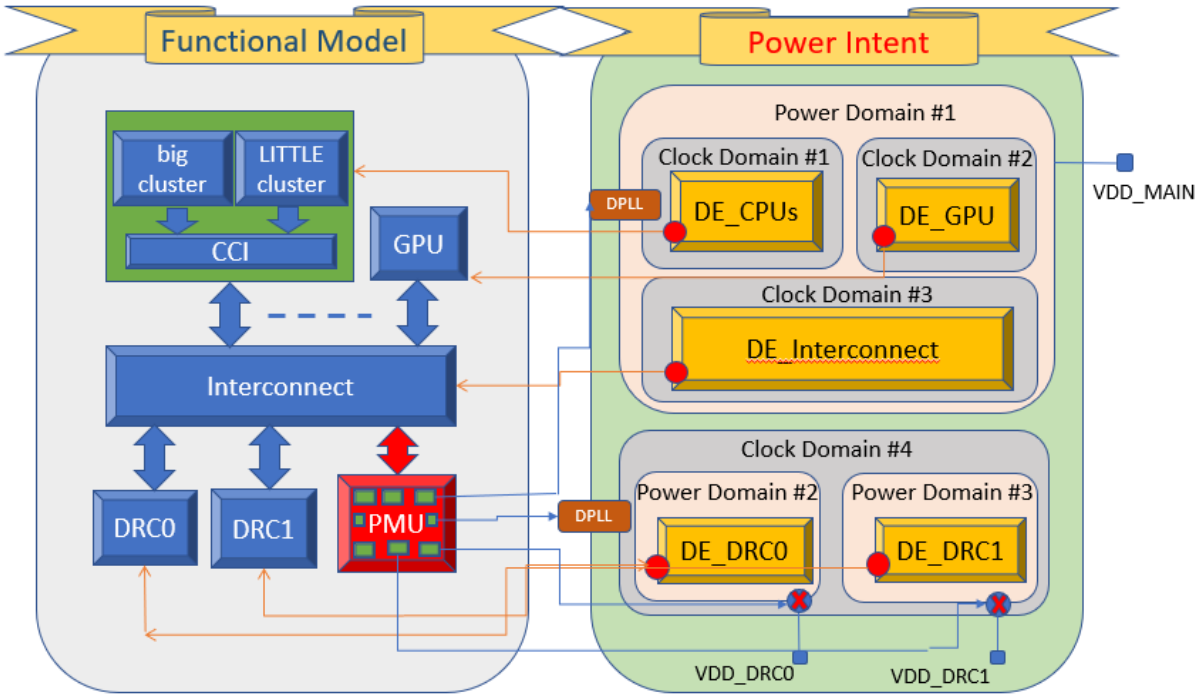


Figure 3.13: Example of PwClkARCH semantics

3.4.3 Power/Clock intents & management - user/designer view

The classes defined in the PwClkARCH library provide an API to control its state changes from a functional module via TLM transactions. To control the power/clock intents according to a power management strategy, a **Power Management Unit (PMU)** (Figure 3.13) must be added to the functional model. It establishes the connection between the functional and power models by adjusting the voltages and clock states of the components according to their current operating point. The PMU contains many elements necessary for power management (described later in this section).

Methodology

The methodology behind PwClkARCH can be divided into 4 phases.

- ***Power and Clock intent specification stage*** - in this step, architects analyze the behavior of the device and define the possible power intents based on the activity of the different components. They choose the power structures that are worth testing and divide the design into several power and clock domains. At this point, all necessary components are instantiated, such as DPLLs, generated and external clocks, supply networks and power switches.
- ***PMU modeling stage*** - at this point, we define the module establishing the connection between the functional and power models. This module acts as a relay and switches between power and clock modes based on requests (TLM transactions) from an initiator module in the functional model. This initiator module can be any processing unit or it can also be a specific controlling module defined for the purpose of power management control.

- ***Full Power-aware simulation stage*** - once we have defined the power intent and modeled the PMU, we are ready to simulate the complete architecture composed by the functional model and its power model. During the simulation, several power and clock/activity values are automatically extracted to binary data log files that can be plotted and analyzed. All architecture options can be simulated and compared in order to choose the optimal solution.
- ***Verification stage*** - the verification step is conducted in parallel with the three previous steps. It allows us to verify that the simulations were successful and that the power intentions and power management strategies do not generate functional design errors. This step is based on an assertion mechanism that verifies the power and clock management properties during the simulation. It ensures the correctness of the model by asserting different clauses, such as a warning if an activity is captured in an idle block, or indicating if components of the power intent are missing or misconnected. An exception is thrown if a clause is violated and an error report is written to a report file, called the Design Error Report File (DERF).

These steps are illustrated on Figure 3.14.

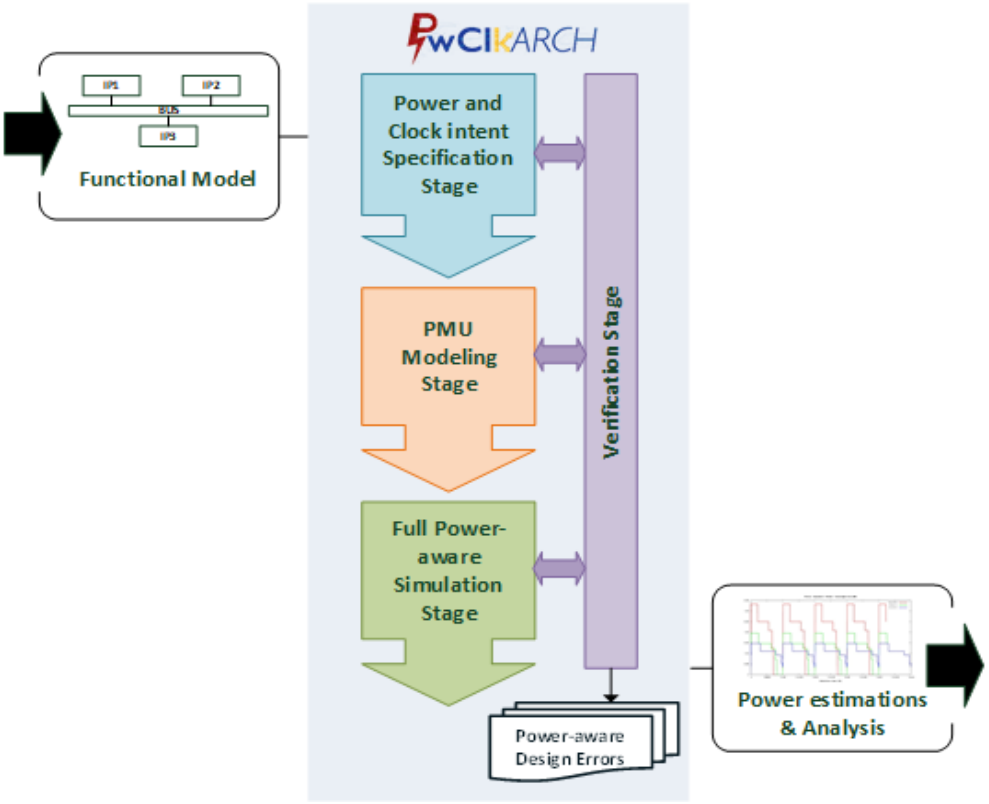


Figure 3.14: PwClkARCH modeling flow [5]

Power intent structure

Figure 3.15 shows the components needed to define and configure the power and clock intent. The instantiation of these components is done in the main/top file of the project.

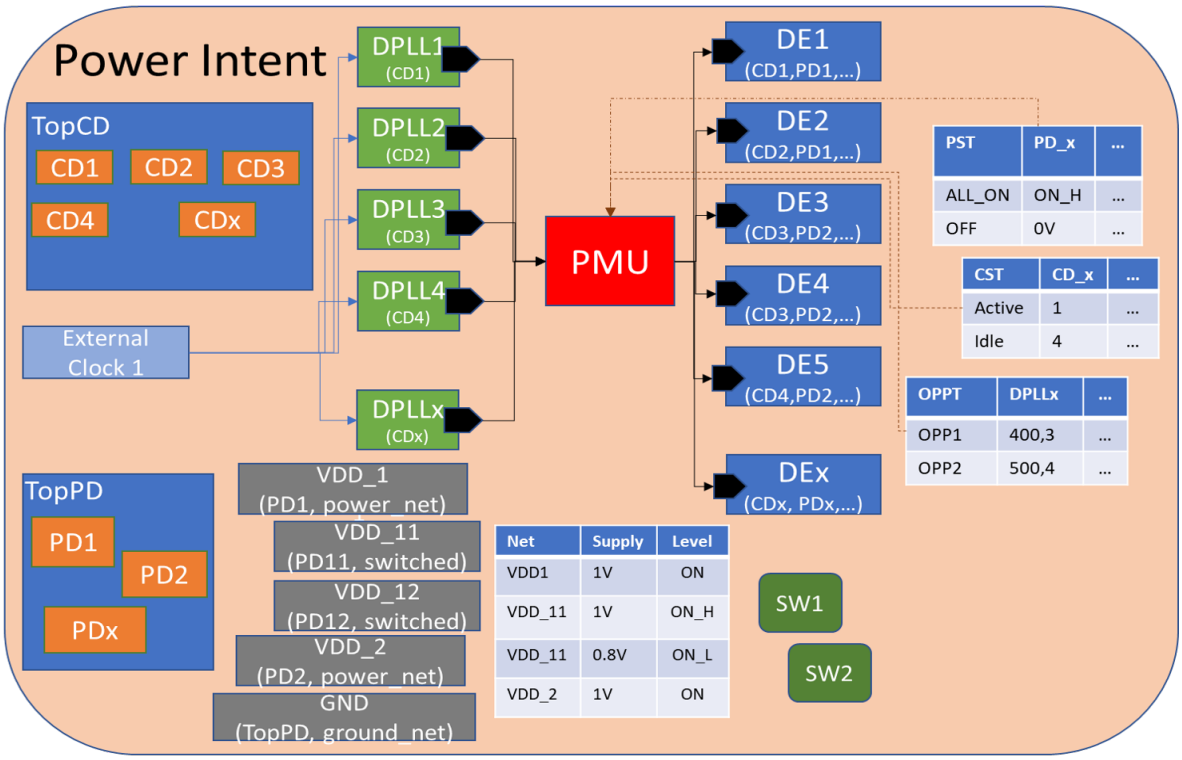


Figure 3.15: Power Intent specification

A brief presentation of each element is made here in the order of declaration.

- **Clock Domain (CD)** - following the definition given in section 3.3.2, we divide the architecture into several clock domains clocked by synchronous or constant phase clock signals. A pair of a DPLL and a CM is associated with each CD. The DPLL provides a clock signal to the CM, which reconfigures it and passes it to each individual block in the given clock domain. As already mentioned in the clock domain definition, the CM allows us to use a division factor of the input clock and pass the result to the IP. An example of clock domain instantiation is given in Listing 3.1.

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    // STEP 1.1: CREATE CLOCK DOMAINS - (container, name, scope)
    Clock_Domain* TopCD = new Clock_Domain(NULL, "TopCD", "TOP");
    Clock_Domain* CD1 = new Clock_Domain(TopCD, "CD1", "TOP/CD1");
    Clock_Domain* CD2 = new Clock_Domain(TopCD, "CD2", "TOP/CD2");
    ...
}
```

Listing 3.1: Clock domain example

There is always a top level *topCD* clock domain containing all the others. This is the only possible hierarchy in the clock domains structure.

Once all the CDs are instantiated, we can add their corresponding DPLLs (Listing 3.2).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 1.2: CREATE DPLLs - (clock domain, name, penalty delay in
    ns )
    DPLL* DPLL1 = new DPLL(CD1, "DPLL1", 2);
    DPLL* DPLL2 = new DPLL(CD2, "DPLL2", 2);
    ...
}

```

Listing 3.2: DPLLs example

Similarly, we can declare the external clocks (Listing 3.3).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 1.3: CREATE Clocks - (name, type, DPLL, frequency [Hz])
    Clock_Externe* CLK1_R = new Clock_Externe("CLK1_R", "reference",
    DPLL1, 24*(std::pow(10,6));
    Clock_Externe* CLK1_B = new Clock_Externe("CLK1_B", "bypass", DPLL1
    , 32*(std::pow(10,3));
    ...
}

```

Listing 3.3: External clocks example

- **Power Domain (PD)** - There is always a top level power domain *topPD* containing all the others. The power domains can be hierarchical (ex. *PD11* in Listing 3.4).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 2.1: CREATE POWER DOMAINS - (container, name, scope)
    Power_Domain* TopPD = new Power_Domain(NULL, "TopPD", "top");
    Power_Domain* PD1 = new Power_Domain(TopPD, "PD1", "top/PD1");
    Power_Domain* PD11 = new Power_Domain(TopPD, "PD11", "top/PD1/PD11"
    );
    Power_Domain* PD2 = new Power_Domain(TopPD, "PD2", "top/PD2");
    ...
}
```

Listing 3.4: Power domains example

Each power domain is powered by a Supply Net (SN), which can feed several PDs, but allows us to control them separately.

An SN can be of type *"power_net"*, *"ground_net"* or of type *"SWITCHED"* (Listing 3.5).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample {
    ...
    // STEP 2.2: CREATE Supply Nets - (name, type, PD)
    primary_supply_net* GND = new primary_supply_net("GND", "ground_net
    ", "topPD");
    primary_supply_net* VDD1 = new primary_supply_net("VDD1", "
    power_net", "PD1");
    primary_supply_net* VDD11 = new primary_supply_net("VDD11", "
    SWITCHED", "PD11");
    primary_supply_net* VDD2 = new primary_supply_net("VDD2", "
    power_net", "PD2");
    ...
    PD1->set_domain_supply_net(VDD1,GND);
    PD11->set_domain_supply_net(VDD11,GND);
    PD2->set_domain_supply_net(VDD2,GND);
    ...
}
```

Listing 3.5: Supply Nets example

Power gating can be applied to SNs using Power Switches (SW) whose declaration is presented in Listing 3.6. The SN is of type *"SWITCHED"* if it is the output of a power switch, otherwise it is of type *"power_net"* (except the ground).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample {
    ...
    // STEP 2.3: CREATE Power Switch - (PD, name)
    Power_Switch* SW1 = new Power_Switch(PD11, "SW1");
    (SW1->_input_supply_nets).push_back(VDD1);
    SW1->SetOutput_supply_net(VDD11);
    ...
}
```

Listing 3.6: Power Switches example

In order to specify supply voltage values, we must define different states for the supply nets, as in Listing 3.7.

```
template <typename FUNCMOD_NAME>
class PowerIntentExample {
    ...
    // STEP 2.4: CREATE Supply Net States - supply voltage [V]
    GND->net_valid_states["OFF"]=0.0;
    VDD1->net_valid_states["ON"]=1.0;
    VDD11->net_valid_states["ON"]=1.0;
    VDD11->net_valid_states["OFF"]=0.0;
    VDD2->net_valid_states["ON_H"]=1.0;
    VDD2->net_valid_states["ON_L"]=0.8;
    VDD2->Set_Net_Voltage_Scaling_Penalty(100.0);
    ...
}
```

Listing 3.7: Supply Net States example

In this example, if the voltage of *VDD2* changes (between the states *"ON_H"* and *"ON_L"*), a time penalty of *100ns per mV* is applied until the transition to the target voltage is complete.

- **Design Element (DE)** - the concept of DE has also been adopted in PwClkARCH from the UPF standard. Design Elements can be compared to "shadows" of IPs in the functional model. Each IP instantiated in the platform and included in a Clock/Power domain is associated with a DE. The parameters needed for the power calculation, such as capacitance and leakage current (leakage resistance), are specified during the instantiation of the Design Element (Figure 3.16). The power/clock specification is built on top of the DEs (not the functional models), in order to define the power intent. In this way, the separation between the hardware design and the power/clock intent is enforced. The DE block is responsible for computing the generic static and dynamic power equations for its corresponding IP. Whenever there is a state change, an update of the power calculation is performed at the IP level and then transferred to the upper platform level. Thus, at the end of the simulation, we obtain the power consumed per module, per clock domain, per power domain and the total power of the hole platform.

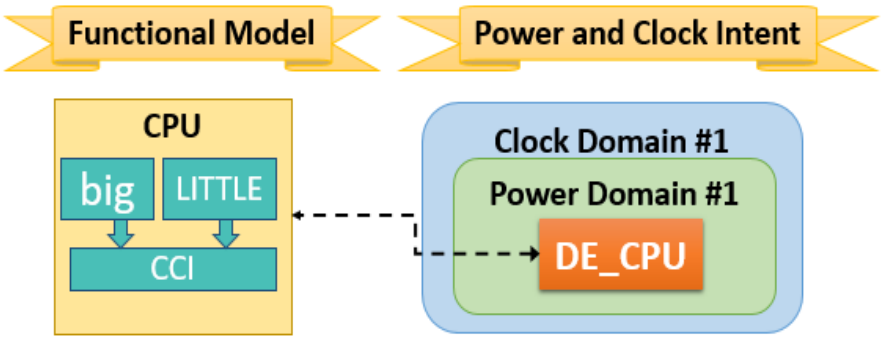


Figure 3.16: Example of Design Element instance

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 3: Create Design Elements - (CD, PD, ptrFuncMod,
    capacitance, initial activity, leakage resistance)
    Design_elem* DE_CPU = new Design_elem(CD1, PD1, (top->ss_cpu),
    CPU_CAPACITY, CPU_ACTIVITY, CPU_LEAK_RESISTANCE);

    Design_elem* DE_MEM = new Design_elem(CD2, PD2, (top->ss_dram),
    DRAM_CAPACITANCE, DRAM_ACTIVITY, DRAM_LEAK_RESISTANCE);
    ...
}
```

Listing 3.8: Design Elements example

- **Clock State Table (CST)** - The CST serves to set a frequency value to each block for a specific clock state in the executed scenario. The goal is to dynamically change the values of the clock frequencies applied to DEs. The Clock state table takes as

parameters the source clock inputs from the Design Elements (clock signals coming from Clock Managers). We can add several clock states and for each one of them we need to set a frequency divider value to all DEs clock inputs. The frequency divider is applied to the reference clock distributed by the CM of the given CD for each IP (or its DE, to be more concrete). Logically, when the value of the frequency divider is zero, the clock is gated.

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 4: Create Clock State Table - (name, CD, nbDEs, div_fact1, div_fact2, ...)
    S2.create_clkst("CST_top",topCD,"3","CM1_DE_Bus","CM1_DE_SRAM","CM1_DE_CPU"); //line 0
    S2.add_clkst_state("CLKST_top","boot","3",    "16","32","8"); //line 1
    S2.add_clkst_state("CLKST_top","low_active","3",  "8","4","2"); //line 2
    S2.add_clkst_state("CLKST_top","cpu_active","3",  "8","4","4"); //line 3
    ...
}

```

Listing 3.9: Clock State Table example

The first line (i.e. line 0) of the CST lists the names of the generated clocks (from the CMs) connected to the DEs. They should be arranged in CDs (one after the other). The other lines are the clock states that can be taken in the specific scenario and the division factors that should be applied to each DE clock for the given state.

- **Power State Table (PST)** - The PST serves to set a voltage value to each supply net for a specific power state in the executed scenario. The goal is to dynamically change the values of voltage of the supply nets and the states of the power switches attached to the PDs (same as UPF).

```
template <typename FUNCMOD_NAME>
class PowerIntentExample {
    ...
    // STEP 5: Create Power State Table - (name, PD, nbNets, netState1, netState2, ...)
    S1.create_pst("PST_top",topPD,"3", "VDD1","VDD11","VDD2"); //line 0
    S1.add_pst_state("PST_top","all_on","3", "ON", "ON", "ON_H"); //line 1
    S1.add_pst_state("PST_top","cpu_off","3", "ON", "OFF", "ON_L"); //line 2
    ...
    //Transitions - (pst_name, from, to)
    PSTrans* Trans = new PSTrans("PST_top","*","*");
}

```

Listing 3.10: Power State Table example

The first line contains all the supply nets in the power intent. The other lines contain the power states with the valid supply net states for each supply net.

We can also specify the legal transitions between power states. All transitions that are not specified here are illegal. The "*" character denotes any power state specified in the PST. Thus, in this example all transitions are legal.

- **Operating Performance Point Table (OPPT)** - The OPPs are used to manage the overall power states of the entire system. From here, we can dynamically control the power and clock domains as well as the DPLL states. This table provides a set of available operating points.

```
template <typename FUNCMOD_NAME>
class PowerIntentExample
{
    ...
    // STEP 6: OPP Table - (name, CD, PD, nbdplls, opState1, opState2, ...)
    S2.create_OPP_table("OPPtable", topCD,topPD,"3","DPLL1","DPLL2","index");
    S2.add_OPP_state("OPPtable","boot","3", "8,1","4,8","1,1"); //OPP1: PD: "all_on"; CD : "boot"
    S2.add_OPP_state("OPPtable","cpu_off","3",  "0,0","4,8","2,2");//OPP2: PD: "cpu_off"; CD: "low_act"
    ...
}

```

Listing 3.11: Operating Performance Points Table example

In the first line we list all DPLLs in the design. The last column of this line must contain the string *"index"*. The following lines correspond to the possible OPP states. For each state, we give a pair of multiplication and division factors for each DPLL. In the last column of each OPP state, we have a index 2-tuple. The first value in the 2-tuple is the index in the PST of the state of power domains and the second value of the 2-tuple is the index in the CST. Applying an OPP to the architecture results in specifying voltage values on supply nets, states of power switches and the clock frequencies of DEs.

- **Power Management Unit (PMU) instantiation** - The PMU is instantiated in the power intent and connected to a system master block that sends transactions to the PMU and implements the power management strategy. The PMU is instantiated by:

```
template <typename FUNCMOD_NAME>
class PowerIntentExample {
    ...
    // STEP 7: PMU instance - (name, vpAll_PDs, pPST, vpAll_CDs, pCST, vpAll_DPLLs, pOPPT)
    PMU pmu("PMU", topPD->power_domains_list, PST::get_PST("PST_top"), topCD->clock_domains_list, ClkST::get_ClkST("CST_top"), DPLL1->dpll_list, opp_table::get_opp_table("OPPtable");
    ...
}

```

Listing 3.12: PMU instantiation example

The arguments are vectors of pointers to all PDs, CDs, DPLLs, and three pointers to PST, CST and OPPT. The Master block (initiating transactions to the PMU) can have different forms. It can be any initiator module (CPU, GPU, HW Accellerators ...) initiating transactions to a PMU connected to the HW transactional bus, or using the approach that we have defined and used it can be an isolated Master block receiving simple control signals from initiators and communicating only with the PMU. Figure 3.17 is a simple example of how the architecture will look for the first case:

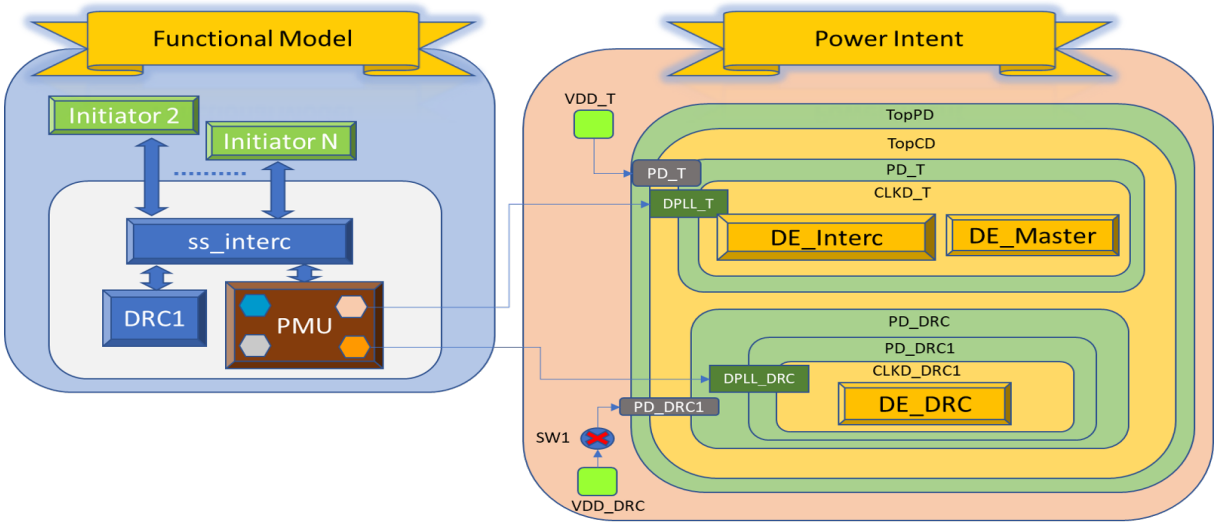


Figure 3.17: Example of Power-aware architecture without Master module

In the second case, it is not necessary to connect the PMU to the functional model. In this way, the power-related code in the functional initiator modules is considerably reduced and the power management strategy is applied in a single module. The master module and PMU are responsible for the entire power management. Figure 3.18 is a simple example of how the architecture will look for the second case:

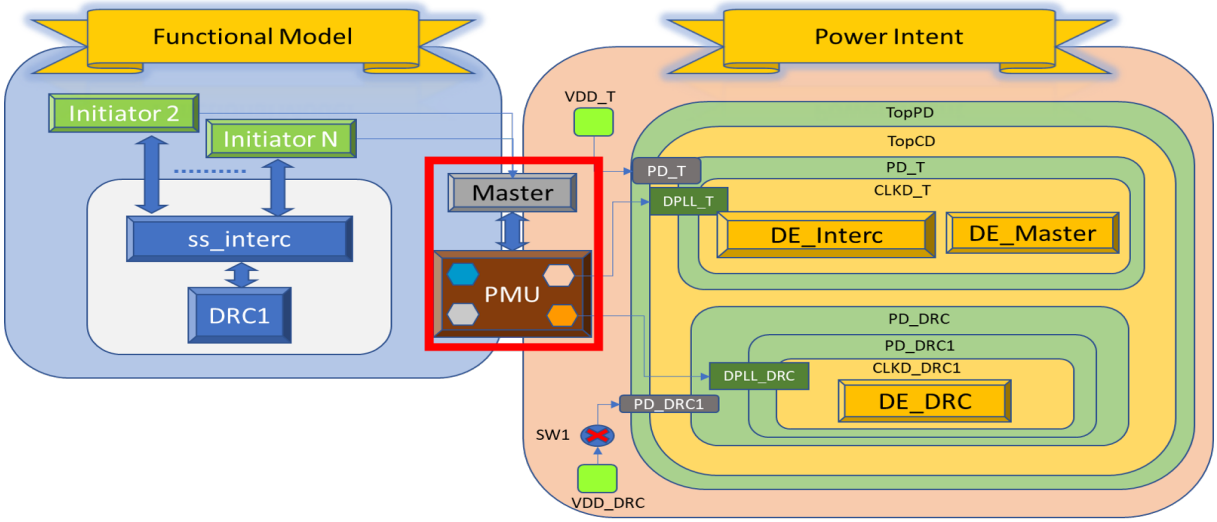


Figure 3.18: Example of Power-aware architecture with Master module

PMU structure

The PMU contains ”copies” of the CST, PST and OPPT plus several important blocks and signals responsible for the automated power and clock management.

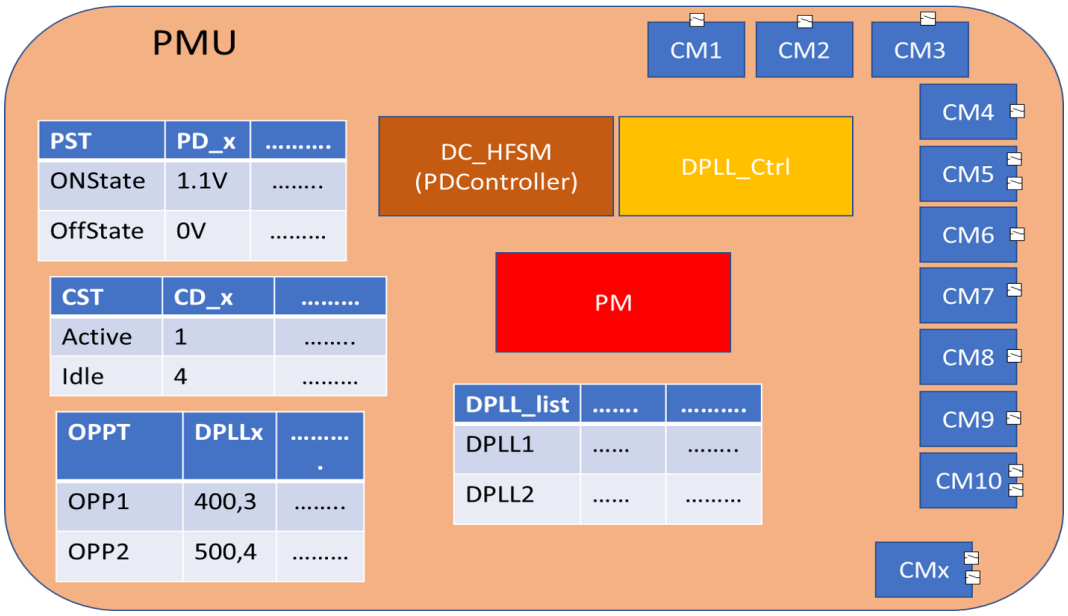


Figure 3.19: Example of PMU structure

- **Power Manager (PM)** - this block is the primary power management component and is the block that contains the TLM socket and provides the interface to the master block. It includes all function calls to the power/clock intents and is responsible for implementing the power management strategy defined in the master. It interprets requests from the master module and automatically executes the necessary actions.
- **Power Domain Controller (PDC)** - this block is responsible for controlling the power domains. It is activated by the PM block whenever a supply network update operation is performed in the power domain. The number of PDCs in the PMU is equal to the number of power domains including a power switch in the architecture.
- **DPLL Controller** - there is a single DPLL controller for all the DPLLs in the architecture. It receives the configurations of the DPLLs from the PM and executes them to update their states.
- **Clock Managers (CMs)** - There are as many CMs as there are clock domains in the architecture. They are activated by the PM when an operation has to be performed on a clock (one or more clocks). This operation is applied on the clock inputs of the DEs.

Master/Initiator implementation

As mentioned earlier, the Master module can be implemented in multiple ways. We have given two examples of how the block can be integrated into the structure. Regardless of the approach chosen, there are mandatory fixed implementations (included in the PwClkARCH library).

Each Master block coordinates the power management strategy based on the application activities. In a given strategy, when the power/clock state needs to be changed, the Master sends first a transaction to the PMU to activate the new OPP (ex. deactivate/activate different modules). In this way, we can apply the multiplication and division factors on the different clocks and readjust the voltage supply with a single update of the OPP state. Thus, the master module can be considered as a power supply control unit with different switching statements. If a condition is met, then change the OPP.

A simple functions *write_mem()*/*read_mem()* is used to send write/read transactions to the PMU with the *b_transport()* method via a *simple_initiator_socket*. The *write_mem()* function takes as parameters an address and a data. The address indicates the command that we want the PM to execute and the data may indicate different things depending on the command. The Power Manager accepts a relatively large set of commands that allow a master to set a power state to the functional model. A command corresponds to an address offset added to the base address of the Power Manager.

The first most commonly used command is *PM_STATUS_REG* which indicates a change to a new OPP whose value is a line number in the OPP Table passed as data in the transaction.

The second command that we use is *PM_ENABLE_AUTO_CLK_GATING* which activates the **auto-clock-gating mode** at DE level according to the integer value transmitted in the transaction (as a data argument), one bit per DE, in the order of creation of the DEs. In the case of Auto-clock gating, the functional module executes the function *set_functional_state (true/false)*, i.e. when it is in the idle/active state and if the associated DE is enabled for auto clock gating, the clock manager automatically gates/activates its clock. Listing 3.13 shows how the *write_mem()* looks like:

```
1  void write_mem(unsigned int addr, int data)
2  {
3      sc_time delay;
4      tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
5
6      trans->set_command(tlm::TLM_WRITE_COMMAND);
7      trans->set_address(addr);
8      trans->set_data_ptr(reinterpret_cast<unsigned char*>(&data);
9      trans->set_data_length(4);
10     trans->set_streaming_width(4);
11     trans->set_byte_enable_ptr(0);
12     trans->set_dmi_allowed(false);
13     trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
14
15     socket->b_transport( *trans, delay);
16
17     tlm::tlm_response_status trans_status;
18     trans_status = trans->get_response_status();
19     if (trans_status == tlm::TLM_OK_RESPONSE && delay > SC_ZERO_TIME)
20         wait(delay);
21     delete trans;
22 }
23
```

Listing 3.13: write_mem() definition

For example, the transaction *write_mem(PM_STATUS_REG, 0x1);* asks the PMU to set up the OPP1 (0x1 is the first line in the OPP table, so it is the OPP1 state) in the architecture.

The transaction *write_mem(PM_ENABLE_AUTO_CLK_GATING, 0x1000)* activates the auto-clock gating mode for the fourth DE created in the power intent structure. From there, the functional module to which this DE corresponds, reports its functional state to the PMU using the function *de->set_functional_state(state)* where *"de"* is a pointer to its design element and "state" is either true or false. True means that the functional unit is in the IDLE state while false means that it is in the *ACTIVE* state. Changing the functional state to false (resp. true) enables (resp. disables) the clock to the unit. This is only necessary for the modules that are in auto-clock gating mode. The other modules are controlled by the master and the OPP. The rest of the Master module implementation depends on the designer's choice and the power management strategy.

SystemC/TLM modules activity

The internal activity factor of each SystemC/TLM functional module is very important for dynamic power consumption. It depends on the functional state of the module, which means that when the module is active, it can have a much higher activity factor than when it is in idle mode. For this reason, it is important to consider this factor and make it as accurate as possible. The range of this factor is [0;1]. In PwClkARCH, this can be done by adding the function *setActivity(float activity)* (Listing 3.14) to each functional module, and having it track the active elements in the module. Each time the *setActivity(float activity)* function is called in a module, the current activity rate is provided to the DE and its dynamic power consumption is updated. This update is then performed at the PD and CD level (where the DE is located).

```
1 void SetActivity(float activity)
2 {
3     sc_core::sc_object& obj = dynamic_cast<sc_core::sc_object&>(*this);
4     Design_elem* de=Design_elem::get_DE(obj);
5     de->Update_dpow(activity);
6 }
```

Listing 3.14: setActivity() function definition

An important and very intuitive question comes to mind:

How do we adjust this activity factor?

This largely depends on the information we have. The initial approach that can be used is to use 0 and 1 as the activity factor. To do this, we need to make maximum estimates of the value of the load capacity. Then we can try to estimate the number of transistors that a given system may have and the blocks that are activated when transactions pass. In this way, we can estimate a ratio of active blocks for a transaction. We can then track each block in the system model and automate the calculation of the average value of the activity factor. In later steps, we can use the toggle ratio defined by the physical engineers for more accurate estimates.

However, this function is not mandatory and if we do not need a dynamic value for the activity factor, the library uses the fixed initial activity passed as an argument when instantiating the DE.

Consistency between power states and functional states

In order to maintain consistency between functional and power states, PwClkARCH uses an optional assertion mechanism. This mechanism uses a power observer class attached to the functional module. It is usually performed in the block model constructor. There are three types of tests that can be enabled/disabled in the power observer - AssumeEnabled, GuaranteeEnabled and SatisfyEnabled.

```
class Bus: public Subject, public Assertions
{
    Bus(sc_module_name name, protocol::bus_type protocol, sc_time clk)
    {
        ...
        sc_object& obj = dynamic_cast<sc_object&>(*this);
        power_observer = new Observer<sc_object>(obj,"pow_ob");
        this->attach(*power_observer);
        AssumeEnabled=true;
        GuaranteeEnabled=true;
        SatisfyEnabled=true;
        ...
    }
}
```

Listing 3.15: Power observer

The Subject class contains the method *notify()*, which is used to notify the power observer whenever a read or write transaction is received. The observer checks the state of the clock and verifies whether the module is active when the transaction is received. If it is not, an error is triggered in the error log file.

Results visualisations and reports

At the end of the simulation, two gnuplot [103] scripts (and multiple binary files) are generated to plot the results provided by the simulation.

These scripts allow us to view the different results separately (system by system) or all at once. The first script contains all the extracted metrics related to the clock frequency and activity, the supply voltage and the OPP state transitions. The second contains all power-related metrics, such as total, dynamic and static consumption and energy consumption. Some examples are shown in figures 3.20 and 3.21.

Using PwClkARCH, the power intents and power management strategies are very easy to modify in order to test different options. We can easily modify the OPP, PST or/and CST tables by adding, deleting or replacing states. In addition, the Master module can be easily modified to change the power management strategy and the OPP activations. Furthermore, we can change the power and clock domains fairly easy and test different structures. None of these modifications require changes in the SystemC/TLM functional models. Thus, the study of power strategies is greatly simplified and can be evaluated in conjunction with system performance.

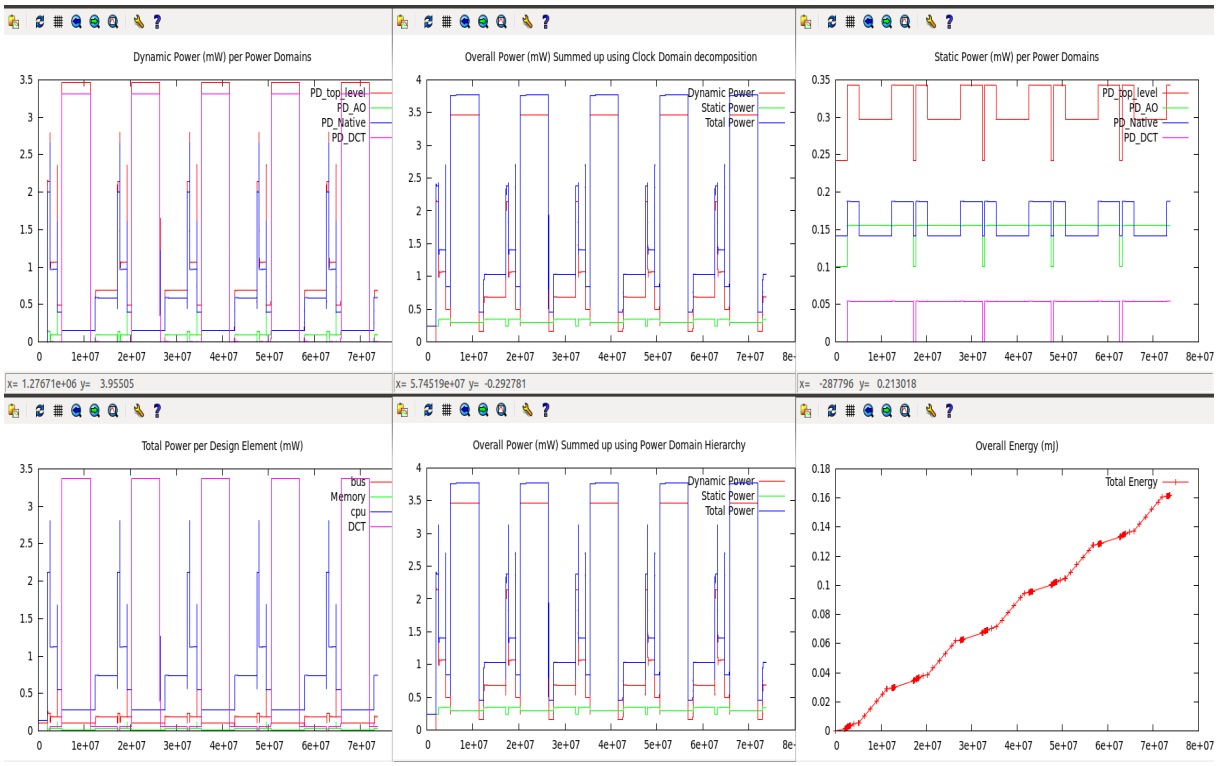


Figure 3.20: Power plots examples

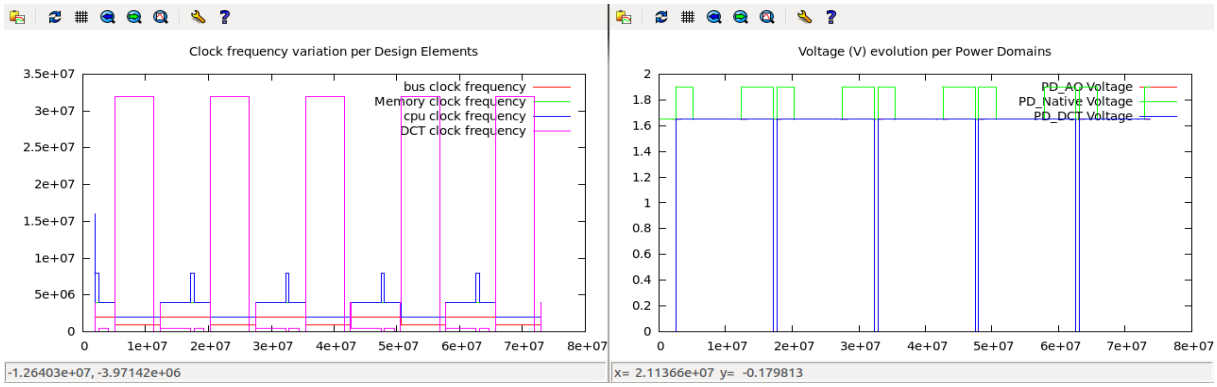


Figure 3.21: Clock plots examples

3.4.4 Model-Driven Engineering - TUI and GUI

PwClkARCH also has a model-based layer built on a Model-Driven Engineering (MDE) approach. It further simplifies the specification of the power/clock intent and the definition of the energy management strategy. *The GUI and TUI are not part of this study and were not used for our model generation. However, they are a very useful feature worth mentioning here in the semantics of PwClkARCH.* Almost all the semantics mentioned above can be generated automatically based on the UML approach presented in [104]. A graphical modeling tool built on Eclipse with the Sirius and Acceleo plugins allows to graphically and/or textually describe the power model and to automatically generate the SystemC/TLM code corresponding to the top level of the DUT model. It is an easy-to-use framework that allows to generate the entire power/clock intent and PMU module from a single metamodel [105], [106], [107], [108]. The code is then included in the hardware architecture model. As men-

tioned at the beginning of this section, the TUI and GUI were not used in the development of our models, and the following sections will instead focus on a coding framework.

3.4.5 Summary and Comparisons

The various academic and industrial solutions do not offer us all the mechanisms that PwClkARCH can provide. Some of them are based on static approaches that require a lot of human effort, others do not allow the use of power reduction techniques or at least not outside the functional models. The majority require a lower level of abstraction and a large part of them are based on power state machines, where we manually associate the energy consumption to each state. PwClkARCH uses a different approach based on relatively accurate information about the area and activity of the tested IP. In Table 3.1 we have made a comparison between the most commonly used industrial tools and the PwClkARCH library.

Table 3.1: Comparison table

Tool	Simulation model	Power estimation	Power management	Thermal impact
Platform Architect	SystemC-TLM	YES	NO	NO
Intel CoFluent Studio	SystemC-TLM	NO	NO	NO
Intel Docea Power	SystemC-TLM	YES	YES (MANUAL-no func/pow co-sim)	YES
Mentor Graphics/Siemens Vista	SystemC-TLM	YES	NO	NO
PwClkARCH	SystemC-TLM	YES	YES	NO

The major drawback of the PwClkARCH library is the fact that we cannot consider the thermal impact on power consumption directly through the simulations. Since component temperature is an important topic that can lead to very significant static consumption values, it deserves to be included in our power estimation methodology. Since in this study we are only ”users” of the PwClkARCH library and our goal is to evaluate this approach, this topic is outside the scope of our study and can be added to the possible improvement perspectives.

3.5 Technology used in this study

3.5.1 Introduction to NXP i.MX SoC family

NXP's i.MX family members are ARM-based, multimedia-oriented microcontrollers. These low-power SoCs integrate a wide variety of analog and digital modules and processing units, such as clusters of multiple Core Processing Units (CPUs), Video Processing Units (VPUs), Graphics Processing Units (GPUs), Display Controllers (DCs) and other advanced functions.

Figure 3.22: NXP i.MX family applications processors [6]

The wide architectural variety of i.MX members (Figure 3.22) allows customers to make the right choices and trade-offs depending on the intended application. These SoCs are constantly being improved and keep up with all the latest technologies. Thus, they have many important customers such as Ford, Sony, Amazon, Garmin, Logitech, Toshiba and many other market leaders.

- The first member of the i.MX family released on the market in 2001 is the ARM920T based i.MX1 series, which was designed for personal digital assistants (PDAs).
- The first i.MX members targeting the automotive market was the ARM11 based i.MX3 series released in 2005, followed by the ARM Cortex A8 based i.MX5 (in 2007) and the ARM Cortex A9 based i.MX6 (in 2011).
- The i.MX6 series offered a scalable and functional multi-core platform that includes single, dual and quad-core structures.
- The next series was the i.MX7, which has a similar architecture to i.MX6, but they are oriented towards IoT (Internet of Things) applications.
- Another more IoT-oriented series is the i.MX RT used for real-time functionality requiring high computing power and multimedia capabilities.
- The i.MX8 series are the ones that will interest us in this report. It is the new generation that is in production and pre-production and starts to be released on the market.

3.5.2 NXP i.MX8 series overview

Designed with advanced multimedia processing, secure domain partitioning and innovative processing vision, the i.MX8 series of application processors revolutionize multi-display automotive applications, industrial systems and vision. It is the current most powerful and elaborated multimedia solution compared to the other i.MX family members. The series includes combined solutions based on Cortex-A72, Cortex-A53, Cortex-A35 and Cortex-M4. The major i.MX8 series - i.MX8M (Figure 3.23), i.MX8X (Figure 3.24) and i.MX8 (Figure 3.25), can differs significantly from each other in terms of architecture and targeted application.

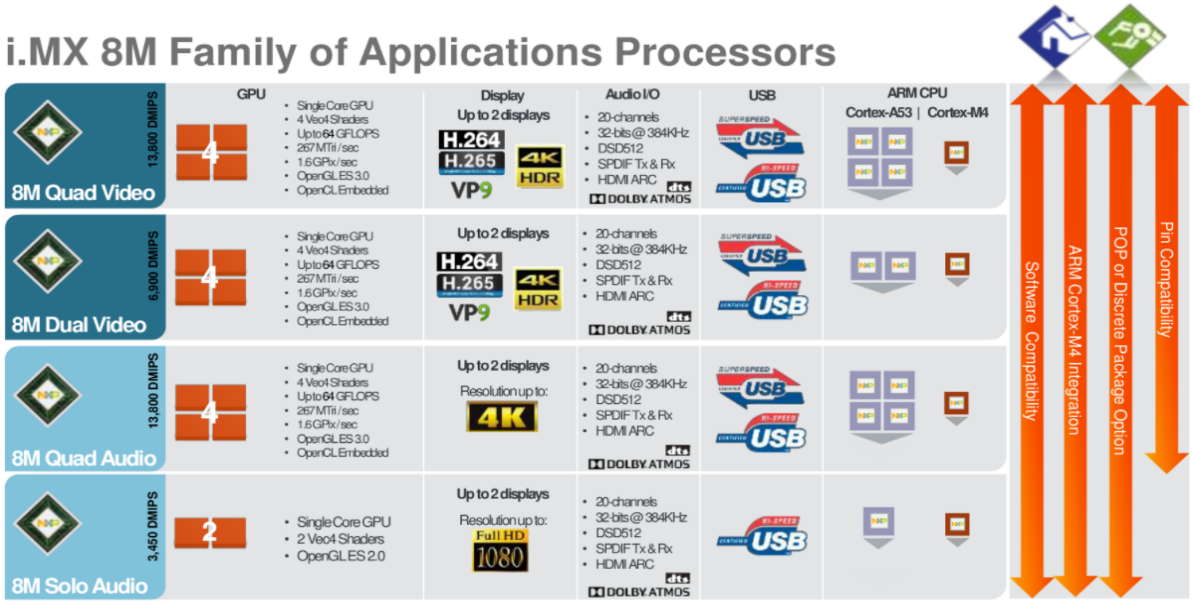


Figure 3.23: NXP i.MX8M serie applications processors

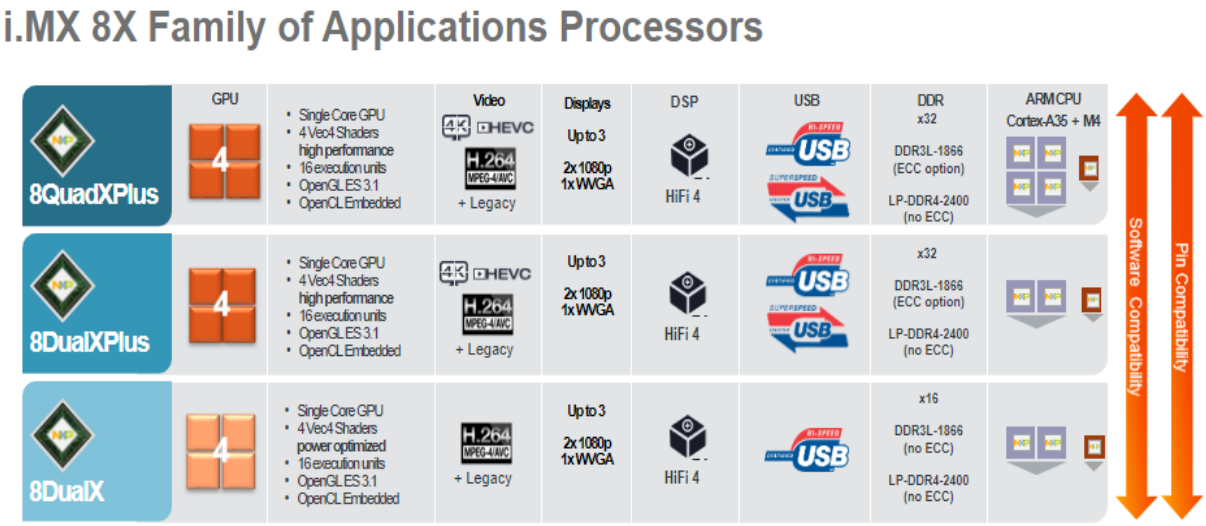


Figure 3.24: NXP i.MX8X serie applications processors

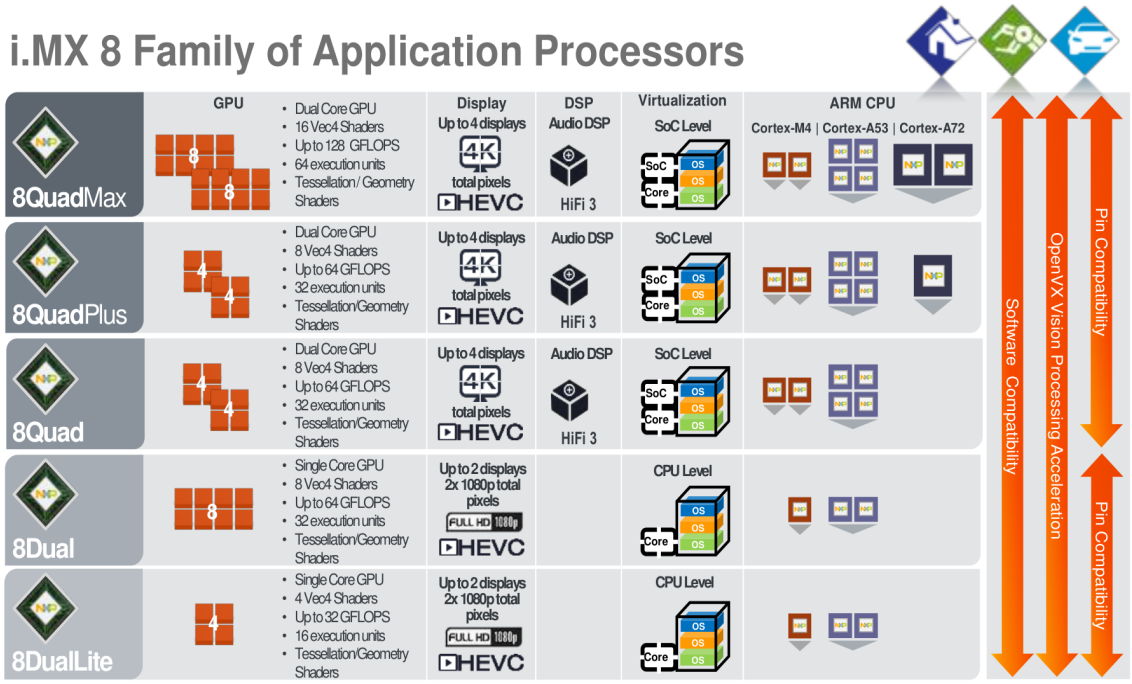


Figure 3.25: NXP i.MX8 serie applications processors

- i.MX8M serie - reinforces the ability of advanced audio, image and video processing
- i.MX8X serie - reinforces the ability of advanced graphics and efficient performance + safety certifiable
- i.MX8 serie - reinforces the ability of advanced graphics, efficient performance and virtualization.

In this study we are working with two platforms of the i.MX8 serie and one platform from i.MX8X. To be more precise, these platforms are the i.MX8QuadMax (QM), i.MX8nextGen and i.MX8QuadXPlus (QXP) respectively. They have very similar architectures, but different complexities and i.MX8nextGen is the new generation of i.MX8 series SoC.

3.5.3 NXP i.MX8QuadMax

i.MX8QuadMax is a fully comprehensive multimedia device built on 28nm FD-SOI technology to achieve both high performance and low power consumption. Automotive and industrial applications are its main market segments. The following diagram (Figure 3.26) highlights the IPs and peripherals present on i.MX8QM.

