

# POLITECNICO DI TORINO

Master's Degree in Computer Engineering



## Politecnico di Torino

Master's Degree Thesis

# SystemC Simulation of Extra-functional properties for RISC-V-based systems

Supervisors

Prof. Massimo PONCINO

Prof. Daniele Jahier PAGLIARI

Prof. Sara VINCO

Dr. Matteo RISSO

Dr. Alessio BURRELLO

Candidate

**Francesco CALICE**

October 2023



# Summary

Nowadays, the RISC-V instruction set architecture (ISA) is gaining traction among engineers and companies thanks to its royalty-free and open-source nature. Following in the footsteps of highly successful open-source software projects (e.g., Linux, GNU, etc.), RISC-V represents an attractive choice to build an open-source hardware community that actively supports updates and ensures continuous improvement. RISC-V can be used without any fees, making it a cost-effective solution for enabling smart heterogeneous embedded systems, which include sensors and actuators, such as nano-drones, smart assistants, and wearables.

In this landscape, the availability of simulation platforms is a key ingredient for making RISC-V based hardware solutions pervasive. Simulators represent a cost-effective and scalable solution that enable rapid prototyping without the need for physical hardware. For these reasons, there is a growing availability of programs designed to simulate entire RISC-V chips. Some options include QEMU, Simulink, Renode, and GVSoC. However, these tools present some limitations. Most of them only simulate the computing core with scarce possibilities to perform system-level simulations. Moreover, they simulate only functional features, with poor support for extra-functional properties such as power consumption.

This work tries to address these challenges by presenting a flexible simulator, developed in collaboration with the University of Bologna, in the context of the TRISTAN European project. The simulator combines the SystemC/SistemC-AMS physical and continuous time modelling capabilities with GVSoC, a lightweight and flexible RISC-V functional instruction set simulator. The simulator takes two inputs. The first is a high-level description of the system in JSON which is then internally translated into SystemC/SystemC-AMS components through a template-based code generator written in Python. The second is the C code to be simulated on the RISC-V core through GVSoC. In this thesis, we focus on the co-simulation of the functionality and power consumption of the system. Nonetheless, the code has been developed to be general, allowing the support of additional extra-functional properties (e.g., temperature, reliability) with minimal intervention.

Overall, the thesis successfully demonstrates how to integrate SystemC/SystemC-AMS simulations of extra-functional properties with a RISC-V functional simulator.

# Acknowledgements

This thesis is the peak of my academic journey. Of course, there are many people I must thank, but there are not enough pages to express all my feelings. This project has been a very different experience from the usual student life. To start with, I would like to thank my academic supervisor, Daniele, Matteo, Sara, and Alessio, for leading me through this journey and providing me with this wonderful opportunity. I want to express my gratitude to my family, who has supported me since the beginning of my university journey. I also want to thank all my friends for always being there during both good and challenging times. Finally, I want to express my gratitude to all the people who have spent time with me; a part of this achievement belongs to you.



# Table of Contents

<b>List of Tables</b>	VIII
<b>List of Figures</b>	IX
<b>Acronyms</b>	XI
<b>1 Introduction</b>	1
<b>2 Background</b>	5
2.1 SystemC . . . . .	5
2.2 SystemC-AMS . . . . .	9
2.2.1 Timed data flow (TDF) . . . . .	10
2.2.2 Electrical Linear Networks (ELN) . . . . .	12
2.2.3 Linear Signal Flow (LSF) . . . . .	12
2.2.4 Scheduler & Differential-Algebraic Equation (DAE) solver . . . . .	13
2.3 RISC-V . . . . .	14
2.3.1 RISC-V Terminology . . . . .	14
2.3.2 RISC-V Software Execution . . . . .	15
2.3.3 Instruction set architecture (ISA) . . . . .	16
2.4 PULP & GvSoC . . . . .	18
2.4.1 PULPissimo . . . . .	19
2.4.2 GAP9 . . . . .	20
2.4.3 GvSoC . . . . .	21
<b>3 Related Works</b>	23
3.1 Modeling Cyber-Physical Electrical Energy Systems . . . . .	24
3.2 SystemC and Simulink Comparisons . . . . .	25
3.2.1 SystemC Cons . . . . .	25
3.3 Methodologies for Extra-Functional Properties . . . . .	26

<b>4</b>	<b>Methods</b>	<b>29</b>
4.1	Extra-Functional Simulator . . . . .	29
4.1.1	Sensors . . . . .	31
4.1.2	Functional Bus . . . . .	35
4.1.3	Power Bus . . . . .	37
4.1.4	Battery, harvester and converters . . . . .	38
4.1.5	Core . . . . .	39
4.1.6	Acknowledgement protocol . . . . .	40
4.2	GvSoC integration . . . . .	41
4.2.1	GvSoC Requests . . . . .	42
4.3	Python code generation . . . . .	44
4.3.1	Template syntax . . . . .	45
4.3.2	JSON file . . . . .	46
<b>5</b>	<b>Experimental Results</b>	<b>48</b>
5.1	Setup . . . . .	48
5.2	Simulator Results . . . . .	50
5.3	Simulator Benchmarks . . . . .	54
<b>6</b>	<b>Conclusions and Future Works</b>	<b>56</b>
<b>A</b>	<b>Code</b>	<b>58</b>
	<b>Bibliography</b>	<b>64</b>



# List of Tables

4.1	Examples of converter, the data integrity must be ensured . . . . .	39
5.1	Comparison between two C programs. The values represent the average command execution time over ten runs for the 'time command' in Linux. . . . .	54
5.2	Comparison between two C programs. The first one does not use external sensors, while the second one does. The values represent the average execution time of the 'time command' over ten runs in Linux. . . . .	55

# List of Figures

2.1	SystemC-AMS logical structure [7]	10
2.2	Example of Timed Data Flow module	11
2.3	Example of Electrical Linear Networks module	12
2.4	Example of Linear Signal Flow module	13
2.5	RV32I Instruction format [8]	17
2.6	RISCY core overview [10]	19
2.7	Zero-RISCY core overview [10]	19
2.8	PULPissimo overview [10]	20
2.9	Example of GvSoC simulation [12]	21
2.10	Example of GvSoC VCD traces [12]	22
3.1	Example of Layered structure for extra-functional property [15]	28
4.1	Power simulator conceptual	30
4.2	Sensor structure	31
4.3	Functional diagram	36
4.4	Power diagram	37
4.5	Acknowledgement protocol conceptual	41
5.1	Plot of functional report	53



# Acronyms

**CPEES**

Cyber-Physical Electrical Energy Systems

**DAE**

Differential-Algebraic Equation

**DMA**

Direct Memory Access

**EDA**

Electronic Design Automation

**EEI**

Execution Environment Interface

**ELN**

Electrical Linear Network

**HPC**

High-Performance Computing

**IoT**

Internet-of-Things

**ISA**

Instruction Set Architecture

**LSF**

Linear Signal Flow

**MOC**

Model of Computation

**PCB**

Printed Circuit Board

**PE**

Processing Elements

**PULP**

Parallel Ultra-Low-Power

**RISC**

Reduced Instruction Set Computer

**RTL**

Register Transfer Level

**SOC**

State of Charge

**SoC**

System-on-Chip

**TDF**

Timed Data Flow

**TLM**

Transaction-Level Modeling

**VCD**

Value Change Dump

# Chapter 1

## Introduction

System-on-Chip (SoC) devices have become ubiquitous in today's technology landscape, finding applications in smartphones, vehicles, industrial machines, smart homes, and even medical environments with health sensors. SoCs encompass a range of electronic devices that include essential components to ensure their proper functioning. Typically, these components consist of a microcontroller, memory, sensors, communication interfaces, and power supply. The complexity of a SoC varies depending on its intended use, and even minor modifications can result in significant differences in the final SoC design.

In a dynamically evolving technological environment like today's, the ability to make agile modifications to the design or even the entire system composition is invaluable for both companies and clients. A relatively recent development in SoC design is the concept of hardware emulation. Hardware emulation is an approach used to create a SoC within a secure, controlled, and digital environment, primarily aimed at testing specific functionalities before the physical board is constructed. This approach offers numerous advantages, such as a fully modular system that allows changes to all components on the fly. These changes can include altering the type of microprocessor, adjusting available memory, or modifying the number and type of sensors, empowering developers to explore design space thoroughly and identify the most suitable solutions for specific devices and applications.

Hardware emulation also offers the benefit of more precise simulation compared to software-based simulators. It can execute and simulate a real board, including internal timing and stages. Sensors can be configured to capture meaningful measurements, providing a more accurate representation of a real-world environment. Additionally, hardware emulation enables the creation of traces to monitor specific components and measurements, making it possible to assess stressful situations, for example.

One of the most significant advantages of hardware emulation is its impact on development speed. Emulating a board can expedite various development phases,

including reducing the development cycle. Engineers can rapidly test numerous changes without waiting for each new circuit, thus accelerating development. The parallelism between software and hardware allows two teams to focus on different aspects of the device, such as application software and the final design of the printed circuit board (PCB). Furthermore, this approach facilitates an agile development process, especially in the early stages, enabling iterative development phases that align with customer feedback within shorter timeframes compared to traditional methods.

Today, there are several key players in the field of hardware emulation, with Matlab and its extension, Simulink, being one of the prominent contenders. Simulink is an environment that excels in supporting model-based systems engineering, allowing for the management of complex systems throughout their lifecycle. It places a strong emphasis on various aspects, including representing components as models, providing a graphical view of the system, and illustrating relationships between components. Simulink is a robust software solution widely adopted in both academic and industrial settings. However, it is often criticized for its complexity and occasional sluggishness during extensive simulations. Additionally, Simulink is a closed and proprietary tool, which means it lacks the flexibility of being easily extendable.

In recent years, other software options have emerged as alternatives to Simulink. Notable among them is QEMU, an open-source software based on the C programming language [1]. QEMU excels at emulating complete systems and is primarily oriented toward running guest operating systems. Consequently, it is better suited for simulating general-purpose systems. On the other hand, RENODE is another software tool based on the Robot Framework and C# [2]. It finds particular relevance in the Internet of Things (IoT) domain and is well-suited for simulating and developing multi-node systems featuring numerous simple devices that need to communicate with each other.

While these software solutions each have their strengths, they also exhibit certain limitations. QEMU and RENODE excel in terms of functional aspects, effectively representing software behaviors. However, they struggle in simulating physical characteristics such as power consumption and energy management. In contrast, Simulink, due to its complexity, faces challenges when it comes to adequately representing and emulating functional features such as programming language instructions.