It is assembled around two ARM clusters using big.LITTLE 64-bit ARMv8 technology. The chip relies on a powerful, fully coherent complex core based on a dual Cortex-A72 cluster for use cases requiring high computational performance and a quad Cortex-A53 cluster for less power-hungry use cases where computational capacity is not paramount. This saves more power than using multiple processors of the same capacity in parallel, while maintaining high computational strength. Graphics processing is handled by two GPUs supporting the latest graphics APIs, including Vulkan and OpenVX for computer vision. Video is handled

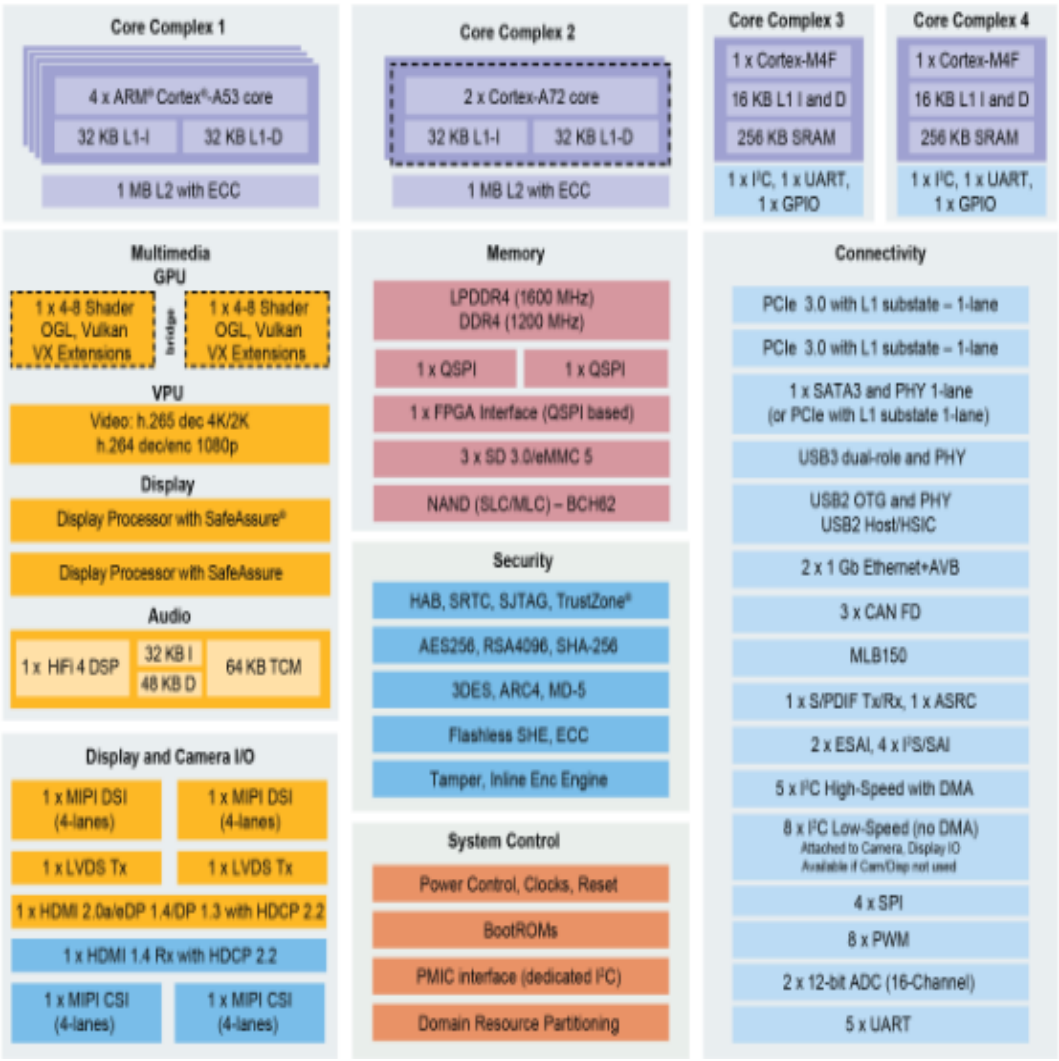


Figure 3.26: NXP i.MX8QM Diagram

by a dedicated video decoding engine for decoding formats including HEVC (H.265) up to 4K60 and H.264 encoding up to 1080p60. The device provides various display interfaces to connect up to four displays. To be able to deliver data to these demanding blocks, i.MX8QM has two DRAM controllers supporting DDR4 and LP-DDR4 memory types. Both DRAM controllers can deliver up to 25.6 GB/s peak bandwidth with 1600 MHz LP-DDR4 memory. i.MX8QM provides additional computing resources and peripherals:

- A dedicated **System Control Unit (SCU)** and a dedicated security subsystem that provides secure boot functionality and cryptographic acceleration.
- An audio subsystem with a wide range of audio interfaces
- Two general purpose Cortex-M4s with their own peripherals
- A large number of peripherals commonly used in the automotive and industrial markets

Figure 3.27 shows a typical platform that can be built around i.MX8QuadMax in the automotive domain:

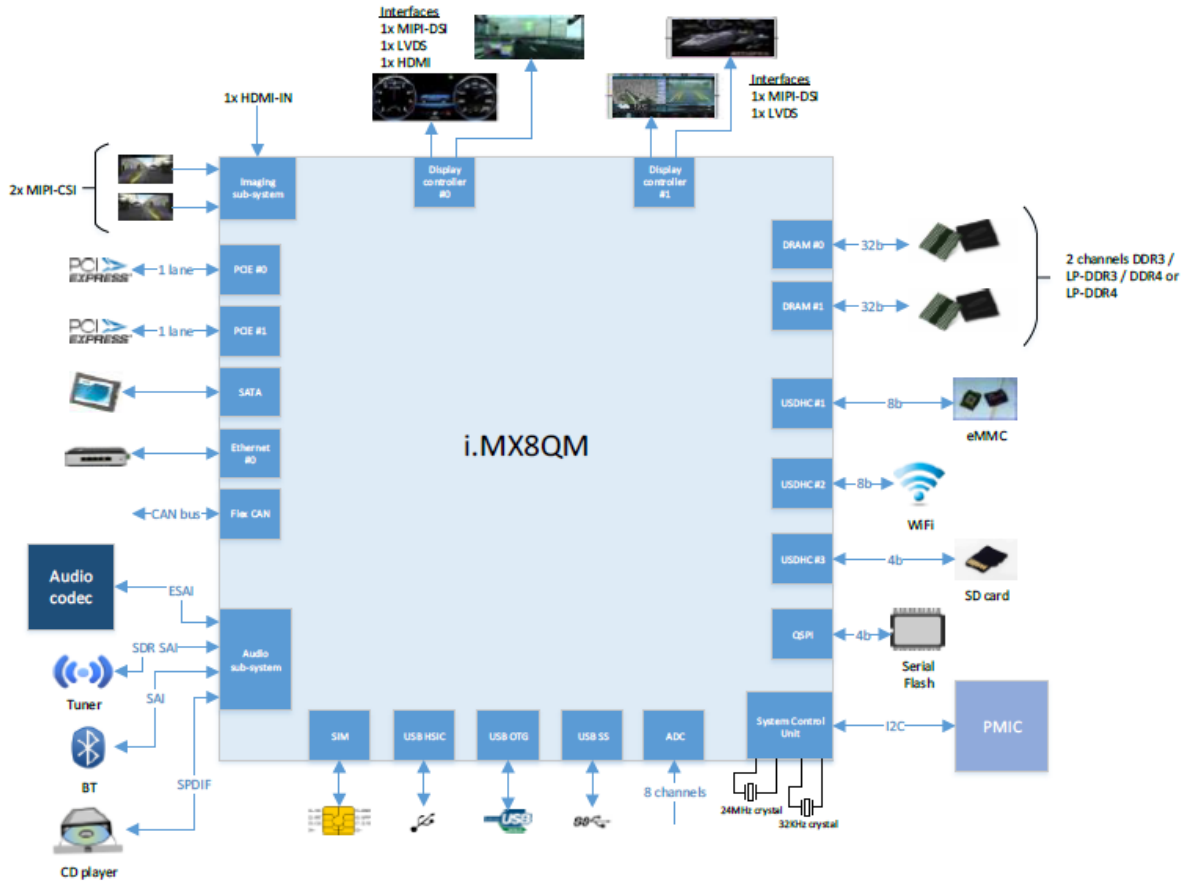


Figure 3.27: NXP i.MX8QM typical platform

3.5.4 NXP i.MX8QXP

i.MX8QXP is a fully comprehensive multimedia device targeting the mid-range automotive and industrial market segments. Similarly to i.MX8QM it is built in 28 FDSOI technology. Its computing power relies on a power efficient quad Cortex-A35 cluster providing full ARMv8-A and ARMv7-A support. Graphics processing is handled by single GPU supporting the latest graphics APIs. The video is handled by similar to i.MX8QM, but with lower FPS (30fps instead of 60fps). The device provides various display interfaces to connect up to three displays.

In order to feed data into those subsystems, i.MX8QXP embeds a single DRAM 32-bits channel running at 1066MHz and supporting DDR3 and LP-DDR3 types of memories. It provides a peak bandwidth of 8.5GB/s.
Same as i.MX8QM it provides similar additional computing resources and peripherals:

- A dedicated **System Control Unit (SCU)** and a dedicated security subsystem that provides secure boot functionality and cryptographic acceleration.
- An audio DMA subsystem with a wide range of audio and generic purpose interfaces
- A general purpose Cortex-M4 with its own peripherals
- A large set of peripherals commonly used in automotive and industrial markets

Figure 3.28 highlights the IPs and peripherals present on i.MX8QXP.

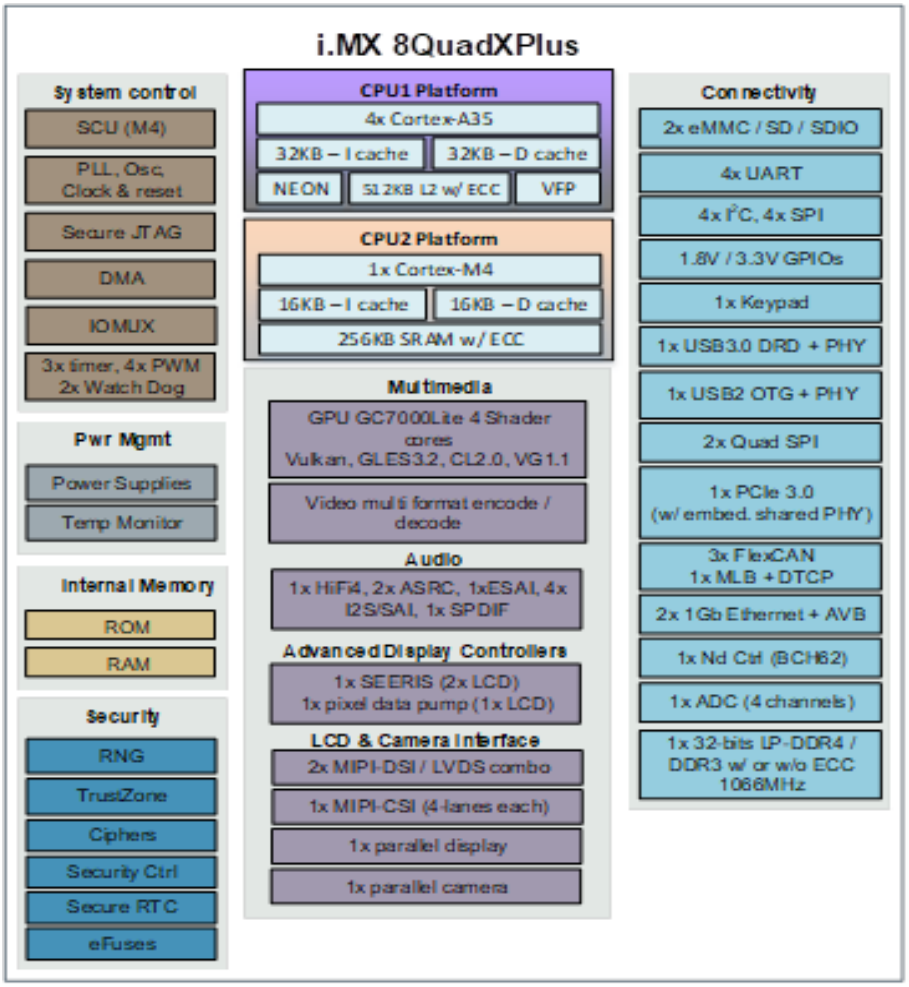


Figure 3.28: NXP i.MX8QXP Diagram

3.5.5 What's next?

The **i.MX8nextGen** devices will also be full-featured multimedia devices targeting the high-end automotive and industrial market segments. They will be built in smaller technology node to achieve both high performance and low power consumption. They will integrate up to four Dual Cortex-A72 clusters, two dual-core GPUs, two ISPs, one VPU, three imaging subsystems supporting up to six input streams each.

In order to feed data to these very demanding blocks, the i.MX8nextGen will likely have two DRAM controllers supporting LP-DDR4 and LP-DDR5 memory types. Both DRAM controllers are expected to be capable of delivering up to 34.1 GB/s peak with LP-DDR4(X) memory at 2133 MHz and 44 GB/s peak with LP-DDR5 memory at 2750 MHz. i.MX8nextGen is expected to provide the same set of computing resources and additional peripherals as i.MX8QM and i.MX8QXP. Platforms incorporating the i.MX8nextGen Quality Managed device should be certifiable with the appropriate ASIL level related to the use cases addressed by the item.

3.5.6 Common structures overview

Each of these platforms groups together modules that contain complex hierarchies (Figure 3.29). In order to facilitate understanding, we will give a common name to all the modules present on the chip (CPU, VPU, GPU, etc...). We will call them ***Subsystems***. All subsystems are composed of several hierarchical ***blocks/units*** and have a similar base structure. Each subsystem has:

- *A block that deals with the distribution and adaptation of the clock*
- *Different peripherals*
- *Interconnection blocks*
- *Blocks that adapt the transactions for communication with the memory or with the other subsystems*

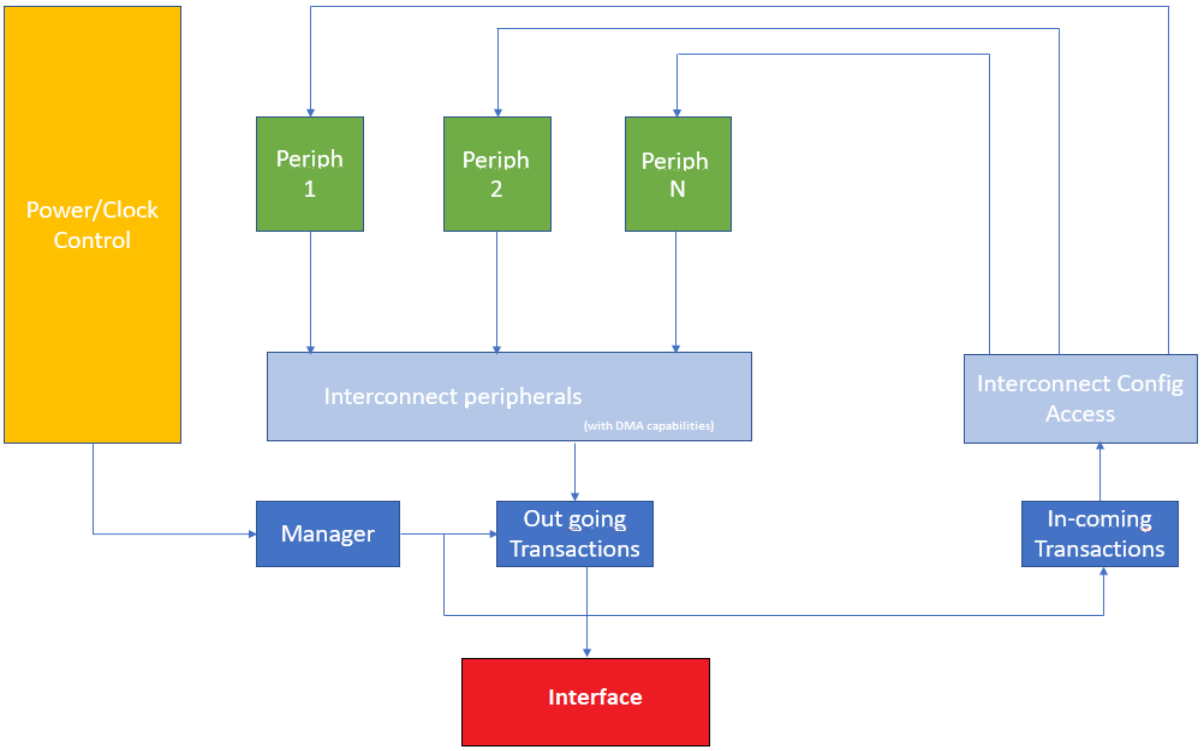


Figure 3.29: General subsystems structure

Each element of the subsystems contributes to the creation or adaptation of a transaction. Each transaction sent to a target, must be able to:

1. Exit the subsystem
2. Reach the target in an understandable form
3. Return to its initiator, or to send back at least one signal that confirms the reception.

There are several subsystems that must communicate with each other and one or two memories that all subsystems must have access to. It is necessary to use a **shared bus** to ensure communication between the SCU, the peripherals and the memory (Figure 3.30).

The structure of this shared bus module is a little more complex than one might imagine. For simplicity, we will call it ***Switch Matrix subsystem*** and we will use the abbreviation ***SM***.

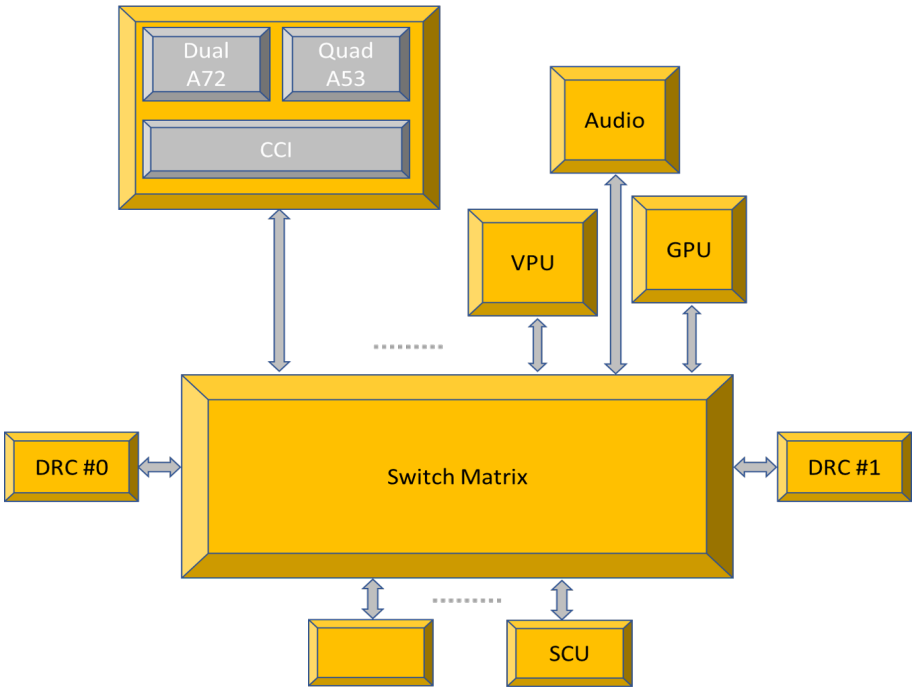


Figure 3.30: General platform structure

This is the main technology targeted in our study and we will build our power model around it.

3.5.7 Switch Matrix (SM) structure

The interface to the SoC has multiple ports to allow multiple subsystem connections to the Switch Matrix. The buses used to connect the subsystems to the SM and to the main memory functions of the chip are asynchronous bidirectional point-to-point interfaces (i.e., at each use there is a master and a slave connection). They carry a bidirectional AXI (or ACE) protocol.

Each subsystem sends data of different sizes. For example, the size of a graphic data sent by the GPU will be larger than a data sent by the Audio subsystem. Therefore, the buses between the Switch Matrix and the subsystems must be able to handle multiple data sizes. Taking this problem into account, an adaptation must be made to the size of the data in the Switch Matrix in order to normalize the transactions. We do not know the order in which the transactions access the memory, nor the time they take to access it, nor the way in which the handshakes are managed. This is a high-performance chip, so there may be several subsystems that require memory accesses at the same time, and the Switch Matrix must take care of their scheduling. Hence the complexity in the structure of the SM. **Therefore, the role of the SM is to manage the scheduling, routing and adaptation of the transactions sent by the subsystems, in order to establish an efficient communication between the subsystems and the two memories.**

In input and output, the SM has *N* Master or Slave interfaces that support the AXI protocol. The SM, like all other subsystems, is composed of several hierarchical blocks and each block performs a processing on the transactions in progress, in order to schedule them and adapt them for sending to the next block. The first adaptation at the SM level consists of grouping the interfaces into blocks that we will call ***Groups (G#)***. Each group is connected, by its interfaces, to several Subsystems (number of Subsystems equal to the number of interfaces). The number of subsystems per Group is fixed and all Groups have more or less the same structure and the same number of subsystems to manage.

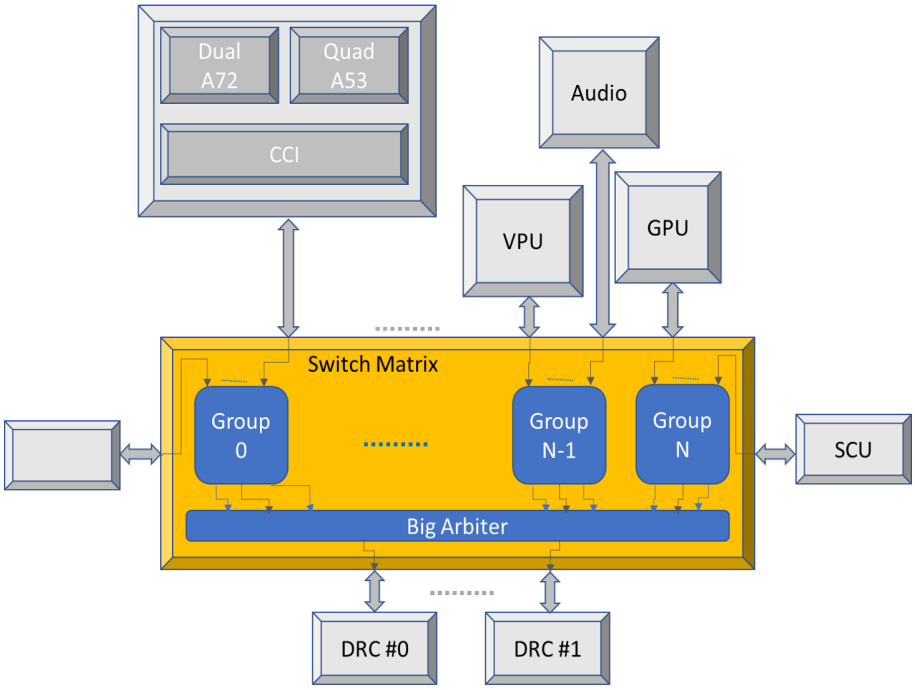


Figure 3.31: Groups inside the Switch Matrix

For the grouping we take into account the location of the subsystems on the chip with respect to the SM, in order to avoid too long and inefficient connections. The other criterion for grouping is to have only one high bandwidth Subsystem per group.

A group returns two or three outputs (depending on the number of external DRAM memories) directed to an arbiter (called Big Arbiter), and each output represents the path the transaction can pursue:

- Output 0: Transaction to DRC0
- Output 1: Transaction to DRC1 (if there is a second memory subsystem)
- Output 2: Transaction to another subsystem (SYS)

Big Arbiter structure

The Big Arbiter takes care of routing the transactions to their targets. The function of the Big Arbiter is to route the targeted DRC (0 or 1) and SYS (subsystems) transactions through the system. The traffic to the memories enters the Big Arbiter in an already interleaved manner (interleaving explained later in this section). Thus, the direction, DRC0 or 1, is already

predetermined and encoded in the transaction. The DRC0 traffic enters into index 0 and DRC1 traffic into index 1. The SYS_NIC simply routes the SYS transactions to the correct destination based on a decoding mechanism. There are one or two DRC arbiters, built using an arbitration policy based on the QOS signal contained in each transaction following the AXI protocol. Thus, the structure of the Big Arbiter is shown in Figure 3.32 as follows:

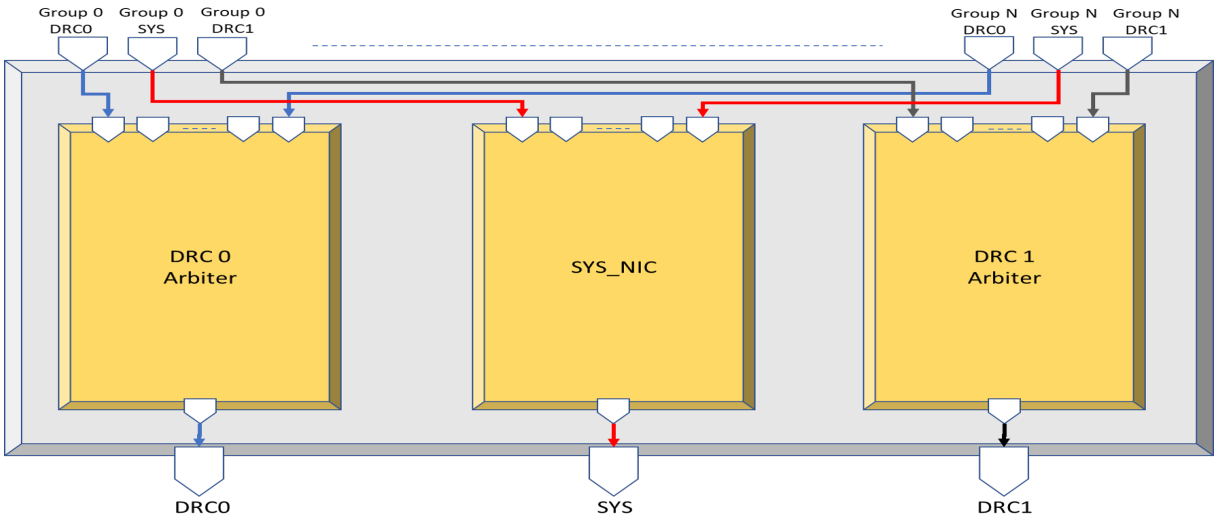


Figure 3.32: Big Arbiter structure

Group structure

The Group is designed to serve ports with an interleaving/rescheduling function. A group contains several sequential blocks that process transactions (Figure 3.34).

- **TBU** - it is the *Translation Buffer Unit* and is actually a component of the SMMU, with N/m instances *(N - number of subsystems and m - number of Groups)* placed in each Group. The AXI Stream interface is multiplexed onto the TBUs in each Group and connected to the TCU via an interface designed for AXI Stream (Figure 3.33).

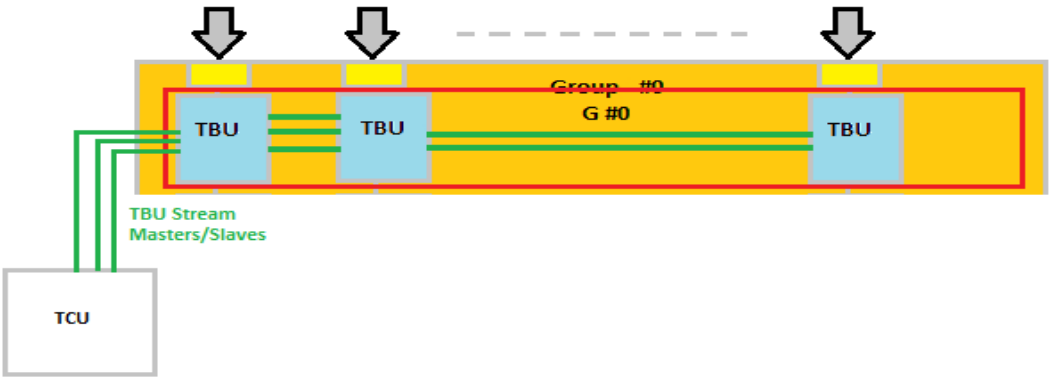


Figure 3.33: TBU structure

The SMMU accepts virtual address domain requests and translates them into the physical address domain through a memory mapping table.

- **Resizer** - there are data size adapters, if needed, under the TBUs. The Resizers structures consist of simple cascaded NIC-301 interconnects.

- **MemISO** - it is the memory region controller. It acts as a firewall controlling the distribution of the memory parts between the entities.

- **QOS** - the QOS block is an inter-leaver and priority manager for transactions, as well as a read replenishment buffer for return data. Since there are two memory channels on the i.MX8QM chip, the QOS interleaves DRC requests evenly across the programmable 4/8/16KB address boundaries. The QOS also routes transactions intended for inter-system communication (sub-system output from each group) to the Arbiter.

- **Little Arbiter** - the function of the Little Arbiter is to route the targeted DRC (0 or 1) and SYS (subsystem) transactions through the system. It has the same structure and operation as the Big Arbiter. The difference is that Little Arbiter takes input from QOS blocks and handles transactions at that level, while Big Arbiter takes input from Group blocks, so it is one level higher in the hierarchy. The Little Arbiters (one in each group) pass the transactions to the Big Arbiter.

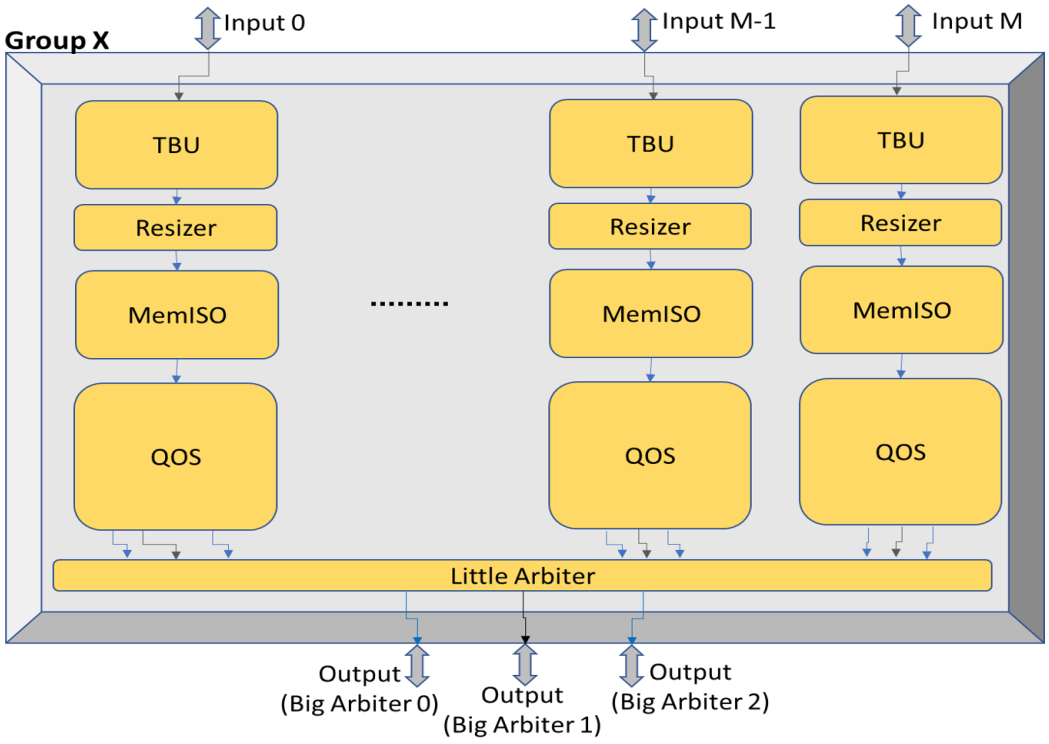


Figure 3.34: Group structure

NOTE: The QOS block hides the main structure needed for the interleaving and scheduling of AXI transactions. The in-depth study of the QOS block was essential for the success of this study. For a question of confidentiality, the structure of the block is not revealed, but its operation is exposed in this part.

QOS structure

- **Definition of Quality of Service (QoS)** - it is the ability to manage the smooth running of a given traffic transmission. The services that are monitored depend on the application, but often they are throughput, transmission delay, availability, information loss and scheduling. Traffic management is a concept that allows us to optimize the use of resources during a process and to ensure good performance for the targeted applications. The AXI protocol has channels reserved for Quality of Service (QOS) that are used for scheduling and routing transactions.

- **Definition of interleaving/scheduling** - interleaved memory is designed to compensate for the relatively slow speed of dynamic random access memory (DRAM) or main memory by distributing memory addresses evenly across memory banks. In this way, contiguous reads and writes take turns using each memory bank. This results in higher memory throughputs due to the reduced expectation of memory banks to prepare for the desired operations. This architecture is different from multi-channel memory architectures, mainly because interleaved memory does not add more channels between the main memory and the memory controller. However, interleaving between two channels is also possible in the NXP i.MX processors, and in fact exactly this type of interleaving is used in i.MX8QuadMax for communication between the Subsystems and the two memories. There are two DRC addressing regions on i.MX8QM, one below 4 GB and one above (Figure 3.35).

Start address	End address	Size	Allocation
0x0_0000_0000	0x0_0001_7FFF	96KB	ROM
0x0_0001_8000	0x0_000F_FFFF	928KB	Reserved
0x0_0010_0000	0x0_0013_FFFF	256KB	OCRAM
0x0_0014_0000	0x0_01FF_FFFF	31488KB	Reserved
0x0_0200_0000	0x0_07FF_FFFF	96MB	Reserved
0x0_8000_0000	0x0_FFFF_FFFF	2GB	DRAM memory
0x1_0000_0000	0x3_FFFF_FFFF	12GB	Reserved
0x4_0000_0000	0x4_3FFF_FFFF	1GB	----------
0x4_4000_0000	0x7_FFFF_FFFF	15GB	Reserved
0x8_0000_0000	0x8_7FFF_FFFF	2GB	Reserved
0x8_8000_0000	0xF_FFFF_FFFF	30GB	DRAM memory

Figure 3.35: Memory mapping from a CPU perspective

These two spaces are merged into one contiguous space, i.e., the lower region (the 2 GB region) is moved into the reserved DRAM hole just below the upper 30 GB (Figure 3.36).

From the Switch Matrix perspective, the memory space forms a contiguous 32GB region (Figure 3.37).

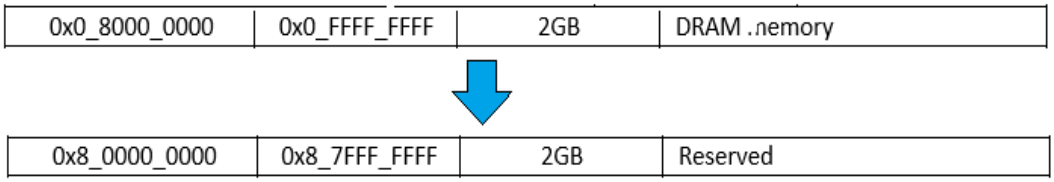


Figure 3.36: Moving the 2GB DRAM space

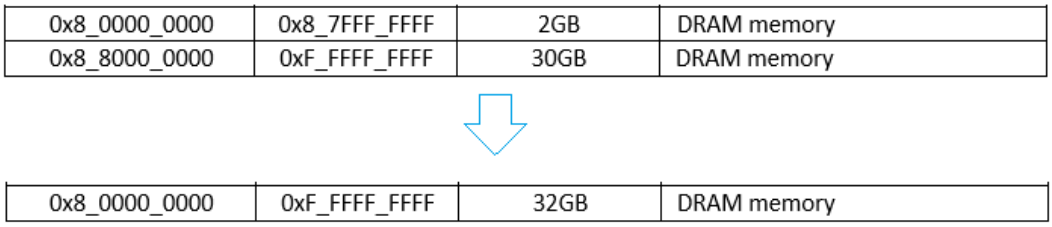


Figure 3.37: 32GB contiguous region

Once the contiguous region is formed, the high-order bit is cancelled so that the memory space reserved for DRAM is moved to address 0 (Figure 3.38).

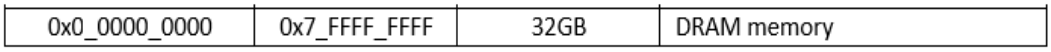


Figure 3.38: Contiguous region seen by the Switch Matrix

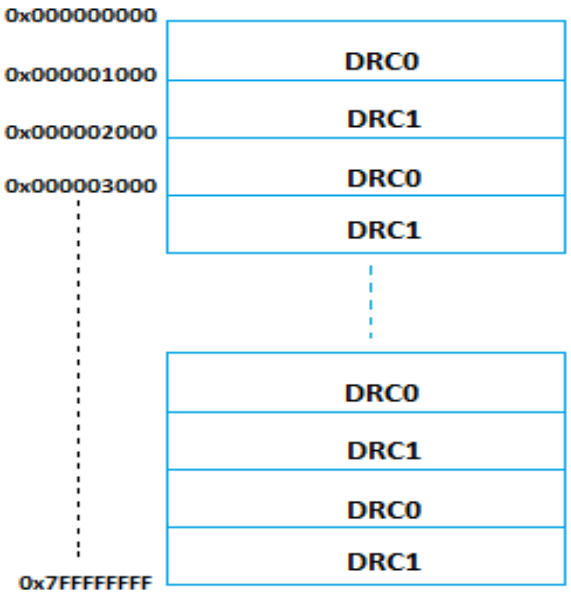


Figure 3.39: Interleaving

The 32 GB memory space is divided into 2 parts of 16 GB and the two memory ports, DRC0 and DRC1, each receive 16 GB of space. They are evenly interleaved on the address boundaries (Figure 3.39). The interleaving is programmable and can perform jumps of 4/8/16KB. In the case of i.MX8QuadMax the interleaving used is programmed to 4KB.

- **Categorization** - thanks to the hierarchy present in a subsystem, and more precisely the interconnection, there are several paths through which the transactions can pass. Therefore, each path must associate different IDs to the transactions in order to indicate the return path when the response is received. There are some modules that send more important transactions than others and they must be given priority during a busy communication (several transactions in progress). The categorization of a request is the process of identifying the initiator or groups of initiators within a subsystem that sent the request. There are *N* categories and each subsystem interface is associated with at least one category. The categorization algorithm contains a set of *COMPARE_IDs* that it compares to the ID bits of the transaction and selects the corresponding category. Each category can store a limited number of transactions. Once categorized, some routing algorithms are applied to the transactions. As mentioned earlier, the QoS value of the transaction is encoded in the QoS bits (AXI - *AWQOS* and *ARQOS* signals). Each category tests the QoS bits and retrieves the highest priority. This value becomes the priority of all transactions in that category over the others.

- **Routing algorithms** - Between these algorithms we recognize the ***HPR*** and ***Under-run Detector***.

  – **HPR (High Priority Request)** - The transaction with the highest priority is determined by its QoS value. After categorization, there is a QoS value that is maintained for all transactions in a category. This value is formed by keeping track of the highest QoS value in the category's queue. There is also a programmable bias QoS offset that is unconditionally added to the *HPR_QOS_OFFSET QoS[N]*. This is:

```
hp_qos[N] = max(req_QOS_in_CAT[N]) + HPR_QOS_OFFSET[N]
```

  Each category has an *HPR_QOS_OFFSET* which can take values between 0-15. This offset allows us to manage the priorities between the different groups.

  Reminder: Each Group contains several identical QOS blocks, so each QOS block contains categories identical to the categories of the other QOS blocks. The only difference between these blocks will be the offset and the number of categories used.

  – **Under-run Detector and Panic offset** - The purpose of this algorithm is to detect the condition in which a request (actually an entire category) is running or to get an external panic signal from the initiator, indicating that service is not being provided often enough. When a transaction is written, a counter for its category is incremented by the number of requests currently residing in the category (including the writing of that current request). When a transaction is read, the counter for its category is decremented by a programmable amount *CAT_SCHEDULED_DECR[N]*. The algorithm representing this operation is:

```
case ({req_write[N], req_read[N]})
2'b00: ud_count_nxt[N] = ud_count[N];
2'b01: ud_count_nxt[N] = ud_count[N] - ud_decr[N];
2'b10: ud_count_nxt[N] = ud_count[N] + ud_incr[N];
2'b11: ud_count_nxt[N] = ud_count[N] + ud_incr[N] - ud_decr[N];
end case
```

Where:

```
ud_incr[N] = req_num_resident[N];
ud_decr[N] = CAT_SCHEDULED_DECR[N];
```

Note that this results in a nonlinear increase in number and a linear decrease in number. Thus, the underfunction detector accelerates the count increase towards the QoS increase threshold as the category comes under pressure (becomes more congested), and a steady decrease in count from the threshold. The *ud_count* may not be reduced to zero, even when all requests in the category have been executed. In this case, this category will be inactive, and *ud_count* will decrease by 1 with each clock cycle passed until it gets to zero, or a new request is pushed into the category to set the condition for increasing it again. There are two thresholds in action based on the *ud_count*, the first threshold being an increase in the QoS value, and the upper threshold being an increase in the PANIC QoS value. When the first threshold is reached, the programmable amount *UD_QOS_OFFSET[N]* is added to the QoS. When the second threshold is reached, the programmable amount *PANIC_QOS_OFFSET[N]* is added. The algorithm representing this operation is:

```
        ud_panic_qos_adder[N] =
            (((ud_count[N]> = UD_COUNT_THRESH2) & UD_ENABLE) |
   ss_panic) ? PANIC_QOS_OFFSET[N] :
               (ud_count[N]> = UD_COUNT_THRESH1) & UD_ENABLE ?
               UD_QOS_OFFSET[N] : 0;

```

The *PANIC_QOS_OFFSET[N]* will be added to QoS even if *UD_ENABLE[N]* is not set.

The contribution of *UD_QOS_OFFSET[N]* is enabled with the *UD_ENABLE[N]* bit. The contribution of *PANIC_QOS_OFFSET[N]* is enabled either with the *UD_ENABLE[N]* bit or with external PANIC input signals.

Both *PANIC_QOS_OFFSET* and *UD_QOS_OFFSET* can be set between 0 and 15.

– **Quality of Service** - The final Quality of Service (QoS) will be the sum of all the QoS adders listed below:

```
final_qos_pre_clamp[N] = hpr_qos[N] + ud_panic_adder[N] + the other
    algorithms.
```

Each category has a *MAX_QOS[N]* and a *MIN_QOS[N]*. These parameters lock the final QoS to set values (maximum and minimum) if the calculated QoS is out of range of *[MIN_QOS : MAX_QOS]*.

Once the QoS is calculated, the QOS block uses these routing algorithms to schedule transaction requests. Requests with the highest QoS go first, followed by lower priority requests. The QOS block hands over to Little Arbiter, which will follow the QoS routing policy defined earlier.

ADB

ADB-400s are the standard connections between Groups and the Big Arbiter; 128 bits for DDR traffic, 32 bits for control traffic (Subsystem output). ADB components are used in pairs - one slave and one master - to provide temporal isolation; the pair is asynchronous to each other. These asynchronous bridges between components allow us to connect systems that are in different clock, power or voltage domains.

Routing

The routing is done according to a QoS (Quality of Service) policy. When sending a transaction using the AXI protocol, several pieces of information are sent with the transaction. According to the AXI protocol, the address bits, the identification bits, the USER bits and the data are sent on different channels (plus other complementary signals). This allows us to send them at the same time and not to add latency while waiting for a response. Each subsystem will transmit QoS related information through these different channels.

- The *address bits* are used for interleaving and to identify the information sought in the memory cells.
- The *ID bits* allow us to recognize the initiator of the transaction and to find the return path during the response.
- The *User bits* allow us to store all the types of information we want to send with the transaction. In our case, we use these bits to move and temporarily store the QoS bits (which are sent on the specific AXI QoS channels). By default, the Adapters take into account the priority bits. It turns out that this is not necessary in our case, so we force the adapter to ignore them. The forcing is done by removing the QoS information (AXI QoS channels) and storing it in the User bits when passing through the adapters.

3.5.8 Switch Matrix power and clock intents distribution

The subdivision of the Switch Matrix into power and clock domains requires that elements powered by the same power source be grouped into a power domain and elements with the same clock source be grouped into a clock domain, yet this is not the most important requirement for mastering power consumption management. We need to split the domains according to the operation and activity of the internal module set.
For confidentiality reasons, we do not describe precisely the decomposition into power and clock domains.
Figure 3.40 shows the chosen subdivision assuming we have N subsystems (The number of subsystems connected to each Group is equal to the number of sequential paths in it, which is equal to the number of clock domains in it plus one for the *Little Arbiter*). In order to manage the power consumption of the SM, i.MX SoCs use several power reduction mechanisms. Two of them are Power and Clock Gating which change the state of the domains according to the state signals (Active and Idle) and the reactivation time. To be more precise, in the SM subsystem and both DRC subsystems, the clock mechanism is automatically controlled by the hardware (hardware auto clock gating). Therefore, it does not use an OPP table for the clock management (present in PwClkARCH), but uses specific cycle counters and direct

clock control commands. This functionality will be presented in more detail in Chapter 4. With this section, we have covered the basics of how the Switch Matrix works and we can move on to the general conclusion of this state-of-the-art chapter.

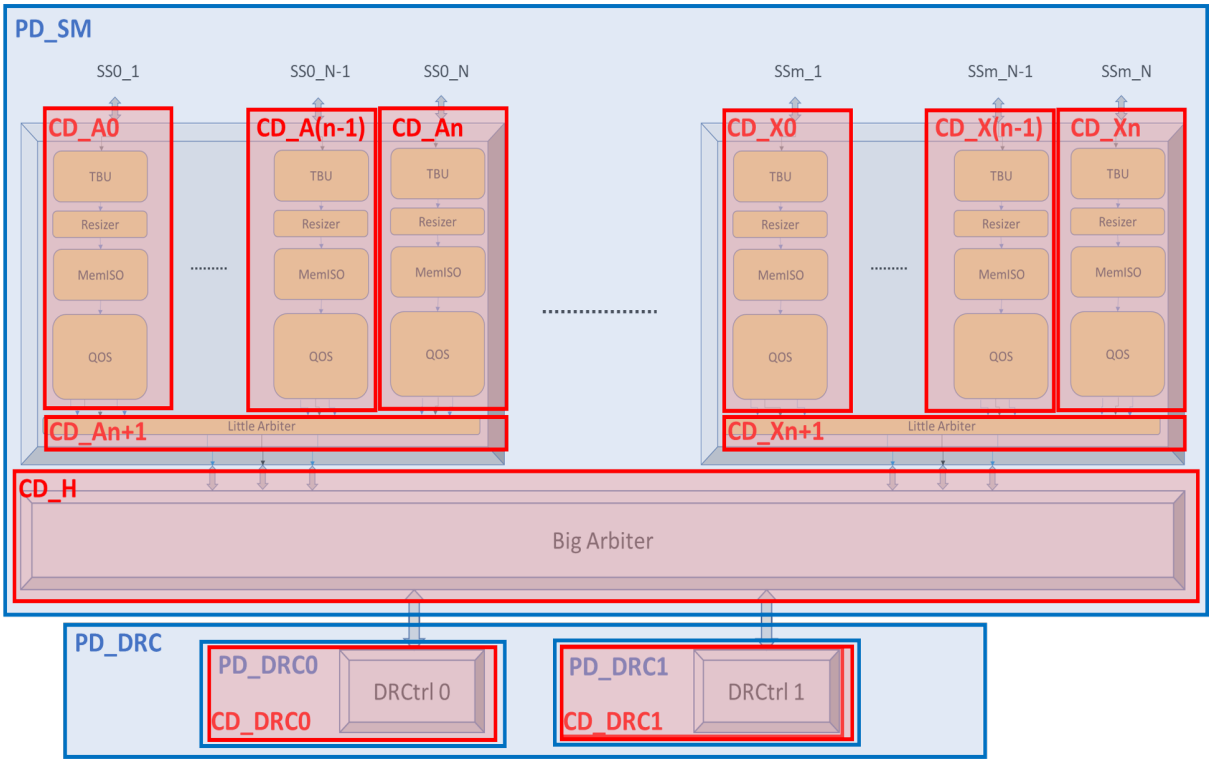


Figure 3.40: i.MX8QM Power Intent decomposition

3.6 Conclusion

In this chapter, we have presented the main challenges present in exploring the system-level design space. We presented the basics and standards for performance and power modeling and estimation at a high level of abstraction and reviewed several existing academic and industrial solutions. In addition, we introduced the PwClkARCH library that we will use for power modeling and compared it to other existing solutions. Finally, we described the technology we are investigating and need to model with respect to its power intent and critical functionality. This project is a collaboration between the LEAT lab and NXP, whose objective is to test and find an optimal solution for power modeling and estimation at a very early stage of the design flow. The choice of technologies imposed by the project was selected because of their high complexity and the lack of standard studies and methods working in interconnect modeling.

In the following chapters, we present the approaches we chose to address the problem, the granularity and abstraction of our modeling strategy, and the framework we created to simplify the application of this methodology on NXP i.MX SoCs.In addition, we present our simplified performance estimation technique and power model structure. Next, we will review the experiments we conducted and the correlation between our simulations and real silicon measurements. Finally, we will review some additional tests performed during the study and conclude with our results and potential future prospects.

Chapter 4

Definition of functional modeling and power/performance analysis approaches

4.1 Abstraction level, reusability and granularity

The transactional level (TL) is currently the most promising level of abstraction for the necessary extension of design methodologies and improvement of design flow. TL is capable of providing easy modeling style and high simulation speed [109], while allowing the creation of sufficiently accurate behavioral models of separate subsystems or complete platforms. By using TLM2.0's modeling styles, we can efficiently handle the modeling of communication protocols, giving us the ability to describe data transfers in complex architectures. The fact that SystemC and TLM2.0 are built on top of the C++ OOP language allows us to use semantics such as polymorphism, inheritance, design patterns and many other features that simplify development, increase interoperability, reusability, configurability and, of course, simulation speed. Analysis and verification mechanisms and tools can be created or integrated relatively quickly. Data transfers are a good indicator of modules activities when coupled with registers monitoring, which can be used for performance or power estimates. The PwClkARCH library can be easily used with these principles, as its calculations can be based on both power management modules or/and local module events related to their activity. Therefore, this is the level of abstraction that best suits this approach.

Usually, new designs are derived from existing ones. Since IPs are reused from one design to another, the goal for TLM models is to be able to reuse them as well. Regardless of the level of abstraction we are working at, there is a set of fundamental requirements for the development and integration of reusable IPs, which include meeting interoperability criteria such as portability, configurability, standardization, quality, mapping strategies, and rigorous verification. Similarly, the approach to describing potential power intent should follow these same requirements. In addition, it is highly recommended to adopt automation of tasks such as consistency checking, code generation and assertions, which can help avoid power-related functional failures and improve productivity. It is very important to build reusability requirements into the design specifications and only make modules reusable when it makes sense, as this is an expensive process. To maintain the reusability of the power intent and the IP, we need easily configurable models and good documentation.

We try to keep the structure of the behavioral models as condensed as possible and base them on a generic skeleton structure with a small number of mandatory processes/threads

and extension possibilities. Our models are not synthesizable, so we don‘t need to create multi-process structures like those needed for RTL. With high-level languages, such as C++ and SystemC, we can create precise behavioral models, without the need to define all the registers, logic, pins and signals as in RTL. The behavioral model must be described in a way that represents the precise functionality of the IP. At the transaction level of abstraction, we need to be able to make fairly accurate and fast power/performance estimates using timed performance models and some high-level estimates of the area and activity of the IP [37] [110]. In addition, we should be able to begin to define and test power optimization solutions and reduce or eliminate their impact on system performance.

Following the idea and semantics of the reverse engineering approach, we started with an in-depth study of the i.MX8QM platform and more precisely, the Switch Matrix module. The first step was to study the possible granularity and decomposition of the blocks for our model. In order to ”mimic” a top-down approach and start with an abstract ”black box” that can be refined later, we first started with a relatively high granularity. Then, we described the main structures and algorithms independently so that we can use them as separate functions. In this way, we can easily move them to a lower child block when refining the model. The first steps of the refinement process are shown in Figure 4.1.

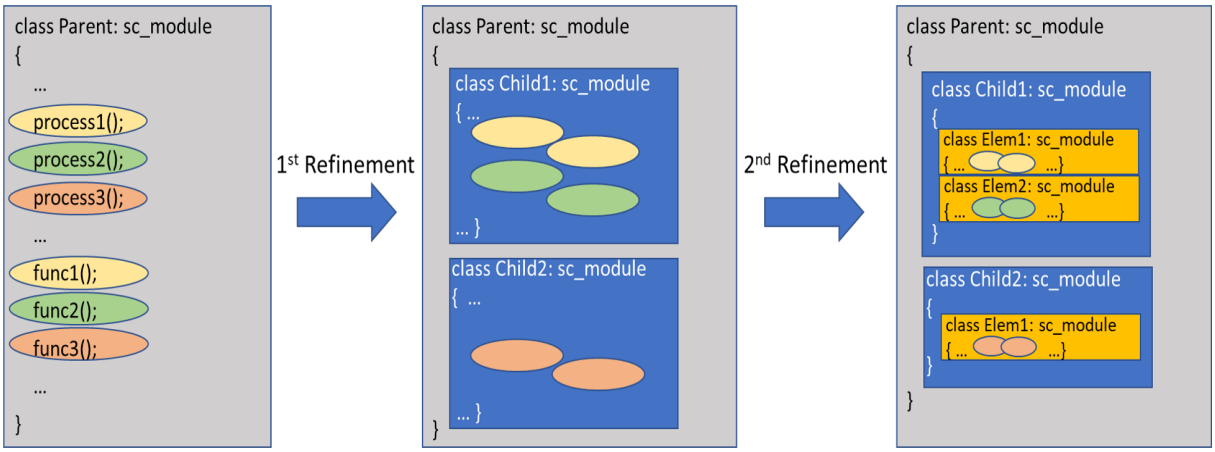


Figure 4.1: Model refinement process

To do so, during the modeling process, we had to define which blocks of the SM subsystem can be reused and which parameters can be reconfigured for these blocks. Then, we limited the granularity of the model to the smallest reusable blocks, allowing to neglect the instantiation of all signals and registers present in the subsystem, without losing the configurability of the components.

Depending on the complexity of the platform and the number of elements/subsystems, the power model can be created at the top level or divided into several groups. For the purpose of our study, which is the Switch Matrix, we can use the top-level technique. Our approach follows and adapts to these reuse criteria and provides the same functionality as UPF and UPM but extended to the transactional level and enhanced by the clock tree description methodology allowing for clock domain distribution. The main concerns addressed by our methodology are to simplify the definition of power intent, extract accurate power metrics, ensure interoperability and portability, unify the language used for the performance model and power intent, and avoid the extensive verbosity of the UPF syntax.

To achieve these goals, we need a framework that allows us to easily build new platforms using module libraries, define different scenarios and test cases, add third-party emulators and libraries and automatically generate model documentation.

4.2 Framework construction

Project construction plays an important role in the development cycle. When working with large platforms that are divided into several blocks, subsystems and modules, it is important to ensure that the build and linkage part is well controlled and that only the modules that need to be tested are built. An essential part of this work was to define a simple and complete solution for hierarchizing projects and to make it as easy and reliable as possible. To this end, we chose to build our framework with a fixed layered structure and non-recursive makefiles for linking and compilation. There are six main fixed top-level directories (Figure 4.2):

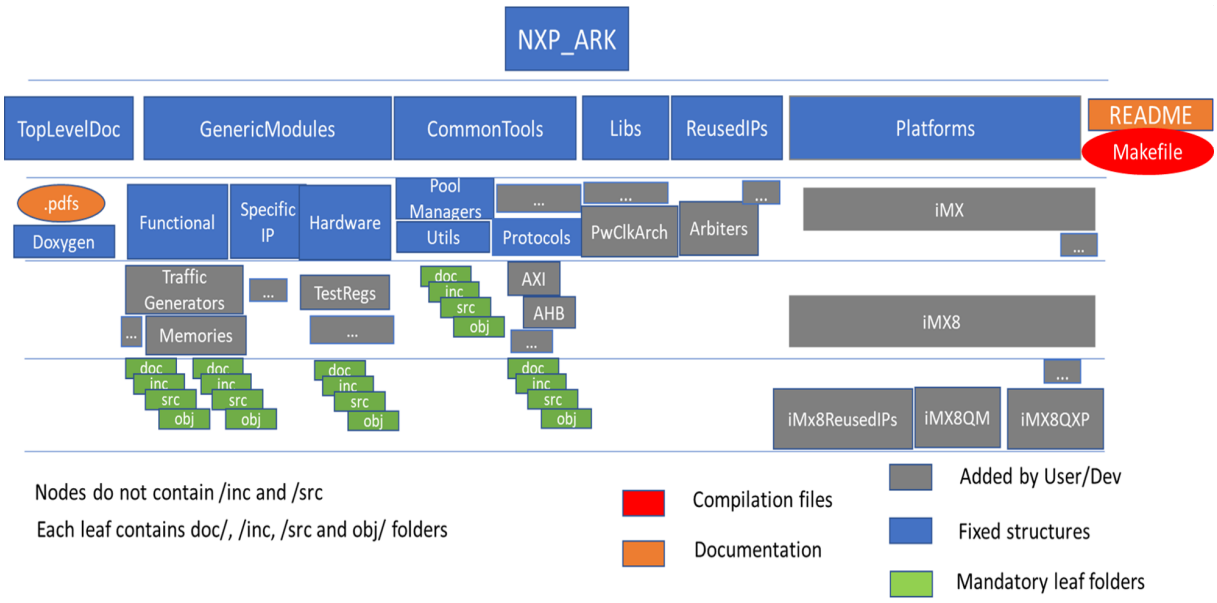


Figure 4.2: Framework structure - Top level

- ***TopLevelDoc*** – contains all the high-level documentation related to the framework, such as project documentation generated by the doxyfile, installation guides, information about using the framework.
- ***CommonTools*** – combines some common headers used in each simulation, such as memory managers, performance estimation observers, functional checkers, transaction pools, protocol definitions, different types of extensions, reporting and tracing mechanisms, etc.
- ***GenericModules*** – contains all generic modules that can be reused in our simulations, such as virtual intellectual properties, traffic generators, memory models, etc.
- ***Libs*** – contains all external libraries that can be used individually or in a co-simulation with our framework. We have developed a solution to easily choose the library we

want to use for the current simulation and a makefile structure to easily integrate new libraries into the framework.

- ***ReusedIPs*** – contains all generic IPs that can be reused in multiple platforms, such as bridges, interconnect modules, etc.

- ***Platforms*** – contains complete platforms assembled with multiple IPs, such as members of the i.MX8 series.