The primary goal of this thesis is to address the challenge of simulating both the functional aspects and extra-functional properties of a System on Chip (SoC). Specifically, the aim is to develop a versatile simulator capable of emulating the functional features of a SoC through the execution of real C language instructions while comprehensively simulating extra-functional properties like power consumption and energy management. This capability will enable a more thorough and

precise examination of SoC designs, allowing for in-depth analysis of the system at minute levels of detail.

To achieve this objective, the open-source Instruction Set Architecture (ISA) RISC-V has been selected as the foundation for the project. RISC-V adheres to the well-established principles of Reduced Instruction Set Computer (RISC) architecture and is characterized by its provision of royalty-free open-source licenses. This open-source nature not only fosters innovation but also nurtures a collaborative community of developers and researchers, [3].

In addition to simulating the functional aspects of the SoC, the simulator developed as part of this research will excel in modeling extra-functional properties. This will make it a valuable tool for engineers and researchers engaged in SoC design across various application domains.

The selection of RISC-V as the foundation is well-founded because one of its primary objectives is to ensure compatibility and portability across various system implementations. Its open-source nature has fostered a large and supportive community, making it increasingly attractive due to its royalty-free licenses in the years to come.

The simulated system will be based on PULPissimo, a microcontroller architecture that incorporates a RISC-V-based core. PULPissimo is the result of an academic collaboration between ETH Zurich and the University of Bologna, initiated in 2013, as documented on GitHub. To simulate the functional aspects of this architecture, GvSoC will be employed due to its high compatibility with the PULPissimo architecture.

For modeling the physical behavior of the system, SystemC and SystemC-AMS will be utilized. These extensions for C/C++ languages are capable of modeling and simulating systems of various types. In particular, SystemC can synchronize computation processes, simulate mathematical models in the time domain, and ensure fundamental hardware design properties such as signals and component reactivity. Moreover, SystemC-AMS offers versatility by providing three types of models: Timed Data Flow (TDF), Electrical Linear Network (ELN), and Linear Signal Flow (LSF), allowing for the most suitable representation for each system under study.

This work also aims to provide valuable insights into the field of high-performance computing (HPC) by leveraging the RISC-V architecture and the developed simulator's role in advancing HPC technologies [4]. It's essential to note that this research is aligned with the TRISTAN project, a major European initiative with the goal of expanding, maturing, and industrializing the European RISC-V ecosystem, allowing it to compete effectively with established commercial alternatives. Within the framework of TRISTAN, a comprehensive European strategy for RISC-V-based designs is being formulated, encompassing a repository of industrial-quality building blocks applicable to diverse application domains, including automotive and



industrial sectors. This initiative is holistic, covering electronic design automation tools (EDA) and the entire software stack. The broad consortium involved in TRISTAN aims to expose a significant number of engineers to RISC-V technology, thereby fostering wider adoption. This ecosystem development aligns with the European Commission’s strategy to support the digital transformation of various economic and societal sectors and advance towards a green, climate-neutral, and digital Europe [5].

The rest of this manuscript is structured as follows. Chapter 2 provides a comprehensive foundation by presenting the essential theoretical information required to grasp the thesis’s subject matter. It begins by elucidating key topics, namely SystemC and its framework SystemC-AMS, shedding light on their functionalities and advantages. Furthermore, it offers insights into RISC-V, detailing its nature and its relevance in System on Chips (SoCs). This chapter concludes with an exploration of various architectures, including PULP and GAP. Chapter 3 delves into studies closely related to this thesis, with a particular focus on the development of simulators for extra-functional properties and their current state-of-the-art. Chapter 4 constitutes the core of the project, offering a detailed exploration of the developed simulator and its seamless integration with GvSoC. It elucidates the details of this integration in meticulous detail. In Chapter 5, readers will discover a comprehensive guide on using the simulator, with meticulous attention to the machine setup process. This chapter also provides concrete examples showcasing the simulator in action, offering a practical understanding of its functionality. Finally, Chapter 6 brings this journey to a close by presenting conclusions drawn from the research and offering insights into potential avenues for future enhancements and expansions of the project.

# Chapter 2

## Background

In this chapter, all the information and knowledge necessary to comprehend the thesis project are presented. Additionally, detailed information about RISC-V and its objectives is provided. The chapter begins with a general overview of SystemC, the framework used to develop the simulator, delving into its internal structure 2.1. Subsequently, there is a dedicated section about SystemC-AMS 2.2, providing a deeper description of its special models and exploring the AD solver. The chapter then moves on to a comprehensive exploration of RISC-V 2.3, covering its history and structure. This section is very important for understanding the academic significance of RISC-V. Finally, there is section 2.4 dedicated to PULPissimo and GAP9, two architectures based on RISC-V. This section describes their structures and introduces GvSoC, the simulator utilized in the thesis to emulate the entire board from a functional perspective.

### 2.1 SystemC

SystemC is a C++ class library built on top of ANSI C++, enabling designers to create precise cycle models of system designs with a focus on hardware architectures and their interfaces. SystemC serves as a crucial tool for validation, optimization, and facilitating easy exploration and design space considerations. It introduces three fundamental properties: time, concurrency, and behavior's activity, all of which are essential for precise hardware modeling and analysis. By doing so, SystemC eliminates the need for the manual translation of C/C++ models into Verilog/VHDL, thereby reducing the potential for errors. In SystemC, systems are described using processing elements (PE) or modules, which exchange data through unidirectional or bidirectional channel classes `sc_signal`. An important concept closely related to SystemC's core functionality is transaction-level modeling

(TLM). TLM offers a high-level approach that strictly separates communication details among processing elements from the implementation details of functional units.

To understand SystemC, first is necessary to be familiar with some of its theoretical concepts and terminologies [6]:

- **Module** Container class, hierarchical, can contain other modules `sc_module`.
- **Process** The core functionality of any modules is contained in its processes, which are C++ methods. A module can have any number of three possible process types.
- **Port** A module sends/receives data to/from other modules via ports SystemC `sc_port` or `sc_export`.
- **Signal** Can be either resolved or unresolved. A resolved signal may have multiple drivers (such as a bus), while an unresolved signal has a single driver. Two- and four-valued signals are allowed, with permissible values being 'True,' 'False,' 'Don't care,' and 'High impedance.
- **Cycle-Based Simulation** Approach that focuses on simulating the behavior of a system in terms of cycles, where each cycle represents a fixed time interval during which certain operations or events occur.
- **Multiple Abstraction Levels** There are untimed models at different levels of abstraction, ranging from high-level functional models to cycle-accurate register transfer level (RTL). High-level models may be iteratively refined into more detailed lower-level models.
- **Sensitivity List** The sensitivity of a process is the set of events or timeouts which trigger that process. A process is sensitive to an event if that has been added to its static sensitivity list or dynamic sensitivity of the process instance.

SystemC offers a reactive, event-driven simulation infrastructure that accommodates two types of processes: spawned and unspawned. Unspawned process instances are generated by calling one of three process macros: `SC_CTHREAD`, `SC_METHOD`, and `SC_THREAD`, which are the most commonly used. Each of these macros is elaborated, unlike spawned processes, enabling efficient resource allocation during compile time. These three macros can be invoked from a module's constructor, along with an appropriate sensitivity list. A sensitivity list is closely tied to the fundamental concept of an event-driven simulator, where a process responds to an event, such as a change in the value of a signal. Consequently, the `SC_CTHREAD` macro requires either the rising or falling edge of any clock.

In contrast, the `SC_THREAD` macro provides greater flexibility and robustness by permitting a general sensitivity list that can include a clock. This allows the associated process to respond to value changes at all ports of its containing module, as specified in its sensitivity list. For example, if process `P` within module `M` is declared using the `SC_THREAD` macro in the constructor of `M` with sensitivities to ports `portA` and `portB` of `M`, then `P` will respond to changes in values received through `portA`, `portB`, or both. On the other hand, an `SC_METHOD` macro is executed each time its containing module is activated and cannot be suspended with a wait statement, unlike `SC_CTHREAD` and `SC_THREAD`. In contrast, both `SC_CTHREAD` and `SC_THREAD` macros are executed only once. Consequently, the code for any process declared as an `SC_CTHREAD/SC_THREAD` contains an infinite loop, causing the process to wait for events in its sensitivity list, with execution only terminating when the container module is destroyed at the end of the simulation.

Additionally, a spawned process can be generated by calling the built-in function `SC_SPAWN` during elaboration or simulation, and it can be utilized in fork-join parallel execution constructs. It is essential to note that fork-join constructs exist strictly in the software realm and do not represent any physical hardware.

Ports represent an indispensable element within any SystemC module. A port has the capability to establish connections with a channel, another port, or an export. Binding of a port or export can occur either by name or by position, but never simultaneously by both methods. For the actual binding process, one can utilize pertinent methods from built-in SystemC classes like `SC_MODULE`, `SC_PORT`, or `SC_EXPORT`. The binding of ports is characterized by its flexibility; for instance, port `A` can be bound to port `B`, which, in turn, may be connected to channel `C`, effectively linking port `A` to channel `C`. It's important to note that all port bindings are exclusively executed during the elaboration phase but can be deferred until the end of elaboration.

The execution of a SystemC application consists in two phases that are elaboration and simulation:

- **Elaboration** The initial and first phase involves establishing the application's module hierarchy, which encompasses the creation of module primitive channels and processes, setting up related data structures, binding ports and exports, executing the outer framework of the public implementation, and building the confidential core of the implementation.
- **Simulation** It involves the activation of the scheduler component within the kernel, which subsequently triggers the execution of the application's processes.

An application may be executed in two ways: application control and direct kernel

control. If executed under direct kernel control, it is managed by the scheduler. The scheduler can execute process only if one of the following conditions is satisfied.

- Process instance has been made runnable during initialization.
- A process has been sensitized to an event and that event has occurred.
- A timeout has occurred.

Scheduler execution is composed by a initialization phase, followed by the evaluation phase, the initialization phase itself consists of sub-phases as update *update phase* and *delta notification phase*.

The SystemC term delta cycle is used to indicate one step of the scheduler, the three sequential step, in detail, are:

1. **Evaluation Phase:** During this phase, the simulation evaluates and computes the values of various signals and variables in the system. It calculates the changes that need to be made based on the current state of the system and the events that have occurred since the last delta cycle. Essentially, it determines what has changed in the system.
2. **Update Phase:** After the evaluation phase, the update phase is where the calculated changes are applied to the system. This means updating the values of signals and variables with the new values computed in the previous step. The system transitions to its new state based on the changes determined in the evaluation phase.
3. **Delta Notification Phase:** In this final phase of the delta cycle, the simulation notifies any processes or components in the system about the changes that have occurred. This notification informs them that something in the system has changed, and they may need to react or perform certain actions in response to these changes. It essentially triggers processes to execute if they are sensitive to the events that occurred in the previous phases.