The complete framework (with all IPs, Platforms, Testbenches, Documentations) can be compiled from the top level with the top level makefile. This takes more time and is not optimal when we want to test specific IPs/Platforms. Two cases where the top-level makefile is useful are the generation of the documentation (Doxygen) representing the entire project structure and the cleanup of the entire project to remove all object, executable and output files. The first layer of IP definitions can be found in the *ReusedIPs* directory. The structure of ***ReusedIPs*** directory is illustrated in Figure 4.3:

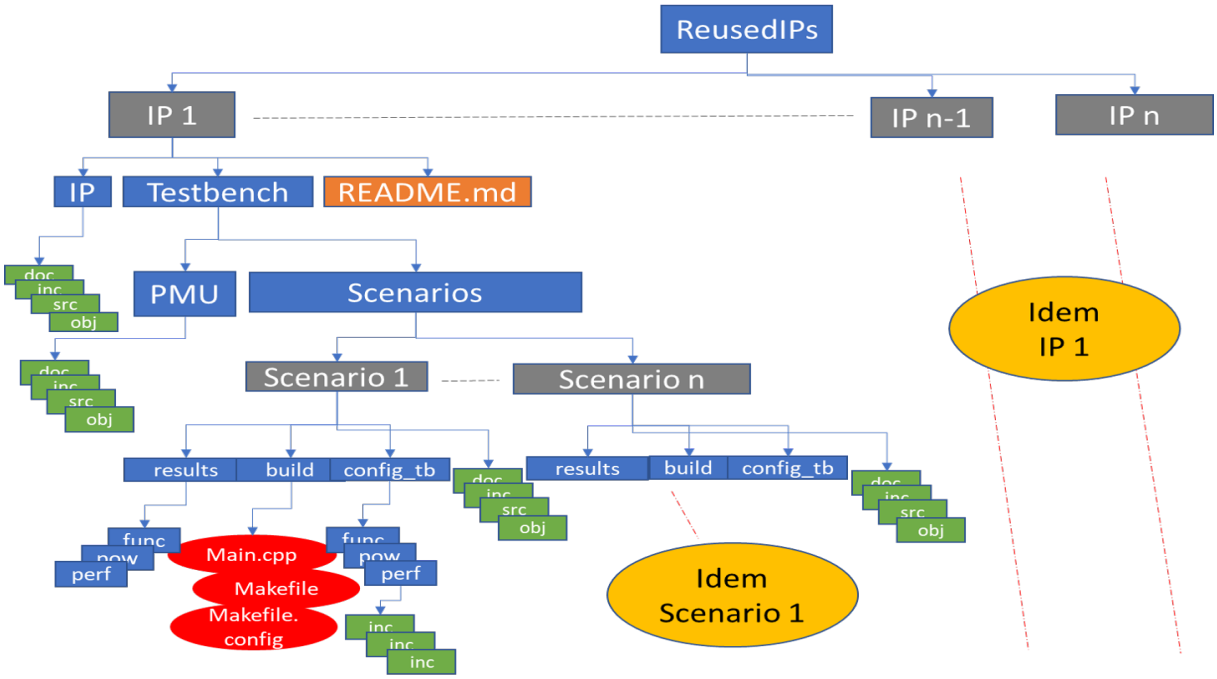


Figure 4.3: Framework structure - ReusedIPs

Each individual IP directory contains its documentation, the functional model and the testbenchs for functional verification at the IP level. Each IP contains two types of local makefiles allowing to compile the IP and create a static library or to directly compile a testbench using the static library of the IP or to compile the scenario with the IP model. The static IP library can be very useful when we have verified and validated the IP design and do not need to modify or recompile them for each simulation. However, each testbench can either use the IP as a static library or rebuild its source files if necessary. The *PMU* folder contains the power-oriented modules, such as the PMU module set for the given testbench, the Master module responsible for the power management, and the description of the power intent of the platform. We can have different power models here, but we should be able to

run all scenarios with each of them. The *Scenarios* directory contains all the use cases defined by the designers that are relevant to this IP. Each scenario has its own directories for the build/compilation process, simulation output, and configurations. Each *Testbench* has its own copy of the reusable fixed Makefile and a unique Makefile.config file containing the use case specific variable definitions and build paths. In the *Platforms* directory (top-level), users can add specific structures containing their full designs. In our case, we added a directory containing all the i.MX designs we want to model and simulate (Figure 4.4).

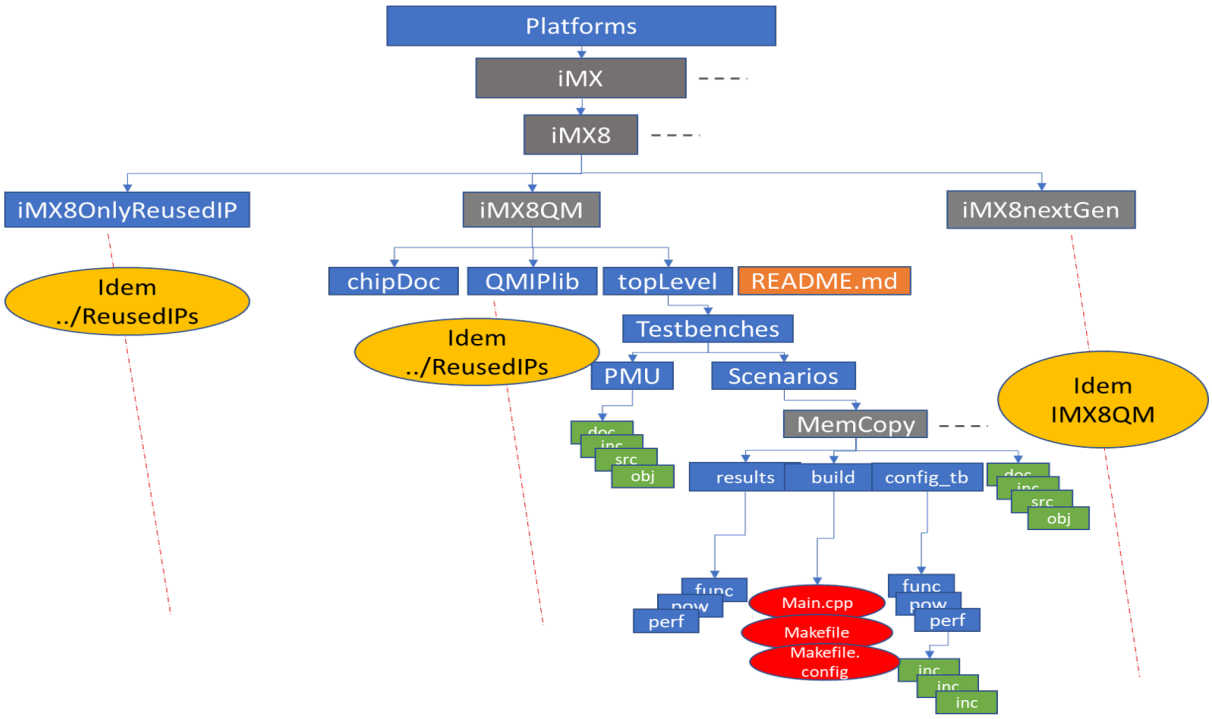


Figure 4.4: Framework structure - Platforms

The structure of each of the added platforms is based on a structure similar to that of the top level and the directory of each component used in the complete platform is similar to that presented in *ReusedIPs*. This structure allows for configurable and maintainable multi-level functional verification. In addition, we have added scripts capable of automatically generating the complete structure in cases where we want to design a new IP, a new testbench or a complete new user directory project.

4.3 Testbench systems modeling

In order to test and analyze all IPs and platforms, we need to be able to easily create new test cases, reconfigure existing cases, and use generic components for integration of the tested design into test benches. We also need to be able to easily switch between communication protocols in order to optimize communication between subsystems on a chip. For this purpose, we have developed several generic modules. In this section, we will present our approach to modeling communication protocols, traffic generation and memory models. All these components drive the activity of the model, so they are essential for visualizing energy consumption.

4.3.1 ARM-based communication protocols modeling

Each of the multiple sets of rules, called communication protocols, developed to create a common communication language between IPs is tailored to a specific application. Thus, we may often need to use multiple communication protocols when designing a SoC. The TLM2.0 transactional level modeling standard has been provided to give a generic protocol base and to allow the modeling of specific protocols using some extension mechanisms. It allows us to abstract all low-level information, such as pin and port descriptions, and establish communication between SystemC processes using functional calls through sockets. Through these sockets, we send packets, called payloads, which contain some generic information such as data, address and size (previously described in 2.3.1). These payloads can be extended with other protocol-specific signals. In addition, TLM2.0 coding styles allow us to pipeline transactions and add approximate timing information to achieve more accurate communication.

As we are modeling a complex module (the Switch Matrix) connecting all subsystems of an heterogeneous SoC, we have to take into account the communication protocols, their pipelining capabilities, the ongoing transactions and the approximate timing of the different phases. For this reason, the LT coding style was not feasible and we had to choose the AT coding style for our modeling framework (which can, of course, contain other hybrid LT and AT modules).

During architectural exploration, we may need to test different blocks and protocols, and thus switch between them. In addition, modeling the protocol can be error-prone and time-consuming if we code it separately in each IP. The main semantics of TLM2.0 is oriented exactly towards this separation of behavioral and communication models. However, the AT coding style contains multiple phases, which can carry over from one protocol to another. Information about these phases may or may not be required by the behavioral model, and additional registers or other features may be needed for the correct implementation of the protocol.

Thus, in order to create reusable, clearer and configurable interfaces and to avoid any protocol-specific code in the behavioral description, we separated the communication part from the behavioral part and **created reusable communication interface modules**. These modules are based on generic protocol initiator and target skeletons, called *SimpleTarget* and *SimpleInitiator* (Figure 4.5).

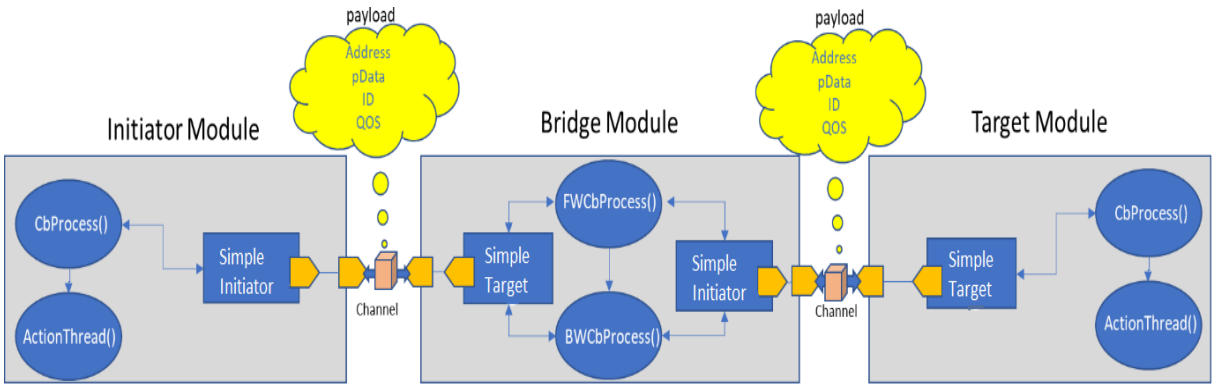


Figure 4.5: Simple Target/Initiator communication modules

- ***SimpleTarget*** – Internal IP module (containing *simple_target_socket*) attached to its corresponding target TLM socket. It uses the target TLM socket to communicate with other IPs and callback functions to communicate with its parent IP.
- ***SimpleInitiator*** – Internal IP module (containing *simple_initiator_socket*)attached to its corresponding initiator TLM socket. It uses the initiator TLM socket to communicate with other IPs and callback functions to communicate with its parent IP.

These are abstract classes containing a tagged *simple_target/initiator_socket*, several pure virtual functions, callback methods, TLM transaction checkers and socket performance observers (Listing 4.1). The implementation of the pure virtual functions is left to the derived classes, where the protocol specifications is built.

```
template<typename MODULE, unsigned int BUSWIDTH>
class SimpleTarget: public ARK_Protocols::IProtocol, public ARK_Utils::
    Observable
{
    tlm_utils::simple_target_socket<SimpleTarget<MODULE,BUSWIDTH>, BUSWIDTH>
    socket;

    ...

    SimpleTarget(const char* name, sc_time period, size_t idx=0, bool
    enableChecks=true, bool enableObs=true);

    void registerModuleCallback(MODULE* mod, void (MODULE::*cb)
        (tlm::tlm_generic_payload& payload, const tlm::tlm_phase& phase,
    const sc_core::sc_time& delay, size_t idx));

    virtual tlm::tlm_sync_enum nbTransportFWCallback
        (tlm::tlm_generic_payload& payload, tlm::tlm_phase& phase, sc_core::
    sc_time& delay);

    tlm::tlm_sync_enum callBWChannel
        (tlm::tlm_generic_payload& payload, tlm::tlm_phase& phase, sc_core::
    sc_time& delay);

    void notifyModule
        (tlm::tlm_generic_payload& payload, const tlm::tlm_phase& phase,
    const sc_core::sc_time& delay);

    virtual tlm::tlm_sync_enum manageFWCallback
        (tlm::tlm_generic_payload& payload, tlm::tlm_phase& phase, sc_core::
    sc_time& delay)=0;

    virtual void sendResponse
        (tlm::tlm_generic_payload& payload, const sc_core::sc_time& delay)
    =0;
    ...

    ARK_Utils::TLMChecker1 mChecker;
    ARK_Utils::SocketsObserver mTracer;
}
```

Listing 4.1: SimpleTarget module

Some of these methods are local (i.e. implemented and called in the modules *SimpleTarget/Initiator*) and others are implemented or called from the derived protocol class:

- Exactly as in TLM2.0 convenience sockets, the *SimpleTarget* (idem *SimpleInitiator*) provides methods for registering callbacks for incoming interface method calls. The method *registerModuleCallback(...)* is used to register a callback method in the module allowing the *SimpleTarget (SimpleInitiator)* to interact with it. It is somewhat equivalent to the *register_nb_transport_fw(...)* method in the convenience target sockets. A callback method must be implemented in the functional module and registered when the module is built. This method can then be called whenever an incoming interface method call arrives. Since this is the method for connecting these communication blocks to the behavioral model, it is mandatory to register a corresponding callback function.

- The method *nbTransportFWCallback(...)* is the registered callback method for the forward channel and is called whenever the socket receives a request on it *(BEGIN_REQ or BEGIN_RESP)*. We use it to easily apply TLM checkers and observers without encroaching on the protocol definition classes.

- The module *callBWChannel(...)* is a method that can be called from the derived communication protocol in order to send transactions on the backward channel when necessary. Its use may differ from one protocol to another, which is why it is implemented in the *SimpleTarget* module, but it can/should be called in the derived protocol. As in the module *nbTransportFWCallback(...)*, we use it to apply TLM checkers and observers.

- The method *notifyMethod(...)* is also implemented here, but called in the derived protocol definition classes. It is used to notify the module of the arrival of a payload or to update its phase. This method allows us to use a stronger connection between interfaces and behavioral models when the protocol requires it. When called in the protocol class, it activates the callback method connected to the *registerModuleCallback(...)*.

- The method *manageFWCallback(...)* is a pure virtual method, which must be implemented in the derived protocol class, because different protocols may require different strategies for transaction control. It is called inside the *nbTransportFWCallback(...)* method whenever the socket receives a request on its transfer channel.

- The method *sendResponse(...)* is also a pure virtual function used in the protocol definition to respond directly to the initiator module (i.e. the update phase) without passing it to the behavioral model.

The *SimpleInitiator* module has a similar structure and semantics, but for the other end of the communication. We instantiate the simple protocol modules in the IP behavioral model without defining the specific communication protocol (Listing 4.2).

```
template< unsigned int WIDTH_in, unsigned int WIDTH_out>
class Bridge: public IPBase {
    SimpleTarget<Bridge, WIDTH_IN>* inSocket;
    SimpleInitiator<Bridge, WIDTH_OUT>* outSocket;

    Bridge(sc_module_name name, protocol::bus_type protocol, sc_time freq):
    ... {
    inSocket->registerModuleCallback(this, &IPBase::request_path_cb);
    outSocket->registerModuleCallback(this, &IPBase::response_path_cb);
    ... }
    ...
}
```

Listing 4.2: SimpleTarget/Initiator instances example

The definition of the communication protocol is done when the IP in question is instantiated in the platform being simulated. The clock frequency of the subsystem is also passed as a parameter to the constructor and is used by the communication modules and the behavioral SystemC models (Listing 4.3).

```
class PlatformSim: public topBase {
    TrafficGenerator<Bridge, WIDTH_IN> trafficGen;
    Bridge<WIDTH_IN, WIDTH_OUT> bridge;
    DramCtrl target;

    PlatformSim(sc_module_name name): topBase(name),
        trafficGen( ... , protocol::AXI, ..., "Cortex-A72", NB_POOLS,
    READ_POOL_SIZE, WRITE_POOL_SIZE),
        bridge( ... , protocol::AXI, ...),
        target( ... , protocol::AXI, ...)
    { ... }
    ...
}
```

Listing 4.3: Protocol definition

The enumeration *protocol::AXI* allows us to generate the allocation function defining the protocol for each module *SimpleTarget/Initiator*. Since **the main components tested in our platform are based on the AXI communication protocol** [111], we will give a quick overview of how we modeled the *AXITarget* communication module (Listing 4.4). This example is important, in order to show the level of abstraction we use when modeling, which is relative to the needs of our model and its power analysis.

- **AXITarget** – Slave communication module derived from *SimpleTarget*. It describes the sequence and control required to handle transactions according to the AXI protocol specification on the slave side.
- **AXIInitiator** – A master communication module derived from *SimpleInitiator*. It describes the sequence and control required to handle transactions according to the AXI protocol specifications on the master side.

The communication blocks *AXITarget/Initiator* trigger a *notifyMethod(...)* that ”wakes up” the behavioral model for processing, when the transaction transmission/reception requires it. *AXITarget* has two operating models to recognize the request phase:

- ***Automatic*** (by default) - *AXITarget* instance acknowledges the request phase as soon as possible.
- ***Manual*** - it's up to the module to trigger the end request.

This allows better control of the relationship between the request and response phases. It is recommended to keep the checkers activated/enabled in manual mode to signal any erroneous scheduling of the module. AXITarget calls the callback method of the module according to the phases:

- ***BEGIN_REQ*** - to signal the module that a new transaction has just arrived. At this point, the module can call:
  1. *sendResponse(payload, delay)*
     (a) If the response can be scheduled immediately, the *END_REQ* phase is ignored *(delay = 0)* or if a response is in progress on the corresponding channel, *END_REQ* is sent one cycle later at the earliest and the response is put on hold until it can be programmed *(delay = SC_ZERO_TIME)*.
     (b) If the response is programmed at the earliest with the corresponding delay *(delay=[value of sc_time])*.
  2. *triggerEndRequest(payload, delay)* - in manual mode only and when the module has full control over the moment when the request is acknowledged and over the temporal relationship between the request and response phases.
- ***END_REQ*** - to signal to the module that the request phase of the transaction is over/-completed. At this point, the module can:
  1. Do nothing if the response has already been programmed at the *BEGIN_REQ* step
  2. Call *sendResponse(payload, delay)* to schedule a response
- ***BEGIN_RESP*** - to signal that a response has been sent to the initiator.

  The delay associated with this phase is always *SC_ZERO_TIME*. In general, the module has no interaction with the *AXITarget* at this stage (but in some cases it is necessary and possible to do so).
- ***END_RESP*** - to indicate that the initiator has acknowledged the response.

  When the delay is not *SC_ZERO_TIME*, it means that the response actually ends at *sc_time_stamp() + delay*. If the module always uses this step to schedule a pending response, it ensures that it has precise control over when the response is sent.

```
template<typename MODULE , unsigned int BUSWIDTH = DEFAULT_BUSWIDTH>
class AXITarget: public SimpleTarget<MODULE, BUSWIDTH> {
    AXITarget(const char* name, sc_core::sc_time clkPeriod, size_t maxRdOT =
    DEFAULT_MAX_RD_OT, size_t maxWrOT = DEFAULT_MAX_WR_OT, size_t maxTotalOT
    = DEFAULT_MAX_OT, size_t idx=0, bool enableChecks=true);
    void arChannelMethod();
    void awChannelMethod();
    void rChannelMethod();
    void bChannelMethod();
    void rChannelEndRespMethod();
    void bChannelEndRespMethod();
public:
    virtual void sendResponse(tlm::tlm_generic_payload& payload, const
    sc_core::sc_time& delay);
    virtual void triggerEndRequest(tlm::tlm_generic_payload& payload, const
    sc_core::sc_time& delay);
    ...
protected:
    tlm::tlm_sync_enum manageFWCallback(tlm::tlm_generic_payload& payload,
    tlm::tlm_phase& phase, sc_core::sc_time& delay);
private:
    tlm_utils::peq_with_get<tlm::tlm_generic_payload> mARChannel;
    tlm_utils::peq_with_get<tlm::tlm_generic_payload> mAWChannel;
    tlm_utils::peq_with_get<tlm::tlm_generic_payload> mRChannel;
    tlm_utils::peq_with_get<tlm::tlm_generic_payload> mBChannel;
    sc_core::sc_event mArChannelEndReqEvent;
    sc_core::sc_event mAwChannelEndReqEvent;
    sc_core::sc_event mRChannelEndRespEvent;
    sc_core::sc_event mBChannelEndRespEvent;
    AXIChecker1  mAxiChecker;
    ...
}
```

Listing 4.4: AXI protocol selection example

The callback method *manageFWCallback(...)* and the SC_METHODs *[X]ChannelMethod()* (sensitive to the *[X]Channel[phase]Event*) are used for synchronization and implementation of this strategy. Payload event queues with the get *(peq_with_get)* mechanism are used to put the back pressure in case of outstanding transactions. The *AXIChecker* checks if the AXI ordering rules are respected. In addition to implementing the AXI protocols, we need to model the AXI signals. For this, we used the generic payload extension mechanisms. At this level of abstraction and for our application, we are not really interested in the presence of all AXI signals. Thus, we have extended the generic payload with only a few AXI signals (of course, we can add them all) selected according to our needs:

- *AxID* - A common extension corresponding to *AR/AW ID*. We only need one extension here, because each payload carries its command type attribute and has only one extension table with it. So there is no need to separate the signals for this purpose.
- *AxQOS* - A common extension for the *AR/AW QOS* signal. Used for the priority-based arbitration mechanism.
- *AxUSER* - Used to carry user bits.

These extensions can be defined and attached to the payload inside the traffic generator modules. They can also be modified from any component of the system responsible for processing that transaction as it passes through the component. In order to generate and send AXI transactions, we created a module *AXITransactor* that serves as the basis for our traffic generators (described in the next section). There are some limitations in the implementation of our *AXITarget* and *AXIInitiator* modules. One of them is that the transactions are atomic, so we cannot model a system interleaving bursts with different *AxIDs*. However, at the level of our DUT, there is no burst interleaving and this is quite sufficient for the level of abstraction targeted in this study. Furthermore, we can use these skeletons and follow the same approach to create and test other communication protocols without significant (or any) change to the behavioral models. If the communication rules allow it, we can switch from one protocol to another very easily and choose the optimal one (after a functional/performance/power analysis).

4.3.2 Testbench - Traffic generation

Payloads memory management

Since we typically use an AT coding style and non-blocking transport interfaces, traffic generation requires the use of an explicit payload memory manager with reference counting. Different traffic generators may require a different memory management strategy. We therefore created an interface class (containing only pure virtual functions) declaring the main generic methods required for each memory manager (Listing 4.5). Using this interface allows us to create traffic generators without specifying their memory manager internally. We can instantiate a memory manager interface instead and pass the necessary memory manager (derived from *IPoolManager*) as a parameter when instantiating the module at the top level (the same way as the communication protocol). In this way we can test different memory management strategies and adapt them to the type of traffic we are targeting.

```
class IPoolManager : public tlm::tlm_mm_interface, public ARK_Utils::
    Observable {
public:
  // main interface
  virtual tlm::tlm_generic_payload* getPayload(size_t poolIdx)=0;
  virtual tlm::tlm_generic_payload* getPayload(const char* poolName)=0;

  // get information from pool manager
  virtual const std::deque<const char*>* getPoolList() const=0;
  virtual size_t getPoolIndex(const char* poolName) const=0;
  virtual const char* getPoolName(size_t poolIdx) const=0;
  virtual bool isExisting(size_t poolIdx) const=0;
  virtual bool isExisting(const char* poolName) const=0;
  virtual bool isEmpty() const=0;
  virtual size_t getNbFlyingPayload() const=0;

  // get information from pools
  virtual bool isEmpty(size_t poolIdx) const=0;
  virtual bool isEmpty(const char* poolName) const=0;
  virtual size_t getNbPayload(size_t poolIdx) const=0;
  virtual size_t getNbPayload(const char* poolName) const=0;
  virtual size_t getNbAvailablePayload(size_t poolIdx) const=0;
  virtual size_t getNbAvailablePayload(const char* poolName) const=0;

  // tlm_mm_interface
  virtual void free(tlm::tlm_generic_payload* payload)=0;
};
```

Listing 4.5: Pool manager interface

Following this approach, we defined a memory manager that implements two transaction object pools (one for read transactions and one for write transactions) with configurable sizes. At each transaction initiation, we allocate a payload object from its corresponding pool and release it when its reference counter reaches 0. The transaction object pools implemented in the memory manager are constructed using instances of the *GenericPool* objects (Listing 4.6). The capacity of the pool (read or/and write) is passed to them as a parameter. The number of available payloads is tracked internally in the *GenericPool* so as not to exceed the capacity of the pool (using the *mAvailPayloadList* map). For auditing purposes and transactions tracing, two IDs *(LocalID and GlobalID)* are associated with each object/transaction leaving the pool.

```
class GenericPool {
public:
    // Constructor
    GenericPool(const char* name, size_t nbPayload = DEFAULT_NB_PAYLOAD);
    virtual ~GenericPool();

    // interface for pool manager
    tlm::tlm_generic_payload* getPayload();
    bool recyclePayload(tlm::tlm_generic_payload* payload);
    bool isEmpty() const {return (mNbAvailPayload == 0);};
    size_t getNbAvailablePayload() const {return mNbAvailPayload;};
    size_t getNbPayload() const {return mNbPayload;};
    ...
private:
    std::map<tlm::tlm_generic_payload*, bool> mAvailPayloadList;
    size_t mNbPayload;
    size_t mNbAvailPayload;
    static size_t mGlobalId;
    size_t mLocalId;
    ...
};
```

Listing 4.6: Generic pool structure

An example of a CPU pool manager is given in Listing 4.7. It generally defines the functions declared in *IPoolManager* according to the structure and the strategy. Generic pools are generated in a deque, in case we want to extend their number to more than two (one read & one write).

```
class CpuPoolManager : public IPoolManager {
public:
    CpuPoolManager(const char* name, uint8_t nbCpu, uint8_t rdCap, uint8_t
    wrCap); // nbCpu is useful to manage exclusive transaction, strongly
    ordered or device memory type accesses

    // main interface (IPoolManager)
    virtual tlm::tlm_generic_payload* getPayload(size_t poolIdx);
    virtual tlm::tlm_generic_payload* getPayload(const char* poolName);
    // get information from IPoolManager
    virtual const std::deque<const char*>* getPoolList() const;
    virtual size_t getPoolIndex(const char* poolName) const;
    ...
    // get information from pools (IPoolManager)
    virtual bool isEmpty(size_t poolIdx) const;
    ...
private:
    std::deque<GenericPool*> mPools;
    uint8_t mRemainRd;
    uint8_t mRemainWr;
    ...
};
```

Listing 4.7: CPU pool memory manager example

The main interface function *getPayload* is the one that extracts the payload from the pool and sets some of its generic attributes (Listing 4.8). At the end of each transfer, the payload is recycled and can be reused.

```
...
tlm::tlm_generic_payload* CpuPoolManager::getPayload(size_t poolIdx) {
    tlm::tlm_generic_payload* payload = NULL;

    switch (poolIdx) {
        case CPU_POOL_RD_IDX:
        if (mRemainRd > 0) {
            mRemainRd--;
            payload = mPools[CPU_POOL_RD_IDX]->getPayload();
            payload->set_data_ptr(mDataBufferPtr);
            payload->set_command(tlm::TLM_READ_COMMAND);
            payload->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
            payload->set_dmi_allowed(false);
            payload->set_byte_enable_ptr(0);
            payload->set_mm(this);
        } break;
        ...
        default: SC_REPORT_ERROR(basename(), "Request payload from unknown
    pool"); break;
    }
    return payload;
}
...
```

Listing 4.8: getPayload() implementation

Traffic Generators and test case scenarios

Using the payload memory management blocks described earlier, we can create our traffic generators. All the traffic generators we create are derived from a base class, called *no_scenario_traffic_gen* serving as the interface linking the callbacks (generator module and communication module).

```
template<unsigned int WIDTH>
class no_scenario_traffic_gen : public sc_module {
public:
    tlm::tlm_initiator_socket<WIDTH> socket;

    SC_HAS_PROCESS(no_scenario_traffic_gen);
    no_scenario_traffic_gen(const char* nm, const char* port_nm, protocol::
    bus_type protocol, sc_core::sc_time clk, IPoolManager* pmi);

    virtual void start() = 0;
    virtual void callback(tlm::tlm_generic_payload& pld, const tlm::
    tlm_phase& ph, const sc_core::sc_time& d, size_t idx)=0;
protected:
    SimpleInitiator<no_scenario_traffic_gen<WIDTH>, WIDTH>* m_protocol;
    IPoolManager* m_pool_manager;
    ...
};
```

Listing 4.9: Generators base module

We have a method *callback* declared as a pure virtual function, which must be implemented in each traffic generator to schedule the generation and transmission of transactions. The *start()* method is a SC_THREAD that can be used to start the traffic generation at the beginning of the simulation. We mentioned earlier that we have created a *AXI transactor* module that is used to generate AXI compliant transactions. This module is nothing more than an abstract helper class derived from the *no_scenario_traffic_gen* class and includes some functions that simplify the creation of traffic generators. There are functions such as:

- *do_valid_write(size_t addr, size_t ID, size$_t$ QOS, size_t data, sc_time& delay)* - checks if the AW channel is free and, if so, generates a transaction with a given address, AWID, AWQOS, data and delay.

- *do_valid_read(size_t addr, size_t ID, size_t QOS, sc_time& delay)* - checks if the AR channel is free and if so, generates a transaction with the given address, ARID, ARQOS and delay.

- *do_valid_random(size_t addr, size_t ID, size_t QOS, size$_t$ w_data, sc_time& delay)* - checks if the AR or AW channels are free and if so, generates a random write or read transaction.

- *check_complete (tlm::tlm_generic_payload& transaction_ptr)* - checks if a read transaction was successful

The helper functions *AXI Transactor* and the abstract base class *no_scenario_traffic_gen*, allow us easily create and reconfigure different traffic generators. The three main types of traffic generators used in our testbench are:

- ***Phased traffic generator*** - This is a complete generic solution for traffic generation where we can define the number of transactions we want to send, the number of times we want to repeat the procedure, the type of transactions we want to send *READ-/WRITE/RANDOM* and possibly the delay between the different phases and between transactions. This traffic generator also contains an identification *(AxID)* map for the transactions, which allows us to reproduce some specific CPU traffic scenarios, such as memory copy.

- ***Per file traffic generator*** - Generate traffic from a trace file. We can use two different formats depending on the details we have for the generated traffic. We can create a common trace file for all initiator modules and have them generate only the transactions associated with them. In this case, the format of the commands is as follows:
  <name/identifier>, <command>, <timeframe>, <number of transactions>

  If we have a detailed traffic trace extracted from other simulations (RTL, SystemC, TLM, ...) in the standard STL format, we can associate separate traffic trace files to each initiator. Our traffic generator works only with a subset of commands in this format. The STL format is as follows:
  <timestamp>, <command>, <address>

- ***Display controller traffic generator*** - Generate traffic that mimics the use case of a display refresh. It is configurable and can be easily used for different display resolutions. A header file containing several configurations describing different resolutions extracted from a VESA is attached. Some of them are *QVGA*, *VGA*, *SVGA* and *HD1080*. It has two modes of operation:
  - *with prefetch* - multiple transactions are prefetched and stored in two registers, which ensures that there are always transactions to display. Moreover, this increases performances and reduces the energy consumption.
  - *without prefetch* - the transactions are directly addressed to the memory block and displayed as soon as they are received (if the delay is not exceeded)

On Figure 4.6 we can see the representation of the inheritance of the traffic generators.

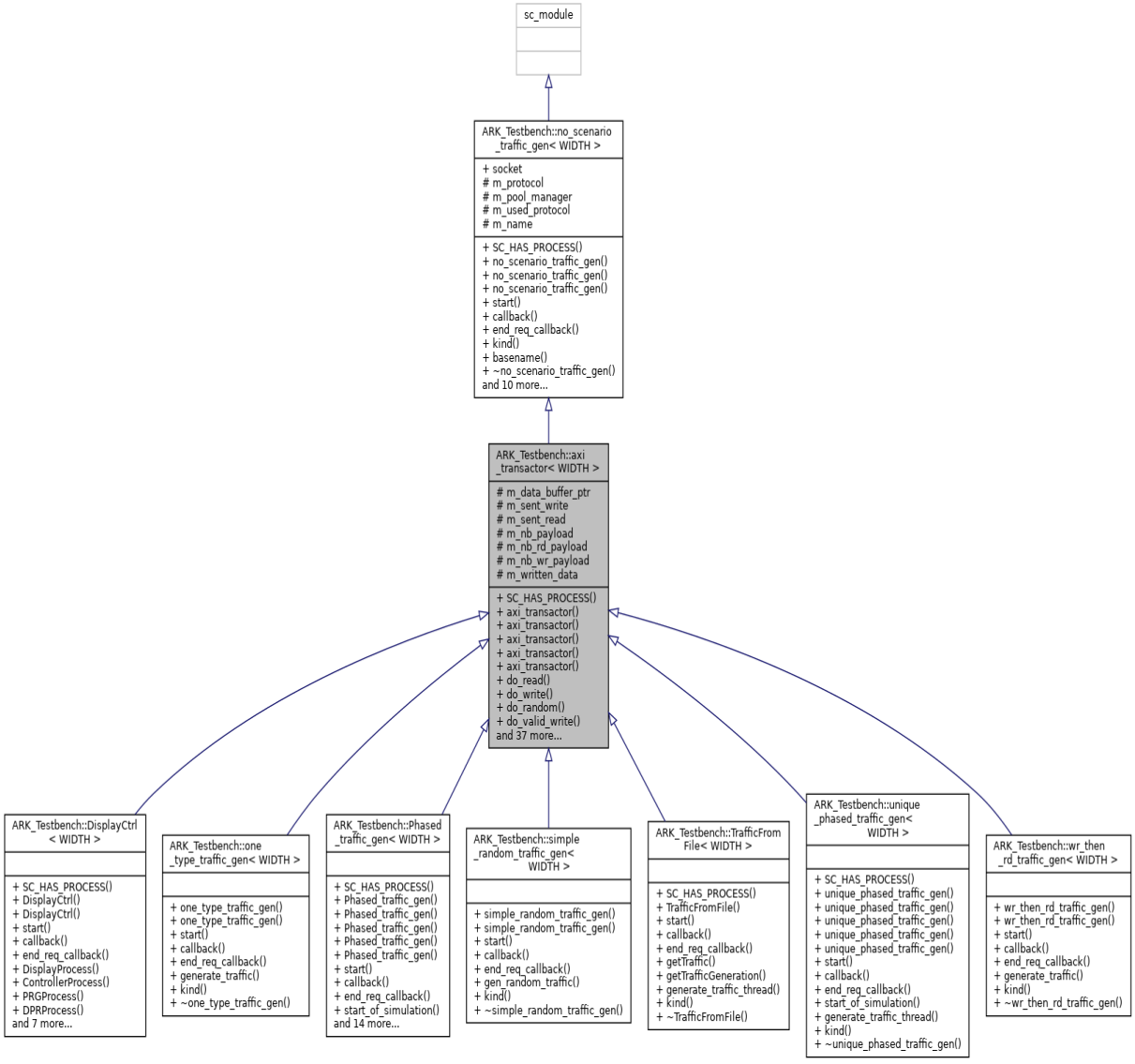


Figure 4.6: Traffic Generators inheritance

Each of these traffic generators also contains a port interface that communicates its status to the Master module via a signal when the status is updated. In this way,

it allows the Master module to move to the next power-related OPP if necessary. It is only used for power management. The power consumption of the traffic generators is outside the scope of this study.

4.3.3 Testbench - Memory model and transactions monitoring

We are not interested in developing a cycle-accurate memory model, as this is not the main objective of the thesis. There are various open-source and non-open-source tools that we can reuse. Considering that the accuracy of the memory model has a significant impact on the accuracy of the interconnect power estimation, we made several observations at different levels of abstraction for the memory system and compared the results. The main goal is to be able to find a sufficiently accurate solution that can be tested and evaluated with the PwClkARCH library. To this end, we implemented several approaches:

1. We created a **Simple Target/Memory module** that receives transactions, checks their validity and addresses, and adds delays for the accept and read/write process. A mapping process is implemented to track the order of transactions (using *AxID*) and to ensure that responses with identical *AxID* are sent in the order of reception.

2. We used the **DRAMPower memory power consumption estimation tool**. DRAMPower performs high-precision modeling of the power consumption of various DRAM operations, state transitions, and power-saving modes at the ESL level. We have created a parsing module to co-simulate our AT functional model with PwClkARCH and DRAMPower. In this way, we have a more accurate estimate of the functionality and power of the memory.

3. We used the **DRAMSys4.0 library** to completely replace our target/memory module. We added a simple power description using the PwClkARCH library and we have extracted the power related metrics (without DRAMPower).

4. We integrated our platform model into **Platform Architect Ultra** and simulated it with its memory models. This solution did not allow us to fully integrate PwClkARCH and extract power metrics. With the UPF methodology available in the tool, we cannot describe a power management strategy, so even if we create a power model, we will not be able to correlate the results with simulations or measurements. Moreover, we did not have access to a ”power model development guide” for custom created IPs (with TLM Creator). This solution will be presented at the end of this thesis as an additional work.

Each of these approaches has its advantages and disadvantages in terms of simulation time, accuracy and effort required.

Simple Target/Memory module approach

In the first one, we have a rather simple solution of a memory controller module with DRAM memory implementing the request callback. In this module, we define our strategy for memory access. Once *BEGIN_REQ* is received, it tests the payload, accepts/rejects it, and if it is accepted, it performs the *READ/WRITE* operation using the DRAM memory module. Then

we add the delay of the operation to the current simulation time and notify the target socket to send *BEGIN_RESP*. Configurable parameters in these models are *READ/WRITE* memory acceptance and response delays, memory size and width, and power contribution for different states, such as clock gated, IO down, and PLL down states. We automate the functional/power state evolution using simple module activity monitoring and hysteresis counting, which is a mechanism allowing to switch the power state after a given number of inactivity cycles (in our case we wait 32 cycles before switching the state). It is common for initiators (ex. CPUs) to sequentially send transactions with the same *AxID*, but the responses do not arrive in order, because for some reason the latest transaction is processed faster than the earliest (eg. opening/closing rows and banks in DRAM memory). In this case, we need to make sure that all transactions with the same *AxID* are returned to the initiator in order, so transactions should go First-In-First-Out (FIFO) when they have the same *AxID*. Therefore, the controller must be able to store and release them in the correct order. For this purpose we implement a request reordering map which is a container using *AxID* as the key and a queue of payloads as the value. It is used to track the order of execution of transactions and ensure correct rescheduling. We use a vector for the response register because the responses are stored in an unordered fashion and we need to retrieve them in order according to the request reordering map.

Usually, all modules use the PwClkARCH design elements for power calculation. Since the memory module is more specific and its power consumption is not similar to most other existing systems, we had to create a specific Design Element for it. The generic model of a Design Element in PwClkARCH is based on the classical equations for calculating dynamic power consumption and static power consumption. Here, the power states of the DRC are a bit more complex because, in addition to the clock gated state, we need to take into account at least the state where the IO buffers can be set to the OFF state, while the internal PLL is still active, and the state where the PLL is also OFF. To do this, we have defined a class *Design_elem_DRC* which inherits from the PwClkARCH class *Design_elem* in which we redefine the power consumption calculation model (Listing 4.10). In addition to the normal active mode of the module, we define 3 states: *CLK_GATED*, *IO_DOWN* and *PLL_DOWN* in which we directly set the value of the dynamic power consumption.

```
#define ACTIVITY_CLK_GATED (int)-1.0
#define ACTIVITY_IO_DOWN (int)-2.0
#define ACTIVITY_PLL_DOWN (int)-3.0
#define DRC_POWER_CLK_GATED (int)294.0 // 294.0
#define DRC_POWER_IO_DOWN (int)52.0 //  52.0
#define DRC_POWER_PLL_DOWN (int)0.0 // 0.0
#define PLL_DOWN_DELAY (int)5 // in US
#define IO_UP_PENALTY (int)30 // in NS

class Design_elem_DRC : public Design_elem {
public:
    Design_elem_DRC(Clock_Domain* pClock_Domain,Power_Domain* pDomain_name,
    sc_core::sc_object& a_obj,float capacitance, float activity, float
    Rleakage){}

    void Update_dpow(float activity) {
    Clock_Domain::log_power_update();
    Power_Domain::log_power_update();

    //Update DRC dynamic power using the new activity ratio
    if (activity >= 0)  {
      _dynamic_pow = _dynamic_pow_base*activity;
      _activity = activity ;
    } else if (activity == ACTIVITY_CLK_GATED)  {
      _dynamic_pow_fixed = DRC_POWER_CLK_GATED ;
      _dynamic_pow =  _dynamic_pow_fixed;
      _activity = 0.0;
    } else if (activity == ACTIVITY_IO_DOWN)  {
      _dynamic_pow_fixed = DRC_POWER_IO_DOWN ;
      _dynamic_pow =  _dynamic_pow_fixed;
      _activity = 0.0;
    } else if (activity == ACTIVITY_PLL_DOWN) {
      _dynamic_pow_fixed =  DRC_POWER_PLL_DOWN ;
      _dynamic_pow =  DRC_POWER_PLL_DOWN ;
      _activity = 0.0;
    } else{} //False Dynamic Power activity value

    Clock_Domain* CD_to_update = this->GetClock_Domain();
    CD_to_update->update_all_activity(CD_to_update);
    Power_Domain* PD_to_update = this->GetPower_Domain();
    PD_to_update->update_all_activity();
    }
};
```

Listing 4.10: DRC Design Element

Since the DRAM controller modules use the auto clock gating mechanism, they communicates the functional state of the module via particular values of the activity parameter *(negative values)*. For example, we can use a hysteresis approach, if no transactions are received for 32 cycles, then the DRAM controller transmits a corresponding activity to the *Design_elem_DRC* and puts it in the *CLK_GATED* state. One way to do this is to add a SC_THREAD in the functional module that is evaluated each time the callback method is activated (each time the payload phase is updated). We called this thread *check_state()* and the event it is sensitive to is *m_check_mode*. The implementation is given in Listing 4.11.

```
void DramCtrl::request_path_cb(tlm::tlm_generic_payload& pld, const tlm::
    tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) {
    drc_state = ACTIVE;
    check_idle_mode.cancel(); //cancel the idle mode if a new pld arrive
    check_idle_mode.notify();
    ...
}

void DramCtrl::check_state() {
    int activity_ratio = 0;
    while (true) {
        wait (m_check_mode);
        sc_core::sc_object& obj = dynamic_cast <sc_core::sc_object&>(*this);
        de=Design_elem_DRC::get_DE(obj);

        switch (drc_state) {
            case  ACTIVE :
                de->set_functional_state(false);
                this->notify();
                SetActivity(1.0);
                drc_state = CLK_GATED;
                m_check_mode.notify(32.0*pow(10,9)/m_Frequency,SC_NS);
                break;
            case CLK_GATED :
                if (m_pendingCounter == 0){ //CLK_GATED
                    activity_ratio = ACTIVITY_CLK_GATED;
                    SetActivity(activity_ratio);
                    de->set_functional_state(true);
                    drc_state=IO_DOWN;
                } else if (m_pendingCounter > 0)  drc_state = ACTIVE;
                else SC_REPORT_ERROR(basename()," Neg pending pld value");
                m_check_mode.notify(32.0*pow(10,9)/m_Frequency,SC_NS);
                break;
            case IO_DOWN :
                activity_ratio = ACTIVITY_IO_DOWN ;
                SetActivity(activity_ratio);
                drc_state=PLL_DOWN;
                m_check_mode.notify(ceil(PLL_DOWN_DELAY/2.0),SC_US);
                break;
            case PLL_DOWN :
                activity_ratio = ACTIVITY_PLL_DOWN ;
                SetActivity(activity_ratio);
                break;
        }
    }
}

void DramCtrl::SetActivity(float activity)
{
  sc_core::sc_object& obj = dynamic_cast <sc_core::sc_object&>(*this);
  de = Design_elem_DRC::get_DE(obj);
  de->Update_dpow(activity);
}
```

Listing 4.11: check_state implementation

DRAMPower/PWClkARCH co-simulation approach

The second solution was inspired by the work done in [5] presenting the co-simulation between a simple LT SystemC/TLM model, PwClkARCH and DRAMPower.

DRAMPower [112] [113] is an open source tool for DRAM power and energy estimation. It includes models of DDRs, LPDDRs and Wide IO DRAM JEDEC standard memories. These models are created using data sheets from memory vendors, including architectural, timing and current/voltage specifications. The tool uses two types of traces as inputs, which also determines its abstraction. These two types are command-level traces and transaction-level traces.

- At the control level, it is assumed that we are able to provide a DRAM controller model or directly a trace generated by such a controller. The command level traces contain the detailed memory commands (such as *ACT, RDA, WRA, SREN, SREX*...). The format used in the trace files is as follows:
  $$<timestamp>, <command>, <bank>$$
- At the transaction level, a simple DRAM controller (command scheduler) is considered internal to the tool. Thus, it only requires read or write commands for transactions, instead of full memory commands. The format it uses is as follows:
  $$<timestamp>, <READ \mid WRITE>, <address>$$

In the representation of a general SDRAM controller (Figure 4.7), we can see that there are front-end and back-end processing elements. At the front-end interface, the controller receives transactional type commands, schedules them to the appropriate time slots, and sends the scheduled requests to the back-end. At the back-end interface, transactions are transformed into command-level commands with physical addresses before being passed to the memory itself. Power consumption is dynamically evaluated at the back-end based on memory specifications, timing, states and traffic to memory.

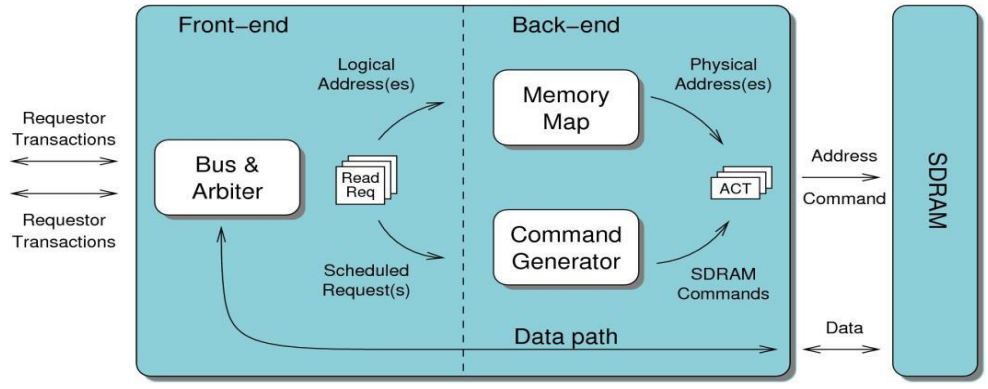


Figure 4.7: General SDRAM controller [7]

In our study, we developed a parser module, similar to the one presented in [5], but for the AT SystemC/TLM model, allowing to create transaction-level trace files. The connection between DRAMPower and PwClkARCH was set up in order to obtain the most accurate power consumption values while maintaining optimal simulation speed. For this purpose, it is preferable to calculate and extract the average power consumed for windows of several transactions and to reduce the exchanges between the two simulators. Calculating the power

consumed by DRAM on a transaction-by-transaction basis can lead to very erroneous power values. Thus, DRAMPower is invoked by the SystemC-TLM functional model only once per window of N read/write transactions. It then returns the average power consumed by the memory for that window and updates the overall consumption calculated by PwClkARCH (Figure 4.8). The larger the window, the more we lose accuracy and increase the simulation speed, as DRAMPower is rarely invoked. And conversely, the smaller the window size (maybe = 1), the more we decrease the simulation speed (but with limited accuracy improvement - window size must be greater than 1).

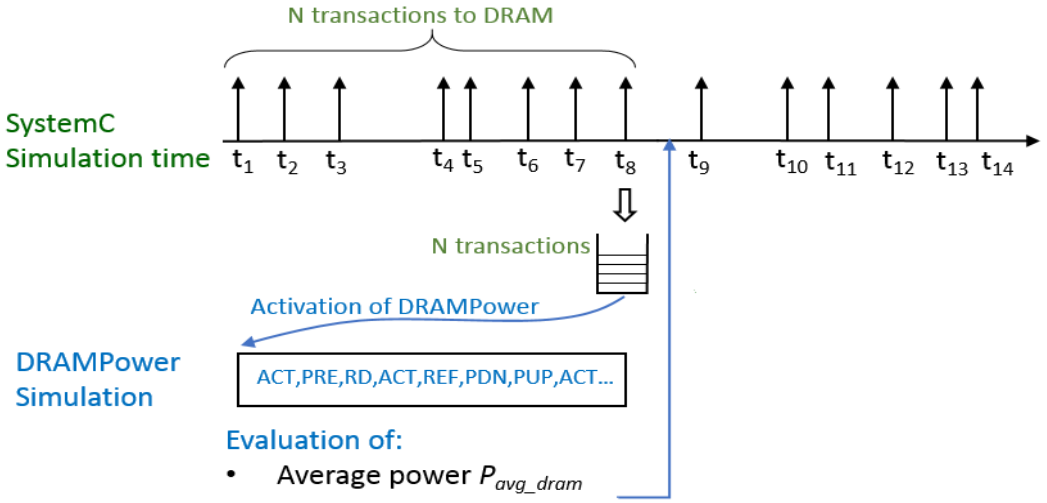


Figure 4.8: Connection of DRAMPower/SystemC-TLM model/PwClkARCH [5]

We also enhanced the DRAMPower tool to calculate the average response time per transaction for each window to improve the timing accuracy of the functional model. In order to handle the swapping between the two simulators (DRAMPower and PwClkARCH) and not lose too much accuracy in the power evaluation, we implemented a solution similar to the one cited above that allows to take into account the periodic refresh commands generated at periods t_{REFI} by the command scheduler. As in the first solution, we need to generate a specific design element for the memory module, which is part of the parsing between tools. We will call this design element *Design_elem_DRC* again.
The power consumption evaluation is divided into two steps:

1. In the **first step**, for each window, DRAMPower is invoked only for the first transaction in the window and evaluates its power. The particularity here is that the timestamp of this transaction is not when it occurs, but when the previous refresh command occurred. In this way, the power consumption we extract for this transaction is the sum of the IDLE power consumption and the power consumption of a single transaction.

2. Then, in the **second step**, we invoke DRAMPower for the evaluation of the entire window of transactions and the result is equal to the sum of the IDLE power consumption and the approximate power consumption of *N(=window size)* transactions. In order not to omit the IDLE power consumption of long idle periods, in the parser DRAM module we always keep track of the last occurred transaction (including refresh) and keep a constant power consumption, representing the power consumed during the idle

phase. This constant value is extracted from the DRAM characteristics. When a new transaction is received, we call the design element to stop using this constant value and revert to the DRAMPower model.

Consider the example in Figure 4.9 for the sequence of read/write transactions at the DRAM controller:

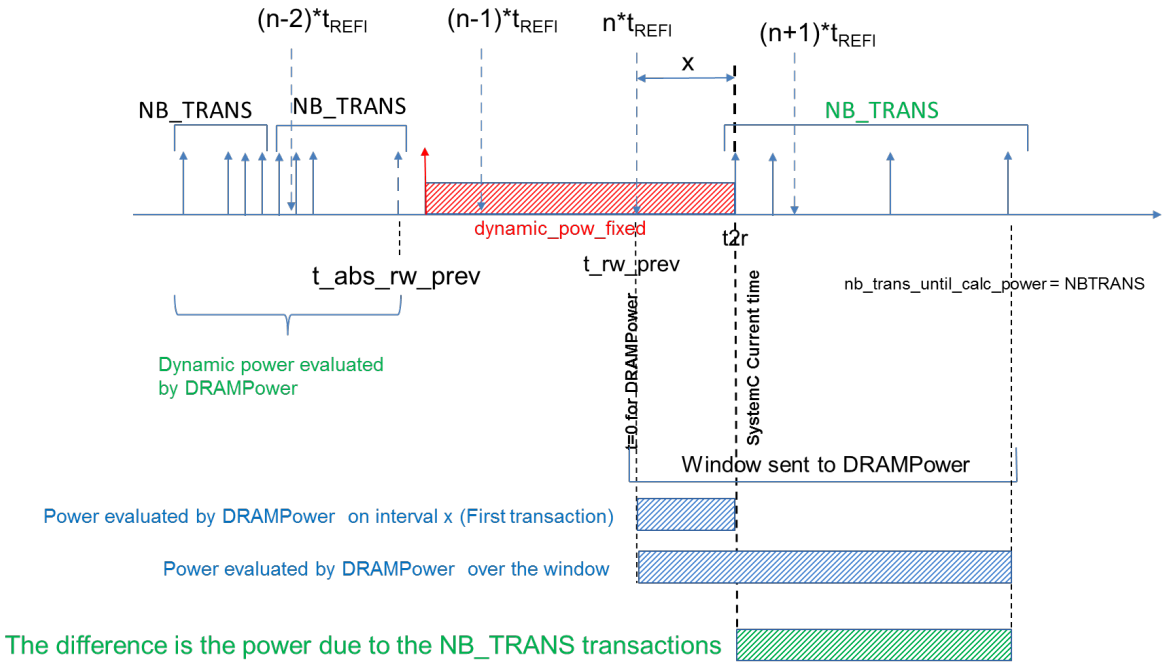


Figure 4.9: RD/WR Sequence example

The current time of the SystemC simulation is indicated by *t2r* with the first transaction of a new *NB_TRANS* transaction window. Prior to *t2r*, the DDR was in idle mode and the value of *dynamic_pow_fixed* set the power consumed by the DDR. The refresh events occur at regular intervals with a period equal to t_{REFI}. Thus, in the example, there are 2 refreshes that occurred during the idle state of the DDR and there will be another refresh $((n+1)*t_{REFI})$ that will occur during the new transaction window.

If at the end of the *NB_TRANS* transaction window we activate DRAMPower only on the window, DRAMPower may not take into account power due to refreshment because this refreshment will be simulated by DRAMPower at the t_{REFI} date of the beginning of the window and not relative to the date of the last refreshment $(n*t_{REFI})$. Similarly, we activate DRAMPower on a window that starts at SystemC's $(n * t_{REFI})$ date (thus in the past) to ensure the correct occurrences of refresh events in the future. However, during the interval *x*, the DRAMPower is already accounted for with *dynamic_pow_fixed*. Therefore, from the average power returned by DRAMPower over the window originating at *t_rw_prev*, we need to subtract the power consumed in the *x*-interval which corresponds to *[t_rw_prev, t2r]* i.e. from *t_rw_prev* until the first transaction in the window.
The value of *t_rw_prev* is:

$$t\_rw\_prev = \lfloor \frac{t2r}{t_{REFI}} \rfloor \times t_{REFI} \tag{4.1}$$

where $\lfloor x \rfloor$ represents the integer immediately below the real number x.

The evaluation in *Design_elem_DRC.cpp* of the average power *_dynamic_pow* of the *NB_TRANS* transactions starting from the first transaction is shown in Figure 4.10:

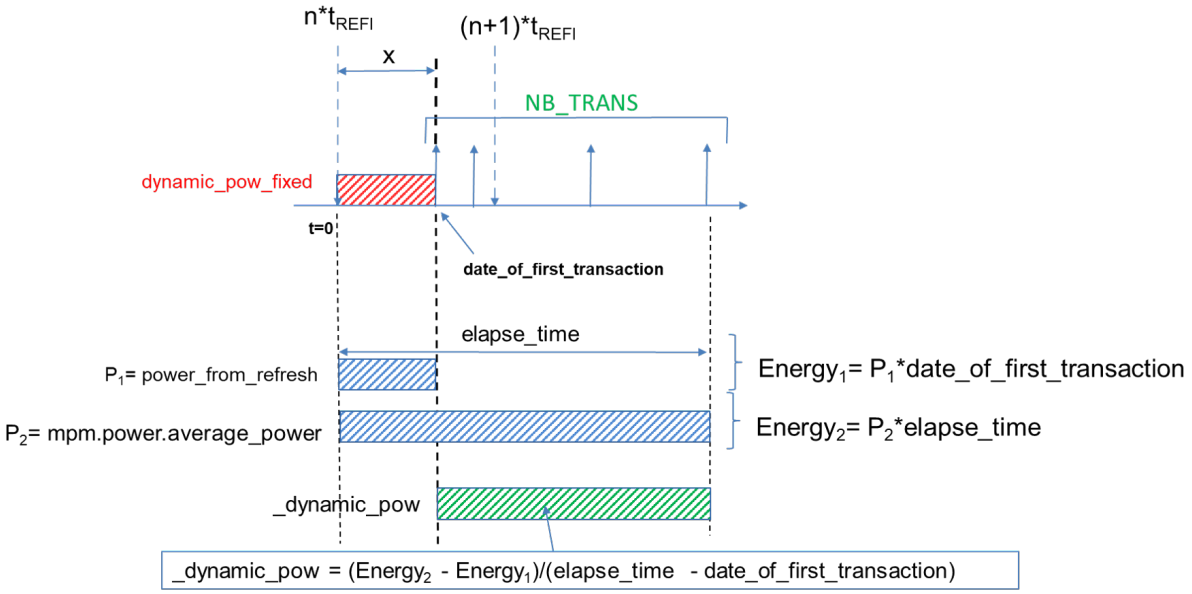


Figure 4.10: NB_TRANS average power evaluation

The evaluation of the average power from $t = 0$ (local time at DRAMPower) to $t = date\_of\_first\_transaction$ is done by sending to DRAMPower a first trace with the only first transaction dated at *date_of_first_transaction*. The value returned by DRAMPower is *power_from_refresh* in the C++ code and also noted P1 in Figure 4.10. Then DRAMPower is used on the complete trace from $t = 0$ to $t = elapse\_time$ and returns the average power *mpm.power.average_power* also noted P2 in the figure. This power can contain the refresh power if such events occur in the window, as is the case in the figure. We can then deduce *_dynamic_pow* by subtracting the two energies produced by P1 and P2, i.e. $P1 * date\_of\_first\_transaction$ and $P2 * elapse\_time$, and dividing the result by *elapse_time*.