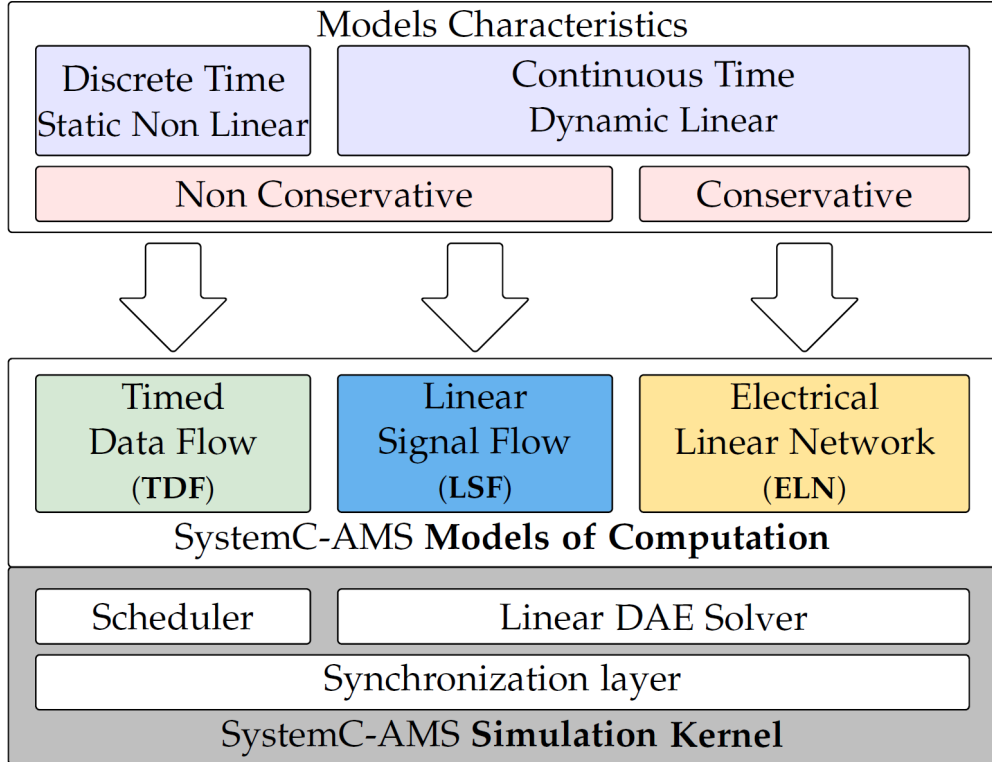
As an ANSI C++ library, SystemC supports each of the standard C++ data types and in addition hardware-specific data types to support concurrency, the concept of time, and events. In the end an example of SystemC module.

```
1 #include <systemc.h>
2
3 SC_MODULE(*Module Name*)
4 {
5     sc_core::sc_in <int> *Input Port*
6     sc_core::sc_out <int> *Output port*
7     sc_core::sc_signal <bool> *Internal signal*
8     SC_CTOR(*Module Name*): //Contstructor
9     {
10        SC_THREAD(*Function Name*);
11        //Sensitivity List
12        sensitive << *signal name*
13                << *signal name*;
14    }
15    //Function Prototype
16    void *Function Name*();
17
18    //Destructor
19    void *Module Name*() {};
20 }
```

## 2.2 SystemC-AMS

While digital circuits belong to a distinct category from analog circuits, approaches and methods for examining and comprehending analog and mixed-signal systems (AMS) have been relatively infrequent or have proven excessively intricate and time-consuming for analog system designers. Extending SystemC-AMS to ANSI C++ serves as a vital initial step in confronting this challenge. It leverages the foundations of the existing SystemC framework as outlined in IEEE 1666–2005 specifications, with the aim of addressing the unique demands presented by analog systems and the integration of digital hardware/software systems within their physical analog surroundings. To illustrate, when digital hardware/software interfaces with RF systems, sensors, actuators, and power electronics, the analysis must not only grapple with the specific issues inherent to purely analog or purely digital systems but also the real-time interactions between them. SystemC-AMS is specifically designed to cater to these distinctive requirements. It utilizes a combination of discrete-time static non-linear (non-conservative behavior) and continuous-time dynamic linear (both conservative and non-conservative behavior) model abstractions to offer three modeling frameworks: timed data flow (TDF), linear signal flow (LSF), and electrical linear networks (ELN). This enables a mixed-signal design scenario where, for instance, a purely digital control signal can govern a purely analog circuit by utilizing suitably conditioned feedback signals

from the analog circuit. Significantly, SystemC-AMS makes use of the identical TLM framework as pure SystemC, permitting users to concentrate on the data processing within a processing element, rather than concerning themselves with data input methods or the transmission of data to subsequent processing elements [6].



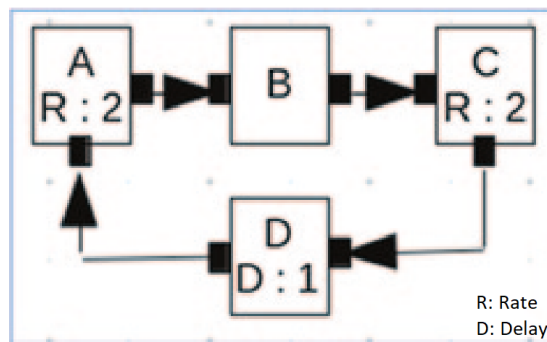
**Figure 2.1:** SystemC-AMS logical structure [7]

For effective utilization of SystemC-AMS, a thorough examination and comprehension of the fundamental principles, formalisms, and framework upon which it is built are essential. We will begin by providing a concise overview of the fundamental concepts that underlie the three integral components of SystemC-AMS: TDF, LSF, and ELN.

### 2.2.1 Timed data flow (TDF)

TDF serves as a model of computation (MOC) based on synchronous data flow MOCs. In this framework, data is conceptualized as signals sampled over time, carrying discrete or continuous information, such as signal amplitude, despite being tagged as discrete in terms of time. A TDF model is essentially an assemblage

of interconnected TDF modules, forming a directed graph referred to as a "TDF cluster." Within this graph, the nodes represent the TDF modules, while the edges denote the TDF channels or signals. Mathematical functions within a TDF module are executed using both direct inputs and internal states. A specific function is processed only when a requisite number of input data values become available, at which point the results are written to the output ports. It's worth noting that the number of input data elements required for a single invocation of a function may not necessarily match the number of data elements produced by that same invocation. Nonetheless, the quantity of input data elements needed for and generated by a single invocation remains constant. Each data element is accompanied by a time tag, referred to as a "time step," which accounts for the designation "TDF."



**Figure 2.2:** Example of Timed Data Flow module

TDF modeling formalism requires to specify the properties of a TDF module and for each of its ports the following properties:

- Time step of the module and each of its ports.
- Port rate of module's port i.e., the number of data elements read or written per each read/write operation.
- Delays and time offsets for each port.

Moreover, there are three strict constraints to follow during the data flow operations:

1. Attributes assigned to ports and modules must be compatible (No mismatch of data type).
2. Rate of data flow (samples/time) must match at sending and receiving ports.
3. All feedback loops must have port delays.



## 2.2.2 Electrical Linear Networks (ELN)

The ELN MOC introduces fundamental electrical components and their interconnections to model and analyze continuous-time, conservative electrical circuits. An ELN model comprises electrical components, such as capacitors, linked to nodes to create an electrical network. The mathematical relationships governing these components, which adhere to Kirchoff's current and voltage laws (KCL/KVL), are expressed as a set of differential algebraic equations and are resolved during the simulation process. An ELN model is essentially composed of a collection of electric components interconnected through terminals, forming an ELN cluster or a system of equations. The ELN primitive modules encompass a variety of components and their respective descriptions, including both dependent and independent sources (current and voltage sources), lumped elements (such as capacitors, inductors, and resistors), linear distributed elements (like transmission lines), ideal amplifiers (ideal operational amplifiers), linear gyrators, and ideal switches. As in the TDF model, the time step for an ELN module can be either explicitly assigned or propagated. In instances where an ELN module is interconnected with a TDF module in a hybrid configuration, the time step from the TDF ports is extended to the ELN model. It is mandatory to maintain consistency between the locally defined time step of the ELN module and the propagated time step to ensure accurate data exchange among the modules. Throughout the simulation process, the equation system is numerically solved using appropriate time steps.

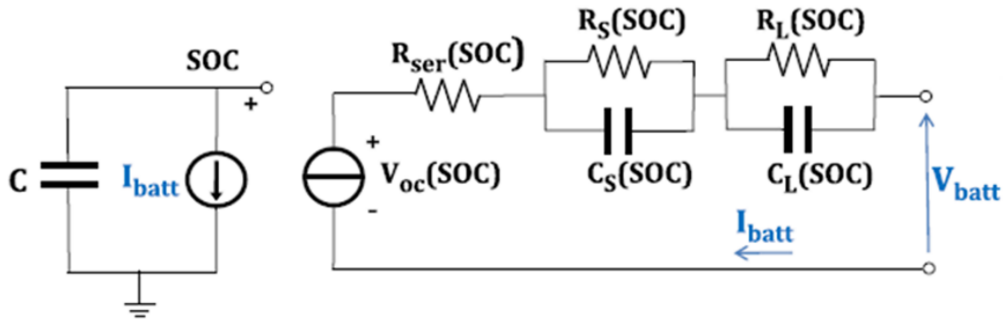
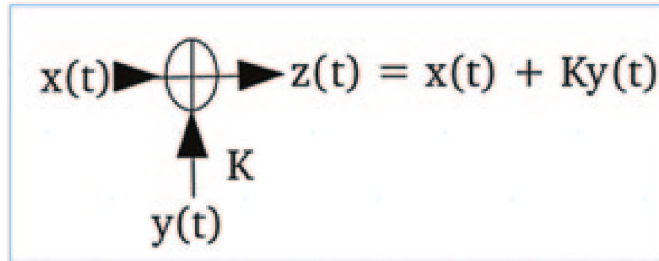


Figure 2.3: Example of Electrical Linear Networks module

## 2.2.3 Linear Signal Flow (LSF)

The LSF computational model provides a framework for modeling and analyzing an AMS system by defining relationships between variables through a system of linear algebraic equations. In other words, it handles non-conservative systems featuring continuous-time, directed real-valued signals. Each real-valued quantity represents

a signal within this context. Visually, a model is represented by a collection of blocks (referred to as LSF modules) interconnected by arrows (designated as LSF signals). These LSF modules feature input and output ports, allowing the model to interact with other components, such as a TDF module. Unlike TDF, it is not permitted to write custom code for the model but instead must employ pre-defined members of a set of LSF models (including operations like addition, subtraction, multiplication, differentiation, etc.) as needed. Much like the TDF model, the time step of an LSF module can be either explicitly set or transmitted. In cases of hybrid configurations, where an LSF module is linked to a TDF module, the time step from the TDF ports is carried over to the LSF model. Ensuring coherence between the locally defined time step of the LSF module and the propagated time step is essential for accurate data exchange between these modules. Throughout the simulation, the LSF equation system is numerically solved using suitable time intervals.



**Figure 2.4:** Example of Linear Signal Flow module

### 2.2.4 Scheduler & Differential-Algebraic Equation (DAE) solver

The SystemC language operates on an event-based architecture, employing a centralized scheduler to govern process execution triggered by events, including synchronizations, time notifications, or changes in signal values. The SystemC-AMS simulation kernel enhances the standard SystemC kernel through the incorporation of three supplementary components. Firstly, a TDF scheduler organizes TDF modules into clusters of interconnected units, constructing a static schedule for each cluster based on factors such as time step, activation rate, and module dependencies. Secondly, a linear Differential-Algebraic Equation (DAE) solver is harnessed for managing ELN and LSF descriptions. It examines the instantiated primitives in ELN and LSF to deduce the underlying equations, which are subsequently solved to ascertain the system's state at any given simulation time. This solver employs lightweight numerical methods, such as backward Euler and trapezoidal techniques,

in combination with optimization strategies like Lower-Upper (LU) decomposition and Woodbury formulas. These techniques expedite matrix factorization, ensuring a satisfactory level of accuracy while concurrently upholding robust simulation performance. In the end, a synchronization layer utilizes the activation time step of each module, primitive, and cluster to integrate the execution of SystemC-AMS elements into the conventional SystemC simulation flow. Moreover, the synchronization layer maintains the static schedule of a cluster comprising components such as modules or primitives, which has been established by the TDF scheduler. This schedule is retained in the form of a list of pointers pointing to the individual components within the cluster (Figure 2.1).

## 2.3 RISC-V

RISC-V (pronounced "risk-five") was originally designed for education and research of computer architecture, it is an instruction-set architecture (ISA) that avoids excessive architectural specificity customized for a particular microarchitecture style (such as microcoded, in-order, decoupled, or out-of-order) or implementation technology (like full-custom, ASIC, FPGA). Instead, it enables efficient implementation on any of these platforms. RISC-V allows 32-bit, 64-bit and 128-bit address space variants for applications, hardware implementations and operating system kernels. Moreover it was developed to fully support highly-parallel multicore or manycore implementations, including heterogeneous multiprocessors, also supports optional variable-length instructions to expand available instruction encoding space and support dense instruction encoding, in order to improve performance, static code size, and energy efficiency.

The RISC-V ISA is deliberately defined to minimize its focus on implementation details (while still providing commentary on decisions driven by implementation considerations). It should be interpreted as the software-visible interface applicable to a broad range of implementations, rather than a specific hardware design [8].

### 2.3.1 RISC-V Terminology

A RISC-V hardware platform may include multiple RISC-V-compatible processing "cores" alongside other cores that are not compatible, fixed-function accelerators, various physical memory structures, I/O devices, and an interconnect structure for component communication. A core is defined as a component that features an independent instruction fetch unit. A RISC-V-compatible core can potentially support multiple RISC-V-compatible hardware threads, known as "harts", through multi threading. A core might also incorporate specialized instruction-set extensions or an additional coprocessor. The term "coprocessor" is used to describe a unit attached to a core, primarily sequenced by a RISC-V instruction stream,

but possessing its own architectural state, instruction-set extensions, and limited autonomy in relation to the primary instruction stream. The term "accelerator" refers to either a non-programmable fixed-function unit or a core capable of autonomous operation, specialized for particular tasks. In RISC-V systems, it is expected that many programmable accelerators will be RISC-V-based cores with specialized instruction-set extensions and/or customized coprocessors. Notably, a significant category of accelerators comprises I/O accelerators, which handle I/O processing tasks separate from the primary application cores. The organization of a RISC-V hardware platform at the system level can vary widely, ranging from a single-core microcontroller to a cluster of many-core server nodes with thousands of nodes. Even smaller systems-on-a-chip may adopt a hierarchical structure of multicomputers and/or multiprocessors to simplify development or provide secure isolation between subsystems.

### 2.3.2 RISC-V Software Execution

The execution behavior of a RISC-V program is contingent upon the specific execution environment it operates within. An execution environment interface (EEI) for RISC-V delineates several critical aspects: it establishes the program's initial state, specifies the number and types of harts (hardware threads) available in the environment, including the privilege modes supported by these harts, defines the accessibility and attributes of memory and I/O regions, outlines the expected behavior of all valid instructions executed on each hart, and dictates the handling of any interrupts or exceptions that may occur during execution, including calls to the environment itself.

Illustrative examples of EEIs encompass the Linux application binary interface (ABI) and the RISC-V supervisor binary interface (SBI). The implementation of a RISC-V execution environment can take various forms, such as being purely hardware-based, solely software-driven, or a combination of both hardware and software components. For instance, to support functionality not inherently provided by hardware, techniques like opcode traps and software emulation can be employed. Examples of EEI [8]:

- *bare-metal* hardware platforms, where harts are directly instantiated as physical processor threads, and instructions enjoy unrestricted access to the entire physical address space. The hardware platform establishes the execution environment right from the moment of power-on reset.
- RISC-V operating systems create numerous user-level execution environments by allocating user-level harts across the accessible physical processor threads and managing memory access through virtual memory mechanisms.

- RISC-V emulators like Spike, QEMU, or rv8, emulate RISC-V harts on an underlying x86 system. They offer the capability to furnish either a user-level or a supervisor-level execution environment.

The execution environment has the duty of guaranteeing the continuous advancement of each of its harts. However, this responsibility is momentarily paused for a specific hart when it engages in a mechanism explicitly designed to await an event, like the "wait-for-interrupt" instruction.

During the execution every hart has single byte-addressable address space of  $2^{\text{XLENGTH}}$  bytes for all memory accesses. There are *word* defined as 32 bits (4 bytes), *halfword* of 16 bits (2 bytes), *doubleword* of 64 bits (8 bytes) and *quadword* of 128 bits (16 bytes) and them are units used to manage memory space. The memory address space is circular, that means the byte at address  $2^{\text{XLENGTH}-1}$  is adjacent to the byte at address 0. The execution environment plays a central role in configuring how hardware resources are allocated within the address space of a hart. Within a hart's address space, various address ranges can exhibit different characteristics they might be unassigned, host primary memory, or host one or more I/O devices. It is important that, interactions with I/O devices through read and write operations can yield observable consequences, whereas interactions with main memory cannot. While it's possible for the execution environment to designate all elements within a hart's address space as I/O devices, it's typically expected that a portion of it will be explicitly designated as main memory.

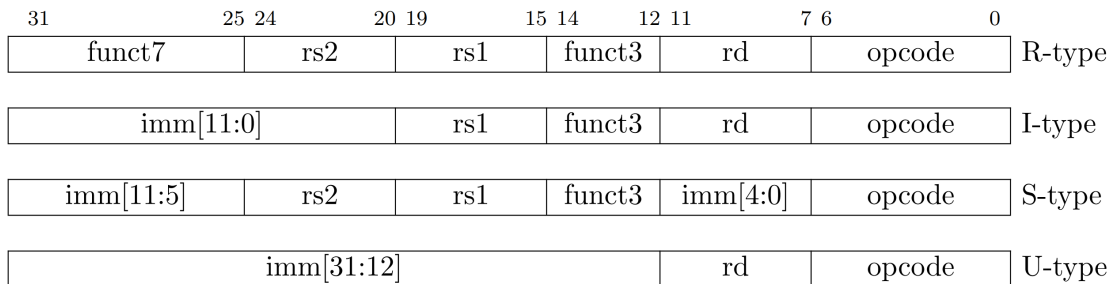
In cases where a RISC-V platform features multiple harts, the address spaces of any two harts can assume different configurations. They may either entirely coincide, diverge completely, or take on a mixed form where some resources are unique, and others are shared, potentially mapped into distinct address ranges.

### 2.3.3 Instruction set architecture (ISA)

The RISC-V ISA comprises a fundamental integer ISA, which is mandatory in every implementation, along with potential extensions to enhance the base functionality. These base integer ISA closely resembles those of the early RISC processors, albeit without branch delay slots and with the inclusion of optional variable-length instruction encodings. The base ISA is intentionally restricted to a minimal set of instructions, strategically chosen to offer a practical foundation for compilers, assemblers, linkers, and operating systems. This design creates a convenient ISA framework and software toolchain "skeleton" that can be customized to construct more specialized processor ISAs.

The RV32I is the base integer ISA for 32-bit systems, it has fixed-length 32-bit instructions that are naturally aligned on 32-bit boundaries. Anyway, RISC-V standard encoding scheme is designed to allow variable-length instructions, in this case each instruction can be any number of 16-bit instruction "parcels" and

parcels are naturally aligned on 16-bit boundaries. RV32I has also 32 "x", general purpose registers each 32 bits wide. Register x0 contains all bits equal to 0 and it is hardwired, registers from x1 to x31 hold values that some instructions can interpret as a collection of boolean values, or as two's complement signed/unsigned binary integers. There is one additional unprivileged register: the program counter (PC), it holds the current instruction's address. Within the base RV32I, there exist four primary instruction formats (R/I/S/U). All of them maintain a fixed length of 32 bits and are required to be positioned on a four-byte boundary in memory. An exception related to instruction address misalignment is triggered when a branch or unconditional jump is taken, and the intended destination address is not aligned to a four-byte boundary. This exception is associated with the branch or jump instruction itself, not with the instruction at the target address. Notably, no instruction-address-misaligned exception is raised when a conditional branch is not taken [8].



**Figure 2.5:** RV32I Instruction format [8]

Below, there is a quick explanation of the instruction format:

- **R-Type:** Instructions that are used for operations involving two registers. These instructions typically perform operations like addition, subtraction, logical operations, and comparisons.
- **I-Type:** These are used for operations that involve an immediate value (an immediate operand) and a register. These instructions can include operations like immediate value loads, immediate value arithmetic, and immediate value logical operations.
- **S-Type:** S-type instructions are used for store operations, where data from a register is stored into memory. These instructions encode the destination address in an immediate value.
- **U-Type:** The last type are used for operations that require a wide immediate value. These instructions include operations like adding an immediate value to a register or setting a register to an immediate value.