As in the first solution, we need to add the methods *check_state()* and *SetActivity()* in the functional model, which are responsible for updating the calculations in the design element. The implementation of the *SetActivity()* method is the same as before (and as for any IP model), but the callback method (*request_path_cb*) and *check_state()* are modified to achieve the behavior shown here. The implementation should look like the simplified code snippet given in the appendix A. The approach we took to this solution is explained in the additional work section 5.7.1 from Chapter 5.

NOTE: In the latest version of DRAMPower, transaction-level traces are an obsolete feature and are no longer supported. The future version of DRAMPower will rely on simulators like DRAMSys4.0 and Ramulator for this. This was not the case during our study.

DRAMSys4.0/PWClkARCH co-simulation approach

In this third solution, we used the DRAMSys4.0 flexible and open source DRAM subsystem framework based on SystemC/TLM2.0 [8]. This framework incorporates a fast and fully cycle-accurate SystemC/TLM2.0 DRAM models and supports the latest JEDEC DRAM standards (e.g., DDR4, LPDDR4, GDDR6 and HBM2). The simulator uses the Approximately Timed (AT) coding style for DRAM behavior, which allows modeling pipelined behavior and out-of-order responses to initiators. This means that we can couple it directly to our TLM model. It also has its own custom protocol, called DRAM-AT, which defines application-specific phases for all DRAM commands [114] and allows projection of DRAM into TLM with cycle accuracy.

The framework can be used for timing, thermal, error and power modeling. Its polymorphic structure allows for easy configuration of individual blocks and can be reused for newer standards. Moreover, it is relatively new (github release in July 2020) and still under development. It is greatly improved over its predecessors (DRAMSys3.0 and DRAMSys2.0) in terms of simulation speed, accuracy, and configurability.

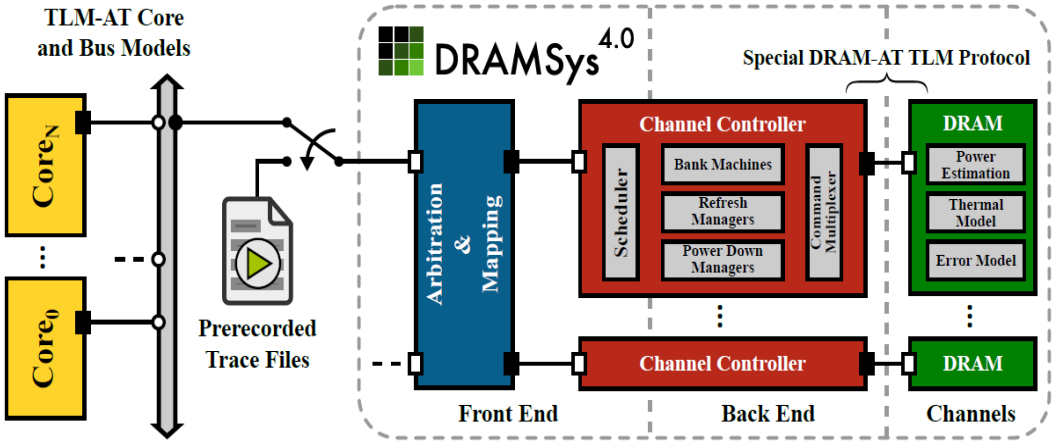


Figure 4.11: Architecture of DRAMSys4.0 [8]

For the power modeling, DRAMSys4.0 usually uses the DRAMPower tool.

In our case, as an initial study, we want to integrate DRAMSys4.0 into our framework and co-simulate it with PwClkARCH. Therefore, we decided not to use its DRAMPower extension and simply use it as an accurate DRAM functional model in our platforms. We wrapped it with its PwClkARCH power description and reused the same design element that was defined for the first solution (Simple Target/Memory module approach 4.3.3). Thus, we increase the accuracy of the functional module, which results in increased power consumption accuracy without the simulation speed penalties added by DRAMPower. Since it is entirely based on SystemC/TLM2.0, its integration into our project is quite easy. Exactly as we did with the DRAMPower solution, we cloned the project into our *Libs* directory (project structure described in 4.2) and compiled it as a library. Then we derived a new class from the top DRAMSys class (called *DRAMSys*). The constructor parameters of the new class are those required by DRAMSys (Listing 4.12).

```
1
2 class DRAMSysPow: public DRAMSys, public ARK_Utils::Observable, public
     Subject, public Assertions {
3 public:
4     SC_HAS_PROCESS(DRAMSysPow);
5     DRAMSysPow(sc_module_name nm, std::string simToRun, std::string
     pathToRes);
6
7     virtual ~DRAMSysPow();
8
9     void check_state(void);
10    void  SetActivity(float activity);
11    ...
12    ARK_Utils::ModuleObserver mModuleObserver;
13    ARK_Utils::PerfObserver mPerfObserver;
14    ...
15    int m_drc_state;
16    sc_event m_check_mode ;
17    Design_elem_DRC* de;
18    sc_core::sc_port<sc_signal_in_if<double>,0> activity_in;
19    sc_core::sc_signal<double> m_activitySig;
20 };
```

Listing 4.12: DRAMSys derived module

```
1 DRAMSysPow::DRAMSysPow(sc_module_name name, std::string simulationToRun, std
     ::string pathToResources):
2             DRAMSys(name, simulationToRun, pathToResources)
3                         ...
4 {
5     m_clock_period = Configuration::getInstance().memSpec->tCK;
6     m_Frequency = 1/(m_clock_period.to_seconds());
7     this->arbiter->activity_out(m_activitySig);
8     activity_in(m_activitySig);
9     drc_state = ACTIVE ;
10    ...
11    SC_THREAD(check_state);
12    SC_METHOD(update_activity);
13    sensitive << activity_in ;
14    dont_initialize();
15 }
16
17 void  DRAMSysPow::update_activity() {
18   drc_state = ACTIVE;
19   if (activity_in->read() == 1.0) {
20     m_check_mode.cancel();
21     m_check_mode.notify();
22   }
23 }
```

Listing 4.13: DRAMSys derived module constructor

As in this case we do not use our *SimpleTarget interfaces*, but the DRAM TLM sockets, we do not define a callback function. However, it is already defined in the *DRAMSys* module from which we are derived. The methods *check_state()*, *SetActivity()* and the specific design element *Design_elem_DRC* are exactly the same as in the first solution, but here we added an

additional SC_METHOD, called *update_activity()*, to activate them (Listing 4.13). In order to avoid the inclusion of intrusive code (and relationships) inside the DRAMSys library, we simply added a port/export connection between *DRAMSysPow* and the *Arbiter* inside DRAMSys by passing a signal that indicates when the module is active and when it is not. There are several ways to realize this simple connection between them, but the purpose of this solution is to make a first test if the co-simulation between these frameworks is possible.

The solution combining our SystemC/TLM2.0 framework, PwClkARCH and DRAMSys4.0 could be the optimal solution for accurate power estimation. Furthermore, it will be interesting to use this approach in combination with the DRAMPower power simulator. We have already done this work separately, once only with DRAMPower and once only with DRAMSys4.0. Thus, it could be quite easy to combine the two. In addition, DRAMSys introduced new JEDEC standard memory specifications (i.e. LPDDR4), which were not available in the DRAMPower tool. However, for this study, targeting the power consumption of the Switch Matrix, **we need a more accurate behavioral representation of the memory module, which is available in DRAMSys4.0 even without the DRAMPower extension.**

Platform Architecture Ultra approach

The fourth solution to provide our framework with an accurate memory model was to integrate our models into the Platform Architect Ultra tool. However, this solution is not open source and for this reason, we only had ”user level” access to this tool and we were not able to integrate PwClkARCH within it. It is worth mentioning that we ported our behavioral model to this platform using the TLM Creator (part of Platform Architect Ultra) and modifying our interfaces *(SimpleTarget/SimpleInitiator)* with their *FT* sockets (part of *SCML*) with AXI extensions. We were able to use models already available in the tool’s library. We used cycle-accurate memory controller and memory models, as well as trace-based and/or task-based traffic generators, generating workloads from *.STL* files. The integration was fairly straightforward, but it left us with limited options for estimating the power of our platforms. The approach we took to this solution is explained in the additional work section 5.7.2 from Chapter 5.

4.4 Switch Matrix system modeling - Design Under Test

In architectural exploration, the description of the behavioral models should be as close as possible to the actual hardware. At the ESL, the prototyping time is limited, therefore, the granularity and level of abstraction in the definition of the functional model must be kept as high as possible.

Some of the mandatory specifications that need to be considered at this stage are the communication protocol, the frequency of the component’s clock, the synchronous and asynchronous processes, the approximate time required for the component to process transactions, and the maximum number of details about the component’s functionality.

At this level, HDLs are not suitable because their semantics are relative to a lower level of abstraction and their development and simulation time is too large. This is why a high-level language, such as C++/SystemC, is more suitable for this type of modeling. Since SystemC is entirely based on C++, it therefore makes use of object-oriented programming (OOP), C++ data types and the C++ code compilation process. Inheriting these semantics, SystemC

provides an event-driven simulation interface that allows designers to simulate concurrent processes. Combined with its TLM2.0 layer for communication between different IPs, it becomes a rapid reference solution for ESL hardware prototyping.

4.4.1 Skeleton structure

When we want to describe a behavioral model, it is very important to model the entire transaction processing process in order to maintain functional accuracy and to meet the timing and deadlines. In order to reduce the simulation time, we do not use clock-sensitive methods/threads and we call the required processes only when they have to perform a certain task. To do this, we use *payload event queues (PEQ)* and event-sensitive methods and threads. For the sake of simplicity, it is also important to keep a fixed common structure for all IP components and add additional processing threads if needed. With this in mind, we decided to create skeletons that can be reused when creating/coding a new IP model. Following the structure of HDL languages, we created an entity class containing all ports, interfaces and instances of other subsystems. An architecture class is derived from it to define the description of the internal module.

Each IP model contains at least two *PEQ*:

- *AcceptReadPEQ* – notified when read transaction is received;
- *AcceptWritePEQ* – notified when write transaction is received;

These PEQs are used in two SC_THREAD processes that perform scheduling and processing of IP-specific requests:

- *AcceptReadThread()* – waits for the *AcceptReadPEQ* notification and performs its IP-specific request acceptance processing;
- *AcceptWriteThread()* – waits for the *AcceptWritePEQ* notification and performs its IP-specific request acceptance processing;

Once accepted, these threads wake up their corresponding forwarding/transfer methods using the *m_forwardRdEvent* and *m_forwardWrEvent* events. These events can be called at three different times during the simulation:

- When a request is accepted (end of *AcceptReadThread()* or *AcceptWriteThread()*);
- When a request has been transmitted to a targeted IP and that IP indicates receipt of the complete request (phase *END_REQ*);
- When the complete transaction *(Request+Data)* has been sent to a targeted IP and this IP indicates the reception of the complete transaction (phase *END_RESP*);

The forwarding methods are:

- *ForwardReadMethod()* – if the transmission channel is free, it forwards the scheduled read transaction;
- *ForwardWriteMethod()* – if the transmission channel is free, it forwards the scheduled write transaction;

Each intermediate module may require a reordering map. Let's imagine the structure of Figure 4.12. The interleaver will receive ordered transactions from the Initiator and forward them to *Target 0* and/or *Target 1*. These transactions may take a different amount of time to be processed by the targets, and we may have out-of-order responses. However, if the transactions have different *AxIDs*, we can send them without reordering them, but in case we receive transactions with the same *AxID* in disorder, we need to make sure that the first received transaction will be executed first *(FIFO)*. The structure of the reordering map is exactly the same that was used in DRAM memory modules.

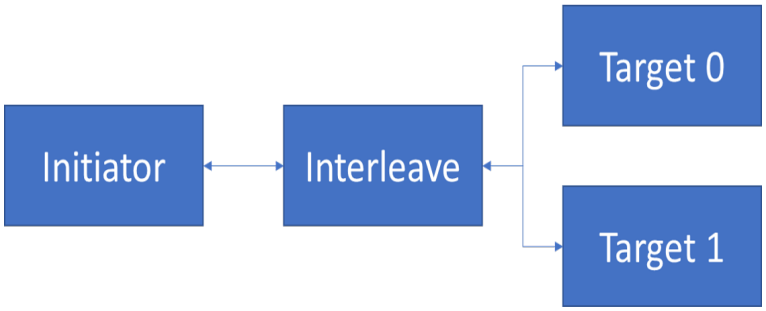


Figure 4.12: One initiator - Two targets structure example

Intermediate processes can be easily added, which guarantees the interoperability of the behavioral skeletons. An example of these skeletons is given in the Appendix B. Moreover, the separation between entity and architecture classes allows us to reuse them separately and test different configurations.

4.4.2 Switch Matrix model quick overview: architecture, power intent and power management strategy

The granularity of our model is the same as presented in the technology section 3.5. Figure 4.13 shows the example of the SM functional model in a i.MX8QM-based tesbench. The lower level modules *(the sequential modules)* are based on the skeletons presented earlier. The upper level modules *(Switch Matrix and Groups)* can be thought of as top-level modules containing the instances and connections of their child modules.

The implementation of the code corresponding to the internal functional behavior of these models cannot be shown here for confidentiality reasons.

However, we can mention that a configuration file is attached to each of these modules. The configuration strategy is passed as a parameter to the Switch Matrix, which in turn passes the correct configuration command to its child modules. Each of these modules is independent and reusable. In order to achieve configurability (and thus reusability), we have separated the algorithmic and transaction processing parts. Reusable algorithms are defined in external classes and their instantiation and application strategy in the modules is defined in the configuration passed by the user. Thus, algorithms, such as those responsible for *interleaving*, *HPR* and *UDP* in the *QOS* module, can be instantiated whenever they are needed. For example, the *interleaving* algorithm is not needed when we have a platform with a single DRAM memory, so the configuration of such platforms excludes it from the model,

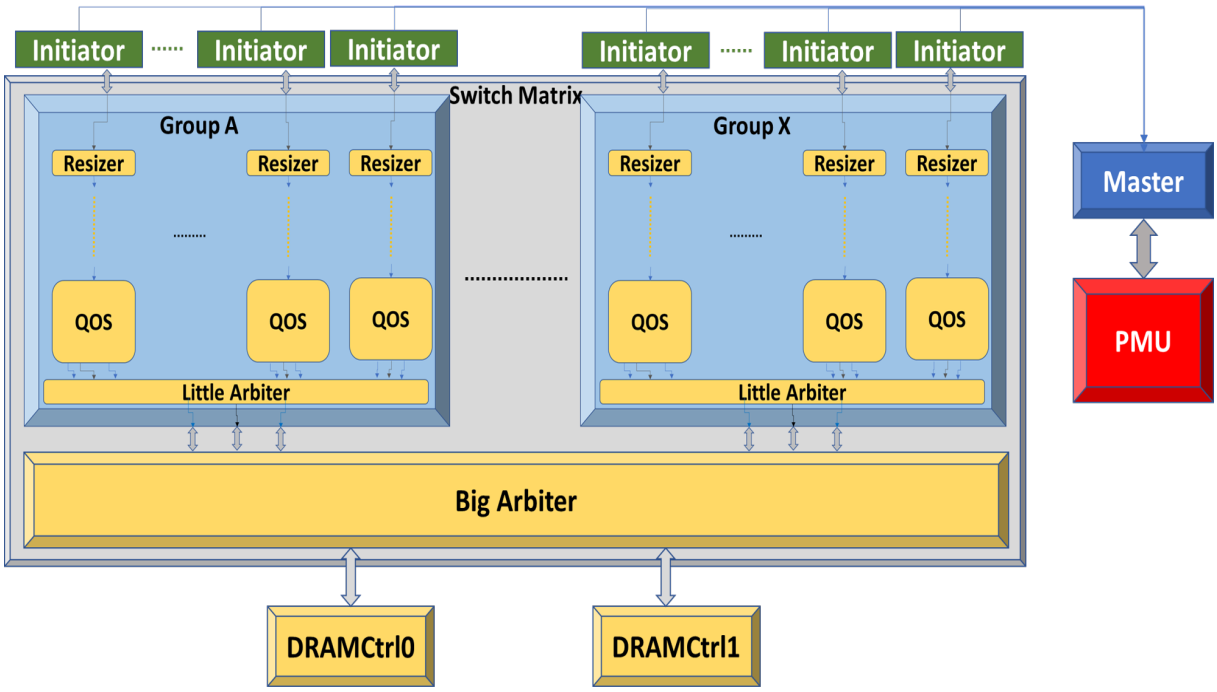


Figure 4.13: SM functional model blocks + Master/PMU

without creating a new separate module for each platform. The same applies to the number of sockets.

Each tool has a referenced method of use and recommended best practices. The same is true for the PwClkARCH library. The methodology behind PwClkARCH is quite simple. Each module must instantiate a signal port that will be connected to the external PwClkARCH Clock Manager. The developer must ensure that the clock frequency used for the synchronous components is synchronized with this external clock generated by PwClkARCH. We do not use real clock signals, because this can be very expensive in terms of simulation time due to all the clock switching and because the TLM2.0 non-blocking transport interfaces provide us with all the necessary information. Thus, the only information transmitted through this channel is the frequency associated with this module. PwClkARCH is responsible for updating the frequency of the component if there is a change (e.g. DVFS and Clock gating) from the Master and PMU modules. Then, following the previous recommendation, we can generate events based on the clock frequency and activity updates in the module.

Decoupling the performance model from the power intent is a very important point for interoperability, simulation speed and debugging. As explained in the section 3.4, we use the same approach as UPF, except that we do not directly associate IP instantiations with power intent (Power Domains and Clock Domains), but rather Design Element components referencing SystemC IP models. In all IP models except the memory and memory controller models, we use the standard Design Elements built into PwClkARCH. Since these DE calculate the static and dynamic power equations only when there is activity in the functional model or an update to the power/clock domain state, the impact on simulation speed is smaller (and also developer dependent). Furthermore, by separating the concerns between the performance and power models, the interoperability of the power intent is increased and we can reuse it for other IPs (e.g., when improving a reusable IP).

The functional models developed prior to the integration of the PwClkARCH library re-

quired minimal and fairly identical adaptations in order to be properly linked to the library. However, it is recommended to use event-sensitive threads or methods for SystemC/TLM model development, as this approach simplifies the control and notifications from the functional models to the power/clock models. This approach allows us to maintain a stronger separation.

Exactly as in the memory models presented in Section 4.3.3 whose clock is controlled internally by the automatic clock gating mechanism, we need to add the methods *check_state()* and *SetActivity()*. We can use a simple generic implementation (*ACTIVE*, *IDLE*, *CLK_GATED*) equivalent for all IPs or we can override it with specific clock management strategies for the IP that requires it. Then, in the behavioral model, it is enough to notify the method whenever there is an activity update (for example, when the IP is idle and *BEGIN_REQ* is received or when all registers are empty and there are no new transactions after the last *END_RESP*).

The Listing 4.14 presents an example of a generic implementation of *check_state()*:

```
#ifdef PWARCH
template(void)::check_state() {
    de=Design_elem::get_DE(obj);
    while(true) {
        wait(m_check_mode);
        if (m_State) { // Activate
          de->set_functional_state(false);
          SetActivity(1.0);
        } else {
          SetActivity(0.8);
          wait((32/m_Frequency)*pow(10,9),SC_NS);
          if (!m_State) // IDLE for 32 cycles -> Clk gating
            de->set_functional_state(true); // Activity=0
          else {} // Activated during the cycles count
        }
    }
}
#endif
```

Listing 4.14: Generic check_state() method

The method *de->set_functional_state(true/false)* allows us to change the clock states to *ON (=false)* and *OFF (=true)* directly from the subsystem. But in order to use it, the Master module must inform the Power Manager that some IPs are running in automatic clock gating mode using the *write_mem(PM_ENABLE_AUTO_CLK_GATING, 0xffff)* (process described in the section 3.4.3).

The Master module is responsible for reporting OPP state changes to the PMU module. The implementation of this Master module is done by the prototype developer, so there are different ways to do it. As we mentioned earlier in the PwClkARCH section (Master implementation, page 66), each initiator module can play the role of Master if the PMU is connected to the bus. In our case, we chose to separate it from the functional model (for the reasons mentioned earlier in the technology overview) and add signal port/export interfaces to the initiator modules and the master module carrying the control commands. The master reads the commands when they are updated and updates the OPP as necessary. A Design Element has also been associated with the Master because this unit receives the Power Manager's state end transition event (from PwClkARCH) when the Master requests a OPP change. To avoid the occurrence of energy consumption by this Master that has no direct

correspondent in the functional model of the Switch Matrix, we have parameterized it accordingly. Regardless of the scenario or testbench, we associate a capacitance of *0.0 F* with the Master design element and a leakage resistance of $10^6\Omega$ so that no significant power consumed by the Master pollutes the simulation results.

We still need to be able to simulate the IP model separately with and without a power intent specification. In addition, we need to ensure that we maintain consistency between these simulations and instantiate the same performance model in both simulations. For this reason, we need to keep a clean and easy way to define the type of simulation we want to perform. To do this, we add the *#ifdef PWARCH* directive to each piece of power-related code and use a simple main function (Listing 4.15) and a few command-line arguments to set simulation conditions, such as the type of simulation (perf or power-aware), verbosity level, and performance tracking and reporting (IP level and socket level).

```
int sc_main(int argc, char* argv[]) {
    ...
    // PERFORMANCE MODEL + TB instantiation
    SwitchMatrixSim0 top("SMarch0");

    // POWER INTENT instantiation
    #ifdef PWARCH
    PowerIntentSM0<SwitchMatrixSim0> PowerIntent(&top);
    #endif

    sc_start();
    ...
    return 0;
}
```

Listing 4.15: Separation between performance model and power intent

The *SwitchMatrixSim0* object represents the functional/performance model, given as a parameter to the *PowerIntentSM0* template containing the full description of the power intent of the reusable IP. Isolating the power intent from the functional model, allows us to reuse it with multiple IP models and test different IP options and their impact on power consumption. It also allows us to define and test different power intents for the same functional model and choose the most optimal solution. Since the functional/performance models and power intents are co-simulated, the accuracy of the power figures from the simulation is strongly related to the accuracy of the corresponding functional models and power intents. In the *PowerIntentSM0* template, all Design Elements, Power and Clock Domains, Supply Nets, Power Switches, the PST and CST and the OPPT are declared. The main components of the SM power intent are shown in Figure 4.14. In addition to the auto clock gating mechanism, we need to add a CST to dynamically change the values of the clock frequencies applied to the design elements, a PST to dynamically change the voltage values of the power supply networks and the state of the power switches attached to the power domains, and an OPPT to define the global states of the system. The tables in Figures 4.15, 4.16 and 4.17 summarize some example scenarios and module states corresponding to the global system state.

The frequency of a clock domain is calculated using its DPLL reference clock, the division factor from the CST and the pair of multiplication and division factors from the OPPT applied to the DPLL clock frequency. For example, the DPLL in the *CD_A0* clock domain have a reference clock running at 24MHz. In the clock state *ALL_ON*, its division factor is

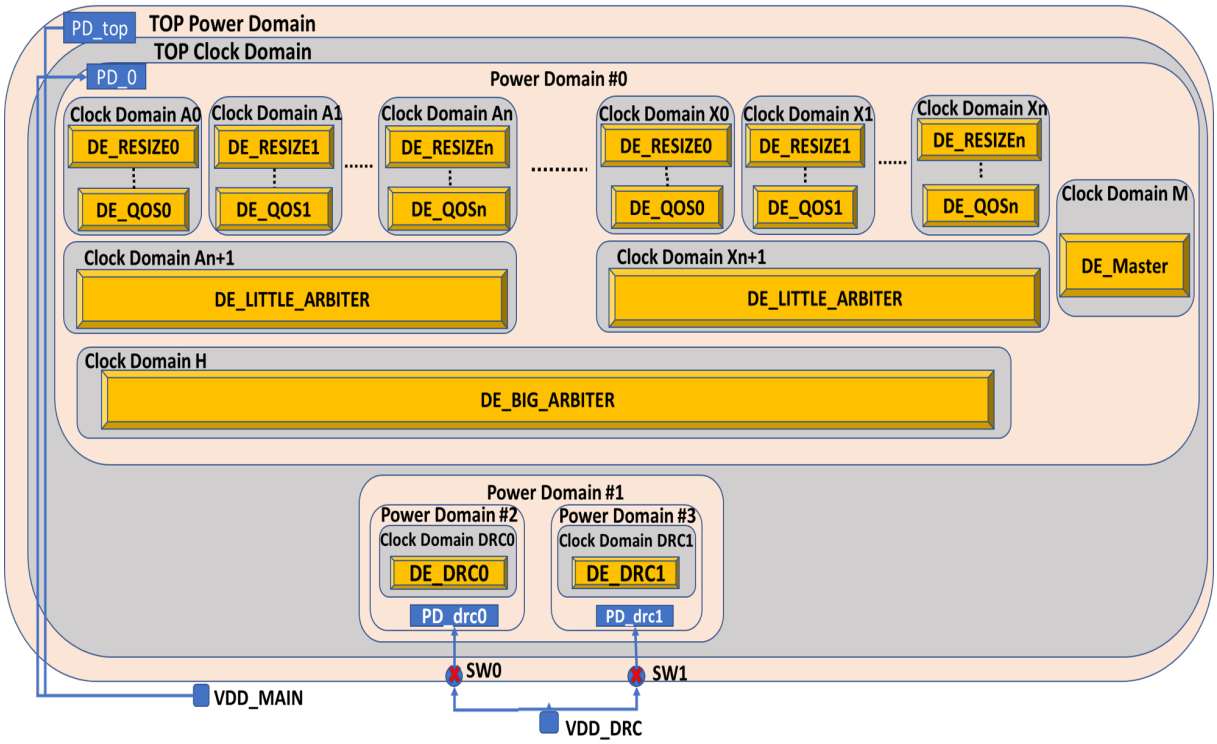


Figure 4.14: i.MX Power intent definition

CLOCK SRCS	States	CD_A0	...	CD_An+1	...	CD_X0	...	CD_Xn+1	CD_H	CD_DRC0	CD_DRC1	MASTER
CLKST_top	ALL_ON	4	4	4	4	4	4	4	4	4	4	4
CLKST_top	ALL_OFF	0	0	0	0	0	0	0	0	0	0	4
CLKST_top	CD_X_only	0	0	0	0	4	4	4	4	4	4	4
CLKST_top	DRC0_OFF	4	4	4	4	4	4	4	4	0	4	4
CLKST_top	DRCs_OFF	4	4	4	4	4	4	4	4	0	0	4

Figure 4.15: i.MX Clock State Table (CST) example

Source	States	VDD_MAIN	VDD_DRC	VDD_DRC0	VDD_DRC1
PST_top	ALL_ON	ON_H	ON	ON	ON
PST_top	ALL_OFF	ON_L	ON	OFF	OFF
PST_top	DRC1_IO_OFF	ON_H	ON	OFF	ON
PST_top	DRCs_IO_OFF	ON_H	ON	OFF	OFF

Figure 4.16: i.MX Power State Table (PST) example

4 (from CST) and in the OPP state *ALL_ON*, its multiplication and division factors are 500 and 4 respectively (from OPPT). Thus, following the simple equation (4.2) we calculate the

frequency applied to the clock domain.

$$f_{CD\_A0} = \frac{f_{ref\_clk}}{DivFact_{CST}} \times \frac{MultFact_{OPPT}}{DivFact_{OPPT}} = \frac{24 \times 10^6}{4} \times \frac{500}{4} = 750MHz \tag{4.2}$$

OPPStates	Nb DPLLs	A0	...	An+1	...	X0	...	Xn+1	H	DRC0	DRC1	Master	Index [PD:CD]
OPP1-ALL_ON	[hidden]	500,4	500,4	500,4	500,4	500,4	500,4	500,4	500,4	400,3	400,3	500,4	1,1
OPP2-DRCs_CLK_OFF	[hidden]	500,4	500,4	500,4	500,4	500,4	500,4	500,4	500,4	400,3	400,3	500,4	1,5
OPP3-ALL_OFF	[hidden]	500,4	500,4	500,4	500,4	500,4	500,4	500,4	500,4	400,3	400,3	500,4	2,2

Figure 4.17: i.MX Operating Performance Points (OPP) example

Figure 4.18 shows an example of a power management strategy:

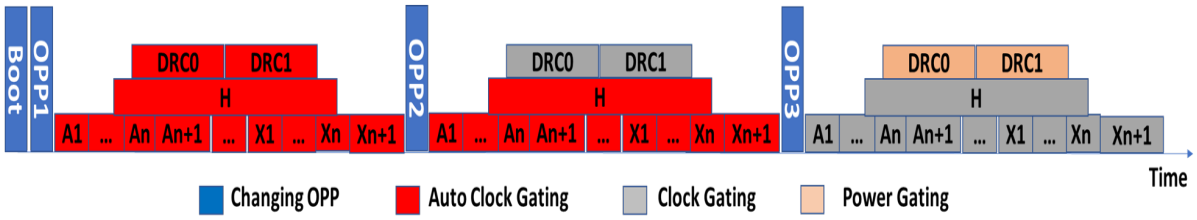


Figure 4.18: i.MX Power management strategy example

At the beginning of the simulation, PwClkARCH should start by setting the operation point *"Boot"* that puts all the system clocks into Bypass mode which results in a wait time equal to the maximum of the clock penalties. Next, we start the communication scenario by setting the operation points (OPPs) with respect to the global system state. In this example, after the Boot, we use the Master module to set the different operation points. We start with the operating point *ALL_ON* to activate all the subsystems.
Once activated, we send a command to the PMU to activate the automatic triggering of the clock (*write_mem(PM_ENABLE_AUTO_CLK_GATING, 0xffff)*). During this phase, all the activity of the scenario requiring a communication between the subsystems and the memories is processed. If we consider that the memories do not use the auto clock gating mechanism and that there is no more activity, we must define the next OPP, which is the *DRCs_CLK_OFF*. During this phase, the clocks of the two memories are gated/cut and if there is any activity inside, the PwClkARCH assertion mechanism will produce an error message indicating the timestamp and information about the activity. If there is no error in the behavior and the scenario is completed, the Master sets the last OPP *ALL_OFF* which performs Power Gating on both memories (OFF state), gates the clock of all other subsystems and changes their supply voltage from *ON_H* to *ON_L*. **We will illustrate it by applying this implementation on an industrialized reusable IP in the next chapter.**

4.5 Performance estimation and transactions monitoring approach

The performance estimation approach we use is based on transaction-level monitoring. It allows us to start with a higher functional abstraction and refine our behavioral model along the way. For the implementation, we use the Observer design pattern [115].

4.5.1 IP-level performance observation

Each IP model in our framework is derived from a subject *Observable*. To simplify this, we have created a module *IPBase* (an abstract class derived from *sc_module* and the class *Observable*) responsible for instantiating, adding and notifying all observers attached to it when necessary. Then, we use this class *IPBase* to derive our IP models. This class contains the request callback function *(request_path_cb(pld, phase, delay,idx))* and response callback function *(response_path_cb(pld, phase, delay,idx))* used for communication between the behavioral model and the interfaces *SimpleTarget/SimpleInitiator*. The request and response callback functions are the ones we attach to the callback registers of *SimpleTarget/SimpleInitiator*, so they are called every time a transaction is passed through the IP model. In their implementation, we simply collect the required data, notify the Observers and call the *manage[Forward/Backward]Path(pld, phase, delay, idx)* which is a pure virtual function in this class and implemented in every IP model derived from this class (Listing 4.16).

```
void request_path_cb(tlm::tlm_generic_payload& payload, const tlm::tlm_phase
    & phase, const sc_core::sc_time& delay, size_t idx) {
    #ifdef PERF
    if ( (phase == tlm::BEGIN_REQ) ) {
        this->m_PldArgs.payload = &payload;
        this->m_PldArgs.phase = &phase;
        this->m_PldArgs.delay = &delay;
        this->notifyObservers(&this->m_PldArgs);
    } else {} //do nothing
    #endif

    manageForwardPath(payload, phase, delay, idx);
}

void response_path_cb(tlm::tlm_generic_payload& payload, const tlm::
    tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) {
    #ifdef PERF
    if ( (phase == tlm::END_REQ) || (phase == tlm::BEGIN_RESP) || (phase ==
    tlm::END_RESP) ) {
        this->m_PldArgs.payload = &payload;
        this->m_PldArgs.phase = &phase;
        this->m_PldArgs.delay = &delay;
        this->notifyObservers(&this->m_PldArgs);
    } else {} //do nothing
    #endif

    manageBackwardPath(payload, phase, delay, idx);
}
```

Listing 4.16: request/response_path_cb() implementation

With these functions, we have access to IP occupancy information (such as transiting payloads, phase updates, transaction delays, and port indexes), which is sufficient to track module activity and thus extract performance information about it.

We can also run simulations without using the performance observers. To do so, we just need to encapsulate all the code related to performance estimation in a *#ifdef PERF* directive. In each *Makefile.config* file, we choose whether we want to compile the power and/or performance estimation code/tools or add other libraries/tools/emulators to the compilation. Then we can use or ignore them in the simulation (only if they have been compiled) by using the UNIX shell run command with some custom arguments. A larger code snippet of the *IPBase* class, containing the instantiation and inclusion of the performance observer, is provided in the Appendix B.

IP-level activity report

Using the performance observer, we can generate an IP activity report file for each IP in the testbench at the end of the simulation. This report contains certain statistics, such as the number of arbitrated transactions, the number of read/write transactions, the clock period, the simulation time, the approximate number of transactions per cycle, the approximate number of cycles per transaction, the number of seconds/cycles the IP was active, and the approximate latency of the requests/responses/complete transactions. This information is available for the joint calculation of read and write transaction data and separately for each type of transaction. An example of a generated activity report file is given in Listing 4.17.

```
Simulation.[module].[...].[submodule]_[enumeration]_PerformanceReport.report
************************************************************
**********************  Statistics  ************************
************************************************************
Nb of arbitrated transactions:[hidden]
Nb of arbitrated READ:[hidden]
Nb of arbitrated WRITE:[hidden]
IP clock period: [hidden] ps
Simulation time: [hidden] ps
Simulation cycles: [hidden]
BANDWIDTH (Approx Transactions per cycle): [hidden]
BANDWIDTH (Approx cycles per transaction): [hidden]
************************************************************
************************************************************
************************************************************
**********************  Performance ************************
************************************************************
The module was active for [hidden] ps or [hidden] cycles.
The maximum transaction latency is L_max = [hidden] ps.
The minimum transaction latency is L_min = [hidden] ps.
Scenario simulation executes [hidden] transactions for [hidden] ps
Theoretical 'wire' throughput = 1/(Tcycle) = [hidden] cycles/sec.
Average throughput = 1/(L_average) = [hidden] cycles/sec.
************************************************************
********************** READ Performance ************************
The maximum READ request latency is L_max = [hidden] ps.
The minimum READ request latency is L_min = [hidden] ps.
The average READ request latency for [hidden] transactions is [hidden] ps
The maximum READ response latency is L_max = [hidden] ps.
The minimum READ response latency is L_min = [hidden] ps.
The average READ response latency for [hidden] transactions is [hidden] ps
The maximum READ full transaction latency is L_max = [hidden] ps.
The minimum READ full transaction latency is L_min = [hidden] ps.
The average READ full transaction latency for [hidden] transactions is [
    hidden] ps
Scenario simulation executes [hidden] transactions for [hidden] ps
The last READ transaction was finished at [hidden] ps
nb_read_transactions*average_latency = [hidden] ps = [hidden] ps
Average READ throughput = 1/(L_average) = [hidden] transactions/sec.
************************************************************
********************** WRITE Performance ************************
The maximum WRITE request latency is L_max = [hidden] ps.
The minimum WRITE request latency is L_min = [hidden] ps.
...
...
************************************************************
```

Listing 4.17: Performance report file

If we need a more detailed report on the execution of each transaction, we can also optionally generate a debug payload tracking report for each IP. It generates a file containing information about all transactions passed through that IP.

```
Simulation.[module].[...].[submodule]_[enumeration]_pldTrack.report
************************************************************
Timestamp: [hidden] ps ; cmd: READ ; addr: [hidden] ; Start request time: [
    hidden] ps ; End request time: [hidden] ps ; Start response time: [hidden
    ] ps ; End response time: [hidden] ps

Timestamp: [hidden] ps ; cmd: WRITE ; addr: [hidden] ; Start request time: [
    hidden] ps ; End request time: [hidden] ps ; Start response time: [hidden
    ] ps ; End response time: [hidden] ps
....
Timestamp: [hidden] ps ; cmd: READ ; addr: [hidden] ; Start request time: [
    hidden] ps ; End request time: [hidden] ps ; Start response time: [hidden
    ] ps ; End response time: [hidden] ps

Timestamp: [hidden] ps ; cmd: WRITE ; addr: [hidden] ; Start request time: [
    hidden] ps ; End request time: [hidden] ps ; Start response time: [hidden
    ] ps ; End response time: [hidden] ps
************************************************************
```

Listing 4.18: Payloads tracking report file

In addition, when we enable the detailed performance reporting option, the performance observer generates a *.CSV* file for each IP containing the latency variations for the arbitrated transactions. In this way, we can track the latency variation for all payloads (Figure 4.19). The axes *[x;y]* are *[Payload Global ID; Payload duration]* where the *Payload Global ID* is a unique identifier to track payloads (payload counter - incremented in the order of transaction generation) and the *Payload duration* which is equal to the transaction end time *(END_RESP)* minus the transaction start time *(BEGIN_REQ)*. This allows us to track if there is any unexpected behavior during transaction processing by a given IP, such as transaction stacking, bottlenecks, etc.

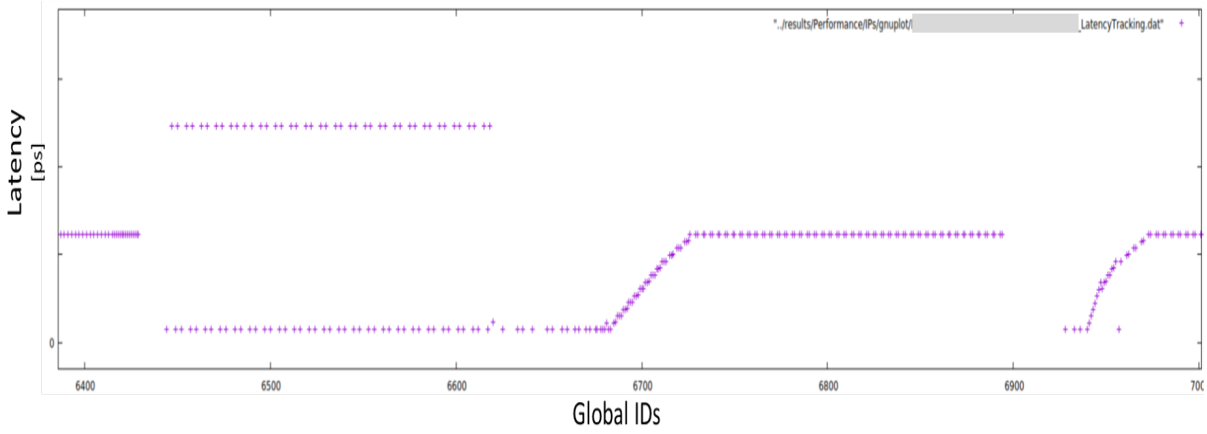


Figure 4.19: IP-level arbitrated transactions latency trace example

IP-level payload trace

The Performance Observer also adds TLM2.0 tracing capabilities to analyze TLM2.0 designs. To this end, it generates *VCD* files, which we can view using a tool such as *GTKWave* [116] for example. The tracing mechanism works by tracking payload objects throughout their lifetime as they travel through the transport interfaces (forward and backward). The performance observer automatically detects when a significant update occurs within the IP (e.g. activation of read/write req/resp channels, phase/address/data updates, number of read-/write arbitrated transactions etc...) and records the information. The traces of all IPs are grouped in one file, this way we can add all/many payload objects of all/many IP traces and we can track the path and timing of the transactions. The transaction view can be separated into *READ* and *WRITE* streams to observe them separately, or it can also be viewed in one stream regardless of the command. We can track recorded information such as command, address, data, state attributes, phase updates + handshakes, and number of arbitrated transactions (Figure 4.20).

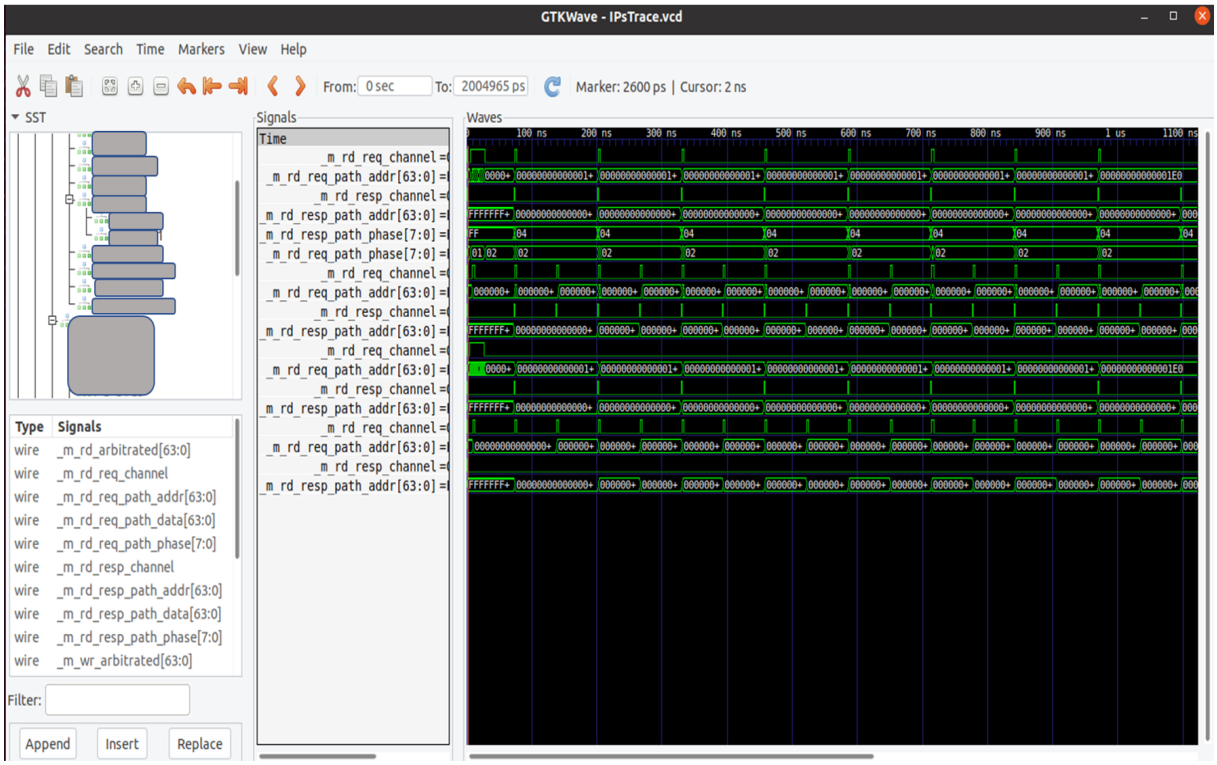


Figure 4.20: IP-level TLM2.0 GTKWave trace example

4.5.2 Socket-level performance observation

At the socket-level, we do not need to use an abstract class *IPBase*, as all protocol interfaces are derived from the abstract class *SimpleTarget/SimpleInitiator*, so they are perfectly suited to be used as subjects/observables. We use a socket observer module that extracts the same metrics as the performance observer, but at the socket level. It is added and notified by the *SimpleTarget/SimpleInitiator* modules. As in the *IPBase* class, the observer is notified whenever the forward and backward callback functions are called and an update occurs (Listing 4.19).

```
template<typename MODULE, unsigned int BUSWIDTH>
tlm::tlm_sync_enum SimpleTarget<MODULE, BUSWIDTH>::callBWChannel(tlm::
    tlm_generic_payload& payload, tlm::tlm_phase& phase, sc_core::sc_time&
    delay) {
  tlm::tlm_sync_enum status;
  mTrans.payload = &payload;
  mTrans.phase = &phase;
  mTrans.delay = &delay;
  mTrans.status = NULL;

  Observable::notifyObservers(&mTrans);
  status = socket->nb_transport_bw(payload, phase, delay);
  mTrans.status = &status;
  Observable::notifyObservers(&mTrans);

  return status;
}

template<typename MODULE, unsigned int BUSWIDTH>
tlm::tlm_sync_enum SimpleTarget<MODULE, BUSWIDTH>::nbTransportFWCallback(tlm
    ::tlm_generic_payload& payload, tlm::tlm_phase& phase, sc_core::sc_time&
    delay) {
  tlm::tlm_sync_enum status;
  mTrans.payload = &payload;
  mTrans.phase = &phase;
  mTrans.delay = &delay;
  mTrans.status = NULL;

  Observable::notifyObservers(&mTrans);
  status = manageFWCallback(payload, phase, delay);
  mTrans.status = &status;
  Observable::notifyObservers(&mTrans);

  return status;
}
```

Listing 4.19: Socket observer notification

In the same way as for the performance observer, the following metrics are observed here:

- *Latency* - the time it takes to execute a task or group of tasks or to read/write a data sample to a channel. Some applications have strict latency constraints (e.g., the execution of a data packet must not exceed a delay of $X\,ms$).

- *Throughput* - average message delivery rate - average rate of data transferred over a communication pattern during system execution *(bytes/sec or bytes/cycle)*

- *Mapping efficiency* - resource utilization and resource access contention. Resource utilization refers to the ability of the modeled system to efficiently use the resources of the architecture (e.g., one CPU is overloaded while another is underused). If the mapping is not appropriate or if the execution requirements are underestimated, the resource utilization will not be optimal.

Socket-level activity report and payload trace

Using this observer, we can extract the socket-level activity report and payload trace containing similar information to that extracted at the IP-level. Examples are given in Figures 4.21 and 4.22.

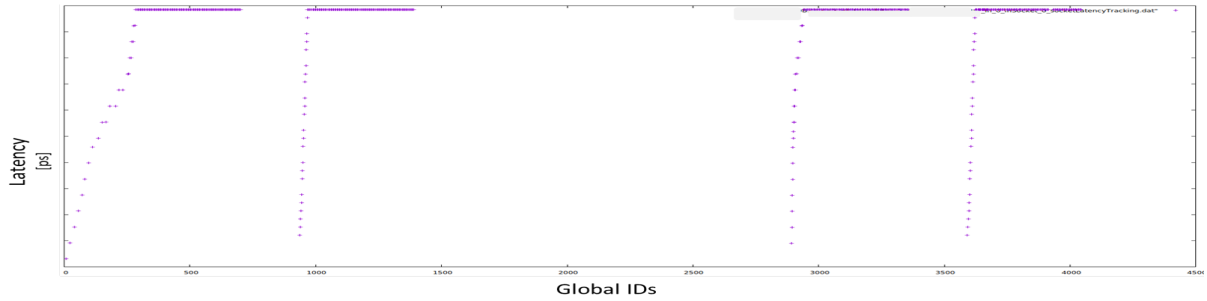


Figure 4.21: Socket-level arbitrated transactions latency trace example

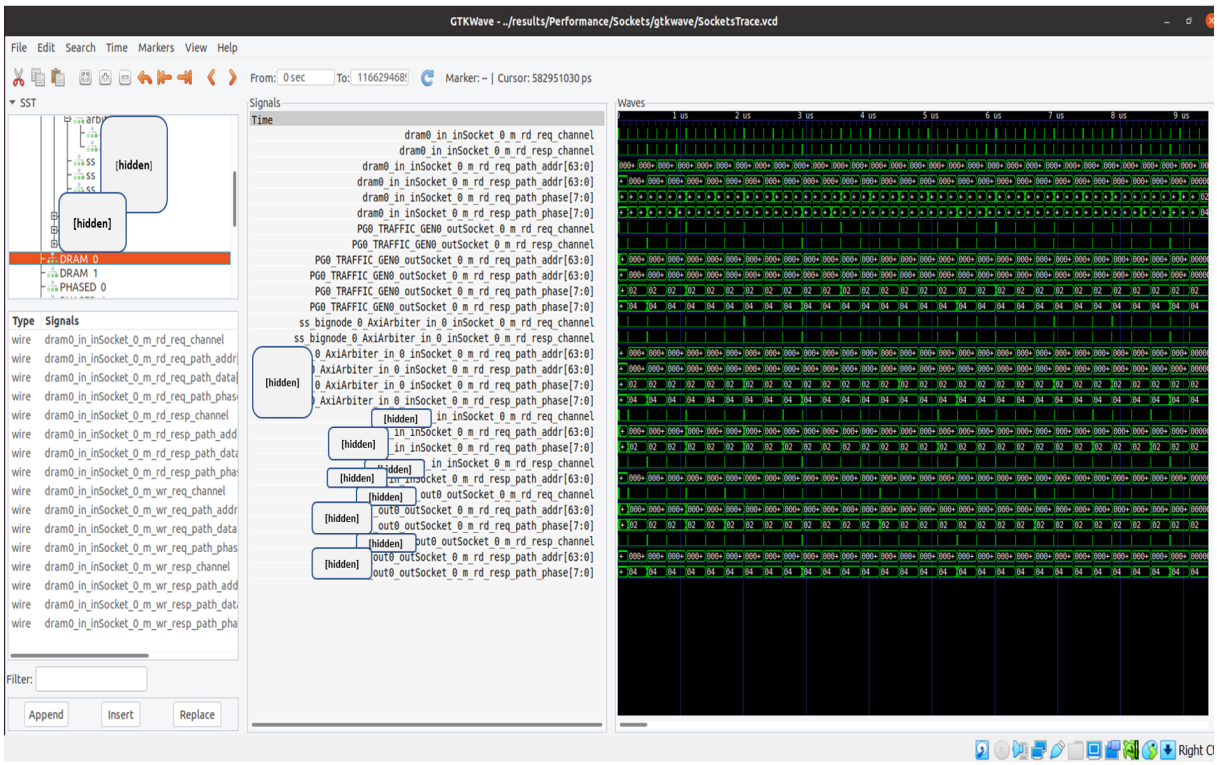


Figure 4.22: Socket-level TLM2.0 GTKWave trace example

4.5.3 Summary

The database generated by these TLM2.0 tracing mechanisms contains a complete history of payload object lifetimes in the simulation. This information is sufficiently representative of the behavior and performance of the simulated design. In addition, we have control over the details generated, which is useful for limiting simulation time.

4.6 PwClkARCH: Power metrics extraction and visualization

In the introductory section of PwClkARCH, we quickly mentioned that we use the *Gnuplot* plotting program [103] for visualization of extracted power metrics. During the simulation, PwClkARCH automatically and dynamically records several metrics in binary files *.gnu*. At the end of each simulation, a global C++ function generates two simple gnuplot scripts:

- ***gnuplot_fvi_script.cmd*** - visualizes the clock/activity variations of all design elements, the voltage variations on the supply networks of all power domains, the power state transitions (from OPPT) and the supply current evolution by power domain.

- ***gnuplot_p_script.cmd*** - visualization of power consumption by power domains, by clock domains, by design elements, static power by power domains, total power consumption and energy consumption.

We use these scripts to visualize the results of the simulations related to power. We can run them directly using the gnuplot command *load gnuplot_[fvi/p]_script.cmd* or we can choose to view separate parts of them. The simulation output traces generated with these gnuplot scripts will be shown in the next chapter (Chapter 5) containing the results and analysis of several use cases and their correlation with the silicon measurements.

In addition, PwClkARCH triggers exceptions in a Design Error Report File (DERF) when a violation occurs between the power model and the behavioral model. These are in the form of warnings with some information written to the DERF without interrupting the simulation to simplify debugging. In Figure 4.23 we have an example of an assertion violation indicating that an activity is noticed in a design element (its name is hidden for privacy reasons) while its clock is disabled. This means that the power management strategy that was applied in this simulation is not suitable for this behavior.

```
1 ******* opening Errors_Report_file *******
2 Specification Error at LINE : 55 at time 73329487 ps FILE : ../src/DE_Observer.cpp
3 Error label : Assume Assertion Violation : An activity is noticed in design element  while its clock is disabled !.
4 The Number of errors is 1
5 Specification Error at LINE : 55 at time 73329487 ps FILE : ../src/DE_Observer.cpp
6 Error label : Assume Assertion Violation : An activity is noticed in design element  while its clock is disabled !.
7 The Number of errors is 2
```

Plain Text ▾ Tab Width: 8 ▾ Ln 7, Col 26 ▾ INS

Figure 4.23: Design Error Report File (DERF)

In addition our framework and the library PwClkARCH included offer configurable verbosity for the reporting mechanism. We can ignore all reports and speed up the simulation, we can have multiple intermediate levels and a full debugging report. The verbosity of the behavioral model is controlled separately from the verbosity of PwClkARCH. This way, if we are aware of the type of problem, we can limit the simulation overhead due to reports.

4.7 Conclusion

In this chapter, we presented how we created the generic SystemC/TLM2.0 framework used for simulating different kinds of IPs and SoC models. We presented the ARM-based AXI communication protocol interfaces we created and their reusable skeleton simplifying the development of other protocols. We presented the generic traffic generators we use to stimulate our designs under test and some configurable application-specific traffic generators. We also presented the four solutions we investigated for power-aware DRAM memory target modeling and provided insights into how we applied them and what their advantages and disadvantages are. Next, we presented the generic and mostly reusable structures that we have created and used for the behavioral and energy modeling of the hierarchical structure of our DUT (the Switch Matrix) and several other blocks. We described the power intent used for all i.MX8 platforms studied during this thesis and gave an example of a power management strategy relevant to the actual power management of the targeted SoCs.

In addition, we presented the structure of our simple performance observation/monitoring approach and a brief presentation of how the power-related output files are dynamically generated by PwClkARCH during simulation, what they contain, and how we can use them. The purpose of this brief presentation is to link the presentation of the PwClkARCH tool (section 3.4) with our modeling strategy and visualization of the outputs to be analyzed.

In the next chapter, we will elaborate on the targeted reverse engineering approach we apply in this study (introduced in the Section 1.2 Context and Objective). We will first present an overview of the experimentation we performed, the power management strategy we used, and how we addressed the problem related to the parameters required by PwClkARCH for power estimation (such as activity, leakage resistance and load capacity). Next, we will take an in-depth look at the simulation of several use cases performed on the i.MX8QM and i.MX8QXP and the analysis of their results. At the end of the Chapter 5, we will present a brief summary of the correlation between the simulations and the silicon measurements and conclude with the analysis and results.

Chapter 5

i.MX8 power estimations, correlation and design flow integration

5.1 Experiments

5.1.1 Simulated use cases

Using the traffic generators we developed, we were able to run multiple scenario simulations on the i.MX8QM and i.MX8QXP Switch Matrix models and extract and visualize the power data for each. The most representative simulations will be presented and analyzed in this chapter. The first use case we will present is a generic multi-task use case that we used for the calibration of our power model. The second is a memory copy use case with different data sizes and the third is a display refresh use case with multiple resolutions and screen configurations.

5.1.2 Power management strategy

For almost all use cases on both platforms, we will use the same power management strategy. We have defined 3 OPPs as shown in Figure 5.1.

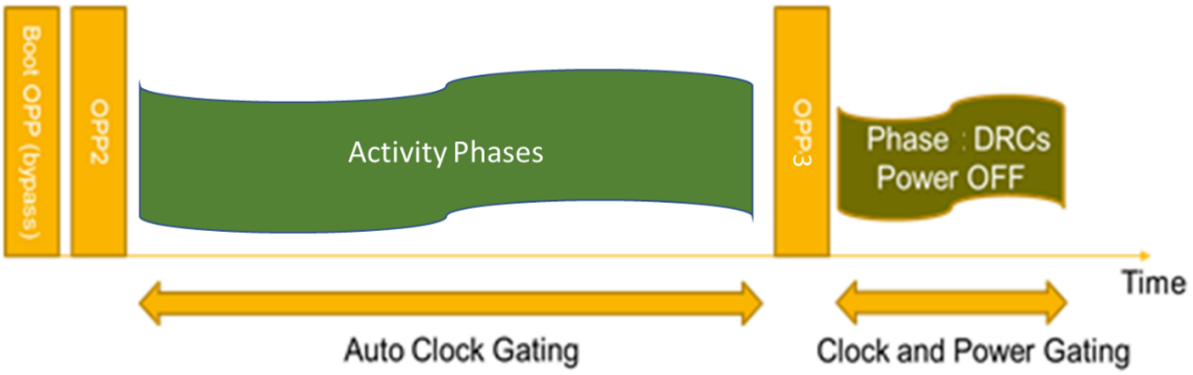


Figure 5.1: Generic Power management strategy

The first OPP *(OPP1-Boot)* acts as a boot and is automatically declared in the PwClkARCH library (no need to declare it in the OPPT). We just have to launch it using the Master

module. It changes the reference clock frequency to a so-called bypass mode, a quick one-cycle operation before traffic generation to reset the process/memories to a default value. The second OPP *(OPP2-ALL_ON)* indicates the activation of all subsystems. Once activated, we trigger the automatic clock gating mechanism on all the internal units of the Switch Matrix and the two memories. This is the main OPP and is kept running for the duration of the traffic generation/activity. Once the traffic is finished, the third OPP *(OPP3-ALL_OFF)* is activated. The third OPP applies clock gating to all subsystems, changes the supply voltage of all Switch Matrix internal units from *ON_H* to *ON_L*, and applies power gating to the memories.

5.1.3 PWClkARCH parameters supply

Estimating or providing the activity factor, leakage resistance, and load capacity values required by PwClkARCH for power calculations (presented on page 39 equation 3.10) can be considered one of the major difficulties of this approach related to the accuracy of power consumption values. In reality, these values are given by the developer only during the creation of the power intention. They are centralized in a header file and can be easily modified and corrected. The same top-down approach applied to functional model development, which starts with a more abstract view and is then refined, can be used for the power model.

We can start with non-accurate proven values, estimated using physical data, data sheets, technology information, and possibly values extracted from the previous generation or similar technologies. The following approaches can be taken sequentially (or individually) early in the power modeling process, depending on the data we have available.

Design initiation approach

As a starting point, we can use activity factors equal to 0 (when inactive) and 1 (when active) or similar strategies based on simple, straightforward values *(e.g., =0 when the clock is gated, =0.5 when the clock is not gated but the module is inactive, =1 when the unit is active, etc.)*.

If we can estimate or extract a top-level value for the leakage current of the entire platform or subsystem, we can estimate the leakage current of the lower-level child units (and the resistance per equation $V = R * I$) using physical data, such as the number of cells or simply a ratio of the estimated sizes/areas of the elements that are part of the target system.

The load capacitance of a transistor depends on the technology used and on parameters such as the permitivity ξ_{xo} and thickness t_{xo} of the insulators (formed between the doped regions, the gate and the substrate), and the height, length and width of the transistors. Thus, in the same way as for the leakage resistance, using the information on the number of cells, the technological data of the transistors and some assumptions, we can have first estimates more or less precise of this value.

At this level and with this approach, we may not have the most accurate power estimation values, but we will have sufficiently accurate power model behavior and an accurate visualization of the impact of power management strategies on the power consumed by the design. We will be able to test different power models and strategies on complex use cases applied to a realistic hardware behavioral model early in the design flow. Thus, we can initiate the built of an power management strategy well before the RTL model development begins.

However, as the phases of the design process progress, we will have more and more information about the physical data of the targeted design and will be able to refine the power model in parallel and further increase the accuracy of the power estimates. To take this a step further, we can then create a database that can be used to configure designs for subsequent generations of similar chips and have more accurate power models earlier in the flow.

Back-end physical data assumptions approach

An example of how to refine these metrics at a slightly later stage of the flow is to use data extracted from back-end teams using Computer-Aided Design (CAD) software tools performing IR-drop analysis reports (e.g. Red Hawk [117]), Chip Power Models (CPM) reports extracted with Spice simulations and synthesis reports.

- From the IR-drop reports, we can use the applied toggle-rates in their IR runs, which can be used as activity factors in our PwClkARCH models.
- From the CPM models in Cdie mode, we can extract information about the values of leakage resistances and load capacities of subsystems.
- From the synthesis reports, we can extract information about the number of cells for the subsystems.

If the CPM reports present information for an entire subsystem, we can use the cell count information to distribute the resistance and capacitance values among the internal blocks.

With this approach, we significantly increase the accuracy of our model, but at a later stage of the design flow. However, this approach is still very useful for design reuse (even for parallel development of similar designs).

Design reuse based approach

The final approach is based on the reuse of IPs from previous generations. We can have accurate data on the different parameters and cell counts, which can be used in the early stages of the development flow of new IPs/SoCs. Typically, if the new chip generation switches to a new transistor technology, multiplication/division factors are available and can be used to adapt the available data.

Approach applied in this study

The approach applied in our study is a kind of mixture of the three previous approaches. We followed them more or less sequentially, except that we initially had more data. Since we started the development of the SystemC/TLM2.0 model and the PwClkARCH model in parallel with the development of the i.MX8QM (which was in the medium to final production phase), we had a test version of the silicon and an initial power measurement curve of the silicon containing information about the power consumption for several states of the SoC. The power model parameters estimates for the functional model modules are defined from the Figure 5.2 below, which represents the results of measurements performed by NXP on a real circuit.

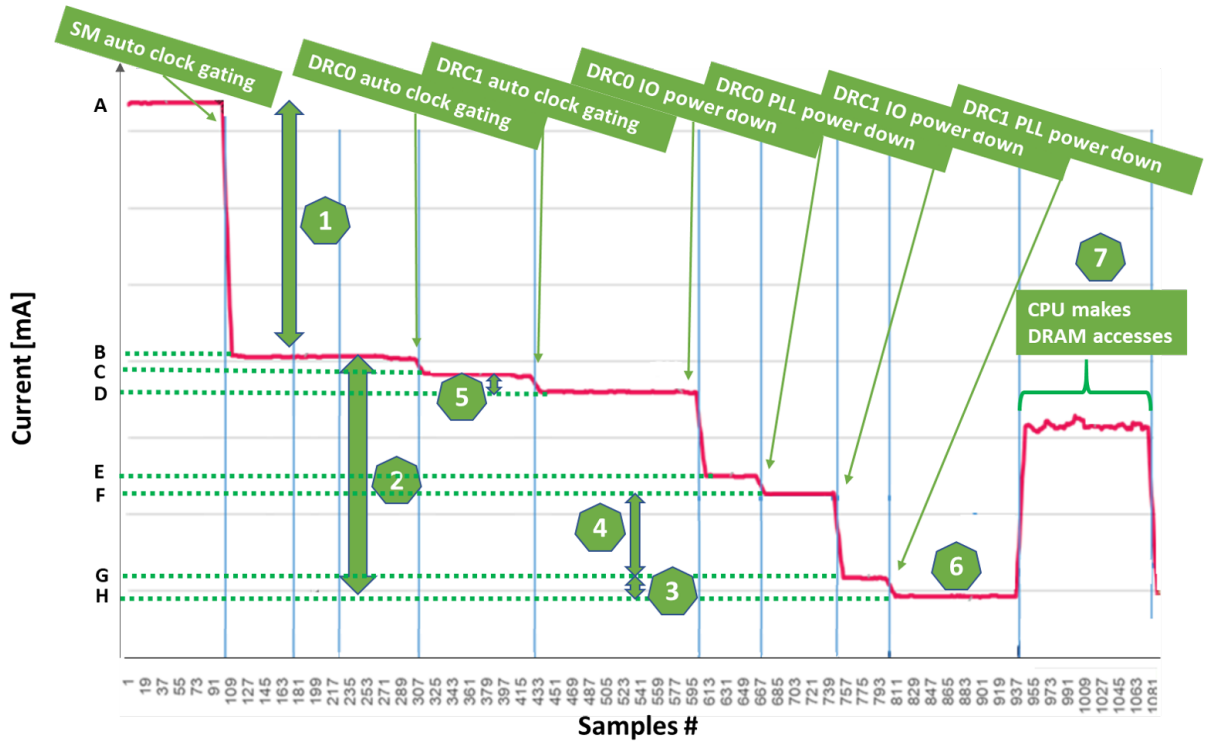


Figure 5.2: Initial silicon power measurements curve

From these measurements, we first estimate the current values for each SM unit described in the SystemC-TLM functional model, and then derive the leakage resistances and load capacitances of these units. From event *109* to event *937* in Figure 5.2, only DRAM controllers (DRCs) are active, except for SM-related leakage currents that are present in this interval. At event *109*, we disable the SM clock. Therefore, the dynamic consumption of the SM with the clock active but without transaction transfer (1) is derived by:

$$I_{dyn_{SM\_notransfer}} = A - B = (1)[mA] \tag{5.1}$$

The two DRCs consume (2):

$$I_{dyn_{DRCs\_notransfer}} = B - H = (2)[mA] \tag{5.2}$$

or, each DRC consumes:

$$I_{dyn_{DRC\_notransfer}} = \frac{B - H}{2}[mA] \tag{5.3}$$

So, for one DRC:

- The PLL of the DRC consumes $I_{dyn_{DRC\_notransfer\_PLL}} = G - H = (3)[mA]$
- The IOs of the DRC consume $I_{dyn_{DRC\_notransfer\_IOs}} = F - G = (4)[mA]$
- The clock of a DRC consumes $I_{dyn_{DRC\_notransfer\_clk}} = C - D = (5)[mA]$

The total power consumption for one DRC is therefore:

$$I_{dyn_{DRC\_notransfer\_total}} = 3 + 4 + 5 = I_{dyn_{DRC\_notransfer}}[mA] \quad (5.4)$$

Since the entire SM consumes 1, we can look at the ratio of the number of cells between the Groups and the Big Arbiter. This approach is not completely accurate for dynamic energy consumption, but it is the only way we have found to go down in granularity and is sufficiently representative in this case. From this, we derive the dynamic energy consumption of the blocks internal to the SMs.

We assume that 6 is the leakage current consumption of the SM+DRCs ($I_{leakage_{SM+DRCs}}$). The approach using the number of cells/ratio is more accurate in this case and we can find the leakage current consumption of each of the internal subsystems and units. In our case, we had the number of cells, but in the case of developing a new design, we need to estimate them at least as size difference ratios.

Using the information from Figure 5.2, we can see that from event *937* to event *1081*, a CPU performs DDR accesses (period marked as 7). This can be seen as the consumption when there is a sequential path activation of one Group (one clock domain in the SM) and one DRC at a time. All other components are clock gated. We have already found the dynamic consumption of each block when there is no activity, so if we subtract from 7 the value of one active sequential path + one DRC and the value of the leakage current consumption, we can deduce the current related to transactions activity. We distribute this value among the activated components (sequential path and DRCs). Using the ratios between the traffic-free consumption of the sequential path and the DRCs, we deduce what percentage of the transaction activity consumption should be associated with each unit.

These values and some assumptions allow us to calculate the values of the load capacity and leakage resistance of all blocks.

1. The capacity is calculated using the equation $C = \frac{I}{VF}$ and the assumption that 100% activity is linked to transactions. The minimum used activity factor of each block is calculated by dividing the dynamic current of the active block without transactions by the dynamic current of the active block with transactions. This is the activity rate when the block is powered up, its clock is active but no transactions are active in the block. This consumption occurs, for example, in the time intervals when the activity counter (hysteresis mechanism) has not yet reached its maximum value and no transaction is present in the block.

2. The leakage resistances (deduced from $V = RI$) of the units can be deduced by knowing the supply voltage. We already know the leakage current of each unit, so we can find an equivalent resistance distributed among the units (with the help of the cell count reports).

In this approach, we considered that the leakage resistance and load capacity values are not available and that we have some reused results from previous simulations/measures or similar technologies. We did not have specific information on the activity ratios that can be

used for activity factors, so we took the assumption mentioned above. This approach allowed us to calibrate our first power model fairly accurately and extract some initial estimates.

Later in the design process, we had the back-end physical information (presented in the Back-end physical data assumptions approach, page 135) and could easily refine our model. The results were more refined, but very close to the initial results, meaning that even with the calibration approach, we got pretty good estimates. Once we had the accurate power models for the i.MX8QM, we could simply reuse them for the i.MX8QXP case. This reuse of the power model saved a lot of development effort and significantly reduced the time required to model and analyze the power of the i.MX8QXP. The simulation results extracted from these models were correlated with silicon measurements for similar use cases. Then, these models and power parameters were reused for modeling the new, more complete generation of i.MX8 SoCs, which we will call i.MX8nexGen. This new generation of SoCs is in the very early stages of the design flow. This means that we are able to extract power consumption metrics and behaviors from system-level simulations as early as the specification phase of the design.

In the next sections of this chapter, we will take a closer look at some of the analysis performed on the i.MX8QM and i.MX8QXP.

5.2 Power and performance investigation on NXP i.MX8QM SoC

5.2.1 Generic multi-task calibration use cases

These types of generic simulations were extremely useful during the development of the functional model, as the power curves gave us a clear view of the activity of each module and simplified the debugging and analysis phase. In addition, we had our first power estimates and a promising first correlation with silicon. This can be a good first step when developing new platforms and we don't have IP models for traffic generation (like CPU, GPU, Display Controllers....).

In one of our first test cases, we consider the i.MX8QM architecture integrating several transaction initiators and two LPDDR DRAMs connected via the Switch Matrix. The tested use case consists of 4 functional phases (Figure 5.3). In the first phase, all data traffic generators send *512 READ/WRITE* transactions to recreate the complete Switch Matrix block activity. In the second phase, there is a short idle period of about *10µs* that is used to test the clock management. In the third phase, only half of our traffic generators remain active for an additional *512 READ/WRITE* transactions, which allows us to test the automatic clock gating mechanism. In the fourth and final phase, when there is no activity, we test the power management. The reason we chose this use case is that we have a real silicon power measurement extracted from an existing chip similar to this type of use case.

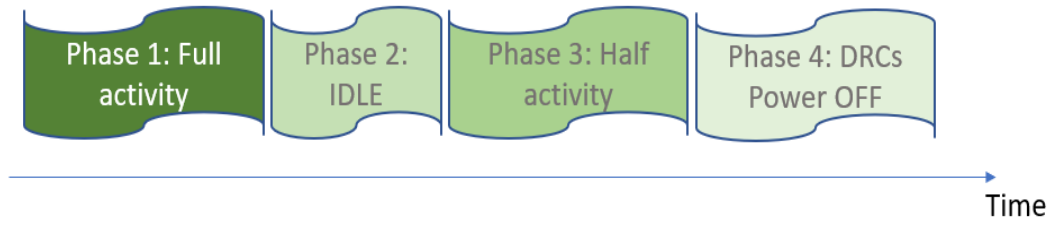


Figure 5.3: Functional use case

We merged our functional use case with the power intent and power management strategy (Figure 5.4) and simulated.

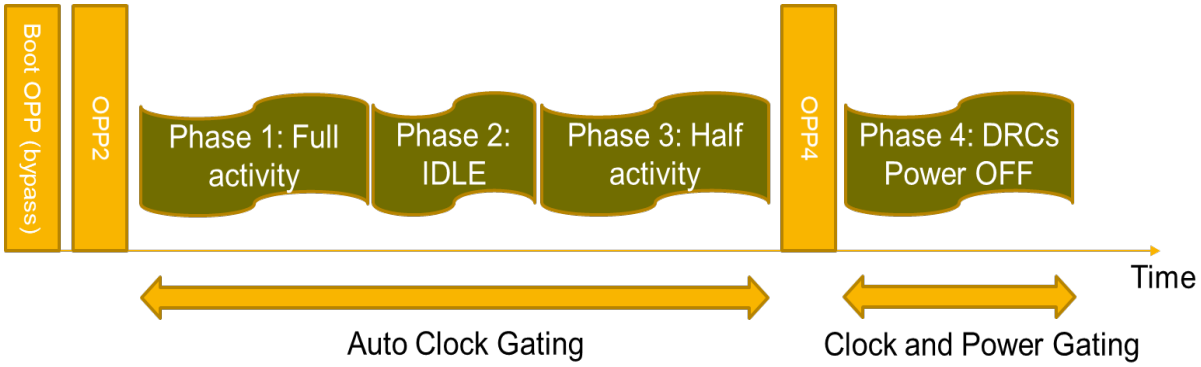


Figure 5.4: Power management strategy

In Figure 5.5, we can observe the different phases of the use case and the effect of the

applied power management strategy. The figure shows the simulated total power consumed by the design (green curve) and the static power consumption (yellow curve).

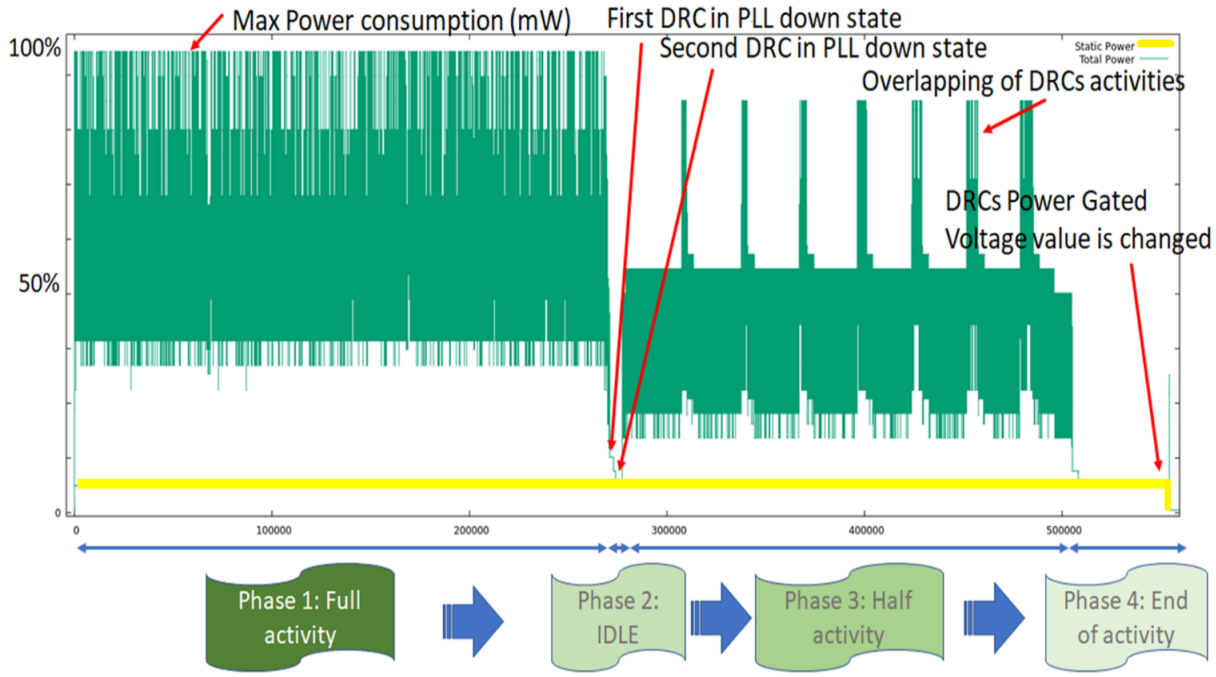


Figure 5.5: Calibration use case: Overall power consumption profile in test case [mW]

In the first phase, all traffic generators are active, so all subsystems are active as well. The first half of the traffic generators start with addresses equal to *0x00000* (i.e. to DRC0) while the other half starts with addresses equal to *0x01000*, i.e. pointing to DRC1 since this is the next 4K data block. In this way both memories are active at the same time. The maximum of the total power obtained by simulation (hidden in the figure) corresponds well to the value deduced from Figure 5.2 using the design element parameters calculated with the approach described in the previous section.

During the second phase, dynamic power is completely eliminated because all clocks are in clock gated mode, the IOs and PLLs of the DRCs are in the OFF state and the only power consumption is static. The state change of the DRCs is sequential, as the execution of the transactions is done sequentially. This may be because the execution time of one transaction is longer than the other (i.e., the read execution is slower than the write execution), or the execution of the transactions started at different times. The PLL state change is also visible through the configurable delay/penalty *(around 2µs)*. Again, the static consumption corresponds to the value deduced from Figure 5.2.

In the third phase, only half of the traffic generators have been activated. Therefore, only half of the SM sequential paths (in a Group) are also active with the Little Arbiter and Big Arbiter units. In Figure 5.5, we can observe that during the third phase, the consumption is almost halved.

The main reason for this is that there is globally only one active DRC at a time, which is more clearly seen in Figure 5.6 where DRC1 (green curve) and DRC0 (yellow curve) change state one after another in the third phase due to the interleaving applied in the QOS unit. Figure 5.6 represents the total power (dynamic + static) of each design element and, therefore, each curve describes the power state changes (and thus the functional idle vs.

active states) performed by each module during the simulation. The power consumption spikes in the first and third phases are due to a short overlap between two pipelined memories, i.e., one transaction started for DRC0 and another ends for DRC1.

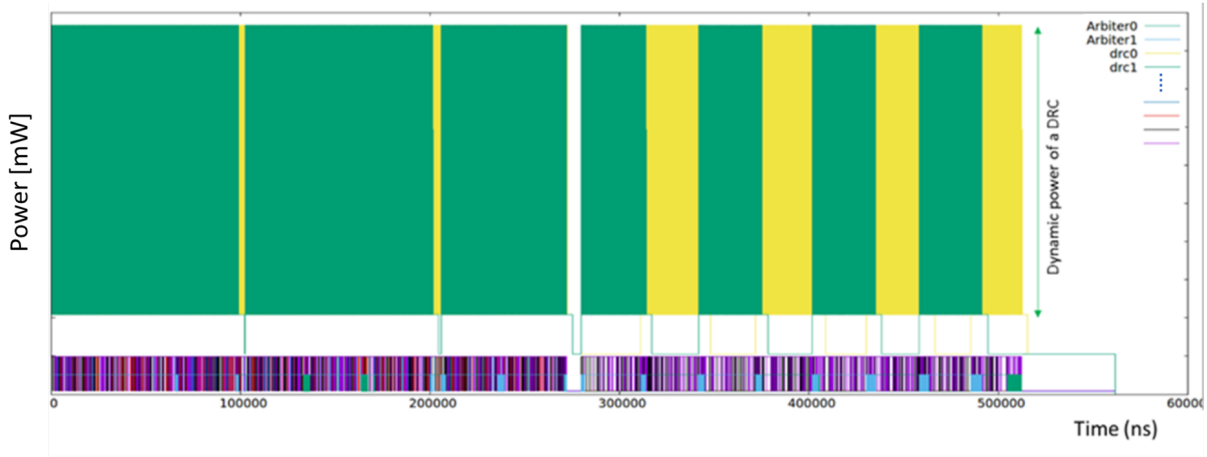


Figure 5.6: Calibration use case: Total power consumption per Design Element [mW]

Note: It is important to mention here that in Figure 5.6 we do not observe the activity of DRC0 (yellow curve) during the first phase, because it is hidden by the activity of DRC1 (green curve) and this is only valid for this visualization. This can be easily verified by hiding the consumption of DRC1 or by changing the colors.

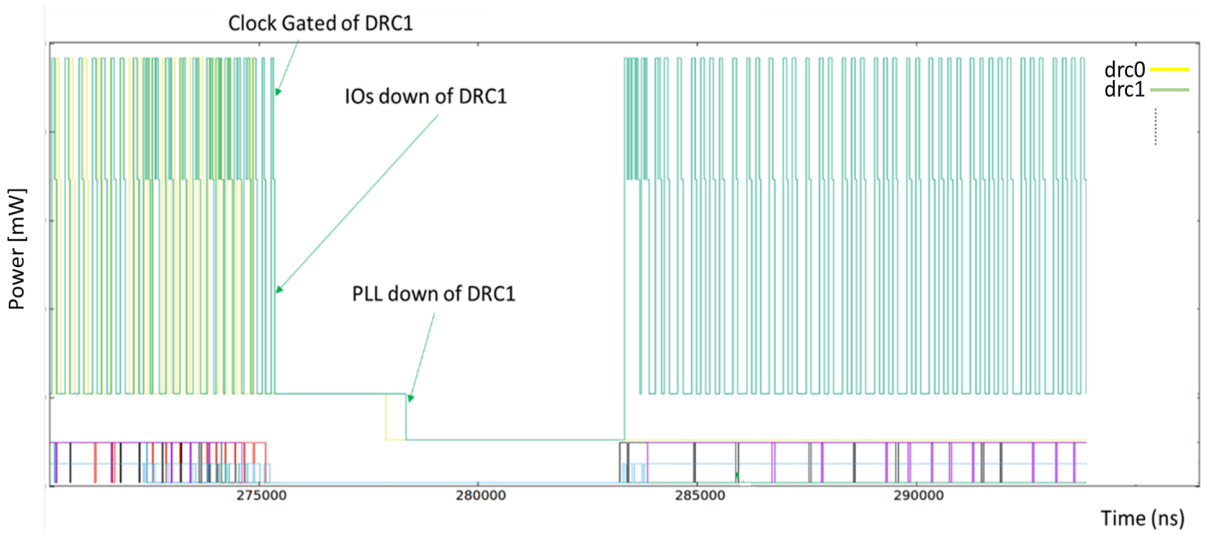


Figure 5.7: Calibration use case: Zoom on total power per Design Element [mW]

Figure 5.7 zooms in on the Figure 5.6 between the *270 µs* and *300 µs* instants. Before *275 µs*, the 2 DRCs are active because the traffic generators are sending transactions related to two contiguous 4K blocks in memory. After time *283 µs*, only one DRC is active because the active traffic generators generate transactions in the same 4K memory block. The transitions to the clock gated, IOs down and PLL down states of the DRC and SM sequential paths appear.

Figure 5.8 zooms in on the third phase of Figure 5.6 and more specifically on the timing of the transition between two 4K blocks pointed to by the traffic generators. At the beginning

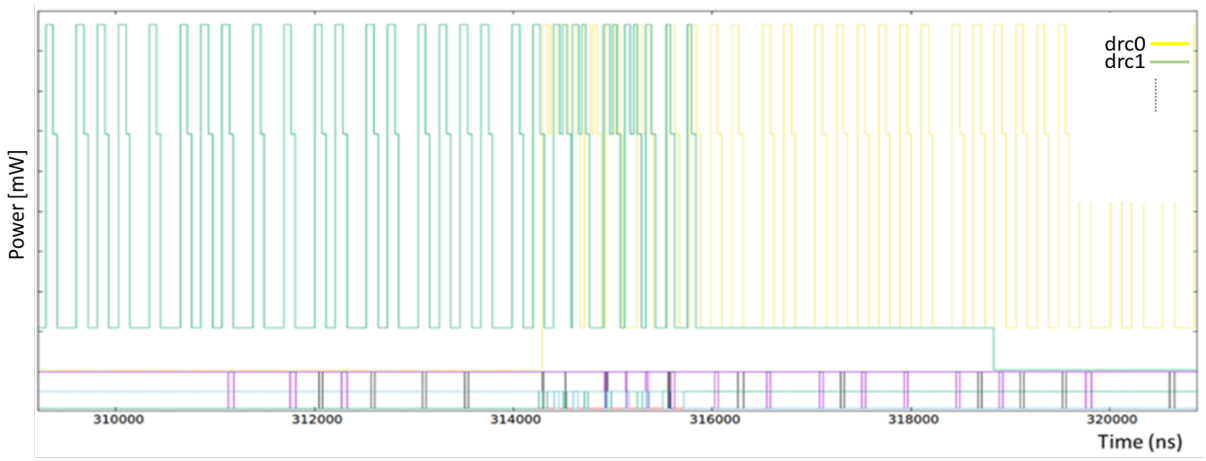


Figure 5.8: Calibration use case: Zoom on third phase in total power per DE [mW]

of the third phase, transactions are directed only to DRC1 as shown in Figure 5.7. In Figure 5.8, in the interval *314 µs - 316 µs*, there is an overlap between the activities of the two DRCs, this is due to the transaction processing pipeline in the SM and DRCs, so the last transactions related to DRC1 are processed while DRC0 receives the first transactions related to the next 4K block.

In the fourth phase, the traffic is terminated, clock gating is applied to all subsystems, and the voltage value changes from *ON_H* to *ON_L* for the SM units and from *ON* to *OFF* for the DRCs, which means that the DRCs are power gated (Figure 5.9).

Figure 5.9: Calibration use case: Voltage evolution per Power domains [V]

In Figure 5.10 we can observe the variation of the clock frequency. The two DRCs are active during the first phase, inactive during the second and interleaved during the third. We can also observe some of the internal units of the SM (e.g. QOS) and the impact of the clock. During the first phase, all subsystems are active, then during the third phase, half of them are clock gated (we illustrate only two subsystems with different behaviors).

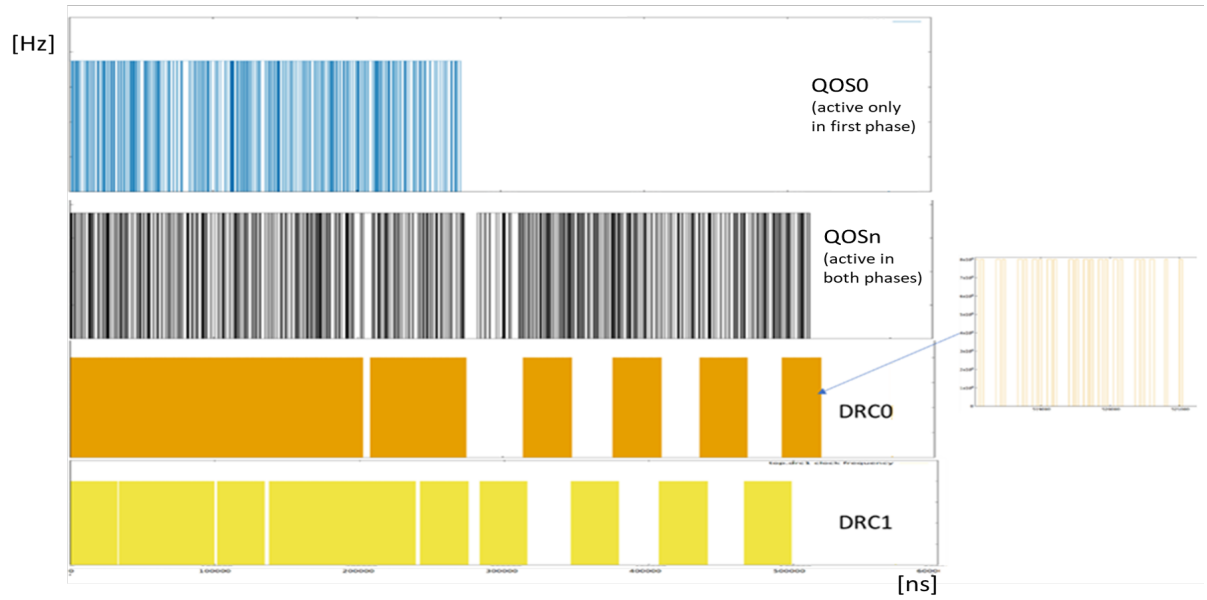


Figure 5.10: Calibration use case: Clock frequency variation during simulation [Hz]

The accumulation of the power consumed over time corresponds to the energy consumed [mJ] whose curve is given in Figure 5.11. The absence of activity (second phase) is visible. Also, the slope of the curve of the third phase is lower than that of the first phase because about half of the modules are in clock gated mode.

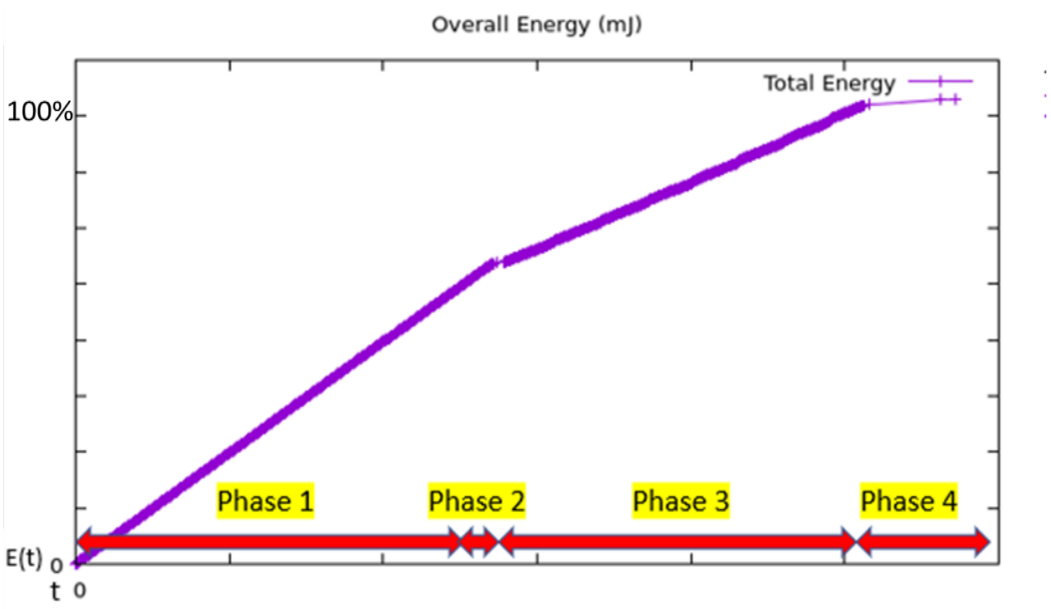


Figure 5.11: Calibration use case: Overall energy consumption profile in test case [mJ]

All the above results are obtained with fixed LPDDR4 read response times of *60 ns* and write response times of *40 ns* (values approximated using the platform specifications).

By changing these values to *30 ns* and *20 ns* respectively for the LPDDR4 connected to DRC1 (and keeping the previous response times for the memory on DRC0), we get different simulation results as shown in Figure 5.12.

The decrease in response times has a direct impact on the transaction execution time.

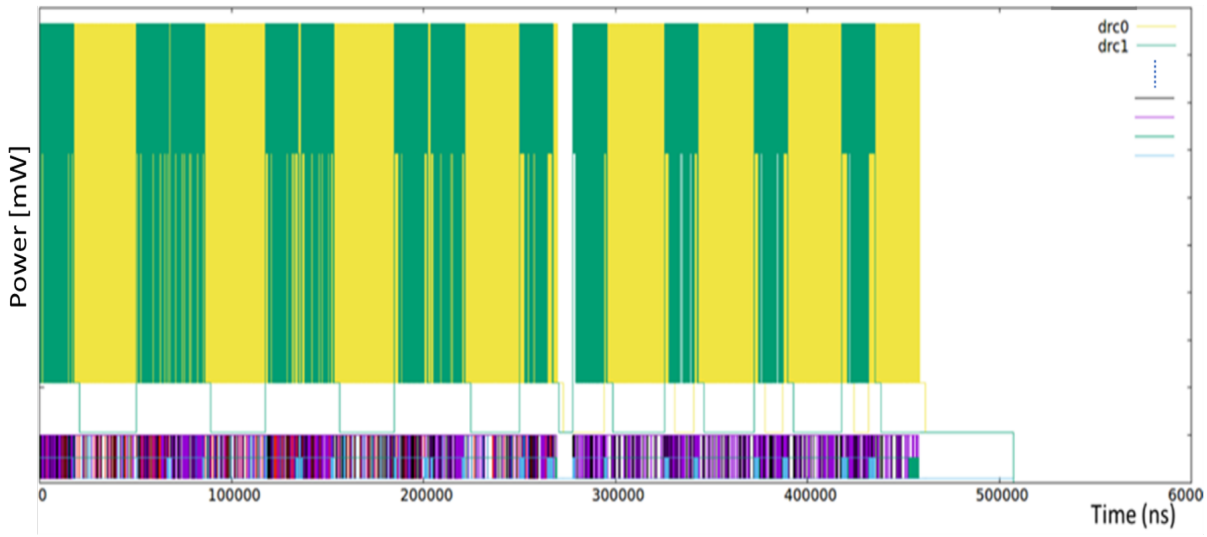


Figure 5.12: Calibration use case: Total power per DE (reduced memory response delay) [mW]

It appears that decreasing the memory access times on DRC1 allows us to almost double the throughput compared to the previous simulation: during the first phase, on the initial *45400 ns* (zoomed in Figure 5.13) we have the processing of the first 4K blocks (blocks sent to DRC0 and DRC1) sent during the activation of all traffic generators. Half of the traffic generators finish their first phase in *17660 ns* (green curve of DRC1 in the Figure 5.13) which leads to initiate from this date by these traffic generators additional transactions to DRC0 (yellow curve) with the rest of the traffic generators.

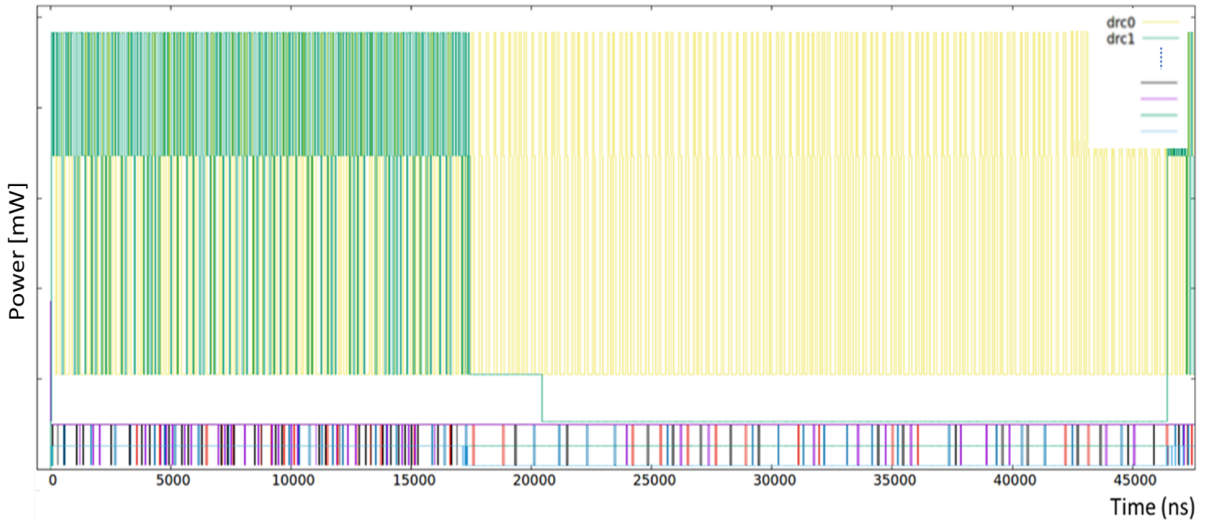


Figure 5.13: Calibration use case: Zoom on first phase in total power per DE (reduced memory response delay) [mW]

By setting the write and read times on the two LPDDR4s to *20ns* for writing and *30ns* for reading, we obtain the behavior described in Figure 5.14. Globally the processing time of all the transactions is almost divided by 2 compared to the Figure 5.5.

Figure 5.15 illustrates that, compared to Figure 5.7, the number of DRC runs in IOs down mode is logically decreased when memory response times are themselves reduced. However,

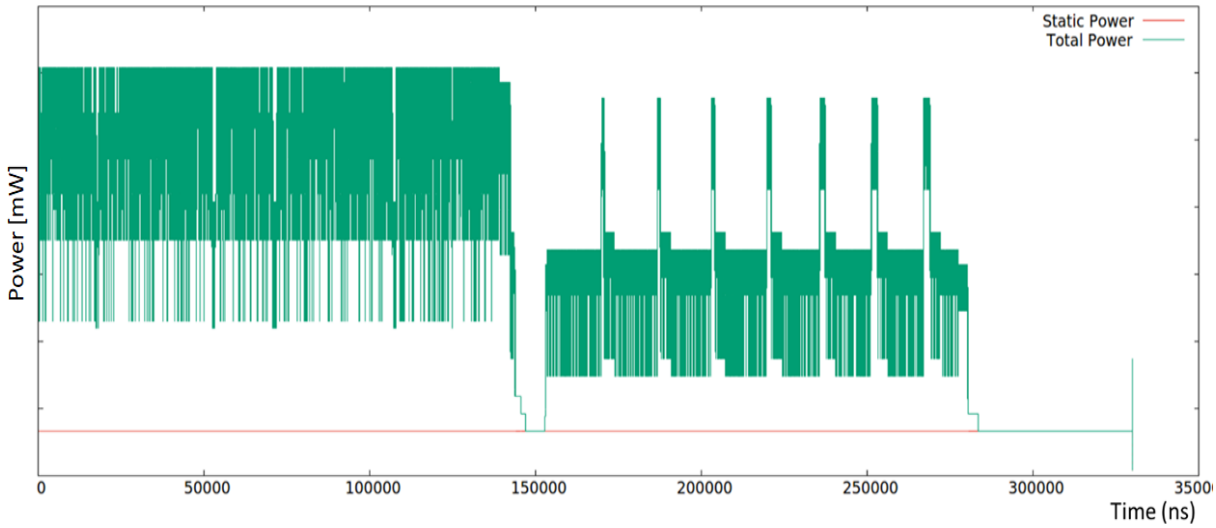


Figure 5.14: Calibration use case: Overall power consumption (reduced memory response delay) [mW]

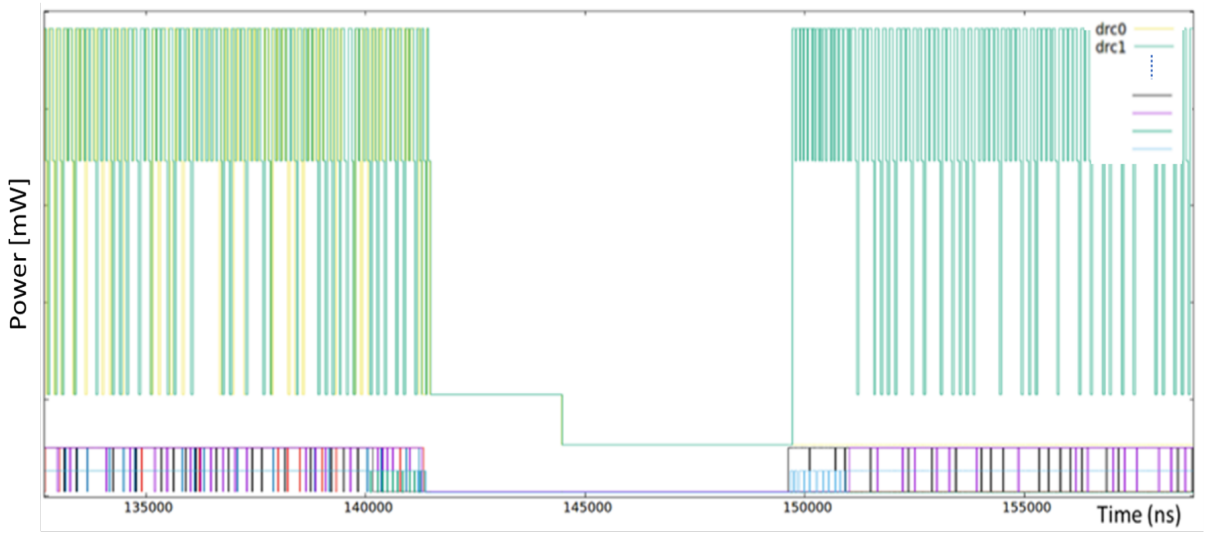


Figure 5.15: Calibration use case: Zoom on total power per DE (reduced memory response delay) [mW]

since the overall processing time of all transactions is lower, the overall energy consumption is reduced.

In all the previous simulations, the value of the *accept_delay* parameter of both DRCs is *20ns*. This is the delay set by a DRC to signify that it has accepted a transaction transmitted by an arbiter. By reducing this delay from *20 ns* to *5ns*, we obtain the curve in Figure 5.16, which gives the total power consumption. Comparing with Figure 5.14, we can verify that the number of DRCs in IO down mode is higher since each transaction is processed faster, leaving the DRC in idle state for a longer time between 2 transactions.

Continuing to reduce the LPDDR4 response times (to *8ns* for reading and *5ns* for writing), we obtain the result in Figure 5.17 which represents the total power consumption. It is clear that the transitions of the DRCs to the IOs down state are very rare in the first phase compared to the previous cases (Figure 5.14 and Figure 5.16). The delay between two transactions

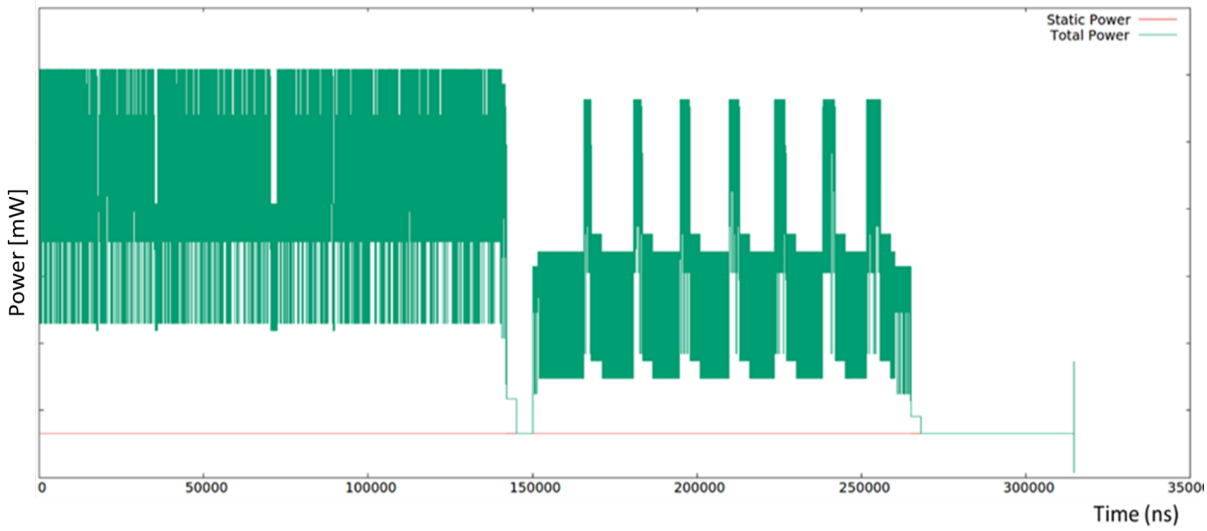


Figure 5.16: Calibration use case: Overall power consumption (reduced memory access delay) [mW]

transmitted to the DRCs becomes less than 32 cycles most of the time after the DRC clock gated (which itself occurs after 32 cycles of inactivity).

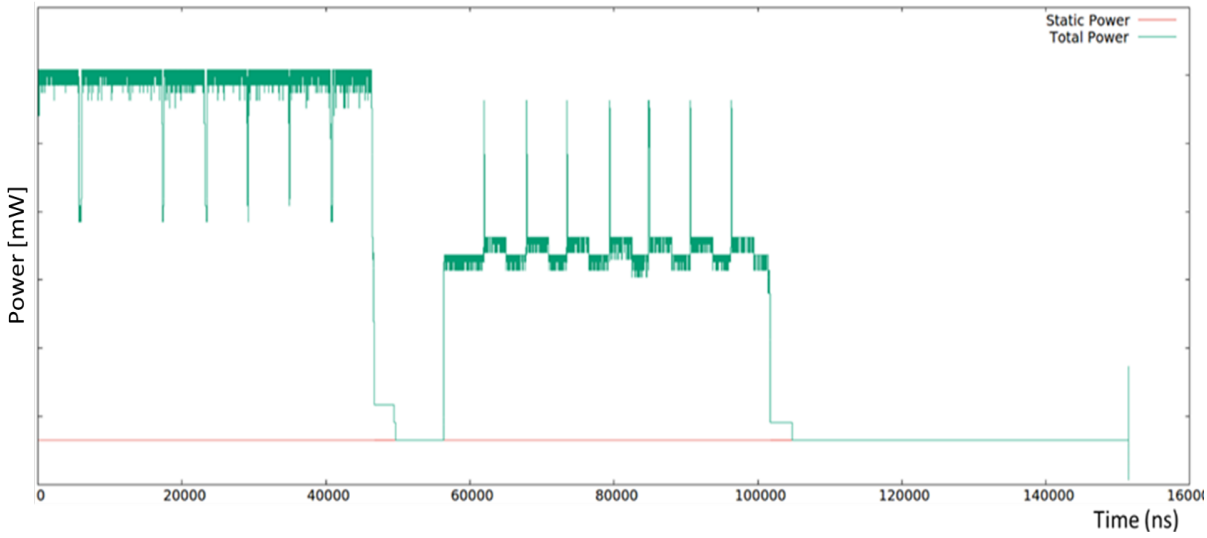


Figure 5.17: Calibration use case: Overall power consumption (reduced memory access/resp delays) [mW]

We can verify on these results the impact of LPDDR4 behavior on the time performance and power consumption. As the response times of LPDDR4 are not at all constant in reality, it seems important to evolve the functional model of SystemC-TLM to include a behavioral model of LPDDR4 in order to obtain a more realistic global behavior of the system.

For this reason, we opted for the DRAMSys4.0 solution, presented in section 4.3.3, to incorporate an accurate DRAM model with variable response and acceptance values.

The same analysis we did in this use case for DRC/DRAM memory models and total high-level power can be done for the internal SM units. The same metrics we estimated for DRAM/DRC were extracted for each of the SM blocks. This analysis is not presented here for confidentiality reasons.

5.2.2 MemCopy use case

The memory copy use case is a relatively straightforward use case with a monotonic energy profile. However, since we use QOS-based arbitration and scheduling (using *AxID*) at multiple levels in the SM, it is very useful to test the implementation of the internal scheduling mechanisms and their energy profile with such a traffic-dense use case. In this use case, we have only one active application-specific traffic generator that performs 256KB, 1MB, and 32MB read accesses to an LPDDR4 performed by a cluster of CPUs. Each data access is 16 bytes and we consider 4K interleaved sequential accesses between the two memories (DRC0 and DRC1). Since we only have one active traffic generator, this means that we only activate one sequential path in the SM (+ Little and Big Arbiters), so all other groups and internal units are clock gated throughout the simulation. The entire SM is installed in a single power domain, so we cannot cut off the power supply net and its static power consumption is the same as if we had full activity in the SM.

256KB memory copy

The first simulation runs 256KB of sequential read accesses to the LPDDR4 memories. Figure 5.18 shows the total power consumption of the simulated platform.

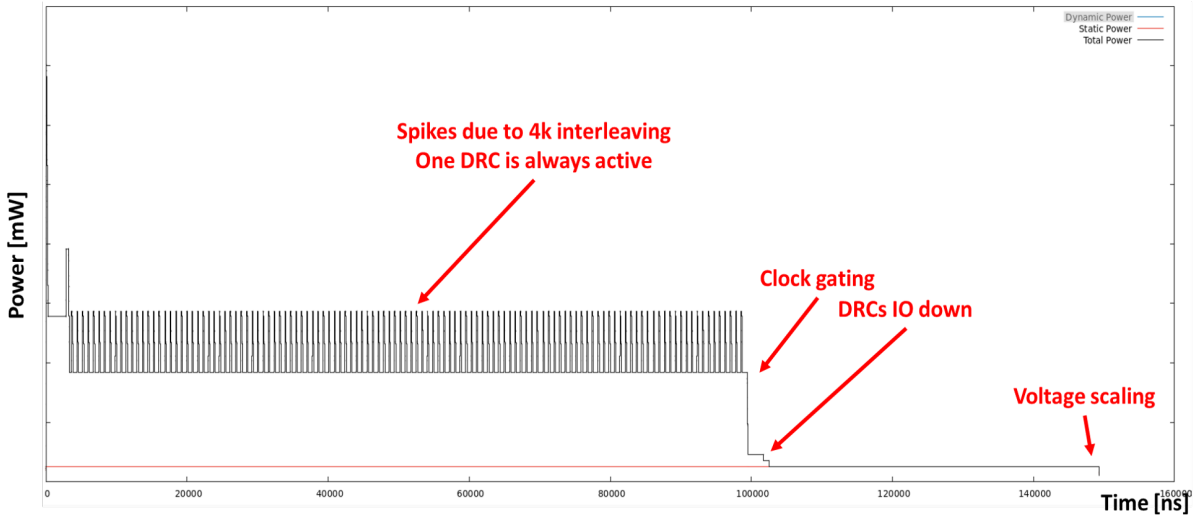


Figure 5.18: 256KB Memory copy use case: Overall power consumption [mW]

We can observe the constant activity due to the continuous memory accesses. Since we interleave the two memories, there are short periods of time when both memories are active at the same time, which is visualized in Figure 5.18 as short consumption peaks. Figure 5.19 shows a zoomed-in view of the interleaving between the two memories.

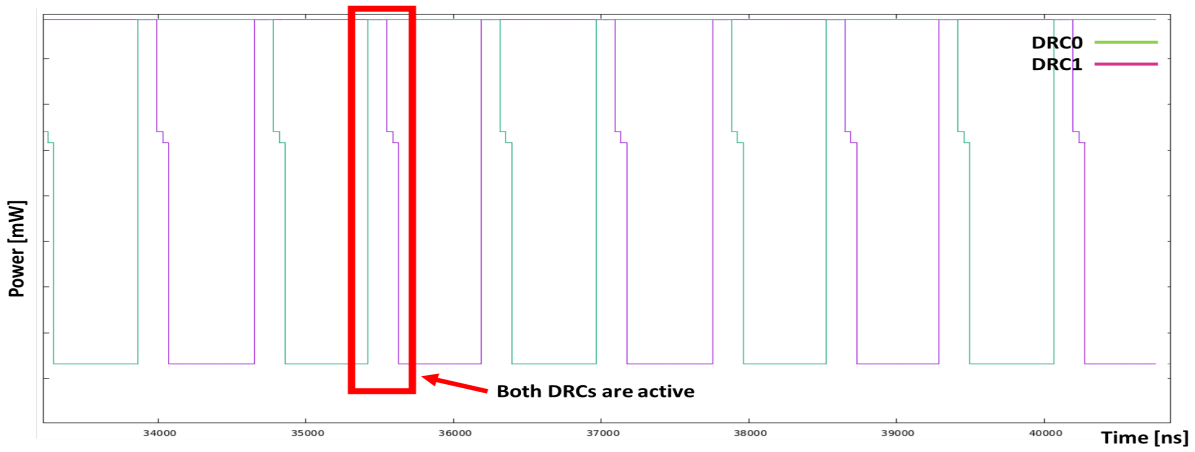


Figure 5.19: 256KB Memory copy use case: Zoom on DRCs power consumption [mW]

Figure 5.20 shows the power consumption of the only active sequential path in the SM. In this use case, we are constantly loading it with transactions, so it is active throughout the entire simulation.

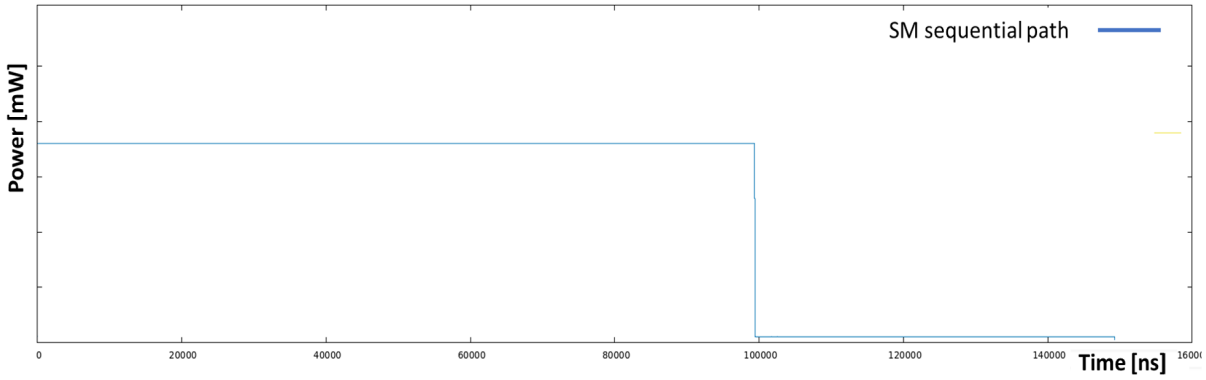


Figure 5.20: 256KB Memory copy use case: SM sequential path power consumption [mW]

Figure 5.21 shows the power consumption of the two internal units of the Big Arbiter module responsible for DRC arbitration.