## 2.4 PULP & GvSoC

PULP (Parallel Ultra-Low-Power) is an open-source multi-core computing platform resulting from a collaborative effort between ETH Zurich and the University of Bologna. This partnership, initiated in 2013, led to the development of the PULP architecture. The primary objective of the PULP architecture is to cater to IoT end-node applications that require versatile data stream processing from various sensors. These sensors may include accelerometers, low-resolution cameras, microphone arrays, and vital signs monitors. PULP features an advanced microcontroller architecture that enhances its capabilities in several aspects. These enhancements encompass autonomous I/O handling, advanced data preprocessing, support for external interrupts, and the inclusion of a tightly-coupled cluster of processors. This cluster enables the offloading of compute-intensive kernels from the main processor, representing a significant advancement in terms of completeness and complexity.

The PULP architecture [9] is composed of:

- Either the RI5CY core or the Zero-RISCY one as main core (Both based on RISC-V).
- Autonomous Input/Output subsystem (uDMA).
- Memory subsystem.
- Support for Hardware Processing Engines.
- Simple Interrupt controller.

And many more components.

RISCY is a single-issue, in-order core with 4 pipeline stages, exhibiting an IPC (Instructions Per Cycle) close to 1. It offers full support for the base integer instruction set (RV32I), including compressed instructions (RV32C) and the multiplication instruction set extension (RV32M). Additionally, it can be configured to include the single-precision floating-point instruction set extension (RV32F). This core incorporates numerous ISA extensions, encompassing hardware loops, post-incrementing load and store instructions, bit-manipulation operations, MAC (Multiply-Accumulate) operations, support for fixed-point operations, packed-SIMD (Single Instruction, Multiple Data) instructions, and the dot product. Its design is meticulously tailored to enhance energy efficiency, making it particularly well-suited for ultra-low-power signal processing applications.

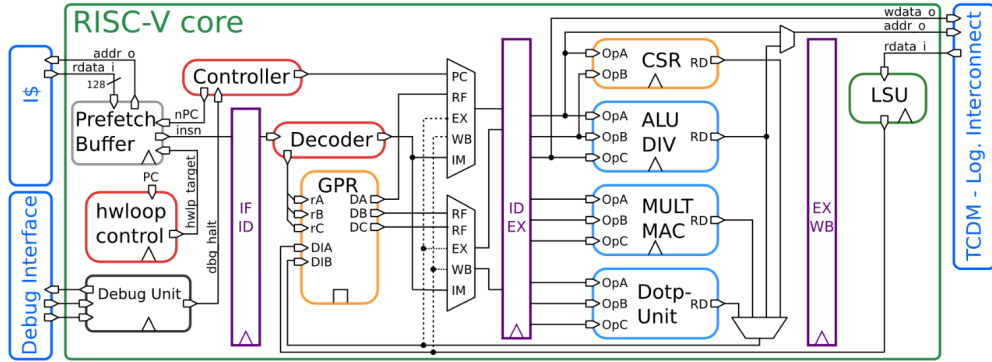


Figure 2.6: RISC-V core overview [10]

Zero-RISCY is a single-issue, in-order core characterized by a well defined 2-stage pipeline. It boasts complete compatibility with the base integer instruction set (RV32I) and the inclusion of compressed instructions (RV32C). Flexibility is a hallmark, as it can be tailored to include the multiplication instruction set extension (RV32M) and the reduced number of registers extension (RV32E). This core has been meticulously crafted to meet the demanding requirements of ultra-low-power and ultra-low-area applications.

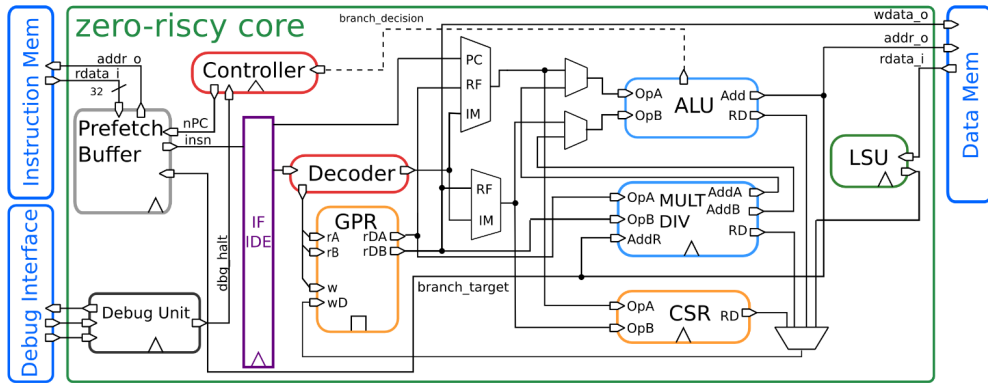


Figure 2.7: Zero-RISCY core overview [10]

### 2.4.1 PULPissimo

PULPissimo is a 32 bit RISC-V single-core System-on-a-Chip equipped with all main components: ROM, RAM, cache, timer, ABP bus, GPIO, I2C, SPI, UART, AXI and DMA. PULPissimo is the upgrade of the first version PULPino system, respect to its predecessor it can be extended to support the multi-core cluster of PULP project. It uses a more complex management memory subsystem, an



I/O subsystem that is autonomous thanks to the uDMA, moreover it has new peripherals (i.e. camera interface) and a new SDK.

PULPissimo provides the flexibility to configure its core architecture during the design phase, offering a choice between RISC-V or zero-riscy cores. Peripheral devices are seamlessly connected to the uDMA (Microcontroller Direct Memory Access), ensuring efficient data transfer to the memory subsystem. The SoC (System on a Chip) also grants access to both JTAG for debugging and the AXI plug for potential extensions such as a multi-core cluster or an accelerator. [10]

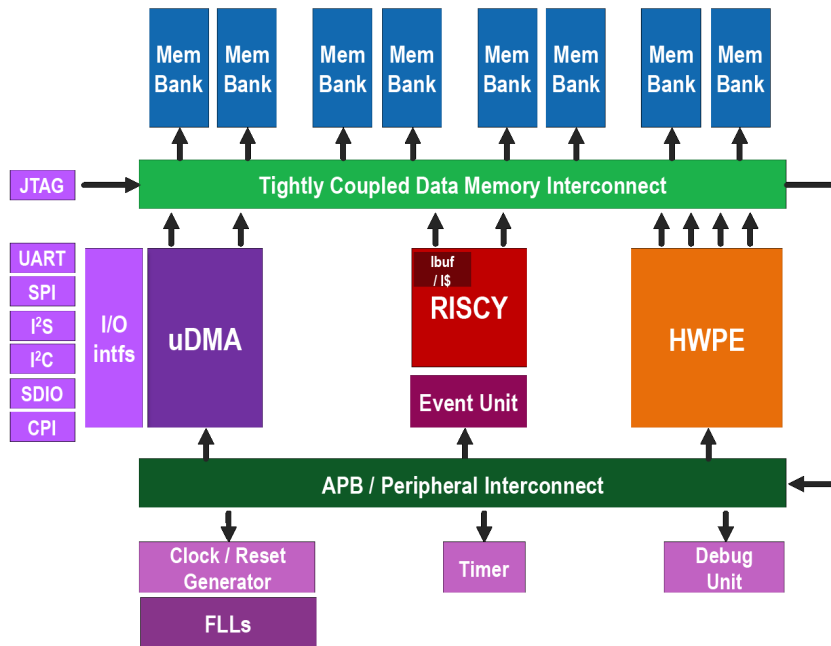


Figure 2.8: PULPissimo overview [10]

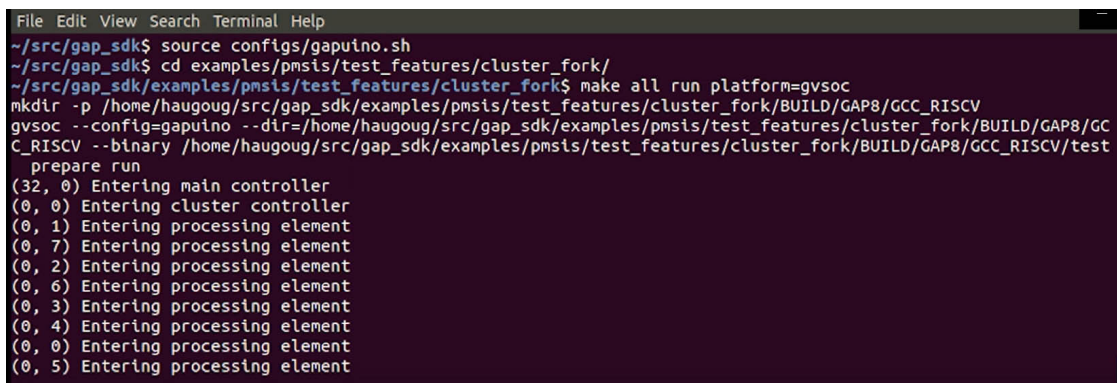
## 2.4.2 GAP9

GAP9 represents the latest offering from Greenwaves Technology [11], distinguished by its adept amalgamation of digital signal processing and cutting-edge neural network algorithms, seamlessly delivered with an unwavering commitment to ultra-low energy consumption and latency. Functionally, GAP9 introduces a comprehensive suite of advanced features, notably harnessing neural networks for audio processing. Furthermore, its versatility extends to encompass multi-sensor analysis, rendering it an ideal choice for battery-powered smart systems. The processor's foundational architecture is rooted in the RISC-V Instruction Set Architecture, with all ten cores tailored to this framework and augmented with bespoke instructions seamlessly

integrated into the GAP toolchain. This configuration empowers the compute cluster with the flexibility to execute a wide spectrum of tasks, spanning from neural network processing to digital signal processing, all while maintaining an exceptional standard of energy efficiency. GAP9's hierarchical, demand-driven architecture stands as an exemplar of adaptability and innovation, ideally poised to shape the trajectory of battery-powered smart sensors. In its capacity as a development platform, it serves as a potent catalyst, streamlining the development process and furnishing a robust foundation for the conception and realization of pioneering solutions within the continually evolving landscape of intelligent devices.

### 2.4.3 GvSoC

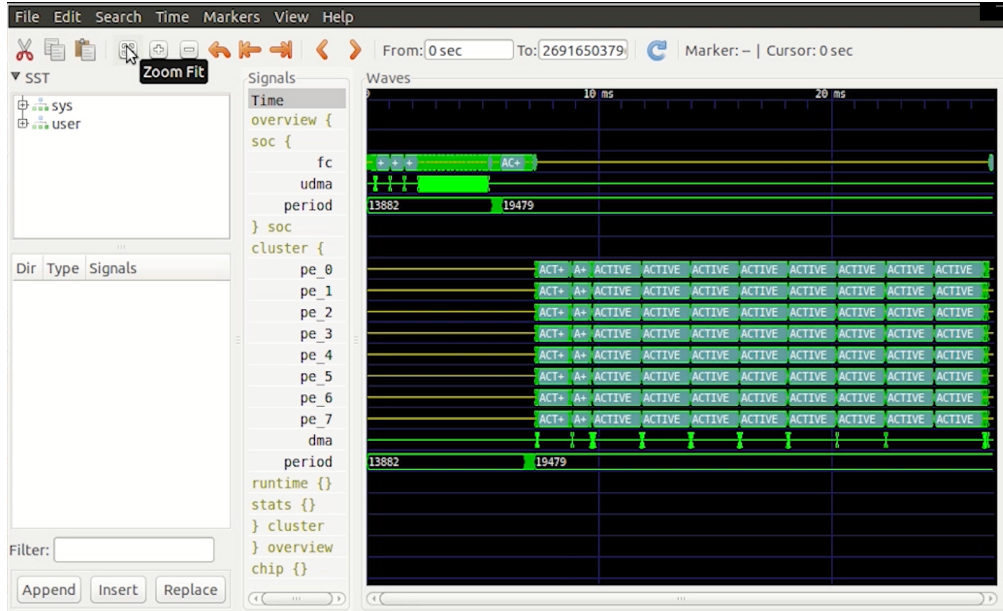
For this thesis, GvSoC has been selected as a high-level simulator. GvSoC is the official simulator available at the GvSoC Official Website, allowing programmers to test GAP applications without the necessity of having the physical chip. It is based on the PULP platform, specifically simulating the environment of GAP9. GvSoC is functionally equivalent to the real chip, which means that compiled code runs in the same way on both the chip and in the simulator. GvSoC ensures timing models that accurately report performance within an error rate of less than 20%. It can simulate up to 20 million instructions per second, which is approximately 10 times less than the actual chip's performance. The simulator is strongly based on C/C++, and the official GitHub repository is available at GvSoC Repository. It is also possible to simulate common devices that can be connected to the real board, such as cameras, microphones, and flash memory. This capability allows for the simulation of a full application rather than just basic code. To use GvSoC with the GAP SDK, it is essential to build the SDK, configure the board (in this case, "GAP9\_EVK\_AUDIO"), compile the application's code, and run it through the simulator.



```
File Edit View Search Terminal Help
~/src/gap_sdk$ source configs/gapuino.sh
~/src/gap_sdk$ cd examples/pmsis/test_features/cluster_fork/
~/src/gap_sdk/examples/pmsis/test_features/cluster_fork$ make all run platform=gvsoc
mkdir -p /home/haugoug/src/gap_sdk/examples/pmsis/test_features/cluster_fork/BUILD/GAP8/GCC_RISCV
gvsoc --config=gapuino --dir=/home/haugoug/src/gap_sdk/examples/pmsis/test_features/cluster_fork/BUILD/GAP8/GC
C_RISCV --binary /home/haugoug/src/gap_sdk/examples/pmsis/test_features/cluster_fork/BUILD/GAP8/GCC_RISCV/test
prepare run
(32, 0) Entering main controller
(0, 0) Entering cluster controller
(0, 1) Entering processing element
(0, 7) Entering processing element
(0, 2) Entering processing element
(0, 6) Entering processing element
(0, 3) Entering processing element
(0, 4) Entering processing element
(0, 0) Entering processing element
(0, 5) Entering processing element
```

Figure 2.9: Example of GvSoC simulation [12]

For the simulations, GvSoC also implements an automatic value change dump (VCD) trace creator, enabling a highly precise visualization of the board's behavior throughout the entire simulation.



**Figure 2.10:** Example of GvSoC VCD traces [12]

# Chapter 3

## Related Works

In this chapter, the current state-of-the-art in hardware simulation is explored. The chapter is organized into multiple sections: first, methods and design approaches for modeling and representing Cyber-Physical Electrical Energy systems (CPEES) in SystemC-AMS are analyzed. Second, a benchmarking comparison between Simulink and SystemC-AMS is presented, highlighting the advantages of the latter. Finally, is presented a paper that describes how to model extra-functional properties in hardware simulation.

These sections provide insights into the importance of having a single program that offers high-level simulations combined with extra-functional simulations. Moreover, they elucidate the real advantages of the SystemC frameworks compared to existing solutions, providing a solid rationale for their choice in this thesis.

Briefly, the studies by [13], [7], and [14] from Politecnico of Torino are examined, all of which focus on the simulation of an electrical energy system (EES). The simulation time of the SystemC simulator is compared to Simulink, with an emphasis on the speedup achieved by the former and the average error between the two simulations. Additionally, the advantages of having a unified simulation environment that enhances model accuracy are discussed.

All of these works aim to address the same issue, which is to propose a new methodology and approach for integrating various domains. The objective is to attain accuracy, flexibility, modularity, and simulation speed within a single application.

The final section delves into [15], elucidating its approach to extra-functional properties. This provides insight into contemporary methodologies, with a particular emphasis on the layered approach for property modeling.

### 3.1 Modeling Cyber-Physical Electrical Energy Systems

One of the main contributions of the aforementioned works is their modeling processes, which involve transforming real systems into simpler module compliant with SystemC and SystemC-AMS syntax. Each work handles a different system, differentiating one solution from the others. However, every solution faithfully adheres to the general model-based paradigm, utilizing the built-in models provided. Thanks to the high flexibility of SystemC, the representation models are close enough to allow for a very low approximation error. A representative example of SystemC modeling is present in [13]. Starting from a CPEES composed by: a wind turbine, a photovoltaic array, a battery pack, a grid-interface, a DC bus and various AC loads. The paper presents a versatile CPPES design approach, where SystemC is mixed with its framework SystemC-AMS, avoiding the integration of external tools and allowing the application of the same methodology to a huge range of components. Still in [13] the possibility of simulate heterogeneous environment in the same program, is concretized in the simulation of AC element and DC element, moreover them are also capable of communicating each other with a special component called "bridge" that manages both AC and DC measurements. Each component is modelled with the appropriate AMS class, taking in example the battery and the wind turbine, the first one is modeled with an ELN module allowing to simulate very well the electrical circuit behind the battery chosen, instead to reproduce the behavior of the turbine, including the gear box and the generator, is used an LSF module with purpose of compute the algebraic formula used for the wind conversion.

Other approaches mentioned are: Hardware-in-the-loop that mix real devices with simulated models through integration of power devices like inverters, with the purpose of test the technology inside a controlled environment. Respect to SystemC these approaches reach an higher accuracy, but their application are restricted to small scale CPEES due to the involving of real hardware. Equation-based approaches, are based on decomposition, briefly, them model all components inside a system into elementary models that integrate basic physics equations. This kind of approaches are limiting an effective high-accuracy modeling and the resulting fidelity to the real CPEES.

In the end [7] and [14] the target is almost the same of above exception for the CPEES under study, also in that works is possible to see the attention put in the design phase, meticulously choosing the right AMS model to better represent the components. In [14] the emphasis is more on the comparison respect to Simulink with the same CPEES. In [7], instead, are better show some issues and limitations about the internal scheduler of SystemC-AMS framework.

## 3.2 SystemC and Simulink Comparisons

Cutting-edge programs offer an innovative approach to realize and simulate a Cyber-Physical Energy-Efficient System (CPEES). This approach involves dividing the simulation into multiple layers, each dedicated to a specific property of the system. This innovative technique is referred to as the Co-simulation approach. With this approach, each heterogeneous aspect of the system is assigned to a specific program. For instance, in the work cited in [15], specifically in section VII where the results are validated, recreating a similar simulation to that achieved with SystemC required the use of three distinct tools, including Simulink and HotSpot. This highlights the need to reconsider traditional approaches, as they often demand a significantly larger amount of effort to achieve comparable results.

Returning to the Co-simulation approach, it's crucial to consider the challenges posed by employing multiple tools. The design of the system can become more complex, leading to a substantial overhead in managing multiple simulators. One significant issue involves the synchronization of multiple timestamps, with each tool having its own timestamp and event queue. This complexity is addressed by SystemC, which aims to simplify the process through a unified application programming language and unified libraries.

Furthermore, using multiple tools requires in-depth knowledge of each of them, as well as the frameworks that facilitate communication among them. This increases the workload for designers. However, there are alternative solutions within the environment. One notable example is Modelica, with its equation-based approaches, which are well-suited for modeling the physical aspects of the system but may face limitations when modeling the "cyber" components. SystemC remains the preferred choice for this type of modeling due to its C-like nature, making it more accessible to programmers. Additionally, SystemC's AMS (Analog and Mixed-Signal) framework contains essential classes that support the modeling of the most critical system components.

As demonstrated in [15], [13], and [14], there are tables that illustrate the substantial advantages of this approach in terms of time efficiency compared to Simulink. These advantages are underscored by the remarkably low average error, often orders of magnitude lower than a percentage unit, making it a highly promising avenue for simulation and modeling in the field of Cyber-Physical Energy-Efficient Systems.

### 3.2.1 SystemC Cons

After mentioning the advantages of SystemC is important to understand also the issues which it is affected. Main issues come from modeling non-standard systems, according with [7], that proposes a detailed exposition of every issue, them can be

collected in five classes:

- Issues related to the scheduler, in particular they rise when tight constraints are applied to the schedulability of system.
- Issues related to the linear DAE solver, its light-weighted nature determines a compromise between accuracy and simulation speed, resulting in certain cases not enough.
- MoC-related issues, tightly connected to every model present in SystemC-AMS and specific for every use case. (Degree of approximation between the real component and the one simulated)
- Kernel issues, this kind of issues are present because of the inefficient memory usage of SystemC-AMS.
- Issues of extensions, them are the minor ones and are related to some features would ease the work of the designer.

As an illustration, is taken from [7] a kernel issue example. In this scenario multiple modules run at different time scales, i.e. from 100ns to 1s, the activation list of the scheduler can be very long because of the multiple scheduling of the smallest time scale. Taking three different modules with: 1s, 1ms and 100ns the last one will be scheduled ten thousand times and the second one will be scheduled one thousand times. This can leads to a crash of the synchronization layer causing a crash in the whole simulation.