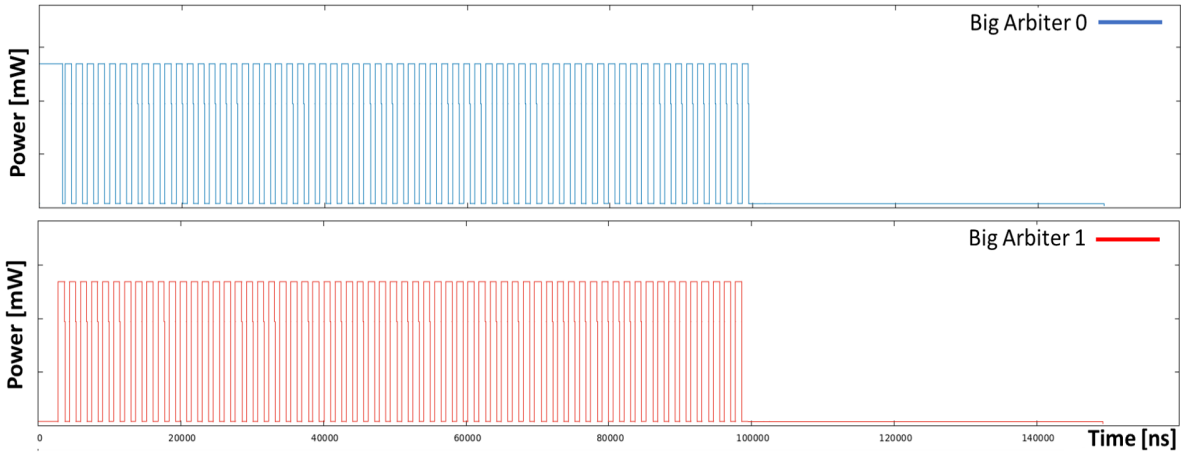


Figure 5.21: 256KB Memory copy use case: Big Arbiter (0 and 1) power consumption [mW]

As explained in the technology section, the Big Arbiter contains one arbiter for transactions destined for DRC0, one for transactions destined for DRC1, and one for communication between the subsystems. In this figure, we can see that since we interleave between memories, we also interleave between arbiters. The same behavior can also be observed using the clock frequency variations for each module.

The total energy consumption for this use case is shown in Figure 5.22. As expected, it increases linearly, as we have constant traffic. If we zoom in, we can see that there are small steps where the energy consumption increases a bit faster, which is also due to the interleaving between memories.

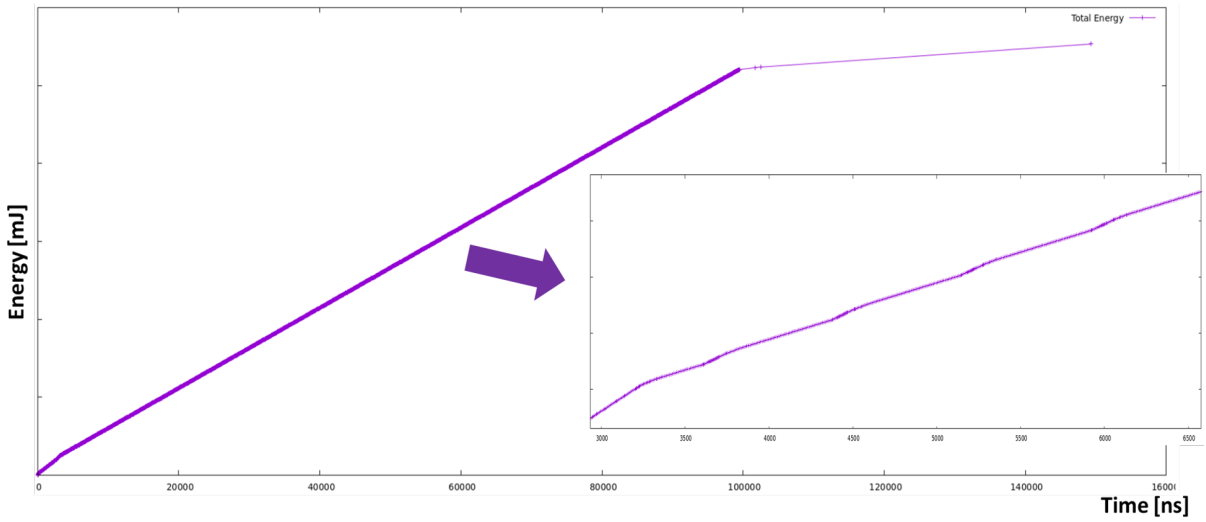


Figure 5.22: 256KB Memory copy use case: Total energy [mJ]

We can also observe the static consumption of the SM and DRCs (by power domain) in Figure 5.23.

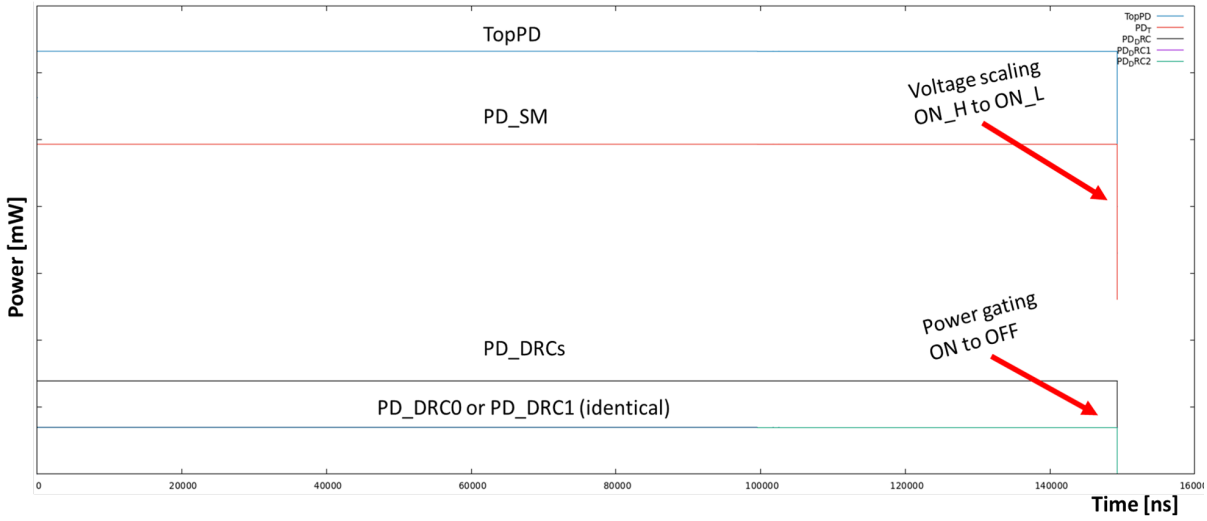


Figure 5.23: 256KB Memory copy use case: Static power consumption [mW]

In order to measure the power consumption gains due to the applied power management strategy, we created another power intent where the entire SM is under one clock domain

and the two memories are under another common clock domain. In this case, the internal hardware mechanism for automatic clock gating is not enabled. The use case executions of the 256KB memory copy using the IP without power management and with it are compared in Figure 5.24 illustrating the total simulated power consumption.

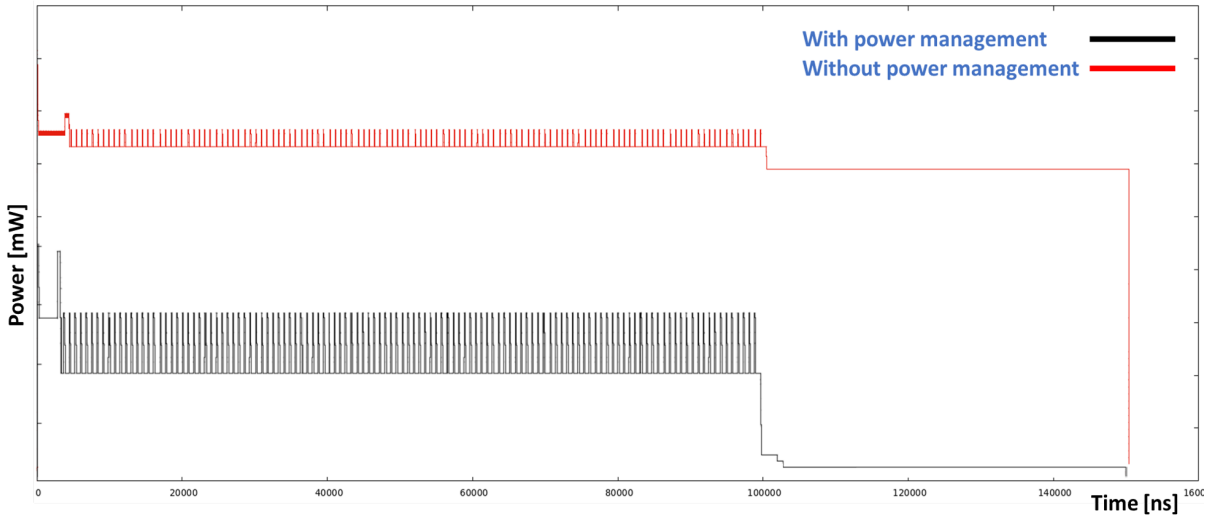


Figure 5.24: 256KB Memory copy use case: With and Without power management [mW]

It can be observed that the power consumption without power management strategy is significantly higher (almost 3x higher) than the power consumption implementing the power management strategy (and more precisely the auto clock gating). In the case of the simulation without power optimization, all unused units of the SM are always active (their clocks are active), so the power consumption is much higher and stable. The only variations are due to the consumption of the transaction transfers passing through the different units.

In Figure 5.25 we can see the total power consumption of a single unused sequential path of the SM subsystem (red curve) and the total static consumption of the SM (for all SM sequential paths and the other units).

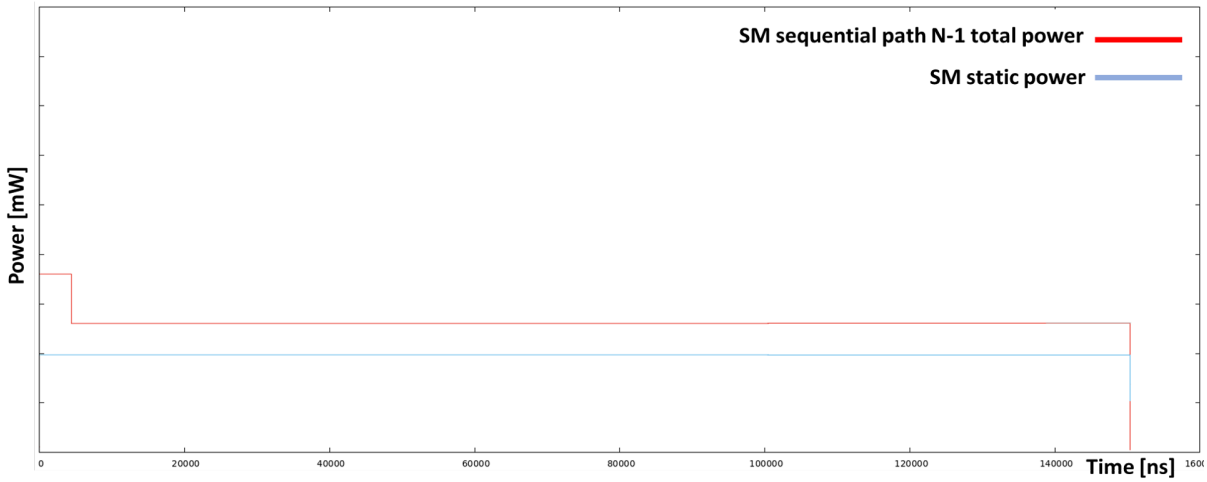


Figure 5.25: 256KB Memory copy use case (without power management): unused SM seq. path (Total Power per DE) + entire SM static power [mW]

The unused units remain active, because the entire SM is in a clock domain and there is activity inside, so the clock and the dynamic consumption of all units cannot be turned off. All unused SM sequential paths have the same behavior and since there are multiple of them in this architecture, it can be inferred that they contribute significantly to the higher total power consumption.

The power consumption of the single SM sequential path used in this use case is shown in Figure 5.26, along with the power consumption of the two units internal to the Big Arbiter responsible for memory accesses.

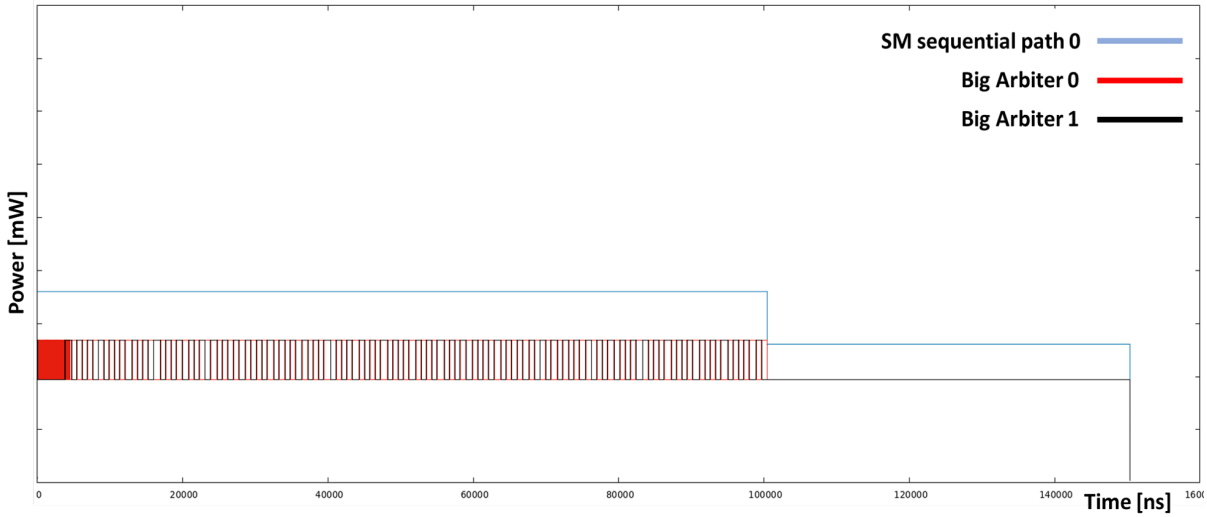


Figure 5.26: 256KB Memory copy use case (without power management): active SM seq. path + Big Arbiter total power [mW]

Due to the dense traffic, the power consumption of the used SM sequential path is constant as long as there are transactions. We interleave between the two memories, thus the two internal units of the Big Arbiter. Therefore, we have variations in their power consumption due to the consumption of transactions when they are present or not. We have the maximum toggle rate/activity factor when we have transactions and a lower toggle rate/activity factor when there are no transactions.

This comparison allowed us to highlight the benefits of power management strategies, and specifically the ability to observe their influence at the system level using this framework and the PwClkARCH library.

1MB and 32MB memory copy

The same memory copy use case was simulated for 1MB (Figure 5.27) and for 32MB of read memory access. The observations on power consumption and management are quite similar to the previous ones. There are no significant behavioral changes (functional and power), except for the longer simulation window and simulation time.

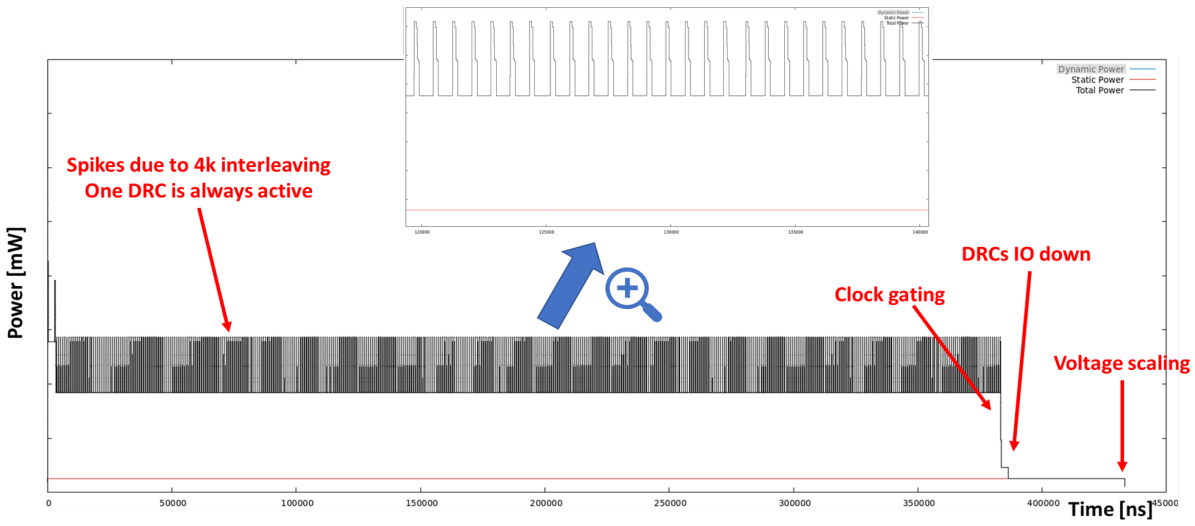


Figure 5.27: 1MB Memory copy use case: Total power consumption [mW]

5.2.3 Display refresh VGA-32bpp use cases

In this use case, we consider a VGA display with a resolution of 640x480 and we execute a frame of transactions. The data paths are 256 bits and each pixel is 32 bits, so we have 8 pixels per transaction. We consider the pixel frequency to be 23.5 MHz and the refresh rate to be 60 Hz (frame rate of 60 fps). One line of active display data represents 640 pixels, which means 80 transactions. We have a total of 480 lines, so a total of 38400 transactions for one image. We also take into account horizontal and vertical blanking periods to increase the accuracy of the generated traffic. Each data access is 32 bytes and we consider sequential memory accesses *(previous_address+32bytes)*.

Using prefetch mechanism

Figure 5.28 shows the total power consumption of the SM+DRCs testbench.

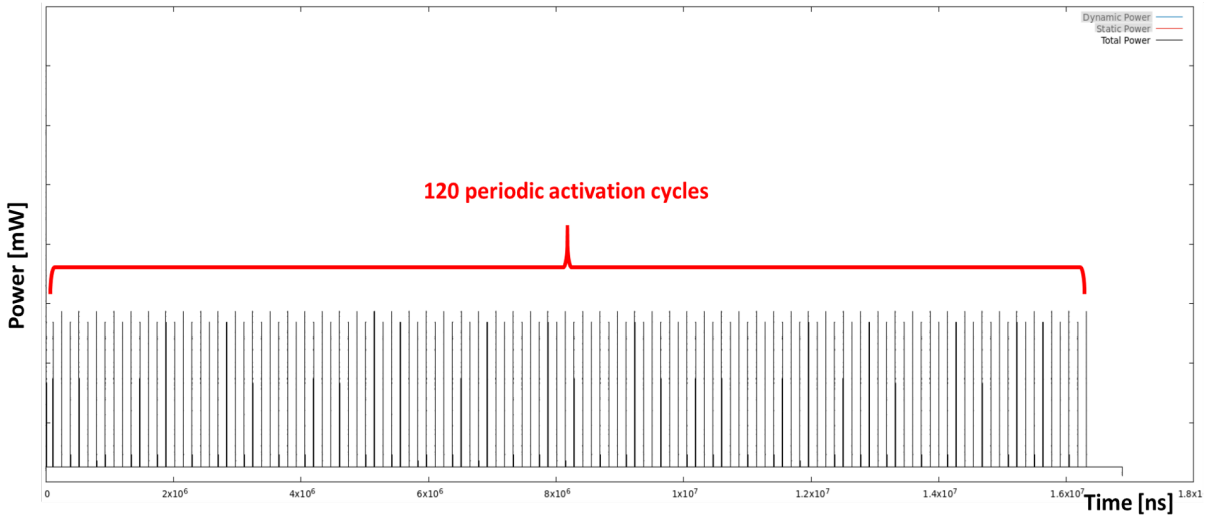


Figure 5.28: Display refresh VGA use case: Total power consumption [mW]

In this use case, we use the prefetching mechanism of the display controller, which allows us to always store about 4 rows in advance before displaying them. The power profile is periodic, because we retrieve several lines, then let the display consume them, so the SM and DRCs are in idle (and clocked) mode during this time. Once the available lines are consumed, it starts consuming the prefetched lines, while we prefetch another bunch of lines for the next cycle. In Figure 5.28 we see a constant static activity (reduced at the end) and a periodic dynamic activity. There are 120 peaks for the dynamic consumption. Since we have 80 transactions per line and we prefetch 4 lines in advance, we have 320 transactions in each activation cycle. One frame is equal to 38400 transactions, so we have about $\frac{38400}{320} = 120$ cycles, which is the currently observed behavior.

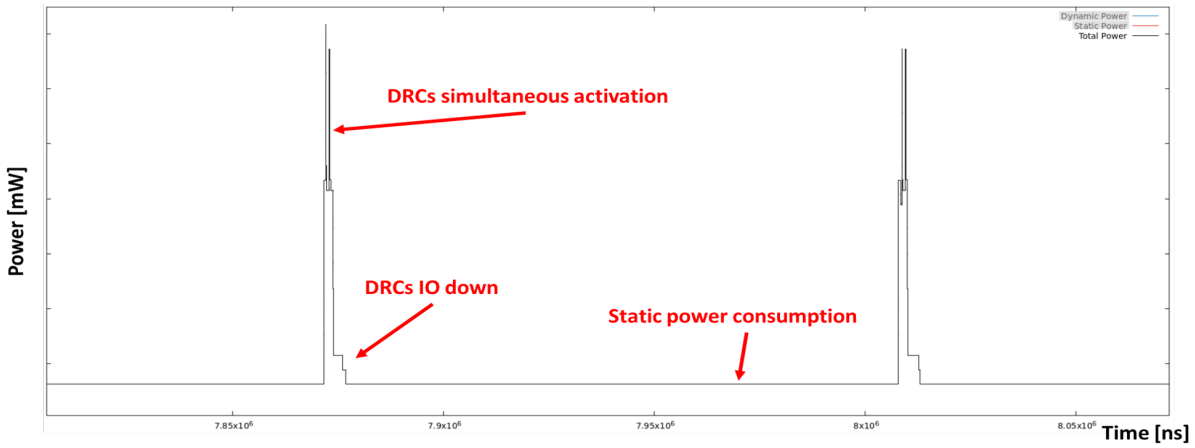


Figure 5.29: Display refresh VGA use case: Zoom on total power consumption [mW]

We can also see that the power consumption during almost half of the cycles is higher than the rest. As with the memory copy use case, this is due to the 4k interleaving mechanism and the short parallel activation of both DRCs. This is visualized more clearly in Figure 5.29 by zooming in on the total power consumption peaks and in Figure 5.30 by zooming in on the DRCs' power consumption.

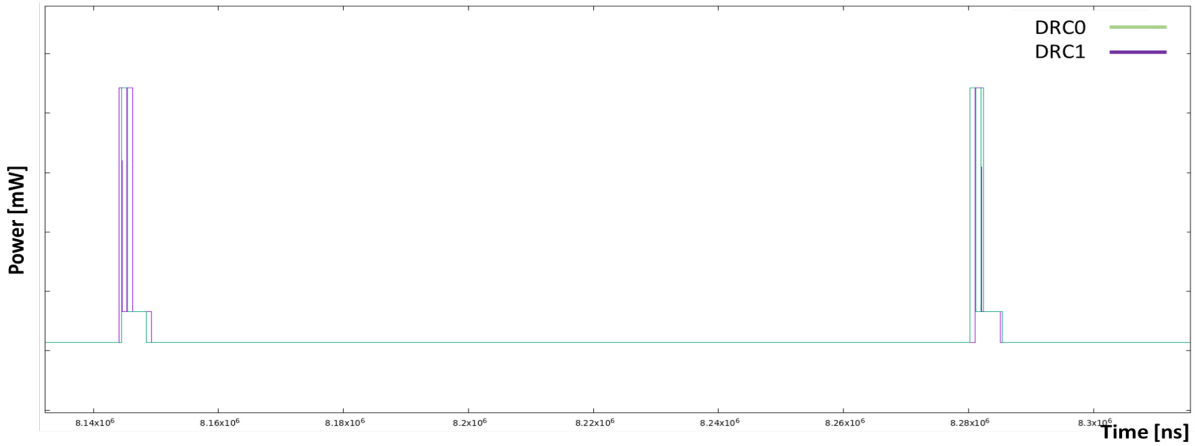


Figure 5.30: Display refresh VGA use case: Zoom on DRCs power consumption [mW]

The consumption of the SM sequential path is shown in Figure 5.31. Transaction processing is scheduled and stored or transmitted, and then if there are no transactions in progress,

the hysteresis mechanism starts counting 32 cycles before triggering the clock gating.

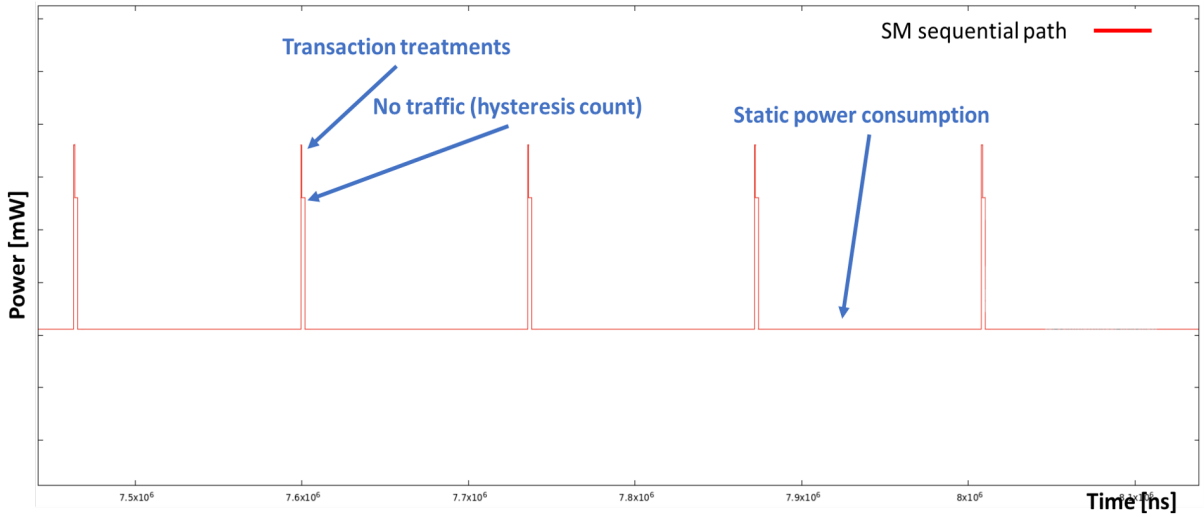


Figure 5.31: Display refresh VGA use case: Zoom on SM sequential path consumption [mW]

In Figure 5.32 we can see the evolution of the clocks of the Big Arbiter module (and its internal units), which have a very similar behavior to the clocks of the DRCs. The clocks are enabled only when there is traffic passing through the arbiter module and disabled when the module is not in use.

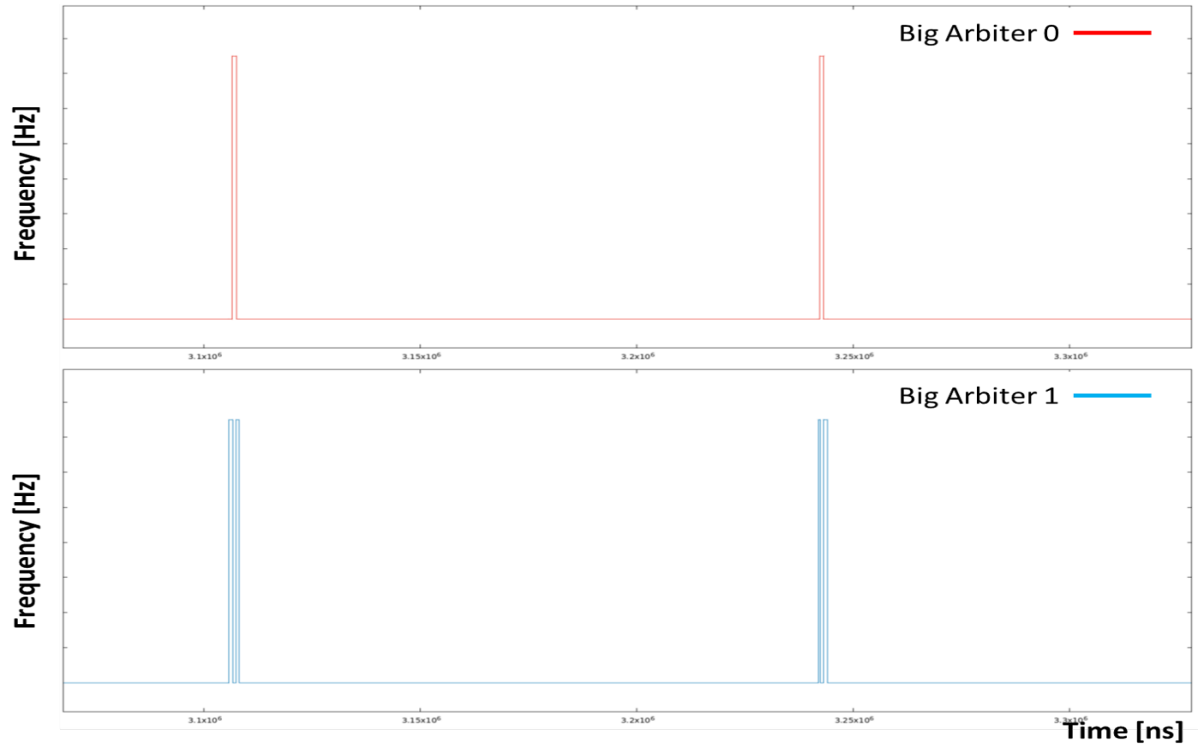


Figure 5.32: Display refresh VGA use case: Zoom on Big Arbiter clock evolution [Hz]

Figure 5.33 shows again the reduction in power consumption due to the use of more refined power management compared to the non-optimized power management strategy.

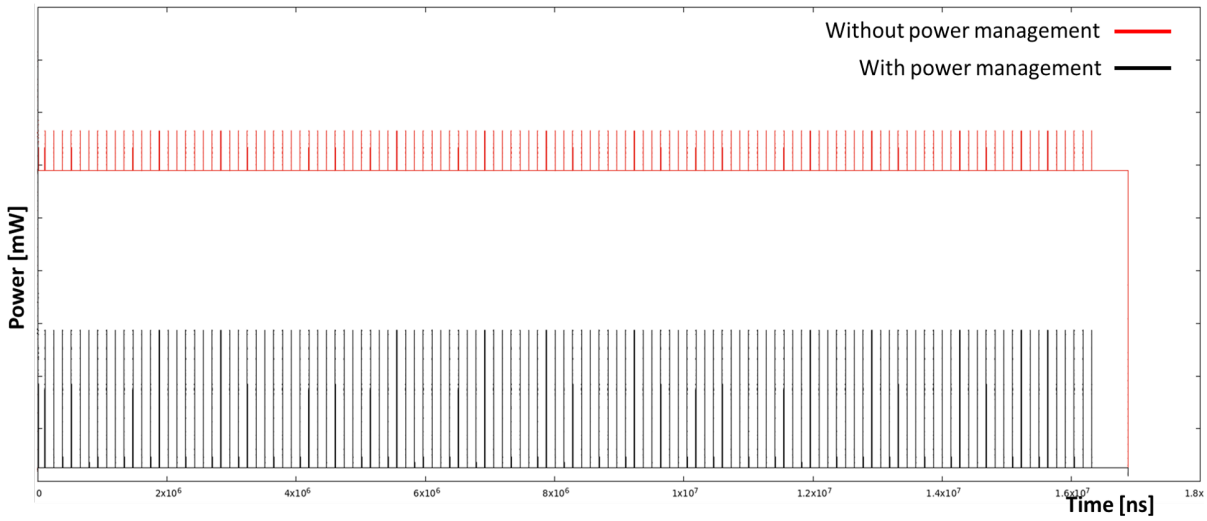


Figure 5.33: Display refresh VGA use case: Total power with and without power management [mW]

Not using prefetch mechanism

Without the display controller's prefetching mechanism, we must access memory for each fetched line before the display consumes it. Figure 5.34 shows the power behavior of this simulation. It can be seen that the number of memory accesses is approximately equal to the number of display lines. The switching activity is much higher, because for each transaction we reactivate the SM and DRCs.

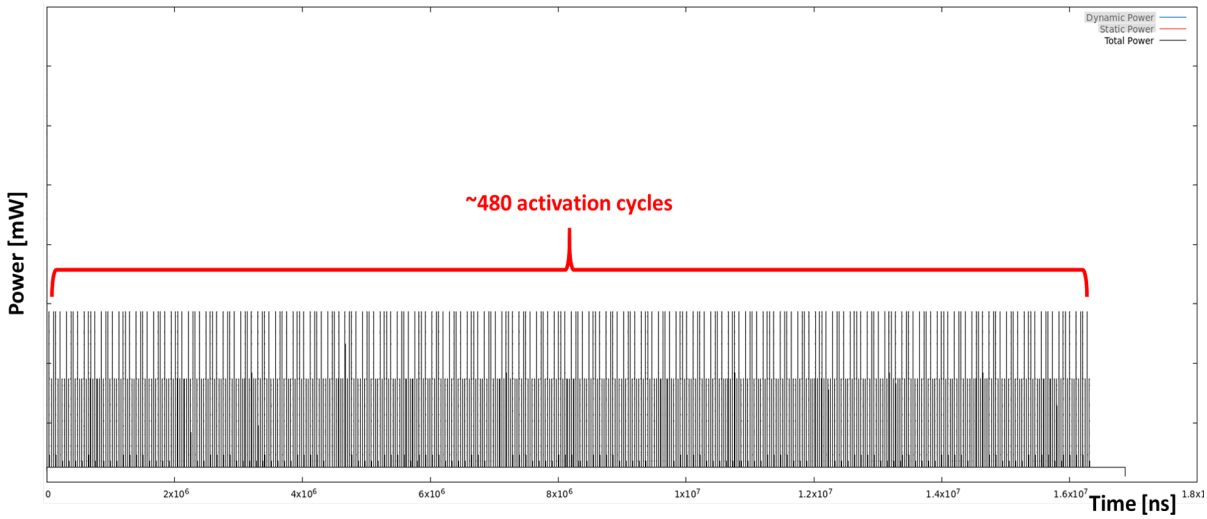


Figure 5.34: Display refresh VGA without prefetch use case: Total power consumption [mW]

Moreover, in this use case, the display controller is the only active traffic generator, so there is not much contention for memory accesses. If we have multiple active traffic generators, we may not meet the display refresh rate. There are many more memory activations, so its occupancy is higher and the PLLs are never put in a down state (Figure 5.35).

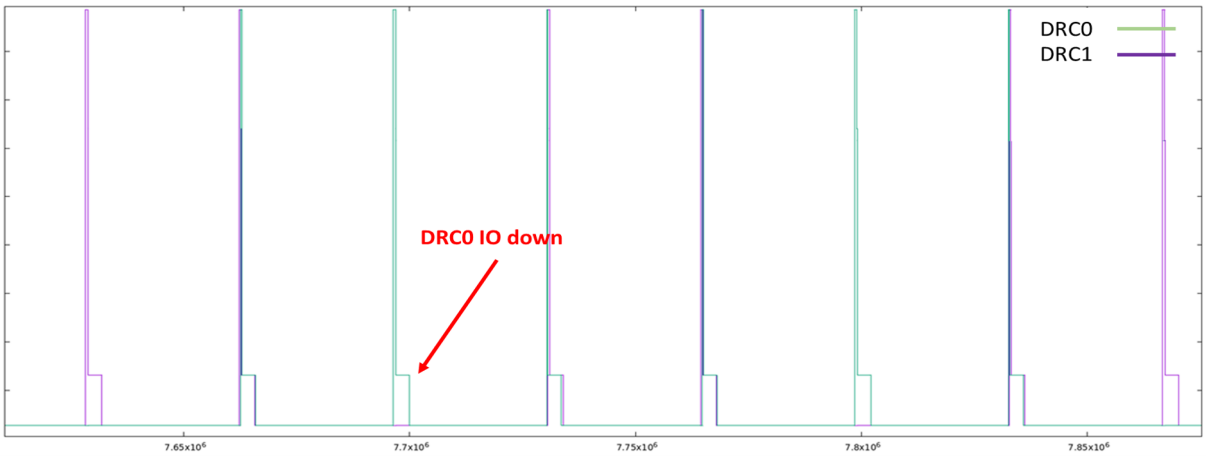


Figure 5.35: Display refresh VGA without prefetch use case: Zoom on DRCs power consumption [mW]

In Figure 5.36, we can compare the energy profiles with and without the prefetching mechanism and observe the higher switching activity of the display refresh simulation without the prefetching mechanism. The prefetching mechanism provides a better power and performance profile, ensuring that the bus and memories are less loaded and the possibility of bottlenecks is significantly reduced.

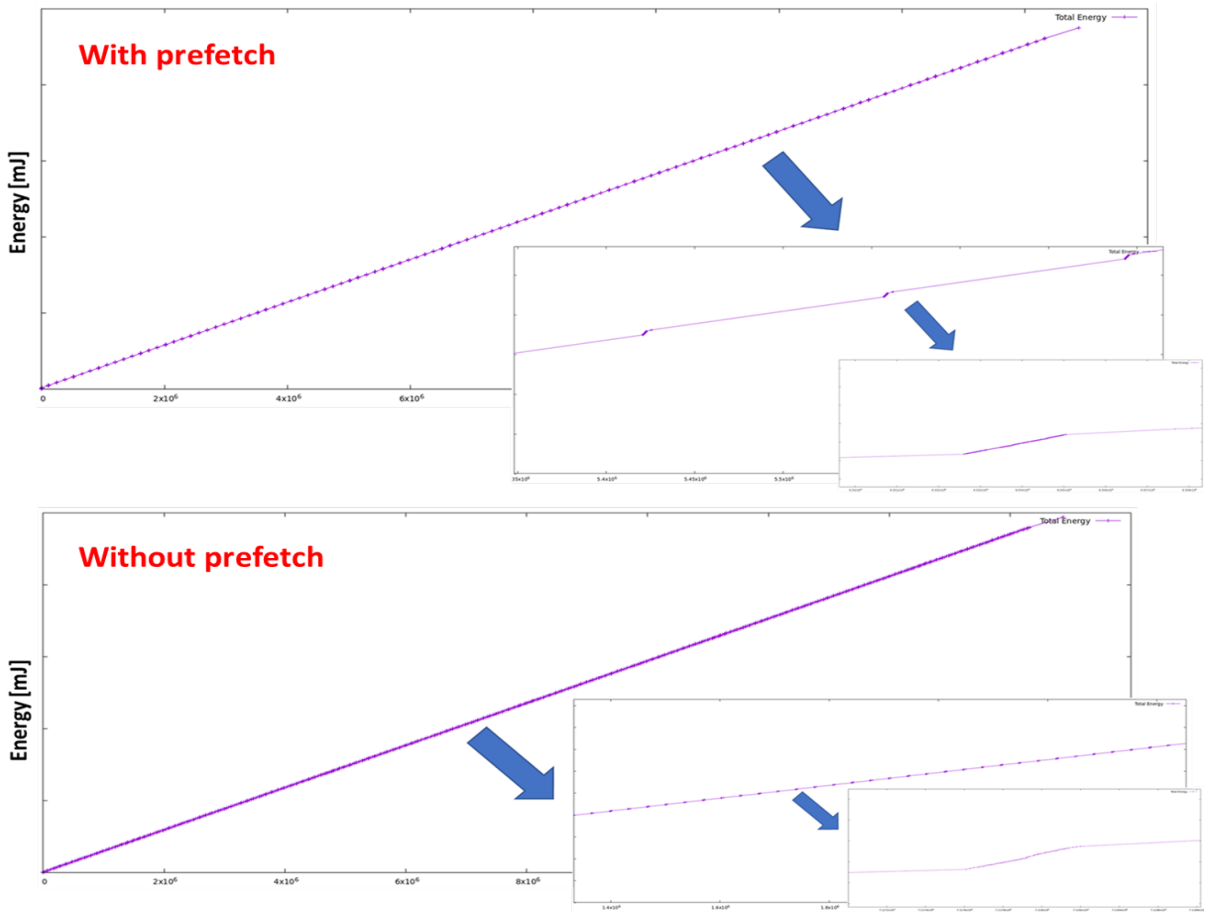


Figure 5.36: Display refresh VGA with/without prefetch: Overall energy [mJ]

5.2.4 Display refresh HD1080-32bpp use cases

In this use case, we consider an HD display with a resolution of 1920x1080 and we execute a frame of transactions. The data paths are always 256 bits and each pixel is 32 bits. We use a pixel frequency of 138.5MHz and a refresh rate of 60Hz. One line of active display data represents 1920 pixels, which means 240 transactions. We have a total of 1080 rows, so a total of 259200 transactions for one image. Again, each data access is 32 bytes and we consider sequential memory accesses.

In Figure 5.37, we can again observe the simulations with and without the prefetch mechanism, but for a higher resolution display.

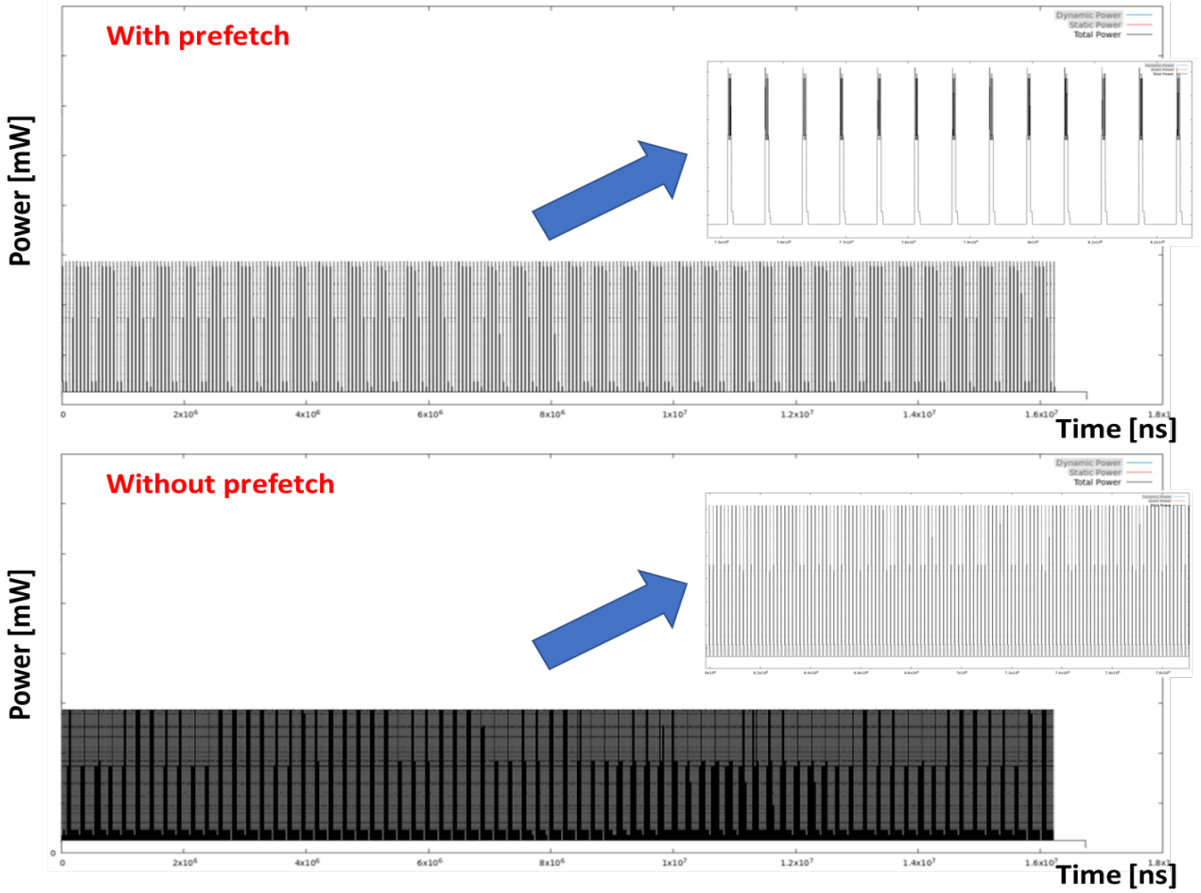


Figure 5.37: Display refresh HD with/without prefetch: Total power [mW]

5.2.5 Multiple displays and CPUs use case

Another more complex and realistic use case is presented in Figure 5.38. In this simulation, we consider that two VGA displays simultaneously execute a display refresh use case, a cluster of two traffic generators (ex. cores) executes two sequential 256KB read memory accesses with a *20us* idle time between each iteration, and a cluster of four traffic generators (ex. cores) executes four sequential 256KB read memory accesses with a *50us* idle time between each iteration.

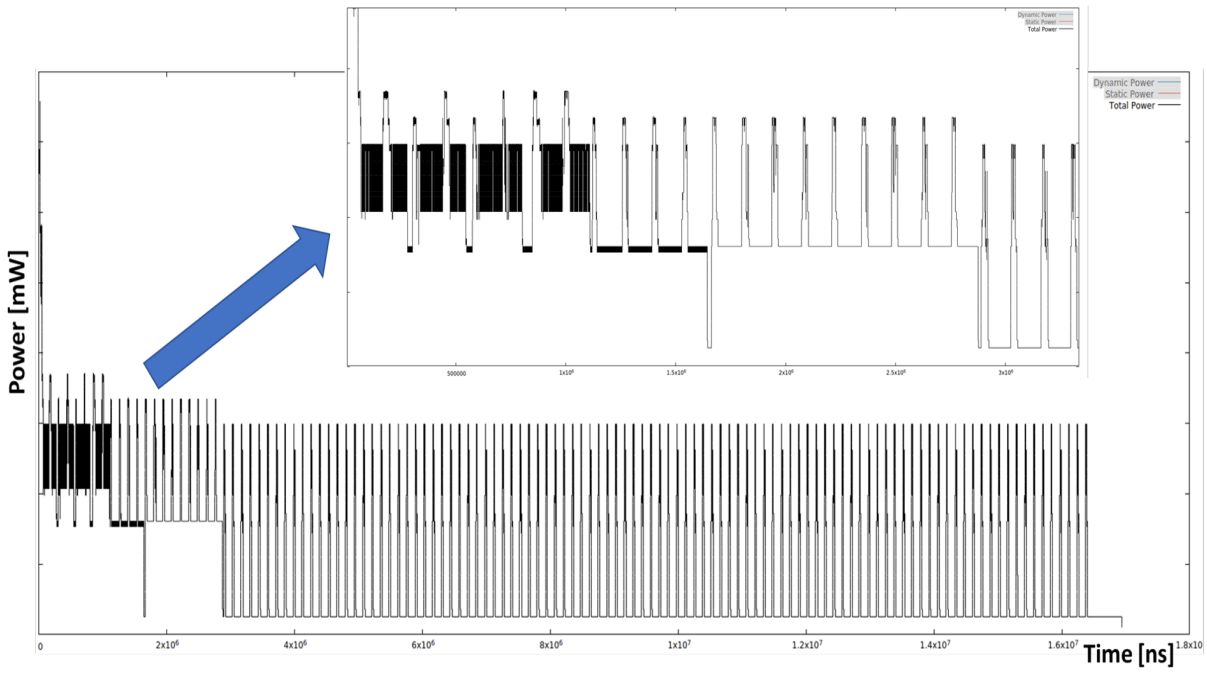


Figure 5.38: Display refresh 2 VGAs + 2 processor cores + 4 processor cores: Total power [mW]

All traffic generators are started at the same time at the beginning of the simulation, which also activate the SM sequential paths (Figure 5.39 shows a zoom on the energy consumption when all use cases are active at the same time).

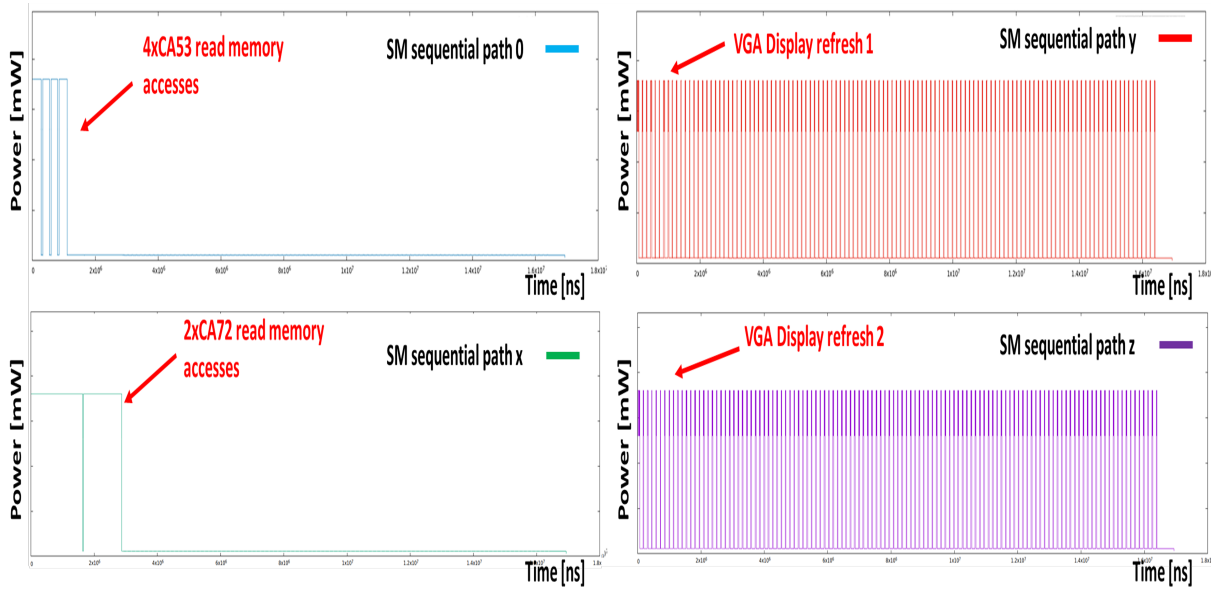


Figure 5.39: Display refresh 2 VGAs + 2 processor cores + 4 processor cores: SM seq. paths - Total power [mW]

The two DRAM memories are the ones that contribute the most to the power consumption. Thus, their behavior and impact can be easily observed and recognized on the total power curve. In Figure 5.40, we highlight this by placing the curves next to each other. DRAM 0 is always active during this period (all traffic generators start with accesses to this memory) and the variation in DRAM 1 activations is visible on the total power when we transpose them.

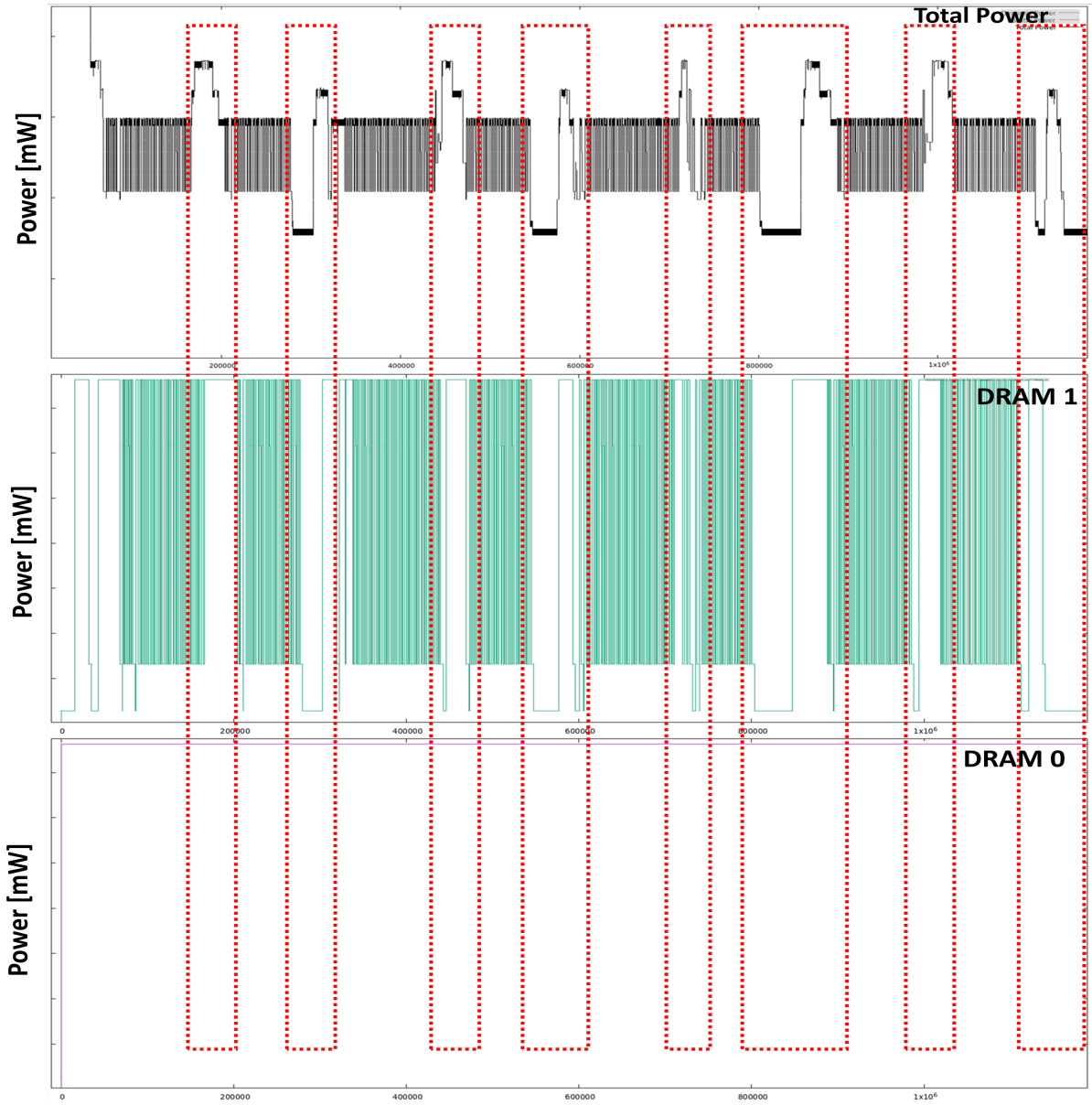


Figure 5.40: Display refresh 2 VGAs + 2 processor cores + 4 processor cores: DRAMs total power [mW]

5.3 Power-aware IP-reuse strategy applied on NXP i.MX8QXP SoC

In the next phase of this study, using the configurable and power-aware SM model, we were able to easily reconfigure and reuse it for another existing SoC platform of the same family, called i.MX8QXP. Again, the silicon is already available and the power intent already defined, which allowed us to successfully correlate our simulations with the silicon measurements. The real goal here was to test the interoperability and effort required to integrate the power-aware soft macro IP into the new platform. The i.MX8QXP has a similar architecture to the i.MX8QM, but with fewer subsystems, a smaller SM (fewer Group blocks), a lower clock rate, and a single external LPDDR4 memory. Thus, in order to model it, we reused the already available components (models of the SM sequential path units, arbiters, groups and testbench structures). As these are reconfigurable components, we reconfigured them (without changing their code). The same is almost true for the power intent. We reused the one defined for QM, but we defined a different number of instances of power components. Thanks to the granularity chosen for modeling the SM, we only had to disable some functional and power blocks when instantiating the module using its parameters. It took less than 2 days of work (one person effort) to complete the full integration (+verification) and simulate the HD-1080 display refresh and 1MB memory copy use cases considered for the silicon power consumption correlation.

5.3.1 MemCopy use case

Figure 5.41 illustrates the total power consumption that shows the main difference between the i.MX8QM and i.MX8QXP power profiles.

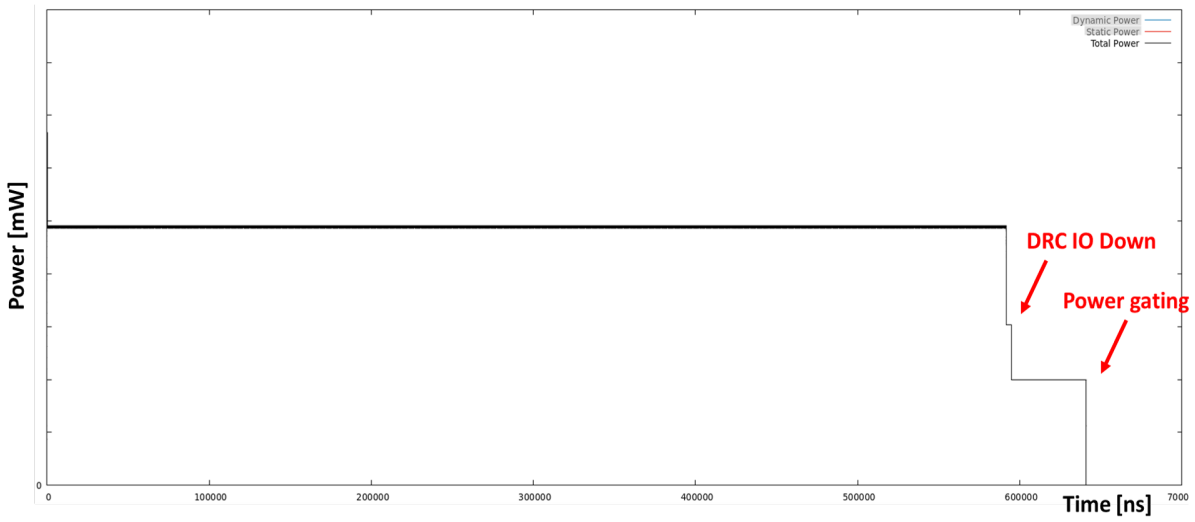


Figure 5.41: Memory copy: Total power [mW]

Since there is only one external LPDDR4 memory, there is no interleaving between memories, so the total power consumption is almost halved and kept constant during this high traffic use case simulation. Since all memory accesses are performed on this memory, there is not enough time to change states and its clock is constantly active (Figure 5.42).