### **3.3 Methodologies for Extra-Functional Properties**

This last section is dedicated to [15] that is a work which this thesis strongly depends. In the article are described the main issues this thesis try to cover, starting from the implementation of extra-functional properties in a simulator through the division in layers of sensor's characteristics. For the purpose of the thesis is taken the idea of this paper and following is extended with the interaction with a complete high-level simulator.

In [15] two features are important to distinguish the proposed methodology. First, the methodology adopts a layered approach, where the simulated system is structured into different views or layers, each dedicated to a specific property. This stratified design permits the independent handling of information related to each property, while also enabling the simultaneous simulation of multiple layers within a single simulator instance, thereby capturing the interplay among these

layers efficiently. This layered paradigm is further complemented by the use of a unified functional language across all layers, simplifying its adoption by functional designers who are already familiar with the language and alleviating the need for in-depth knowledge of the underlying physics or property-specific tools.

Second, the methodology employs a bus-centric modular architecture for each layer, wherein each layer operates on its dedicated bus structure. This architectural choice aligns with the legacy of the functional simulation layer, where buses mimic the logical organization of blocks and facilitate the exchange of information between components within a layer. Importantly, this architecture remains highly scalable, accommodating the addition of components at any layer without necessitating complex interface adjustments.

Each layer is characterized by four key attributes, including layer-specific signals, inter-layer signals, the role of the bus and layer-specific data or information. Signals are central to understanding property behavior, while inter-layer signals enable real-time interaction between properties. The role of the bus defines the simulation semantics of each layer, and layer-specific data complements the simulation process with essential information not directly related to semantics. By integrating these characteristics, the methodology offers a comprehensive analysis of electronic systems, encompassing power, temperature, reliability, and the potential inclusion of other essential attributes.



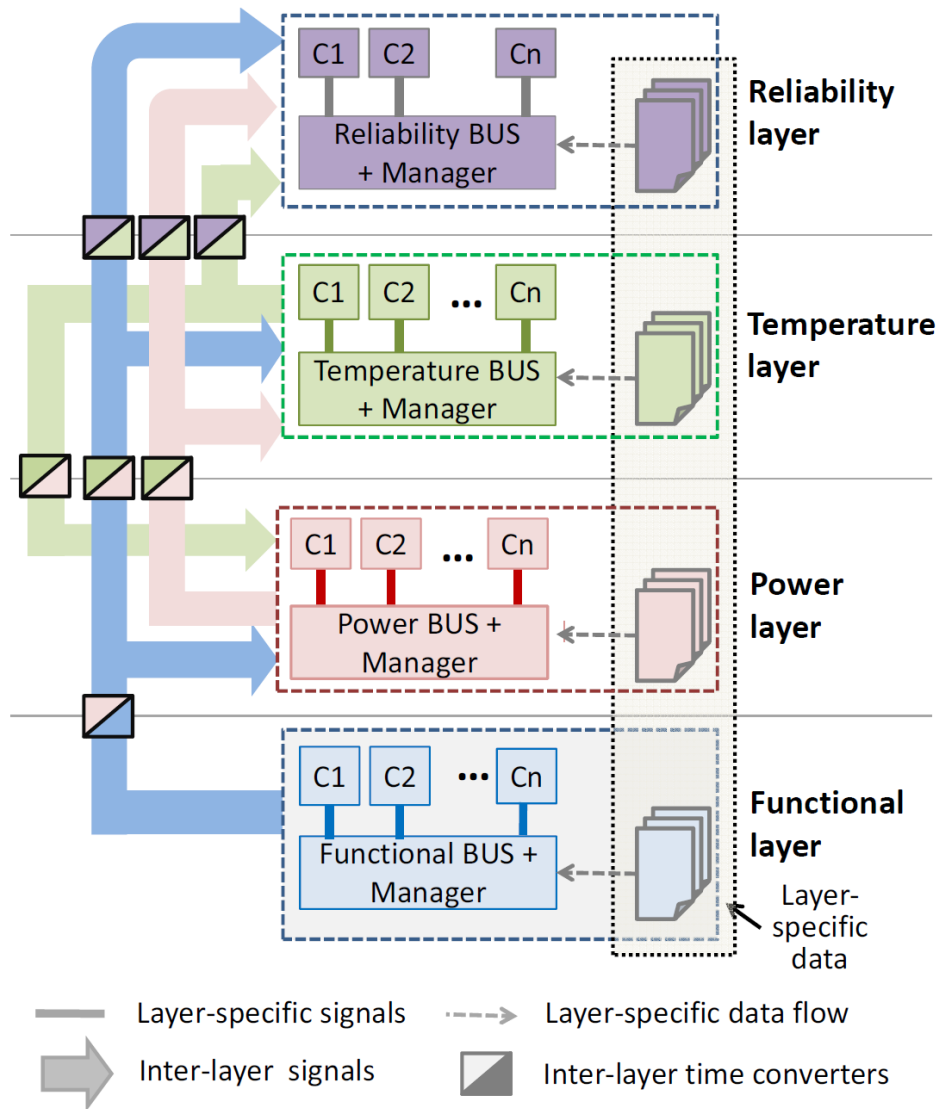


Figure 3.1: Example of Layered structure for extra-functional property [15]

# Chapter 4

## Methods

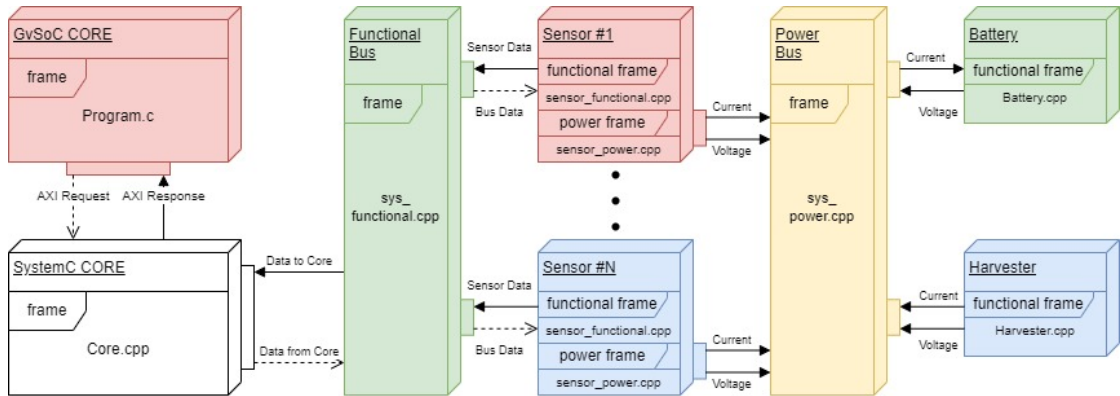
This chapter provides a detailed description of the work undertaken in this thesis, with a particular focus on the actual implementation of the simulator. It begins in Section 4.1 by introducing the core concept, explaining the underlying idea and how it is constructed. Subsequently, the chapter delves into the realization of the core structures, placing emphasis on two critical aspects: the internal structure of each component and the synchronization between components of different types. Once the standalone system is developed, the attention shifts from the simulator's internal workings to its scalability. In the section, 4.2, the chapter explains how two simulators communicate. In this case, the SystemC simulator manages all the physical aspects of the system under test, from the sensor to bus communication and through to the core. Conversely, GvSoC is responsible for controlling the entire program flow written in the system's RAM, including peripheral activations and memory access. SystemC represents the extra-functional simulator in parallel with GvSoC, which represents the functional simulator. Section 4.3 presents the entire process behind the automation of the simulator, ensuring the flexibility to modify the system's configuration on-the-fly. Specifically, it allows for the modification of the type and quantity of specific components like batteries, harvesters, sensors and actuators. This adaptability is facilitated through a set of scripts created using the powerful Python programming language, particularly leveraging the Jinja framework [16], and a JSON configuration file that can be adjusted as needed. Throughout the chapter, code examples, design models, and program screenshots are provided to enhance understanding and clarity of the entire project.

### 4.1 Extra-Functional Simulator

First, the simulator is constructed following a scheme adaptable to all extra-functional properties. The key aspect is to represent a property, define its purpose

inside the system, and determine how it influences the simulation and propagates through the components. The approach followed is similar to the one presented in Section 3.3. The following sections can serve as general guidelines for any kind of extra property. For the goal of this thesis, power consumption is chosen as the property to be represented. Starting with a board equipped with at least a power source, power storage, and some sensors, all of these components are connected to the GAP9 processor through memory addresses accessible by the core, enabling simple read and write C-like operations. The board’s purpose is to collect data from these sensors within a defined timeframe and periodically transmit this data over the network using the transmitter. The simulation must accurately track all information sent and compute the maximum system lifetime, accounting for different battery representation models and a solar panel that receives irradiance traces. These operations are intercepted by SystemC, which has instances of these sensors, simulates their behavior to obtain meaningful data, and computes power consumption.

The information exchanged between every component can be heterogeneous and can represent various kinds of data. As mentioned earlier, power is selected as the property to represent in this case, with quantities such as voltage and current being relevant. Thus, the developed simulator is designed as a power simulator. The power simulator is designed with two buses: one for functional communication `functional_bus` and one for managing power-related aspects `power_bus`. There is a SystemC core unit that acts as the master of the simulation and serves as the bridge between SystemC and GvSoC. Additionally, there are peripheral components such as the battery, harvester, and sensors, all equipped with appropriate converters and correctly connected to their respective buses. Figure 4.1 provides a conceptual design of the simulator.



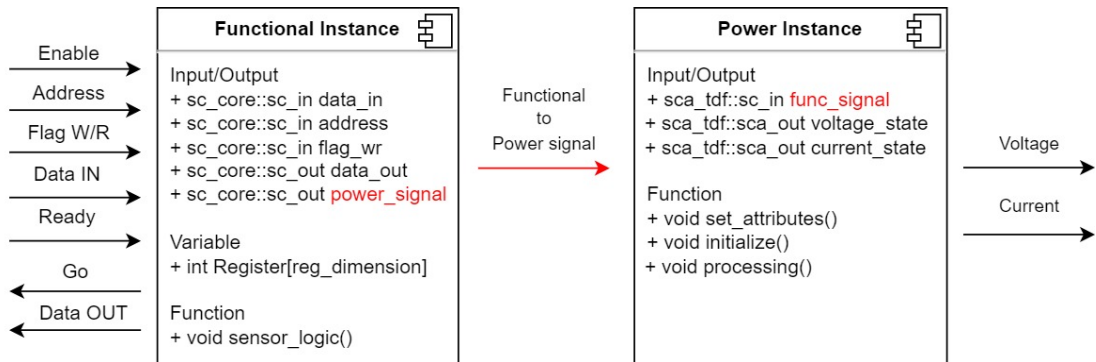
**Figure 4.1:** Power simulator conceptual

In the following sections, each component related to the power simulator will be

analyzed in detail. These components serve as abstract representations of real-world components. However, it is also possible to define custom components with their own specific behaviors.