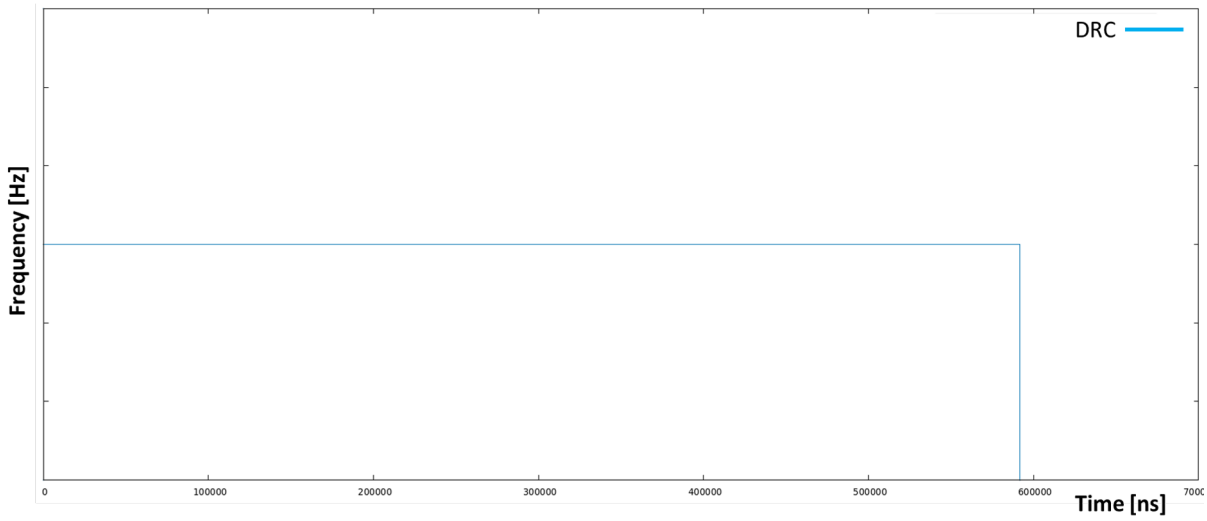


Figure 5.42: Memory copy: DRC clock evolution [Hz]

5.3.2 Display refresh QVGA-32bpp use case

Similarly to the i.MX8QM, the power behavior of the QVGA use case (such as for the VGA and HD) is periodic, but with a single memory. Figure 5.43 shows the total power of SM+DRC during this use case.

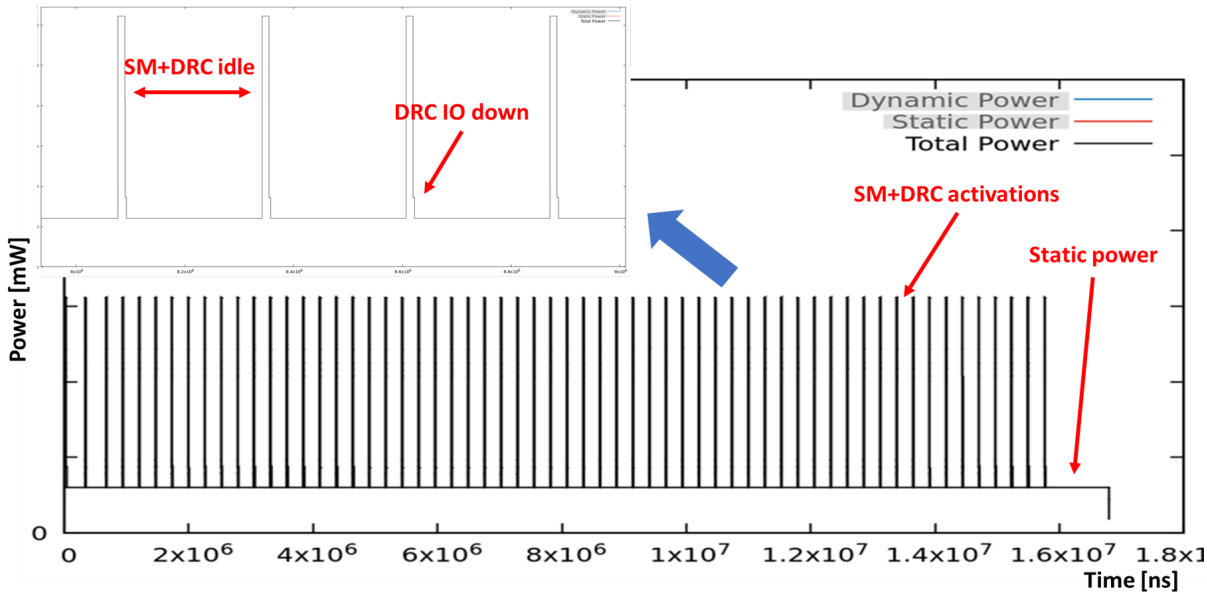


Figure 5.43: QVGA use case: Total power [mW]

In reality, the memory power profile will be more dynamic even during these types of simulations, but the variations are minimal and often can be neglected at this level of abstraction. Creating an accurate memory power model with PwClkARCH or including DRAMPower in this co-simulation between PwClkARCH and DRAMSys4.0 may be a good approach to catch more details of memory consumption. However, the main goal of this study was to evaluate the power consumption of the SM (which is confidential), so the most essential part was to have an accurate functional model of the LPDDR4 memory.

5.4 Correlation

To correlate the results of these simulations with silicon measurements, we extracted power measurements from the i.MX8QM and i.MX8QXP using *Baylibre ACME+Beaglebone* board for power measurements [118] and a *maxTC* temperature forcing system [119]. We placed *Analog-to-Digital Converters (ADCs) and 50 mOhms* shunt resistors in series with the load and the voltage of the SM+DRCs power networks (PMIC level) and connected them to the various probes on the *Baylibre ACME* chip. The main components are shown in Figure 5.44.

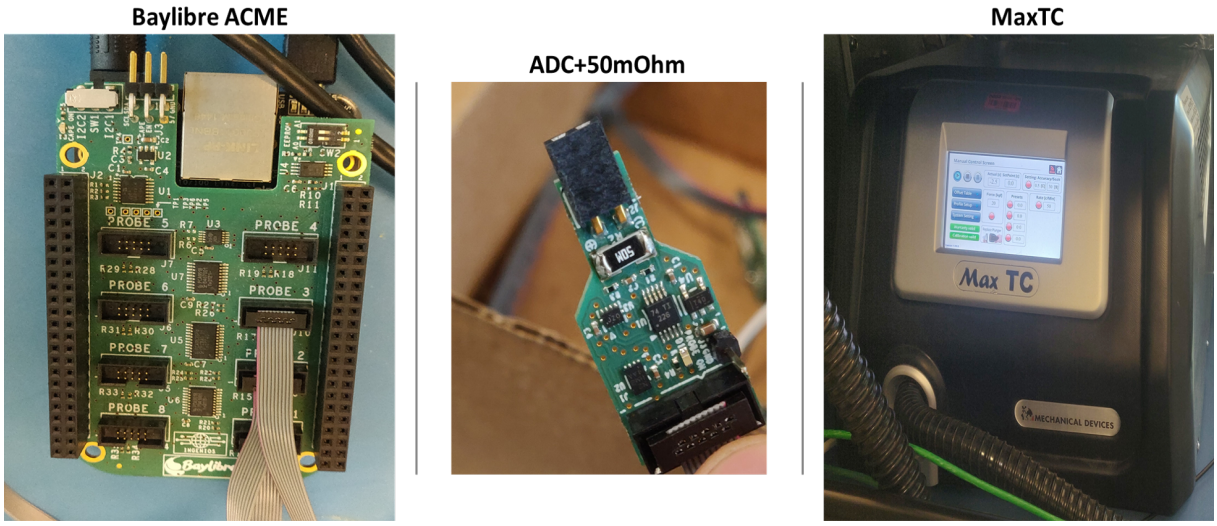


Figure 5.44: Power measurement tools

The results were visualized with the *PyACMEGraph* [120] tool which automatically calculates the current using Ohm's law. We monitored the temperature of the SoCs using the temperature forcing system *maxTC* while running the use cases and extracted the measurements for different temperatures. Figure 5.45 shows the power measurement benches of the i.MX8QM and i.MX8QXP using the tools mentioned above.

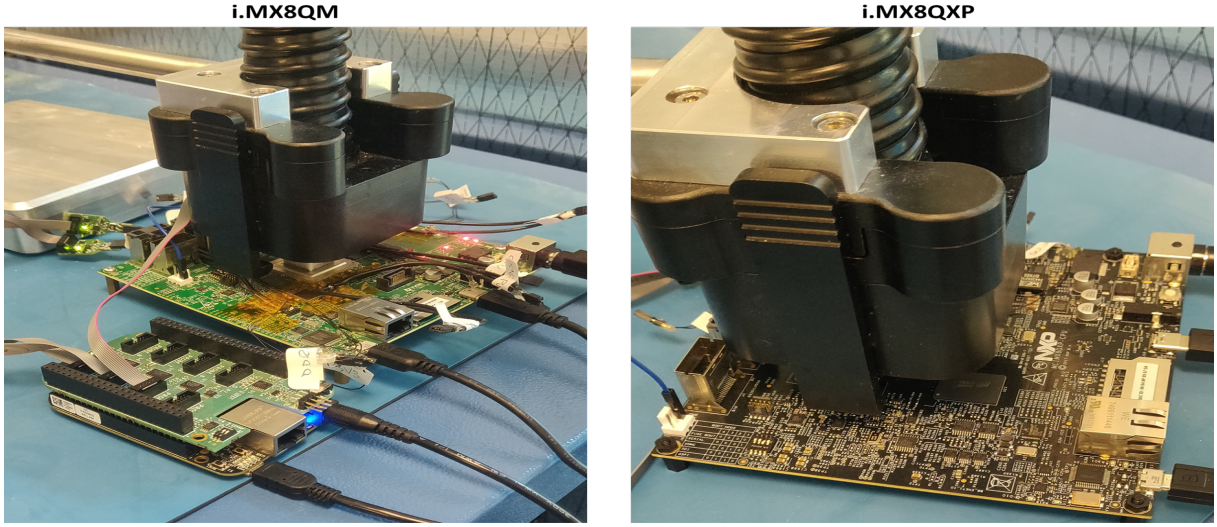


Figure 5.45: i.MX8QM and i.MX8QXP benches

In addition, in Figure 5.46, we can observe the process of extraction of power measurements for a temperature of zero degrees.

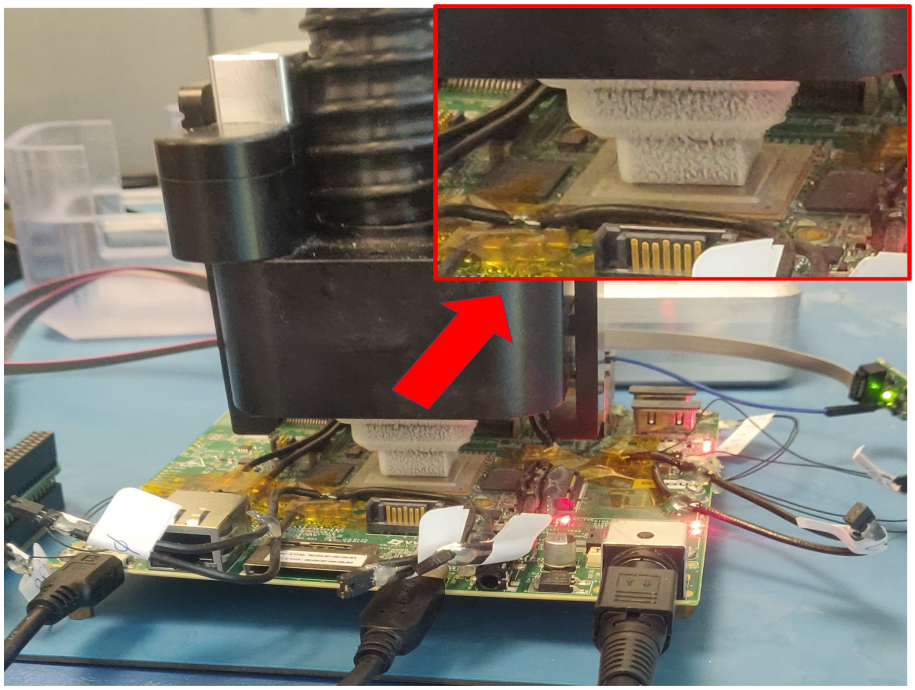


Figure 5.46: i.MX8QM bench for zero degrees temperature

In order to track the temperature of the SoCs, we used a debug monitor embedded in the System Control Unit Firmware (SCFW) that has a temperature monitoring function. The test case simulations were performed under a Linux OS. The test cases we simulated correspond to four important key states:

- *Key State 1 (KS1)*: System idle - This is a short term idle, the system has been running and can resume immediately based on an interrupt from a timer or a peripheral. Usually there is no memory traffic going on, but the resumption can be done immediately and does not require any software or firmware interaction. The PLLs and analog functions are all enabled and in “run mode”. Its likely that the SCU is in an idle state waiting for an interrupt. The CPU and SM+DRCs are powered up. This implies that the screen is blank and no I/O traffic is taking place, the CPUs are currently idle and waiting.
- *Key State 2 (KS2)*: Leakage measurement for tester - This is a measurement intended to be performed on the tester and provide a simple summary of the overall ”leakage” of the device. As such, it should include all components of the device, even if it is not a realistic use case.
- *Key State 3 (KS3)*: System idle with display - The intention is to replicate the case where the entire system is idle (similar to KS2 above) with the rather significant difference that a screen (potentially more in a real system but for this case we define the case as having a screen) is added. This implies that there is some memory traffic going on and some level of CPU activity.
- *Key State 4 (KS4)*: Stream - In this key state, we execute sequential memory read transactions from a CPU cluster where all 4 CPUs are active.

KS1 and KS2 were used to find the leakage current for both platforms and to extract some information about the power consumption when the systems are active, but no transactions are sent. The goal was to compare once with our simulation results and once with the calibration model used at the very beginning with i.MX8QM. KS2 was replicated for the measurement using some of the power commands available in SCFW that allow changing the power state of each component. Thus, we used a script that places all subsystems in the correct power state.

KS3 and KS4 are very similar to the simulation we performed using our framework with very minor differences, so they were used for correlation. The results of the i.MX8QM measurement for KS3 and KS4 are shown in figures 5.47 and 5.48.

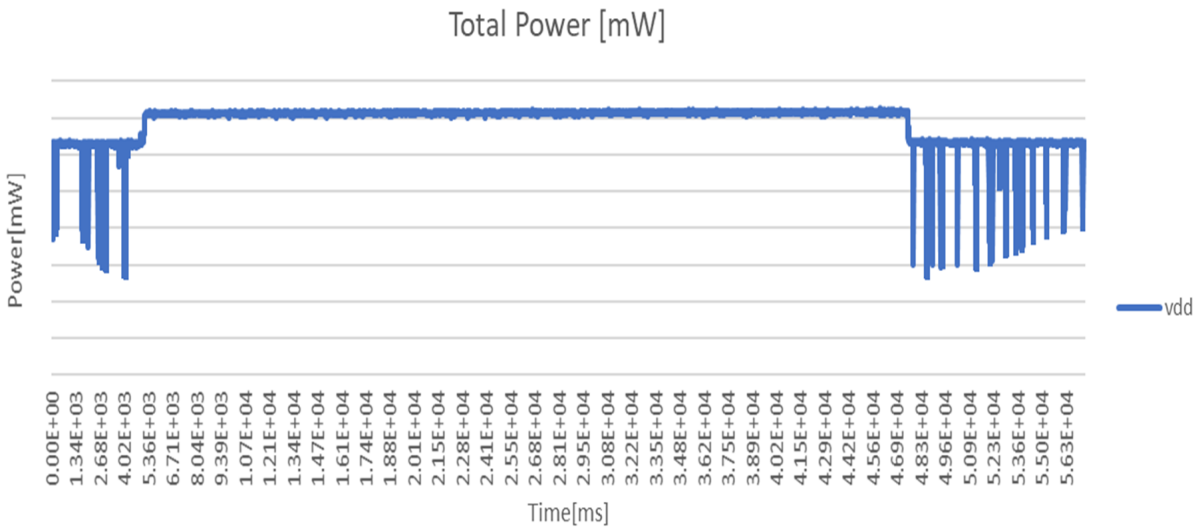


Figure 5.47: i.MX8QM: KS3 - Stream measurements

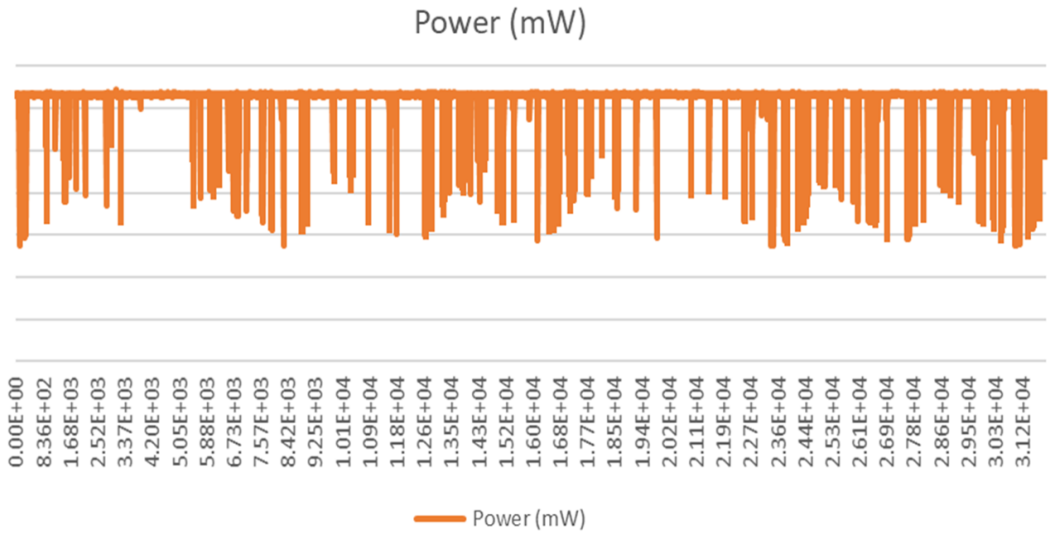


Figure 5.48: i.MX8QM: KS4 - Display ON measurements

The KS3 test case executed here represents 5000 iterations of 32MB memory copy. The resolution of the measurement is very low compared to our system-level simulations, which show the detailed power evolution during the execution of the use case. Due to the lower

measurement resolution, these curves sequentially show the average power consumption calculated for different windows of time. To establish the correlation, we extracted the average value from our memory copy simulation (1 iteration of 32MB) and compared it to the visibly stable power consumption extracted from the measurement. The correlation between the simulation and the temperature-controlled KS3 measurements for the i.MX8QM showed an accuracy of about 95% (at ambient temperature).

The KS4 visualize the power consumption of the system when a display is connected to it. Thus, again due to the lower resolution, we compared the average values extracted from the HD display refresh simulations and the KS4 power measurements (for a short period of time). The average power consumption correlation obtained from several simulations and measurements is about 90% with temperature control and 80% without temperature control. The power consumption correlation of this use case appears to be slightly less accurate than that of the memory copy (KS3), but this is not due to the accuracy of the power consumption modeling, but to the additional activity of the IPs and SM during actual display activity (and eventually the activity of the CPUs accessing the memory), which was not modeled here.

Using some physical data, the calibration model, and the results of KS1 and KS2, we estimated the accuracy of the separate units. The summary of the power correlation is summarized in Table 5.1.

i.MX8QM Hardware	Silicon/Simulation correlation
SM dynamic power	**85-99%**
SM static power	**85-90%**
2 DRCs dynamic power	**85-90%**
2 DRCs static power	**90-96%**
Total power	**88-92%**

Table 5.1: Silicon/Simulation range of power correlation observed between multiple runs

In the case of the i.MX8QXP, we have a few additional components that are part of the power domain where the SM and DRCs are located. These components were not modeled, but their power consumption was estimated using the spreadsheet approach. For the KS3 use case, the power consumption is very stable for this platform (in both simulations and measurements), which facilitates comparison. We deduced the power consumption estimates for the additional components from the total power consumption and found that the power correlation results show an accuracy of about 90-95%.

The correlation of power consumption between the simulation of these use cases and silicon measurements showed that the methodology applied with PwClkARCH and the tool itself are applicable to heterogeneous SoC systems of high complexity and IPs with a large number of elements in their power intent. The separation of concerns between functional and power models has proven to be extremely useful in the case of modeling reusable IPs. It allows us to easily swap and test different power intents/power management strategies for an IP or swap and test different functional models with one power intent/power management strategy. The correlation results are sufficiently accurate for this level of abstraction. In addition, at present, the silicon power measurement tools accessible without significant investment have relatively low resolution. Their results are usually an average value for a period of time/number of samples. For example, the resolution of our measurement tools is almost equal to the time needed to execute a display refresh. Thus, at the end of the measure-

ment, the value we observe is the average value of all power consumption variations during the retrieval of an image. With high-level simulation tools like PwClkARCH, we can extract a more detailed power consumption profile than with these measurement tools and closely observe its behavior.

The accuracy of our correlation allowed us to move to the final phase of the study and return to the top-down development approach. In this final part, we reuse the functional model units, parameters, power intent, and power management strategy to create a power-aware model of the next generation of the i.MX8 SoC series, which we will refer to here as i.MX8nextGen.

5.5 Application on new upcoming generation NXP SoC

In this final step, we reuse the units from the functional/power model again and integrate them on a higher performance and more complex next-generation i.MX8 platform. On this platform, the SM is separated into two smaller, connected SMs, which are responsible for the communication between more subsystems and external LPDDR memories. The clock frequency of the components is higher and the clock and power domains follow the same strategy as before, but their number is significantly higher.

The architecture of this platform is confidential (including all clock and power information), so we will focus the presentation on the effort required to generate the model with the reusable components and comparing parts of its power profile that show the increased performance.

5.5.1 Integration effort

The integration of the reusable IPs and simulation of the existing use cases was very intuitive and took less than a week (one person effort), which is slightly longer than the first case due to the use and verification of additional logic and algorithms embodied in the IP model, but not used for the previous platforms. All functional components have their automatic clock mechanism already associated and the necessary observers related to PwClkARCH and performance monitors. The granularity chosen for the modeling allowed us to maximize reuse, so we only needed to redefine the top-level modules of the SMs (more than one here) and add the additional power intent components into the already available power description file. Therefore, we reused the IP functional models, power intent descriptions, power management strategy, PMU and Master module. We reused the same testbenchs used in the previous two platform designs, but reconfigured with the appropriate identification indices, power supply and clock frequencies.

We first recompiled and re-tested each unit separately after reconfiguration and then the entire platform to ensure proper functionality.

5.5.2 i.MX8QM, i.MX8QXP and i.MX8nextGen power profile comparison

In Fig. 5.49, we illustrate the total power obtained after executing a QVGA display refresh use case on each of the three platforms.

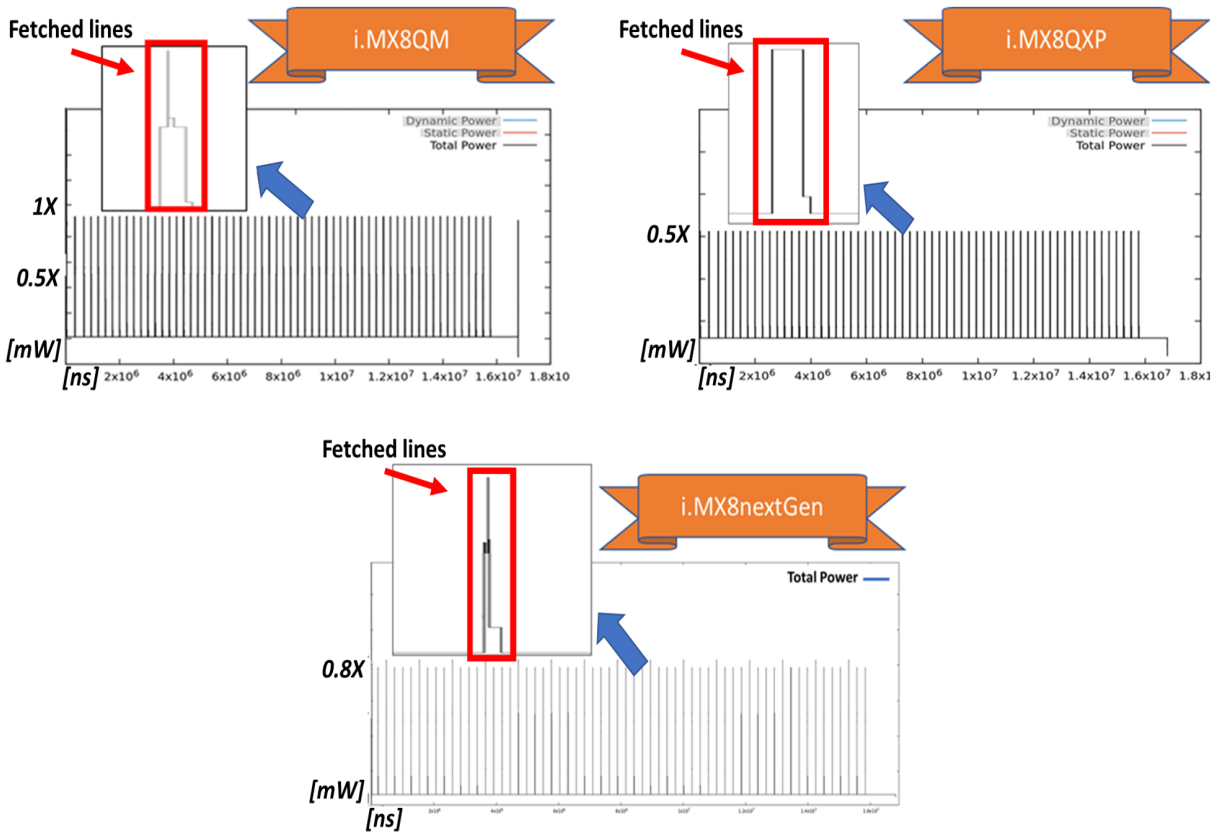


Figure 5.49: QVGA use case: i.MX8 platforms comparison [mW] *All power-related axis scales had to be masked for confidentiality reasons. Let us consider the fixed value "X" [mW] as the most significant power consumption between the 3 platforms presented here.*

We have a periodic generation of traffic corresponding to the different lines fetched from the memory. The width of each pulse, which represents a packet of a few fetched lines, depends on the screen resolution, SM bandwidth occupancy, transaction priorities, memory availability and SM+DRAM clock frequencies. In the case of the i.MX8QM, we have some peaks in maximum power consumption *(= 1X)* when both memories are accessed simultaneously and relatively stable power consumption *(about 0.6X)* when only one memory is used and the second is clocked. While the display consumes the captured lines, the dynamic consumption of the SM and DRAMs is reduced because they are not used. Using the i.MX8QuadMax results, we are able to compare the total consumption pulse width and height of the QVGA use case with the other two platforms. This may be important for the comparison of total power consumption.

In the case of the i.MX8QXP, we can see that the maximum power consumption per line retrieved *(about 0.5X)* is lower than that of the i.MX8QuadMax *(= 1X)*. On the individual pulses (line retrieval), we can observe that there are no consumption peaks, which is due to the single memory architecture of the i.MX8QXP. On the contrary, the width of individual

pulses for i.MX8QXP is about three times larger than for i.MX8QuadMax. Thus, we can conclude that in the case of i.MX8QuadMax, we have high peak power consumption, but lower latency and in i.MX8QXP there are no peaks, but stable power consumption and higher latency. This is due to the lower clock frequency of the i.MX8QXP.

In the case of the i.MX8nextGen, we can observe that the maximum power consumption is about *0.8X*. The maximum power consumption is higher than the i.MX8QXP, due to the dual memory architecture, and lower than the i.MX8QM, due to the higher frequency and lower voltage. When comparing the pulse width, the latency is 0.2 times lower than the i.MX8QM. The energy consumption of the i.MX8nextGen can also be compared to the other two platforms.

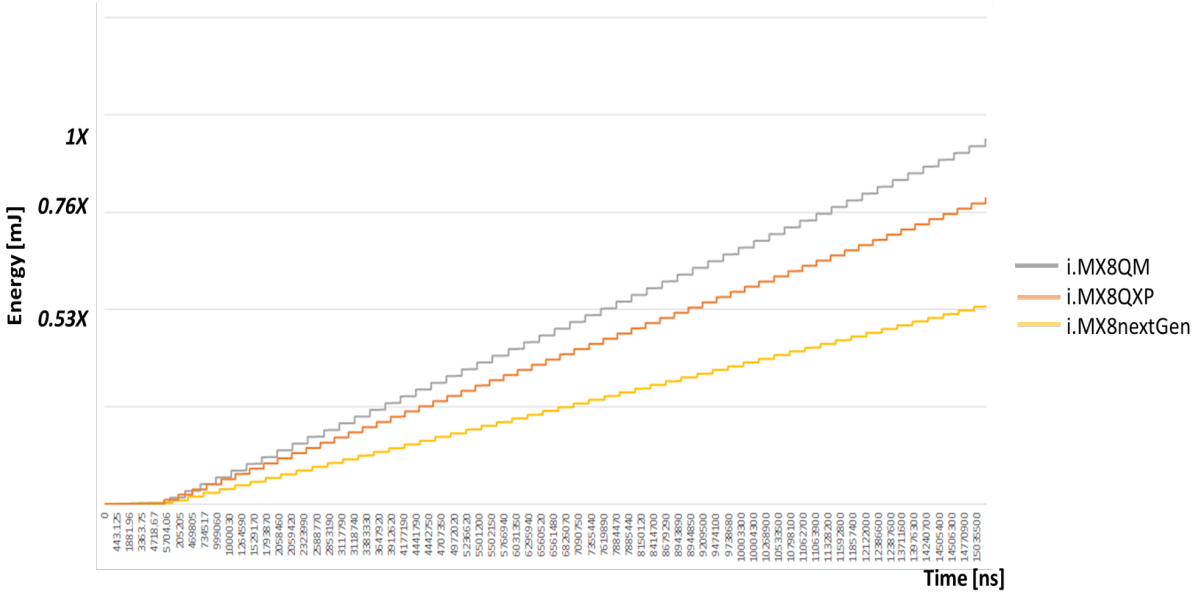


Figure 5.50: QVGA use case: i.MX8 platforms energy comparison [mJ] *All energy-related axis scales had to be masked for confidentiality reasons. Let us consider the fixed value "X" [mJ] as the most significant energy consumption between the 3 platforms presented here.*

In Figure 5.50, we can observe that the energy consumption of the i.MX8nextGen for the QVGA use case is significantly lower than its predecessors. This is due to the much lower latency compared to i.MX8QXP, the lower voltage and the higher clock rate. Even though the maximum power consumption of i.MX8nextGen is higher than i.MX8QXP, the final power consumption is much lower and its performance is significantly improved.

This can also be observed for the use case of the memory copy (Figure 5.51). In this case, there is no fixed simulation time, as with the display refresh where all simulations end almost at the same time (due to the similar vertical frequency around 60Hz) and the simulation is finished when the memory copy is complete. i.MX8QM and i.MX8nextGen are much faster than i.MX8QXP, due to the higher clock frequency and the use of two memories instead of one. The power consumption slope of i.MX8QXP is lower, but it needs much more time to perform the task.

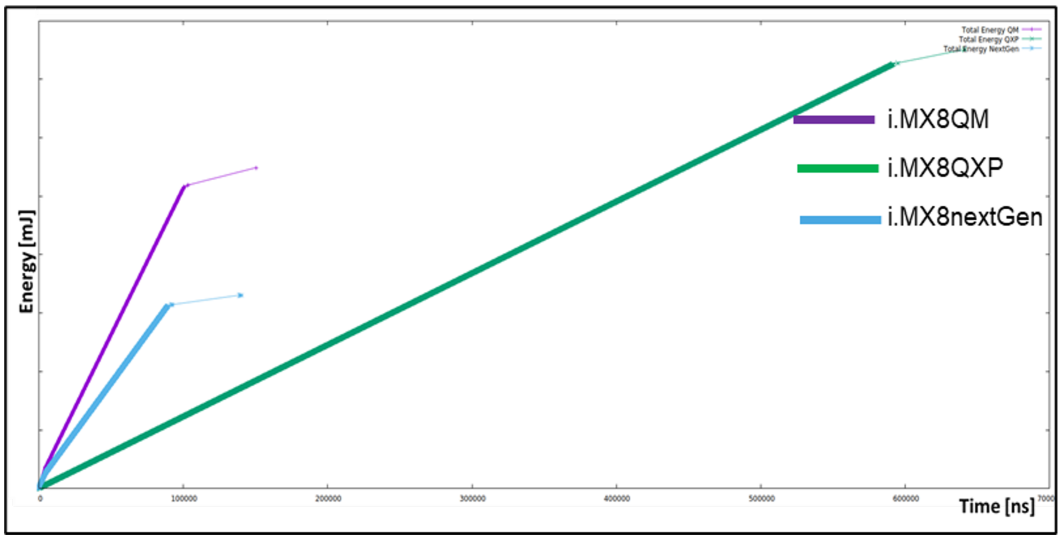


Figure 5.51: Memory copy use case: i.MX8 platforms energy comparison [mJ]

The correlation will be done when RTL or silicon is available. This will serve as proof-of-concept for this new methodology.

5.6 Simulation time impact of PwClkARCH

In order to scale the simulation time overhead due to PwClkARCH and the power model, we extracted and compared the simulation time of each use case presented in this study. The results are presented in Figure 5.52.

Platform	Period [ns]	MaxOT	MEMCOPY - 256 KB		Display Refresh QVGA		Display Refresh VGA		Display Refresh HD1080		MultiTask Display	
			CPU time (No PwClkARCH)	CPU time (PwClkARCH)	CPU time (No PwClkARCH)	CPU time (PwClkARCH)	CPU time (No PwClkARCH)	CPU time (PwClkARCH)	CPU time (No PwClkARCH)	CPU time (PwClkARCH)	CPU time (No PwClkARCH)	CPU time (PwClkARCH)
i.MX8QXP	1.666	16	14 sec	35.99 sec	8.114 sec	20.1 sec	27.5176 sec	78.34 sec	238.857 sec	585.4 sec	20.879 sec	65.58 sec
i.MX8QM	1.333	16	13.45 sec	35.94 sec	9.39 sec	20.02 sec	34.13 sec	80.9 sec	212.19 sec	514.8 sec	21.346 sec	64.5 sec
i.MX8nextGen	0.756	16	11.843 sec	36 sec	8.446 sec	21.3 sec	41.362 sec	80.26 sec	223.831 sec	621.74 sec	19.78 sec	65.37 sec

Figure 5.52: Approach simulation time overhead

The simulation time overhead due to PwClkARCH is about 3x compared to simulating the functional models alone. This simulation time overhead is relatively high, but it is mainly due to the opening/reading/writing of all log files with the power values. Furthermore, it can be reduced also by several optimizations that are part of future PwClkARCH work, such as distributed PMU for example. However, since we do not have access to solutions that can be compared to the one applied here (no open source code or licenses available), it is difficult to compare it to other studies. However, we can compare the run time of similar use cases at the RTL level, executed for performance analysis (without power analysis). For example, running a memory copy use case using the full RTL model of a similar platform takes from one to several hours (depending on the size of the memory copy). Executing a similar use case on a similar platform using UVM takes between 30 minutes and an hour and a half. Thus, the speedup in system-level simulation using SystemC and TLM2.0 is significant.

5.7 Additional co-simulation tests conducted with third-party tools

Some additional work has been done to increase the capabilities of our framework. The main objective is to integrate more accurate IP models for testbench construction.

5.7.1 Co-simulation with DRAMPower

In Figure 5.53, we illustrate a comparison between identical simulations on i.MX8QM with the only difference that for the first we use a simple memory model and for the second we use DRAMPower with an LPDDR4 specification derived from the *MICRON-16Gb-LPDDR3-1600_32bit* specification. The reason for this is that DRAMPower does not support LPDDR4 and therefore there is no associated specification file. The main difference between LPDDR3 and LPDDR4 architectures (except for the values of internal parameters) is that LPDDR3 provides a single data channel, while LPDDR4 provides a dual channel (width of 16 bits each). Fortunately, in the i.MX8 series platforms, the dual channel of LPDDR4 is used as one large channel. Thus, we were able to reuse the LPDDR3 model, with an LPDDR4 specification.

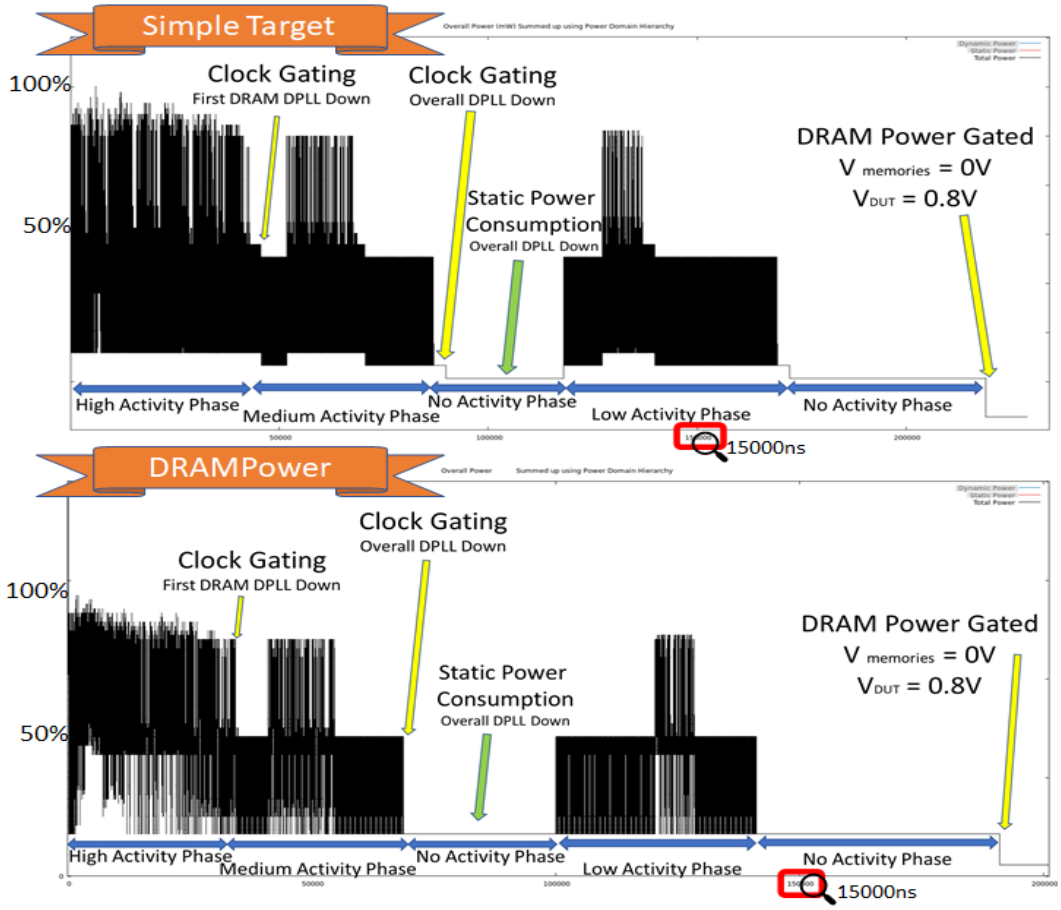


Figure 5.53: Generic Multi-task with DRAMPower: Total power [mW]

During the high activity phase, all traffic generators and both DRAMs are activated, thus achieving the maximum power consumption. During the medium and low activity phases,

we activate several (not all) traffic generators, but most of them access only one memory. There are small fractions of time when both memories are activated simultaneously. The non-activity phase is a rest phase added to test the clock triggering mechanisms. We can clearly observe the differences between the two simulations. The PwClkARCH simulation with DRAMPower uses more realistic and variable response timing values (considering refresh rates, bank and rank interleaving and more), while the single memory module uses constant values deduced and approximated from previous hardware measurements. The red rectangle shows a selected time in the simulation and highlights the timing error introduced by the constant value methodology. Thus, the PwClkARCH/DRAMPower combination improves our functional model and thus increases the accuracy of the power and performance estimation.

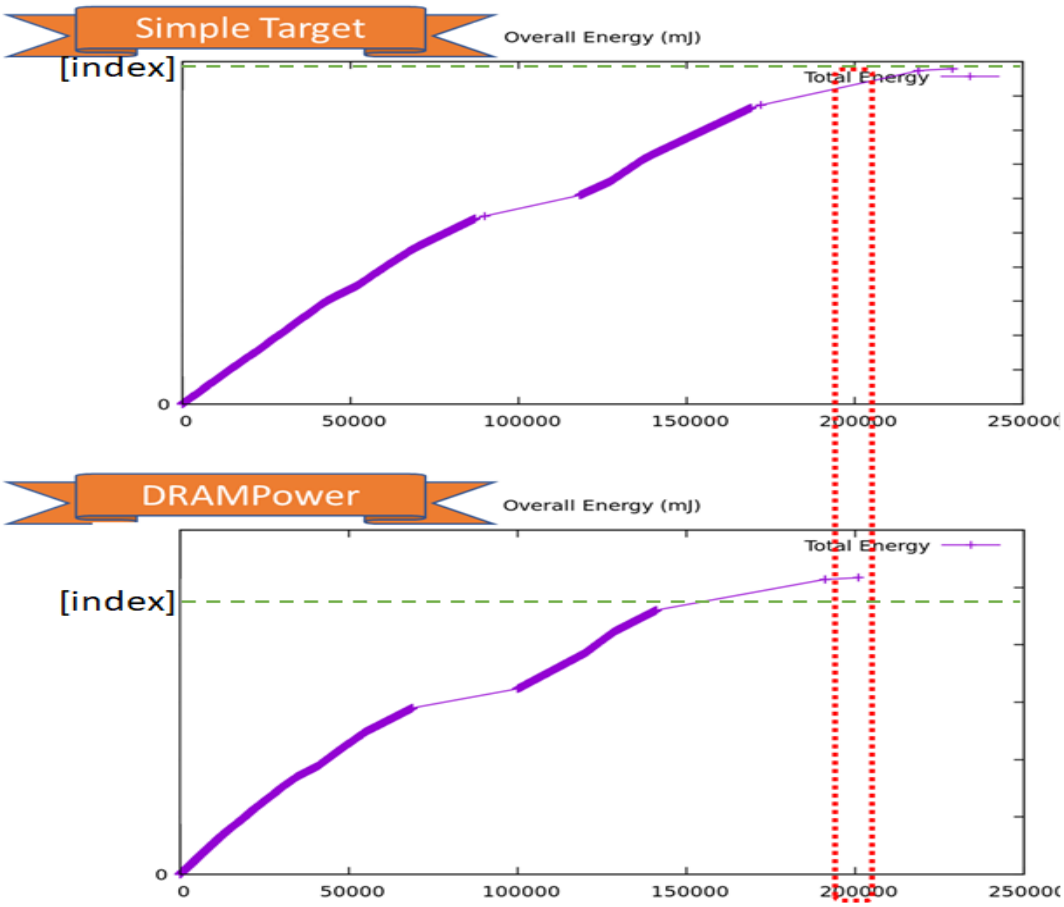


Figure 5.54: Generic Multi-task with DRAMPower: Overall energy [mJ]

In Figure 5.54, we illustrate the overall energy consumed during the two simulations. The *[index]* represents a fixed value used to compare the two energy consumptions. We can distinguish that the simulation using DRAMPower executes the same number of transactions in less time than the one using simple memory. Therefore, the slope of its curve is steeper. At this level, with these generic traffic generators, we can have a good correlation for the maximum, average and minimum energy consumption (with or without DRAMPower). However, if we want to simulate a real use case, we need more application-specific traffic generators, such as those used for the Memory Copy and Display Refresh use cases.

Memory copy

In this use case, we have an application-specific active traffic generator running 256 KB DRAM read accesses performed by a CPU cluster. Each data access is 16 bytes and we consider sequential memory accesses. To optimize memory utilization, we apply the 4K interleaving between our two DRAM memories.

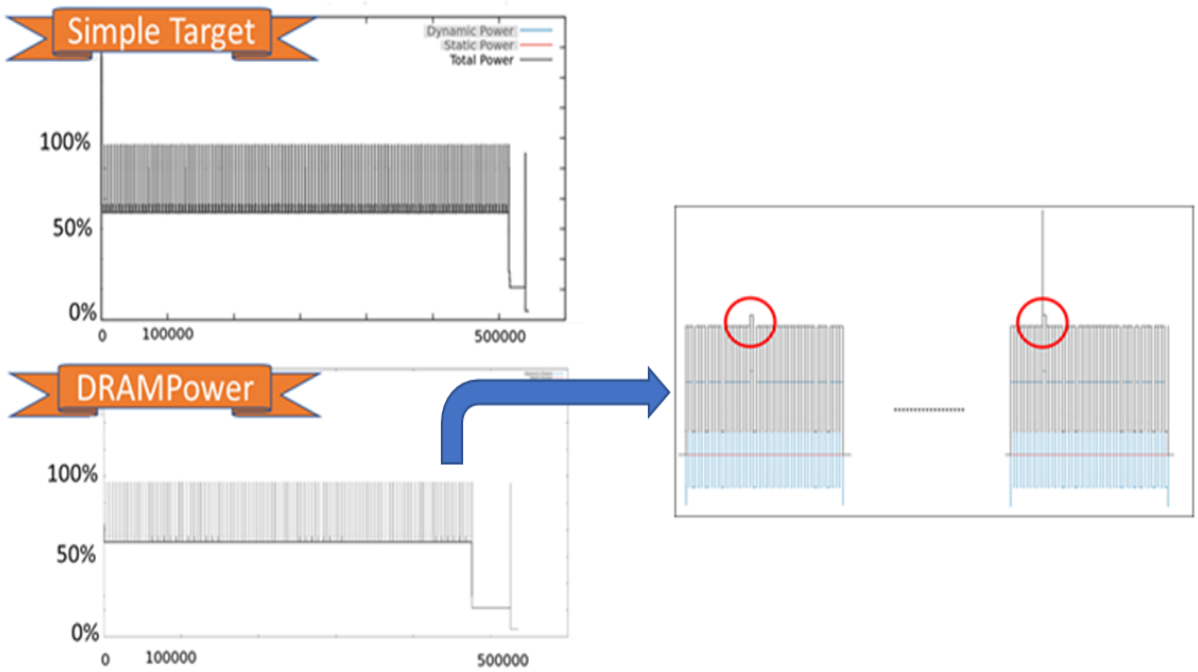


Figure 5.55: Memory copy with DRAMPower: Total power [mW]

In Figure 5.55, we can see the constant activity due to the continuous memory accesses. The spikes come from the interleaving moments between the two memories. The simulation with DRAMPower ignores some of the spikes due to interleaving, because we use multiple transaction windows (described in 4.3.3) and the memory power consumption is normalized to show the average power for entire window.

The main contribution of DRAMPower to our study was the improved temporal accuracy of the memory model and the consideration of memory refresh. However, we found the same contributions using DRAMSys4.0 memory models, but with reduced simulation time (because it is a SystemC/TLM2.0 model and uses the SystemC core) and an accurate functional model. In addition, it is compatible with DRAMPower, which can be very useful in a co-simulation between PwClkARCH, DRAMSys4.0 and DRAMPower.

5.7.2 Co-simulation with Platform Architect (PA) Ultra

The first steps we took with PA Ultra were to adapt our SM sequential path units from the functional model to the SystemC Modeling Language (SCML) FT sockets using TLM Creator and add them into the PA Ultra library for testing with Platform Creator. Then, using the IP libraries defined in Platform Creator, such as VPUs, AXI protocols, buses, clocks, generic memories, and STL workloads, we encapsulated our SM sequential path model and created a Group and SM modules with their corresponding testbench (Figures 5.56 and 5.57).

We performed several simulations (without power management and estimation) to test the functionality and analyze the performance using the PA Ultra tool.

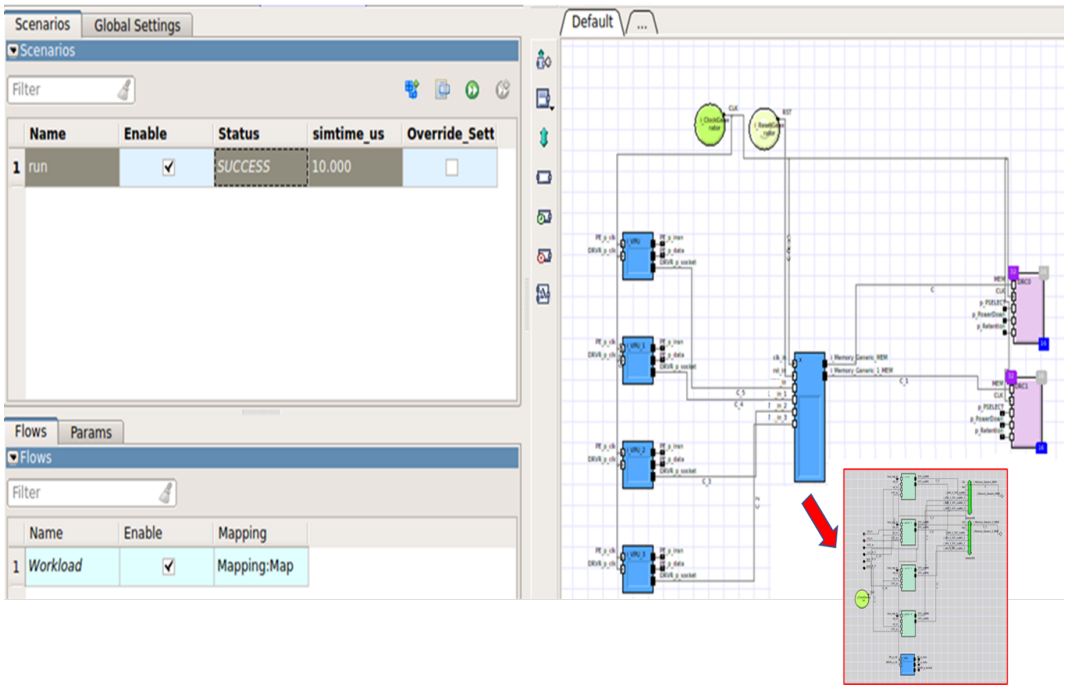


Figure 5.56: Platform Architect Ultra: Example of Group model

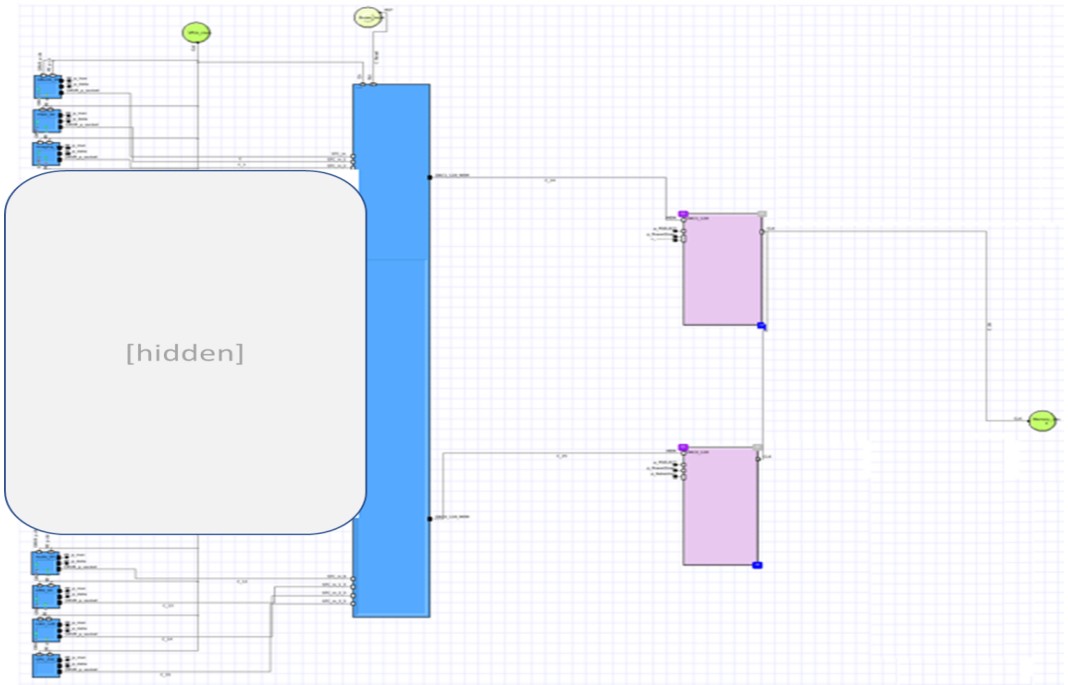


Figure 5.57: Platform Architect Ultra: Example of SM model

Then we found that, due to limited access (as users), at the moment the only way to integrate PwClkARCH into PA Ultra was to put our entire model (including the testbench) into a TOP module. We did this to test if the two tools can be easily compiled together and if they will work the same way as the external definition.

Using TLM Creator, we encapsulated the entire testbench into a TOP project, and included the PwClkARCH library. Then, using Platform Creator, we opened our project and started the simulation (Figure 5.58).

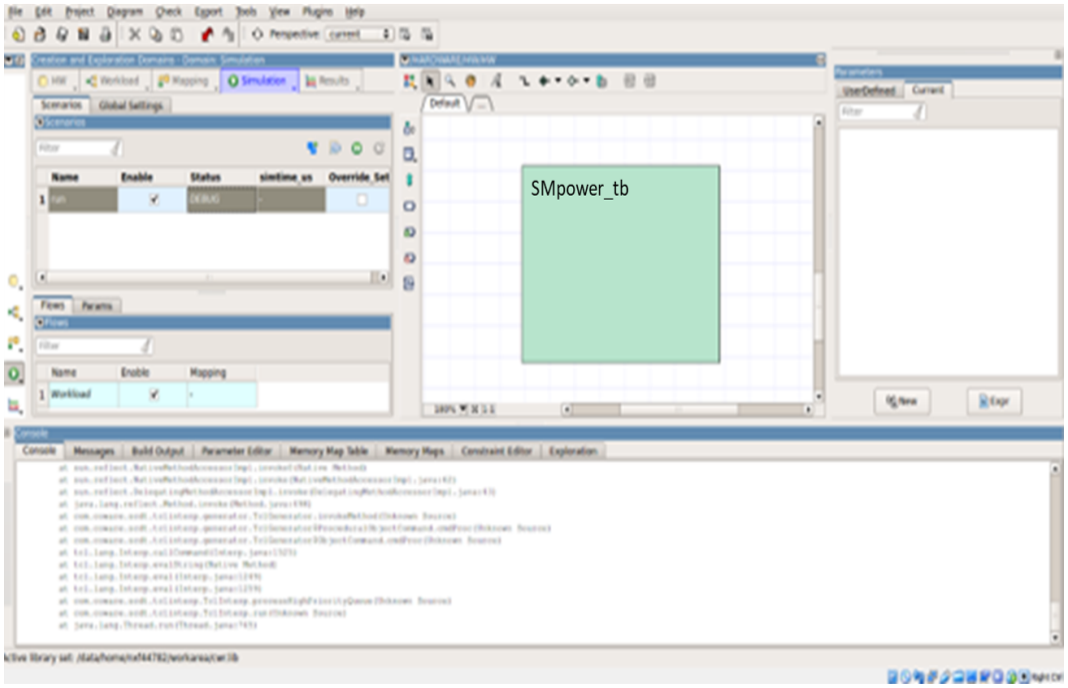


Figure 5.58: Platform Architect Ultra: SM power model

Using VP Explorer, we generated the power-related output files and used gnuplot to obtain the curves (Figure 5.59).

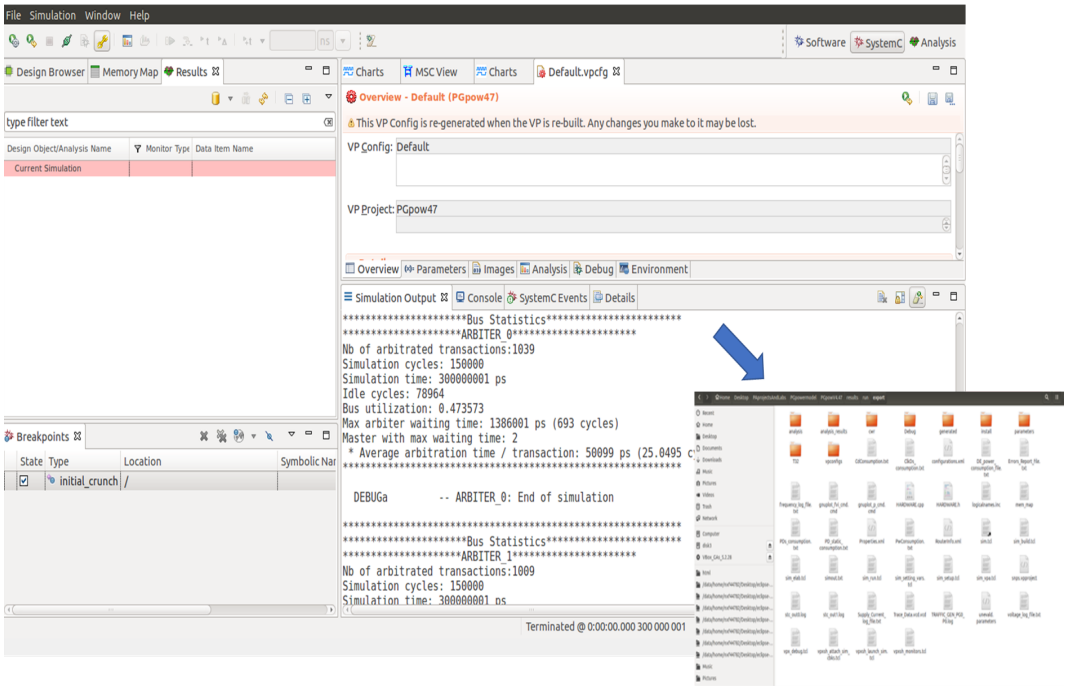


Figure 5.59: Platform Architect Ultra: VP Explorer output files generation

We observed the results of a generic multi-task use case where all traffic generators send

512 transactions at first, and then half of them repeat the procedure with another 512. In this use case, we do not have the IDLE phase between the two traffic phases and the memories are accessed randomly.

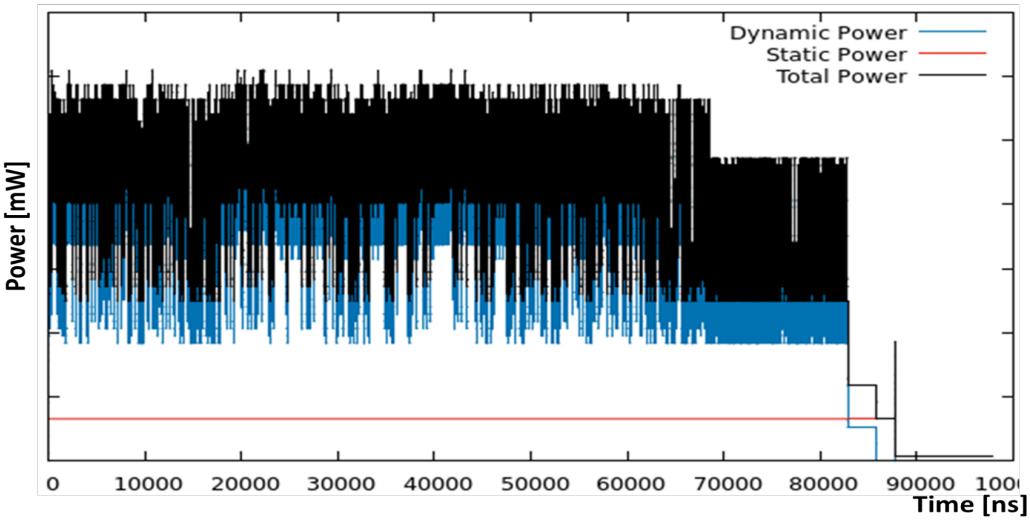


Figure 5.60: Platform Architect Ultra: Total power [mW]

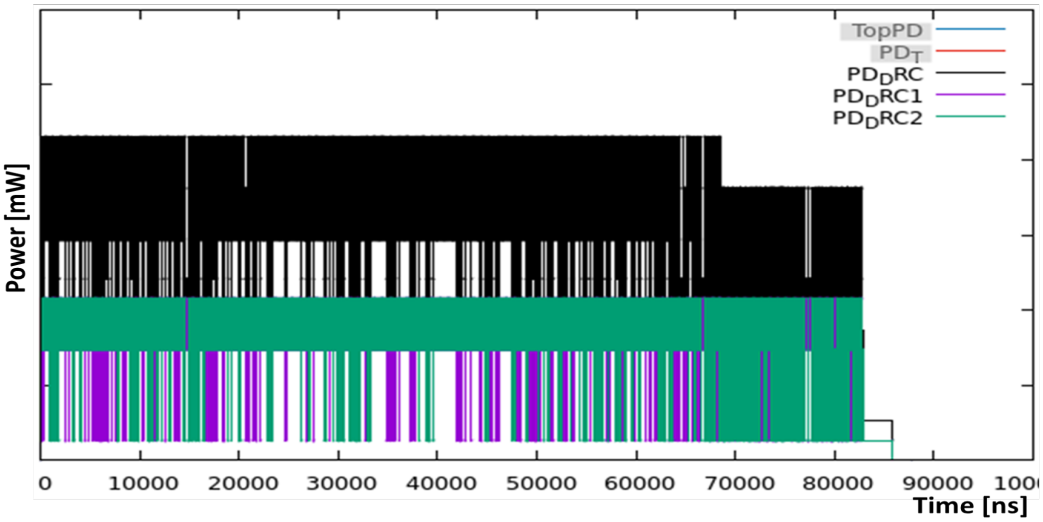


Figure 5.61: Platform Architect Ultra: DRCs dynamic power [mW]

On the total power (Figure 5.60), we observe the same maximum consumption as in the generic multi-task use cases studied outside the tool. We extracted the full dynamic consumption of a DRC using the results in Figure 5.61. Once the DRC is clock gated, we can extract its inactive power consumption (IOs and PLLs are still active). Thus, if we consider that due to the activity of a single SM sequential path, which interleaves between the two memories, we have one memory that is fully active and the other one that is clock gated, but still active (no time to go into IO down mode), we can estimate the total power consumption of the DRCs (one active + one inactive) and the active SM. The value calculated from here is approximately the same as the one found using the silicon measurements.

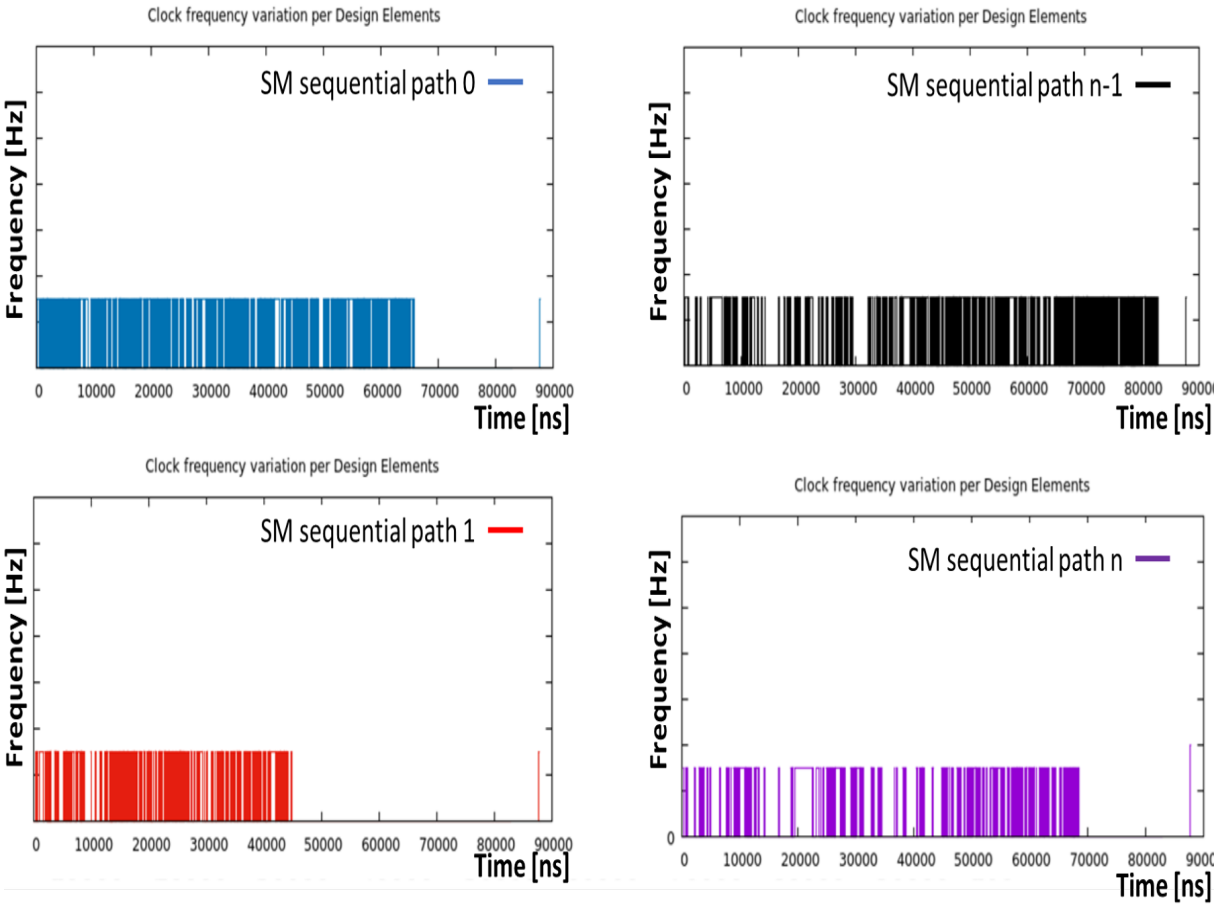


Figure 5.62: Platform Architect Ultra: SM sequential paths frequencies [Hz]

In the Figure 5.62, we can observe the clock trigger activations for several *SM sequential paths*. *SM sequential paths 0* and *n-1* are those activated during the two phases. *SM sequential paths 1* and *n* are those which are active only for one phase. We can see that the transactions on *SM sequential paths n-1* and *n* lose their priority compared to *SM sequential paths 0* and *1* almost during all the first phase and for this reason *SM sequential path n* is finished after *SM sequential path 0* in spite of the fact that it sends the half of the transactions that it sends. The reason for this is that we used fixed priority arbitration in the Arbiters that are connected to the *SM sequential paths*. This proves that PwClkARCH also allows us to observe the behavioral part and the activity report of the clock.

In order to integrate modules (IP models) from the PA Ultra library into our platform, we need to at least have access to the static top file *(HARDWARE.cpp)*, which was not the case for us because this file was generated dynamically at each recompilation in Platform Architect. In addition, we need access to the VPU and Generic Memory code or just their top-level modules. If we have access to the code, we can use the VPUs to initiate OPP state changes, and if we only have access to their top-level modules, we can derive from them and add a strategy similar to the one we use in our framework (i.e., add a port communicating with a master module). However, this was the only reason to integrate all modules (including our testbench) into a top-level project and we managed to prove the first layer of compatibility between the two tools.

5.8 Conclusion

In this chapter, we developed the targeted reverse engineering approach on NXP i.MX8QM and i.MX8QXP SoCs. We presented an overview of the experimentation we performed on each platform and showed some power-related observations in each. We presented our ”go-to” approach for choosing power parameter values (activity, load capacity, and leakage resistance) and model calibration. We presented simulation results for several generic and more realistic use cases for each of the modeled platforms. We have indicated the effort required to reuse the power-aware IP models, which has proven to be a significant benefit for modeling later generation designs. Starting with the model of an existing platform (i.MX8QM), we successfully ported the required units to a lower performance SoC (i.MX8QXP) and correlated the simulation results with silicon measurements. We have also showed the power consumption benefits from the power management definition using a comparison between a model with and without power management. It is important to highlight again that this comparison is done at ESL level, thanks to the PwClkARCH library. In addition, we created a model of our next generation high performance SoC, which is currently in its product definition phase (early design stage), and extracted some power estimates for this SoC, which can be correlated once RTL or/and silicon is/are available.

Due to the confidentiality of the SM architecture information, a significant portion of our power analysis was presented on the memory side, but the same analysis was performed for the entire SM subsystem.

In order to improve and complete our framework, we have done some additional work to include some important IPs in our modeling library. We included DRAMPower and DRAMSys4.0 for cycle-accurate memory models and started the integration of the gem5 tool which has a library containing CPUs, GPUs and other useful models that can further increase the capability of our framework (initial stage - not included in the report). We have shown some initial steps to integrate PwClkARCH into the Platform Architect Ultra tool which might be the most optimal solution, as this tool has a very rich IP library and strong performance estimation capability. PwClkARCH can be a very powerful add-on, as both tools follow the UPF standard for the power modeling approach and PwClkARCH allows the definition of power intent around functional models and the definition of power management strategy without implementing power state machines requiring lower level design information.

We also presented the overhead of simulation time due to the inclusion of a power model and some ideas on how to reduce it. The easy reuse of functional and/or power models showed the usefulness of the separation of concerns semantics applied in our approach. We were able to simulate the model with and without its power intent and observe the differences in simulation speed.

Chapter 6

General conclusion & perspectives

6.1 Conclusion

The increasing complexity and performance of modern integrated circuits is driving the opening of new and challenging application areas. However, this same complexity creates the need to look for improved design flows, especially in the complex heterogeneous SoCs that are increasingly used in different industries. We are creating a world where battery-powered mobile devices are all around us and need to be sufficiently reliable and resilient and especially power-efficient, while executing millions of different commands and tasks. Using old, slow design flow methodologies is no longer possible with these new levels of complexity and power efficiency constraints. Therefore, a proven approach to speeding up the design flow and increasing design performance and power analysis capabilities is raising the level of abstraction and start design explorations earlier. Part of the inevitable improvements is to include techniques such as virtual platforms, hardware/software co-simulations, early exploration of the design space and, more concretely, early power and performance analysis.

In conclusion, during this PhD, we have thoroughly studied the main semantics and approaches to power modeling and estimation that could be applied at the system level. We have created a framework that allows us to develop transactional level models of an SoC described exclusively in C++/SystemC-TLM2.0 and extract power consumption and performance metrics from these models. This type of high-level modeling methodology is not widely adopted in the industry, but it is becoming essential for all companies producing high-performance, low-power SoCs in order to be competitive. This is the first work on modeling and evaluation of an IP part of the i.MX8 SoC family and an introduction to system-level modeling in the NXP team responsible for this project.

The proof of concept for this approach was done on a very interesting custom interconnect subsystem called Switch Matrix, which is also a vital and very challenging component for power analysis. For confidentiality reasons, we were not able to present the concrete analysis of the Switch Matrix block. Therefore, we have provided an identical analysis of the entire voltage domain (including the Switch Matrix and memory systems). Nevertheless, the main contributions have been covered. Since at the system level there are no standards defining the exact granularity and abstraction when creating models for architectural exploration, we had to investigate and define how to approach this problem. The Switch Matrix is a relatively large and complex custom IP, where we cannot reuse existing models, so we had to develop it from scratch with an appropriate level of abstraction.

The power estimation and modeling approach we adopted (as users/testers of PwClkARCH) is unique because its implementation is less dependent on low-level component information and at the same time, it does not use power state machines for power modeling, but a few physical parameters allowing for more accurate power estimates. The accuracy of this approach is due to the full automation of dynamic power calculations performed in parallel of a functional model and the reduction of human intervention in the power consumption estimated data supply (e.g., there is no need to provide power consumption estimation for different power states). In addition, this approach allows the most common power reduction techniques to be applied in a simplified and straightforward manner. The strong UPF-like separation between the functional code and the power intent description significantly increases the design space exploration and power modeling capabilities, as we can easily test different combinations of power/functional models (jointly or separately) without having to modify the code. In addition, the PwClkARCH library follows the UPF standard, but it also adds the inclusion of DPLL components and the definition of clock domains, which are not present in other currently available tools and solutions at the same level.

The type and complexity of the subsystem, the use of the TLM2.0 AT coding style in the functional modeling and the silicon-power correlation, make this study a first industrial proof-of-concept of the power estimation capabilities of the PwClkARCH library for complex architectures (i.e. more than 40 clock domains, automatic clock gating mechanisms …).

This approach has the potential to improve the exploration of the design space at the system level and even the entire design flow, as it can help us detect bugs or sub-optimal power management strategies early in the flow. By avoiding the discovery of bugs at later stages of the design flow, we can avoid very significant expenses when developing new circuits.

6.2 Perspectives & future works

The prospects for future work after this study are numerous. With this and previous studies on PwClkARCH, we have proven that this approach is not IP specific and allows for accurate modeling and power estimation of different types of components with different levels of complexity and different use cases. We performed a power analysis on a much more complex SoC subsystem than any previously studied. However, the approach has not yet been tested on processors or full SoC platform with its hundreds of clock domains and design elements or with the implementation of more complex use cases (e.g., GPU and CPU traffic), which may give further conviction of the accuracy and capabilities of this framework. Both of these perspectives require the availability of such SystemC/TLM2.0 models, which can only be achieved by accessing a large library of IP components, or by developing an effortful model for each block of the SoC.

We have shown the importance of memory model accuracy in energy estimates and have proven the interoperability of PwClkARCH with other tools and models, such as DRAMPower, DRAMSys4.0 and Platform Architect. A logical future work is to co-simulate the PwClkARCH library and DRAMSys4.0 tool with the enabled DRAMPower memory power emulator. Using this setup, we can extract accurate models of memory power, as well as the rest of the systems. In addition, the inclusion of the gem5 or QEMU tools in this framework

can allow for the simulation of more complex use cases. In our study, this was not the goal, as we were looking for specific CPU and display traffic simulations, which were not available with gem5 or QEMU. However, it can have a significant added value for the environment.

On the PwClkARCH side, the modeling work has shown some limitations of the central PMU approach. A future work that can greatly increase the flexibility of the power model and the speed of simulation is to move to a distributed PMU model. In this way, it will simplify the power modeling of very complex architectures. This will allow the creation of multiple PMUs in a design, which means smaller clock, power and operating performance points tables to define and traverse and more efficient and concise code.

Furthermore, the PwClkARCH library does not take into account the thermal impact on power consumption. If we take the example of our platform, the simulation/measurement power correlation without temperature control during measurements (with maxTC) is significantly reduced. The thermal impact on the static and total power consumption is important and its modeling can significantly increase the capabilities of the tool. Hypothetically, the optimal solution would be to include an already existing thermal estimation tool in PwClkARCH (or co-simulation).

Another future work on the side of PwClkARCH, is its adaptation to new semiconductor technologies. Several extensions can be considered in order to adapt to semiconductor innovations and to allow internal consideration of the technology used for the design, while being careful not to require too low-level information.

PwClkARCH is based on the UPF standard. Its textual user interface contains a simple UPF-like language parser to integrate a UPF-like power partitioning file and automatically generate the code related to the PwClkARCH power model. This is very useful, as this file can be reused at multiple levels in the design. The same type of integration for the new UPM standard, targeting system-level power modeling, can be a very logical step in the future development of PwClkARCH.

The most optimal solution to enhance the capabilities of this framework may be to integrate it into a tool containing a library of cycle-accurate functional models and comprehensive testbenches, allowing for the inclusion of third-party IPs.

6.3 Previous publications

- A. Genov, L. Leconte, F. Verdier “Mixed Electronic System Level Power/Performance Estimation using SystemC/TLM2.0 Modeling and PwClkARCH Library”; DVCon EUROPE, Virtual, 27th –28th October 2020.
- A. Genov, A. Ben Ameur, F. Verdier, L. Leconte “Timing-Aware high-level power estimation of industrial interconnect module”; DVCon EUROPE, Virtual, 27th –28th October 2020.
- A. Genov, F. Verdier, L. Leconte, “Expand Reuse Strategy to ESL Power Modeling in SystemC/TLM”; 28th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2021, Dubai, 28th Nov – 1st Dec
- A. Genov, F. Verdier, L. Leconte, “Extension of the power-aware IP reuse approach to ESL”; DVCon U.S., 2022, San Jose (CA, USA), 28th Feb – 3rd Mar

Appendix A

DRAMPower/PWClkARCH parsing and co-simulation

```
void DramCtrl::request_path_cb(tlm::tlm_generic_payload& payload, const tlm
    ::tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) {
    ...
    check_mode.cancel();
    de->set_functional_state(false);
    switch(phase) {
        case tlm::BEGIN_REQ:
        if (payload.is_read()) {
            ...
            if(nb_trans_until_calc_power == NB_TRANS ||
    nb_trans_until_calc_power == 0) {
                ...
                if (t2r - t_abs_rw_prev > tREFI) t_abs_rw_prev = t2r;
                t_rw_prev = tREFI*(t_abs_rw_prev/tREFI);
                int x = freq_ghz*(t2r-t_rw_prev); //freq*(Cycles last RD/WR
    req)
                ...
                first_trans_file_DRAM1 << std::dec << x;
                first_trans_file_DRAM1 << ",READ,0x" << std::hex << payload.
    get_address() << std::endl;
                ...
            } else ...

            de->Remove_dynamic_pow_fixed(); //idle consumption

            int tt = freq_ghz*(t2r-t_rw_prev);//freq*(Cycles last RD/WR req)
            trace_file_DRAM1 << std::dec << tt ;
            trace_file_DRAM1 << ",READ,0x" << std::hex << payload.
    get_address() << std::endl;
            ...
            t_rw_prev = t2r;
            t_abs_rw_prev = t2r;
            current_nb_trans = nb_trans_until_calc_power;
            nb_trans_until_calc_power++;

            if (nb_trans_until_calc_power == NB_TRANS)
                check_mode.notify();
        } ...
```

```
        case tlm::END_REQ:
            ...
            mean_t_acces = de->get_access_time();//req acc delay
            LATENCY = sc_time(mean_t_acces,SC_NS);
            resp_delay = LATENCY+delay;
            ...
        case tlm::END_RESP:
            drc_state = ACTIVE;
            ...
            drc_state = CLK_GATED;
            check_mode.notify();
            ...
    }
}
void DramCtrl::check_state() {
  while (true) {
    wait (check_mode);
    ...
    int max_trans_from_prev = (current_time - t_prev)/interval_rw;
    int activity_ratio = nb_trans/(float) max_trans_from_prev ;
    if (current_nb_trans == nb_trans_until_calc_power) {
        nb_trans_until_calc_power = 0;
        de->Set_dynamic_pow_fixed(SDRAM_POWER_IDLE_MODE);
        switch (drc_state) {
            case  ACTIVE :
                ...
                SetActivity(activity_ratio);
                ...
            case IO_DOWN :
                ...
                SetActivity(activity_ratio);
                ...
            case PLL_DOWN :
                ...
                SetActivity(activity_ratio);
                ...
            default:
                ...
    }
  mean_t_access = de->get_access_time();
  t_prev = current_time;
  LATENCY = sc_time(mean_t_acces,SC_NS);
  }
}

void  DramCtrl::SetActivity(float activity)
{
  sc_core::sc_object& obj = dynamic_cast <sc_core::sc_object&>(*this);
  de = Design_elem_DRAM1::get_DE(obj);
  de->Update_dpow(activity);
}
```

Listing A.1: Trace file generation code snippet

```
void Design_elem_DRC::Update_dpow(float activity) {
    ...
  TraceParser traceparser;

  trace_file.open(DRAM_FIRST_TRANS_FILE1, ifstream::in);
  traceparser.parseFile(memSpec, trace_file, CMD_ANALYSIS_WINDOW_SIZE,
   grouping, interleaving, burst, power_down, trans);
  mpm.power_calc(memSpec, traceparser.counters, term);
  double power_from_refresh = mpm.power.average_power ;
    trace_file.close();

      trace_file.open(DRAM_TRACE_FILE1, ifstream::in);
      traceparser.parseFile(memSpec, trace_file, CMD_ANALYSIS_WINDOW_SIZE,
   grouping, interleaving, burst, power_down, trans);
  mpm.power_calc(memSpec, traceparser.counters, term);
  long int total_cycles = mpm.total_cycles; //nb tot cycles in curr window
  mean_access_time = clk_period * traceparser.cmdsched.mean_access_time ;
  trace_file.close();

  elapse_time = clk_period*total_cycles; //one window
  date_of_first_transaction = clk_period*traceparser.cmdsched.
   date_of_first_transaction ; //window start-point

  if (activity >= 0)  {
    _dynamic_pow = _dynamic_pow_base*activity + ((mpm.power.average_power*
   elapse_time - power_from_refresh*date_of_first_transaction)/(elapse_time
   - date_of_first_transaction));
    _activity = activity ;
  } else if (activity == ACTIVITY_CLK_GATED){
    _dynamic_pow =   DRC_POWER_CLK_GATED + _dynamic_pow_fixed;
    _activity = 0.0;
  } else if (activity == ACTIVITY_IO_DOWN){
    _dynamic_pow =  DRC_POWER_IO_DOWN + _dynamic_pow_fixed;
    _activity = 0.0;
  } else if (activity == ACTIVITY_PLL_DOWN) {
    _dynamic_pow =  DRC_POWER__PLL_DOWN + _dynamic_pow_fixed;
    _activity = 0.0;
  } else {}

    ...
}
```

Listing A.2: Design Element code snippet

Appendix B

Models skeleton

```
class IPBase: public ModuleBase, public ARK_Utils::Observable {
public:
    SC_HAS_PROCESS(IPBase);
    IPBase(sc_module_name name, sc_core::sc_time clk_period): ModuleBase(name), m_name(name)
        , ...
        , m_AcceptReadPEQ("acceptReadPEQ")
        , m_AcceptWritePEQ("acceptWritePEQ")
        , mPerfObserver("PerfObserver",this) {

        SC_METHOD(writeRequestThread); //AW scheduling
        this->sensitive << forwardWrReqEvent;
        this->dont_initialize();

        SC_METHOD(readRequestThread); //AR scheduling
        this->sensitive << forwardRdReqEvent;
        this->dont_initialize();

        SC_THREAD(acceptReadThread);
        SC_THREAD(acceptWriteThread);

        this->addObserver(&mPerfObserver,this->name());
    }

    IPBase(sc_module_name name, sc_core::sc_time clk_period_in,  sc_core::sc_time clk_period_out): ModuleBase(name)
        , ...
        , m_AcceptReadPEQ("acceptReadPEQ")
        , m_AcceptWritePEQ("acceptWritePEQ")
        , mPerfObserver("PerfObserver",this) {

        SC_METHOD(writeRequestThread); //AW scheduling
        this->sensitive << forwardWrReqEvent;
        this->dont_initialize();

        SC_METHOD(readRequestThread); //AR scheduling
        this->sensitive << forwardRdReqEvent;
        this->dont_initialize();

        SC_THREAD(acceptReadThread);
```

```
39          SC_THREAD(acceptWriteThread);
40          this->addObserver(&mPerfObserver,this->name());
41      }
42
43      //Helper functions - name of the inst where the module is used - @return
        const char*
44      virtual const char* basename()  const  { return m_name; };
45      virtual const char* fullname()  const  { return this->name(); };
46
47      //Behavioral model callbacks
48      virtual void manageForwardPath(tlm::tlm_generic_payload& payload, const
       tlm::tlm_phase& phase, const  sc_core::sc_time& delay, size_t idx) = 0;
49      virtual void manageBackwardPath(tlm::tlm_generic_payload& payload, const
        tlm::tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) = 0;
50
51      /**
52      *  Request Callback (forward)
53      *  @param receive index, payload reference and delay reference from
       protocol interfaces
54      */
55      void request_path_cb(tlm::tlm_generic_payload& payload, const tlm::
       tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) {
56      ...
57      manageForwardPath(payload, phase, delay, idx);
58      }
59
60      /**
61      *  Response Callback (backward)
62      *  @param  receive index, payload reference and delay reference from
       protocol interfaces
63      */
64      void response_path_cb(tlm::tlm_generic_payload& payload, const tlm::
       tlm_phase& phase, const sc_core::sc_time& delay, size_t idx) {
65      ...
66      manageBackwardPath(payload, phase, delay, idx);
67      }
68
69      tlm_utils::peq_with_get<tlm::tlm_generic_payload> m_AcceptReadPEQ;
70      tlm_utils::peq_with_get<tlm::tlm_generic_payload> m_AcceptWritePEQ;
71
72  protected:
73      sc_event forwardRdReqEvent;
74      sc_event forwardWrReqEvent;
75
76      TLMObsArg m_PldArgs;
77      ARK_Utils::PerfObserver mPerfObserver;
78  };
79
```

Listing B.1: IPBase skeleton

```
class BridgeModule: public IPBase
#ifdef PWARCH
, public Subject, public Assertions
#endif
{
public:
    SC_HAS_PROCESS(BridgeModule);
    BridgeModule(sc_module_name name, protocol::bus_type UsedProtocol,
    sc_time period);
    ~BridgeModule();
    ...
    #ifdef PWARCH
    void check_state();
    void SetActivity(float);

    uint32_t m_pendingCounter;
    int m_check_state;
    Observer<sc_core::sc_object>* m_PowObserver;
    Design_elem* de;
    #endif

    tlm::tlm_target_socket<WIDTH_in> input;
    tlm::tlm_initiator_socket<WIDTH_out> output;
protected:
    ARK_Protocols::SimpleTarget<BridgeModule, WIDTH_in>* inProtocolIF;
    ARK_Protocols::SimpleInitiator<BridgeModule, WIDTH_out>* outProtocolIF;
    ...
};
```

Listing B.2: Entity skeleton

```
1
2   template< unsigned int WIDTH_in, unsigned int WIDTH_out>
3   class BridgeArch: public BridgeModule<WIDTH_in, WIDTH_out>
4   {
5     typedef typename std::deque<tlm::tlm_generic_payload*> IdRegister;
6     typedef typename std::map<sc_dt::sc_uint<16>, IdRegister*> ReorderingMap
      ;
7     typedef typename std::vector<tlm::tlm_generic_payload*> RespRegister;
8     typedef typename std::map<sc_dt::sc_uint<16>, RespRegister*>
     RespReorderingMap;
9
10 private:
11     using BridgeModule<WIDTH_in, WIDTH_out>::m_pendingCounter;
12     using BridgeModule<WIDTH_in, WIDTH_out>::inProtocolIF;
13     using BridgeModule<WIDTH_in, WIDTH_out>::outProtocolIF;
14 public:
15
16     SC_HAS_PROCESS(BridgeArch);
17     BridgeArch(sc_module_name name, protocol::bus_type UsedProtocol, sc_time
       period);
18     ~BridgeArch();
19
20     /**ReadRequestThread
21     *  Request Callback (forward)
22     *  @param  payload reference, phase, delay and index
23     */
24     virtual void manageForwardPath(tlm::tlm_generic_payload& payload, const
     tlm::tlm_phase& phase, const sc_core::sc_time& delay, size_t idx);
25
26     /**
27     *  Response Callback (backward)
28     *  @param  payload reference, phase, delay and index
29     */
30     virtual void manageBackwardPath(tlm::tlm_generic_payload& payload, const
       tlm::tlm_phase& phase, const sc_core::sc_time& delay, size_t idx);
31
32     /**
33     *  Scheduling threads
34     *   @brief - new transaction pulling from the max priority queue
35     *   @param  receive index, payload reference and delay reference from
     protocol interfaces
36     */
37     virtual void acceptReadThread();
38     virtual void acceptWriteThread();
39     virtual void writeRequestThread();
40     virtual void readRequestThread();
41
42     /**
43     *  Helper functions - Functions used for Read and Write transactions (
     avoid code repetition)
44     *   @param - payload reference, required reordering tables, delay and
     index from protocol interfaces
45     */
46     virtual void pushToOrderingTable(tlm::tlm_generic_payload& payload,
     ReorderingMap& ReorderingMap);
```

```
47      virtual void reorderAfterEndResp(tlm::tlm_generic_payload& payload,
        RespReorderingMap& RespOrderMap, ReorderingMap& ReorderingMap, const
        sc_core::sc_time& delay, size_t idx);
48      virtual void reorderAfterBeginResp(tlm::tlm_generic_payload& payload,
        RespReorderingMap& RespOrderMap, ReorderingMap& ReorderingMap, const
        sc_core::sc_time& delay, size_t idx);
49      virtual void forwardRequest(ReqQueuesVect& requestQueuesVector, uint32_t
         high_priority_queue_index);
50
51  protected:
52      ReorderingMap m_readReorderingMap;
53      ReorderingMap m_writeReorderingMap;
54      RespReorderingMap m_readRespOrderMap;
55      RespReorderingMap m_writeRespOrderMap;
56    };
```

Listing B.3: Architecture skeleton

Bibliography

[1] Harry D. Foster. Why the design productivity gap never happened. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 581–584, 2013. doi: 10.1109/ICCAD.2013.6691175. xxi, 2

[2] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2nd edition, 2009. ISBN 0387699570. xxi, 23

[3] O. Mbarek, Alain Pegatoquet, and Michel Auguin. Using unified power format standard concepts for power-aware design and verification of systems-on-chip at transaction level. *IET Circuits, Devices & Systems*, 6(5):287–296, November 2012. doi: 10.1049/iet-cds.2011.0352. xxi, 48

[4] IEEE. Standard for Power Modeling to Enable System-Level Analysis. *IEEE Std 2416-2019*, pages 1–63, 2019. doi: 10.1109/IEEESTD.2019.8782907. xxi, 49, 50

[5] Amal Ben Ameur. *Transactional simulation approach for modelling performance and energy of a heterogeneous SoC memory system*. Theses, University of Côte d'Azur, June 2019. xxi, xxii, 56, 58, 110, 111

[6] NXP. i.MX Applications Processors, 2021. https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors:IMX_HOME Last accessed on June 2021. xxii, 72

[7] Benny Akesson and Kees Goossens. *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 144198206X. xxii, 110

[8] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 110–126, Cham, 2020. Springer International Publishing. ISBN 978-3-030-60939-9. xxii, 38, 114

[9] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. doi: 10.1109/N-SSC.2006.4785860. 1, 2

[10] F Rizzante. What does the future hold for cloud and edge computing?, 2021. https://www.information-age.com/what-does-future-hold-for-cloud-edge-computing-123494089/, Last accessed on June 2021. 1

[11] Hubert Kaeslin. *Top-down digital VLSI design: From architectures to gate-level circuits and FPGAs*. Morgan Kaufmann, 01 2014. doi: 10.1016/B978-0-12-800730-3.09988-3. 2

[12] Grant Martin, Brian Bailey, and Andrew Piziali. Chapter 1: What is ESL? In *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 9780080488837. 2

[13] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511. 3

[14] ARM Ltd. ARM Company, -. https://www.arm.com/company, Last accessed on June 2021. 3

[15] Vignesh Adhinarayanan, Indrani Paul, Joseph L. Greathouse, Wei Huang, Ashutosh Pattnaik, and Wu-chun Feng. Measuring and modeling on-chip interconnect power on real hardware. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, 2016. doi: 10.1109/IISWC.2016.7581263. 3

[16] L. Yang. Electrical Interconnect Energy Overhead. Stanford University, 2015. http://large.stanford.edu/courses/2015/ph240/yang2/, Last accessed on June 2021. 3

[17] NXP. About NXP, -. https://www.nxp.com/company/about-nxp:ABOUT-NXP., Last accessed on June 2021. 4

[18] LEAT. Le LEAT, -. https://leat.univ-cotedazur.fr/le-leat/, Last accessed on June 2021. 4

[19] E. Larsson. *Introduction to Advanced System-on-Chip Test Design and Optimization*. Springer-Verlag Boston, MA, 2005. https://doi.org/10.1007/b135763. 11

[20] Kim R. Fowler and Craig L. Silver. Chapter 1 - Introduction to Good Development. In *Developing and Managing Embedded Systems and Products*, pages 1–38. Newnes, Oxford, 2015. ISBN 978-0-12-405879-8. doi: https://doi.org/10.1016/B978-0-12-405879-8.00001-5. https://www.sciencedirect.com/science/article/pii/B9780124058798000015. 12

[21] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, San Francisco, CA, USA, 2007. ISBN 978-0-12-374274-2. https://doi.org/10.1016/B978-0-12-374274-2.50001-7. 14

[22] A. K. Tyagi. *MATLAB and SIMULINK for Engineer*. Oxford, Oxford University, 2012. ISBN 9780198072447. 14

[23] Scott Meyers. *Effective Modern C++*. O'Reilly Media, Inc., 1st edition, 2014. ISBN 1491903996. 14

[24] ARM. About ARMulator, -. https://developer.arm.com/documentation/dui0058/d/armulator-basics/about-armulator., Last accessed on June 2021. 15

[25] IEEE. Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012. doi: 10.1109/IEEESTD.2012.6134619. 15

[26] IEEE. Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual. *IEEE Std 1666.1-2016*, pages 1–236, 2016. doi: 10.1109/IEEESTD.2016.7448795. 15

[27] V. Snyder. Verilator Manual Release 4.213, 2021. https://www.veripool.org/ftp/verilator_doc.pdf, Last accessed on October 2021. 15

[28] Cadence. Synthesis: Creating the best balance of power, performance, and area, 2021. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis.html, Last accessed on September 2021. 15

[29] Synopsys. DC Ultra: Concurrent Timing, Area, Power, and Test Optimization, 2021. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, Last accessed on September 2021. 15

[30] Synopsys. Physical implementation: New RTL-to-GDSII solution, 2021. https://www.synopsys.com/implementation-and-signoff/physical-implementation.html, Last accessed on September 2021. 15

[31] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-Chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, USA, 2001. ISBN 0792372794. 15

[32] William Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 01 2005. ISBN 0137010923 9780137010929. 16

[33] Synopsys. Emulation, 2021. https://www.synopsys.com/verification/emulation.html, Last accessed on September 2021. 17

[34] Cadence. Palladium Emulation, 2021. https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html, Last accessed on September 2021. 17

[35] Erik Seligman, Tom Schubert, and M. Kirankumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann; 1st edition, 01 2015. ISBN 9780128008157. 17

[36] IEEE. Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pages 1–458, 2020. doi: 10.1109/IEEESTD.2020.9195920. 18

[37] Rainer Leupers, Grant Martin, Roman Plyaskin, Andreas Herkersdorf, Frank Schirrmeister, Tim Kogel, and Martin Vaupel. Virtual platforms: Breaking new grounds. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 685–690, 2012. doi: 10.1109/DATE.2012.6176558. 19, 90

[38] H. Foster. 2020 Functional Verification Study, Wilson Research Group and Mentor, A Siemens Business, 2020. https://blogs.sw.siemens.com/verificationhorizons/2020/11/05/part-1-the-2020-wilson-research-group-functional-verification-study/, Last accessed on November 2020. 19, 20, 47

[39] L. dos Santos. Low Power Techniques for SoC Design: basic concepts and techniques. Embedded Systems - INE 5439 Federal University of Santa Catarina, 2014. http://www.kunyuanic.com/upload/20191225091147_573.pdf, Last accessed on June 2021. 20

[40] H. Foster. 2018 Functional Verification Study, Wilson Research Group and Mentor, A Siemens Business, 2018. https://blogs.sw.siemens.com/verificationhorizons/2018/11/19/part-1-the-2018-wilson-research-group-functional-verification-study/, Last accessed on November 2020. 20

[41] D. Lidsky and J.M. Rabaey. Early power exploration-a World Wide Web application [high-level design]. In *33rd Design Automation Conference Proceedings, 1996*, pages 27–32, 1996. doi: 10.1109/DAC.1996.545539. 21, 51

[42] Synopsys. Platform Architect, 2018. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html, Last accessed on October 2021. 23, 32, 38, 53

[43] Accellera Systems Initiative. SystemC Synthesizable Subset Version 1.4.7, 2016. https://www.accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7.pdf, Last accessed on October 2021. 23

[44] Accellera Systems Initiative. About us, -. https://www.accellera.org/about, Last accessed on October 2021. 24

[45] MathWorks. Understanding Discrete-Event Simulation, -. https://www.mathworks.com/videos/series/understanding-discrete-event-simulation.html., Last accessed on June 2021. 24

[46] Open SystemC Initiative. OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL, 2009. https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf, Last accessed on June 2021. 26

[47] Cadence. Xcelium Logic Simulation, -. https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, Last accessed on October 2021. 32, 38

[48] Mentor Graphics. Mentor Graphics Underscores Low-Power Strategy with Vista Architecture-Level Power Solution, 2021. https://eda.sw.siemens.com/en-US/ic/vista-virtual-prototyping/, Last accessed on October 2021. 32, 38, 54

[49] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016. 32

[50] Guillaume Delbergue. *Advances in SystemC/TLM virtual platforms : configuration, communication and parallelism*. PhD thesis, University of Bordeaux, 2017. Dissertation supervised by Jego, Christophe http://www.theses.fr/2017BORD0916/document. 32, 36, 38

[51] Michael Keating and Pierre Bricaud. *Reuse Methodology Manual: For System-on-a-Chip Designs*. Kluwer Academic Publishers, USA, 1998. ISBN 0792381750. 32

[52] E. Sperling. The Limits Of IP Reuse, 2017. https://semiengineering.com/the-limits-of-ip-reuse/, Last accessed on November 2020. 32

[53] A. Viehl, T. Schonwald, O. Bringmann, and W. Rosenstiel. Formal Performance Analysis and Simulation of UML/SysML Models for ESL Design. In *Proceedings of the Design Automation Test in Europe Conference (DATE)*, volume 1, pages 1–6, 2006. doi: 10.1109/DATE.2006.244110. 35

[54] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411, 2011. doi: 10.1109/SAMOS.2011.6045491. 35

[55] Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. *A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems*, volume 33. Kluwer Academic Publishers, USA, 2006. doi: 10.1007/s11241-006-6884-x. https://doi.org/10.1007/s11241-006-6884-x. 35

[56] Fernando Herrera and Ingo Sander. Combining analytical and simulation-based design space exploration for time-critical systems. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pages 1–8, 2013. 36

[57] Z. J. Jia, A. Núñez, T. Bautista, and A. D. Pimentel. *A Two-Phase Design Space Exploration Strategy for System-Level Real-Time Application Mapping onto MPSoC*, volume 38. Elsevier Science Publishers B. V., NLD, 2014. doi: 10.1016/j.micpro.2013.10.005. https://doi.org/10.1016/j.micpro.2013.10.005. 36

[58] Matteo Monchiero, Ramon Canal, and Antonio Gonzalez. Power/Performance/Thermal Design-Space Exploration for Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):666–681, 2008. doi: 10.1109/TPDS.2007.70756. 36

[59] Cliff Chow, Tsong Yueh Chen, and T.H. Tse. The ART of Divide and Conquer: An Innovative Approach to Improving the Efficiency of Adaptive Random Testing. In *2013 13th International Conference on Quality Software*, pages 268–275, 2013. doi: 10.1109/QSIC.2013.19. 36

[60] Gunar Schirner and Rainer Dömer. *Quantitative Analysis of the Speed/Accuracy Trade-off in Transaction Level Modeling*, volume 8. Association for Computing Machinery, New York, NY, USA, 2009. doi: 10.1145/1457246.1457250. https://doi.org/10.1145/1457246.1457250. 36

[61] F. Carbognani, C.K. Lennard, C.N. Ip, A. Cochrane, and P. Bates. Qualifying precision of abstract SystemC models using the SystemC Verification Standard. In *2003 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 88–94 suppl., 2003. doi: 10.1109/DATE.2003.1186677. 36

[62] R. Jonack and J. L. Ambel. VP Performance Optimization - How to Analyze and Optimize the Speed of SystemC Models. In *Proceedings of the Design and Verification Conference Europe (DVCON Europe), Munchen*, 2014. 36

[63] Jan Henrik Weinstock, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, and Laura Tosoratto. Time-decoupled parallel SystemC simulation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, 2014. doi: 10.7873/DATE.2014.204. 36

[64] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous parallel SystemC simulation on multi-core host architectures. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 241–246, 2010. doi: 10.1145/1878961.1879005. 36

[65] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321113586. 36

[66] B. Andrist and V. Sehr. *C++ High Performance: Second Edition*. Packt Publishing, 2020. ISBN 9781839216541. https://google.github.io/styleguide/cppguide.html Last accessed on November 2020. 36

[67] Reinaldo Bergamaschi, Indira Nair, Gero Dittmann, Hiren Patel, Geert Janssen, Nagu Dhanwada, Alper Buyuktosunoglu, Emrah Acar, Gi-Joon Nam, Guoling Han, Dorothy Kucar, Pradip Bose, and John Darringer. Performance modeling for early analysis of multi-core systems. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 209–214, 2007. doi: 10.1145/1289816.1289868. 38

[68] Nourddine Abid, Wissem Chouchene, Brahim Attia, Abdelrim Zitouni, and Rached Tourki. Design and performance evaluation of on chip network with Transaction Level Modeling. In *ICM 2011 Proceeding*, pages 1–6, 2011. doi: 10.1109/ICM.2011.6177373. 38

[69] Huy-Nam Nguyen, Tuan-Anh Nguyen, Amir Nakib, and Eric Petit. Transaction Level Simulation of Network-Based Computing Systems. In *2014 International Conference on Computational Science and Computational Intelligence*, volume 1, pages 268–271, 2014. doi: 10.1109/CSCI.2014.53. 38

[70] Vladimir Herdt, Daniel Große, and Rolf Drechsler. Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 618–621, 2020. doi: 10.23919/DATE48585.2020.9116522. 38

[71] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016. doi: 10.1109/LCA.2015.2414456. 38

[72] SoCRocket. SoCRocket Web Page, 2014. http://www.esa.int/TEC/Microelectronics/SEMW9XK1QAH_0.html, Last accessed on October 2021. 38

[73] Thomas Schuster, Rolf Meyer, Rainer Buchty, Luca Fossati, and Mladen Berekovic. SoCRocket - A virtual platform for the European Space Agency's SoC development. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2014. doi: 10.1109/ReCoSoC.2014.6860690. 38

[74] Kai Hwang and Faye A. Briggs. Computer architecture and parallel processing. In *McGraw-Hill Series in computer organization and architecture*, 1986. 42

[75] S. Bhunia A. Raychowdhury, B. C. Paul and K. Roy. Computing with subthreshold leakage: device/circuit/architecture co-design for ultralow-power subthreshold operation. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 13(11):1213–1224*, 2005. 42

[76] Tamil Chindhu S. and N. Shanmugasundaram. Clock Gating Techniques: An Overview. In *2018 Conference on Emerging Devices and Smart Systems (ICEDSS)*, pages 217–221, 2018. doi: 10.1109/ICEDSS.2018.8544281. 43

[77] NXP. i.MX 8M Quad Power Consumption Measurement, Doc. number: AN12118, Rev. 2, 08/2018, 2018. https://www.nxp.com/docs/en/nxp/application-notes/AN12118.pdf, Last accessed on October 2021. 44

[78] IEEE. Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems. *IEEE Std 1801-2015 (Revision of IEEE Std 1801-2013)*, pages 1–1068, 2016. doi: 10.1109/IEEESTD.2016.7445797. 47

[79] IEEE. Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems. *IEEE Std 1801-2018*, pages 1–548, 2019. doi: 10.1109/IEEESTD.2019.8686430. 48

[80] Jinsong Ji, Chao Wang, and Xuehai Zhou. System-Level Early Power Estimation for Memory Subsystem in Embedded Systems. In *2008 Fifth IEEE International Symposium on Embedded Computing*, pages 370–375, 2008. doi: 10.1109/SEC.2008.48. 51

[81] Nidhi Chandoke and Ashish Kumar Sharma. A novel approach to estimate power consumption using SystemC transaction level modelling. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–6, 2015. doi: 10.1109/INDICON.2015.7443519. 51

[82] Vijay Narayanan, Ing-chao Lin, and Nagu Dhanwada. A power estimation methodology for systemC transaction level models. In *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, pages 142–147, 2005. doi: 10.1145/1084834.1084874. 51

[83] Rabie Ben Atitallah, Smail Niar, and Jean-Luc Dekeyser. MPSoC Power Estimation Framework at Transaction Level Modeling. In *2007 Internatonal Conference on Microelectronics*, pages 245–248, 2007. doi: 10.1109/ICM.2007.4497703. 51

[84] N. Bansal, K. Lahiri, A. Raghunathan, and S.T. Chakradhar. Power monitors: a framework for system-level power estimation using heterogeneous power models. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 579–585, 2005. doi: 10.1109/ICVD.2005.138. 52

[85] Muhammad Irfan, Shahid Masud, and Muhammad Adeel Pasha. Development of a High Level Power Estimation Framework for Multicore Processors. In *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*, pages 1–1770, 2018. doi: 10.1109/IMCEC.2018.8469473. 52

[86] J. Castillo, H. Posadas, E. Villar, and M. Martínez. Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures. In *XXII Conference on Design of Circuits and Integrated Systems*, pages 1–1770, 2007. 52

[87] R.A. Bergamaschi, Youngsoo Shin, N. Dhanwada, S. Bhattacharya, W.E. Dougherty, I. Nair, J. Darringer, and R. Paliwal. SEAS: a system for early analysis of SoCs. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pages 150–155, 2003. doi: 10.1109/CODESS.2003.1275275. 52

[88] Amr B. Darwish, Magdy A. El-Moursy, and Mohamed Dessouky. Transaction Level Power Modeling (TLPM) Methodology. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 61–64, 2016. doi: 10.1109/MTV.2016.21. 52

[89] Amr Baher, Ahmed El-Zeiny, Ahmed Aly, Ahmed Khalil, Adham Hassan, Abdel-Rahman Saeed, Karim Makarem, and Magdy El-Moursy. Dynamic power estima-

tion using Transaction Level Modeling. *Microelectronics Journal*, 81, 09 2018. doi: 10.1016/j.mejo.2018.08.012. 52

[90] Gautier Berthou, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power embedded systems. In *2020 Global Internet of Things Summit (GIoTS)*, pages 1–6, 2020. doi: 10.1109/GIOTS49054.2020.9119593. 52

[91] Marcio F. S. Oliveira, Eduardo W. Brião, Francisco A. Nascimento, and Flávio R. Wagner. Model Driven Engineering for MPSOC Design Space Exploration. In *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design*, SBCCI '07, page 81–86, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938169. doi: 10.1145/1284480.1284509. https://doi.org/10.1145/1284480.1284509. 52

[92] Feriel Ben Abdallah, Chiraz Trabelsi, Rabie Ben Atitallah, and Mourad Abed. Early power-aware Design Space Exploration for embedded systems: MPEG-2 case study. In *2014 International Symposium on System-on-Chip (SoC)*, pages 1–8, 2014. doi: 10.1109/ISSOC.2014.6972450. 52

[93] David Greaves and Mehboob Yasin. TLM POWER3: Power estimation methodology for SystemC TLM 2.0. In *Proceeding of the 2012 Forum on Specification and Design Languages*, pages 106–111, 2012. 52

[94] Jens Rudolf, Daniel Gis, Sebastian Stieber, Christian Haubelt, and Rainer Dorsch. SystemC Power Profiling for IoT Device Firmware using Runtime Configurable Models. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2019. doi: 10.1109/MECO.2019.8759994. 53

[95] Kim Grüttner, Philipp A. Hartmann, Tiemo Fandrey, Kai Hylla, Daniel Lorenz, Stefan Stattelmann, Björn Sander, Oliver Bringmann, Wolfgang Nebel, and Wolfgang Rosenstiel. An ESL timing & power estimation and simulation framework for heterogeneous socs. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 181–190, 2014. doi: 10.1109/SAMOS.2014.6893210. 53

[96] Hugo Lebreton and Pascal Vivet. Power Modeling in SystemC at Transaction Level, Application to a DVFS Architecture. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pages 463–466, 2008. doi: 10.1109/ISVLSI.2008.71. 53

[97] Dominik Macko, Katarína Jelemenská, and Pavel Cicák. Power-Management Specification in SystemC. In *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pages 259–262, 2015. doi: 10.1109/DDECS.2015.16. 53

[98] Dominik Macko, Katarina Jelemenská, and Pavel Cicák. Verification of Power-Management Specification at Early Stages of Power-Constrained Systems Design. *Journal of Circuits, Systems and Computers*, 26:1740002, 02 2017. doi: 10.1142/S0218126617400023. 53

[99] Intel. Intel® Docea™ Power Simulator, 2021. https://www.intel.com/content/www/us/en/system-modeling-and-simulation/docea/powersimulator.html, Last accessed on October 2021. 54

[100] Muhammad Mudussir Ayub, Habibullah Ahmadzay, Josef Eckmüller, and Franz Kreupl. Electronic System Level Power and Performance Analysis for Multi-Processor-System-on-Chip. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–2, 2019. doi: 10.1109/ReConFig48160.2019.8994783. 54

[101] Hend Affes, Michel Auguin, Francois Verdier, and Alain Pegatoquet. A methodology for inserting clock-management strategies in transaction-level models of system-on-chips. In *2015 Forum on Specification and Design Languages (FDL)*, pages 1–7, 2015. 55

[102] Ons Mbarek, Alain Pegatoquet, and Michel Auguin. A Methodology for Power-Aware Transaction-Level Models of Systems-on-Chip Using UPF Standard Concepts. In *Proceedings of the 21st International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization, and Simulation*, PATMOS'11, page 226–236, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642241536. 55

[103] Thomas Williams, Colin Kelley, Russell Lang, Dave Kotz, and al. gnuplot homepage, -. http://www.gnuplot.info/, Last accessed on June 2021. 69, 131

[104] Hend Affes, Amal Ben Ameur, Michel Auguin, François Verdier, and Calypso Barnes. An ESL framework for low power architecture design space exploration. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 227–228, 2016. doi: 10.1109/ASAP.2016.7760801. 70

[105] Eclipse. Eclipse Modeling Framework (EMF), 2016. https://www.eclipse.org/modeling/emf/, Last accessed on October 2021. 70

[106] Eclipse. EcoreTools, 2013. https://www.eclipse.org/ecoretools/, Last accessed on October 2021. 70

[107] Eclipse. Acceleo: GENERATE ANYTHING FROM ANY EMF MODEL, -. https://www.eclipse.org/acceleo/, Last accessed on October 2021. 70

[108] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. doi: 10.1109/MS.2003.1231150. 70

[109] Matthias Jung, Christian Weis, and Norbert Wehn. DRAMSys: A flexible DRAM subsystem design space exploration framework. *IPSJ Transactions on System LSI Design Methodology*, 8:63–74, 02 2015. doi: 10.2197/ipsjtsldm.8.63. 89

[110] Tim Kogel, Malte Doerper, Torsten Kempf, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Virtual Architecture Mapping: A SystemC Based Methodology for Architectural Exploration of System-on-Chip Designs. In *Computer*

Systems: Architectures, Modeling, and Simulation, pages 138–148, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-27776-7. 90

[111] ARM Ltd. AMBA® AXI™ and ACE™ Protocol Specification, 2011. http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf, Last accessed on June 2021. 97

[112] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. DRAMPower: Open-source DRAM Power & Energy Estimation Tool, -. http://www.drampower.info, Last accessed on June 2021. 110

[113] Karthik Chandrasekar, Benny Akesson, and Kees Goossens. Improved Power Modeling of DDR SDRAMs. In *2011 14th Euromicro Conference on Digital System Design*, pages 99–108, 2011. doi: 10.1109/DSD.2011.17. 110

[114] Matthias Jung, Christian Weis, Norbert Wehn, and Karthik Chandrasekar. TLM Modelling of 3D Stacked Wide I/O DRAM Subsystems: A Virtual Platform for Memory Controller Design Space Exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450315395. doi: 10.1145/2432516.2432521. https://doi.org/10.1145/2432516.2432521. 114

[115] Source Code Examples. C++ Observer Pattern Example, -. https://www.sourcecodeexamples.net/2020/09/c-observer-pattern-example.html, Last accessed on June 2021. 124

[116] GTKWave. Welcome to GTKWave, -. http://gtkwave.sourceforge.net/, Last accessed on June 2021. 128

[117] ANSYS. Ansys RedHawk-SC Digital Power Integrity Signoff, 2017. https://www.ansys.com/products/semiconductors/ansys-redhawk-sc, Last accessed on October 2021. 135

[118] Baylibre. ACME CAPE, . https://baylibre.com/acme/, Last accessed on October 2021. 162

[119] Mechanical Devices. maxTC. https://mechanical-devices.com/portfolio-posts/maxtc-high-power-temperature-forcing-system/, Last accessed on October 2021. 162

[120] Baylibre. PyACMEgraph, . https://baylibre.com/acme-and-pyacmegraph-part-2-2/, Last accessed on October 2021. 162