### 4.1.1 Sensors

The first component modeled is the sensor, which is developed with two separate instances. The first one is the functional instance, responsible for managing requests coming from the core through the functional bus. It also defines all the parameters of the sensor, such as the dimensions of the sensor's internal registers. On the other hand, the power instance controls the state of the sensor and exposes its current and voltage to the power bus through a shared signal between the two instances.



**Figure 4.2:** Sensor structure

The functional part retrieves details of the request, such as the address, read/write flag, and data from the core. Its logic consists of a structure of if statements where it queries an array of elements (simulated internal registers of the sensor), sets the correct power state, and waits. After this step, the power instance is activated, which updates the power information for a certain amount of time defined by the sensor's configuration. This is done to simulate the computational time that the sensor takes in real life to analyze the data and update the corresponding register. After that, the power instance sets an IDLE state and gives control back to the functional instance, which can then prepare Data OUT and GO signals to reply to the core. The functional part uses a standard SystemC class, SC\_MODULE, which is well-suited for all kinds of management operations. The power part is embedded in an SCA\_TDF\_MODULE, which belongs to the SystemC-AMS environment. The choice of a TDF module is important because one goal of the simulator is to keep track of power over time, and the TDF structure is perfect for this kind of operation.

## Functional Instance

In detail, the functional part consists of two files: *.h* and *.cpp*. The *.h* file encompasses all the interfaces employed by the sensor for communication. Additionally, this file contains declarations for the sensitivity list and the internal register vector, essential components that facilitate the sensor's operation and interaction within the system.

```

1  /* Sensor_Functional.h file */
2  //Input Port
3  sc_core::sc_in <bool> enable;
4  sc_core::sc_in <int> address;
5  sc_core::sc_in <int> data_in;
6  sc_core::sc_in <bool> flag_wr;
7  sc_core::sc_in <bool> ready;
8  //Output Port
9  sc_core::sc_out <int> data_out;
10 sc_core::sc_out <bool> go;
11 //Power Port
12 sc_core::sc_out <int> power_signal;
13 ...
14 //Declaration of function and its sensitivity list
15 SC_THREAD(sensor_logic);
16 sensitive << ready;
17 ...
18 //Register Map
19 private:
20 int Register[AIR_REG_DIMENSION];

```

All the signals are utilized by the *.cpp* file to execute the correct procedures. Two special signals are introduced here: the first one is the `power_signal`, which is responsible for enabling communication between the power and functional components to ensure coherence between actions performed and their power consumption. Thanks to this signal, it is possible to observe, in the reports, the correspondence between functional and power statuses at a given time. For example, if the sensor is reading in the functional report, the functional read will be reported, and in the power report, the power status of the read will be indicated. The second signal is `ready`, which is used to coordinate requests. Without this signal, when the functional bus sets the address, data, and flag, the sensor can be activated at any moment. However, with the `ready` signal, the bus can wake up the sensor only when it has finished preparing the request, ensuring data coherence and coordination.

```
1  /* Sensor_Functional.cpp file */
2  while (true) {
3      if( enable.read() == true ){
4          if(ready.read() == true){
5              if( flag_wr.read() == true ){
6                  data_out.write(Register[address.read()]);
7                  power_signal.write(1);
8                  wait(AIR_QUALITY_SENSOR_T_ON, SIM_RESOLUTION);
9                  power_signal.write(3);
10                 go.write(true);
11             } else {
12                 //Write Operations
13                 Register[address.read()] = data_in.read();
14                 data_out.write(data_in.read());
15                 power_signal.write(2);
16                 wait(AIR_QUALITY_SENSOR_T_ON, SIM_RESOLUTION);
17                 power_signal.write(3);
18                 go.write(true);
19             }
20             ...
21         wait();
22     }
```

The *.cpp* file houses the core functionality of the sensor, it is only shown the main while cycle that illustrates the sequence of actions undertaken each time a request is directed towards the sensor.

First and foremost, a series of sequential checks is initiated: the enable signal must be active, and the ready signal from the functional bus must be asserted. Subsequently, the type of operation being requested is determined. In the case of a Read operation, the sensor retrieves the value stored at the designated address, sets the power\_signal to the corresponding power status, and introduces a simulated computation delay using the wait() instruction. Following this, the power\_signal is reverted to the idle state, and ultimately, the response is transmitted back to the functional bus. Conversely, for Write operations, the process is analogous, except that data is written to the specified address instead of being read.

## Power Instance

The power part is, also, represented by a *.h* and a *.cpp* file. This component is notably less complex compared to its functional counterpart, primarily because a significant portion of the management is handled by the SystemC-AMS kernel. For the purpose of this project, the power instance is modeled as a power state machine, which switches its status every time an operation is performed. There are as many statuses as functional operations.

Furthermore, SystemC allows for more complex models, granting designers a higher degree of freedom in representing power, for example. To delve into the specifics, this segment focuses on configuring the voltage and current parameters for the sensor, corresponding to various operational states. Once these parameters are established, they are subsequently directed towards the load converter and, ultimately, transmitted to the power bus. This streamlined process ensures that the power requirements of the sensor align with its designated state of operation.

```
1  /* Sensor_Power.h file */
2  //Data from Functional Instance
3  sca_tdf::sc_in <int> func_signal;
4  //Data to Power Bus
5  sca_tdf::sca_out <double> voltage_state;
6  sca_tdf::sca_out <double> current_state;
```

In this case, the interfaces are restricted to a single input port and two output ports. Once more, the `func_signal` plays a pivotal role, determining the power state to be represented. The two output ports are responsible for transmitting the corresponding values associated with that state.

```
1  /* Sensor_Power.cpp file */
2  void air_quality_sensor_power::processing()
3  {
4      if(func_signal.read() == 1){
5          //std::cout << "Air quality in ON state READ MODE" << std::endl;
6          voltage_state.write(AIR_QUALITY_SENSOR_V_ON_READ);
7          current_state.write(AIR_QUALITY_SENSOR_I_ON_READ);
8          return;
9      }
10     ...
11 }
```

The `.cpp` file comprises a series of conditional if statements, each of which corresponds to a potential power state. In the power section, it continuously examines the `'func_signal'` until it discovers a matching condition. Once a match is found, it proceeds to configure both the voltage and current parameters, effectively completing the power management for that specific state.

Additionally, a comment line has been included within each if branch to serve two functions. First, it provides clarity regarding the represented power state within a particular branch, aiding in code comprehension. Second, it serves as a helpful

reference in case there's a need to print messages during the simulation whenever a power state transition occurs. In the event that the 'func\_signal' fails to match any of the if conditions, an error is raised to signify an unexpected condition.

### 4.1.2 Functional Bus

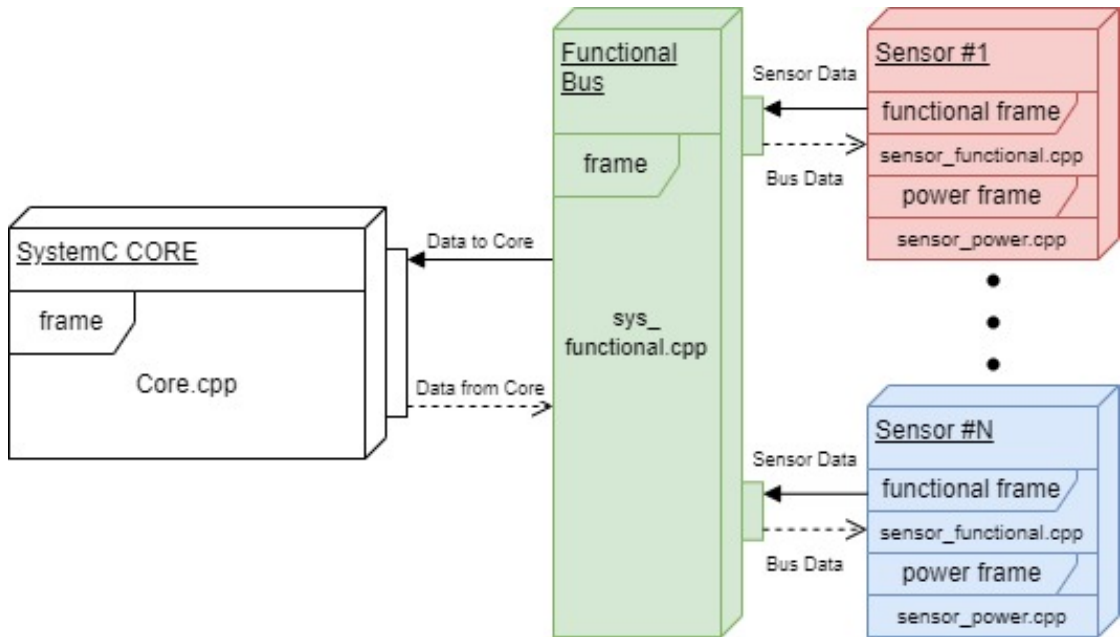
The functional bus is the component that forwards the requests coming from the core to the right sensor. In fact, the core sends all the requests through only one channel connected to the functional bus, which forwards the request to the correct sensor. The functional bus exposes a vector of channels where every sensor can get connected. Each channel is composed of the following elements:

- `address_out_S`: Address that is forwarded to a specific sensor.
- `data_out_S`: Data forwarded to a specific sensor.
- `flag_out_S`: Flag that indicates if the request is a Read/Write for a certain sensor.
- `Ready_S`: Signal used to communicate that the other parameters are correctly configured.
- `data_in_S`: This signal contains the information coming from the sensor after its computation.
- `Go_S`: The sensor raises this signal when it has completed all its internal procedures and replied correctly.

This channel structure is designed to facilitate the exchange of information in a clear manner, ensuring code readability during the analysis of the simulation flow. Moreover, the Ready and Go signals are introduced to enhance reliability among requests, preventing the failure of the grant/reply mechanism on both the sensor-to-bus and bus-to-master sides. There is no standard for the bus request, which means that channels can be easily modified to adopt any kind of standards such as I2C or CAN.

With reference to Figure 4.3, each pair of arrows represents a mentioned channel. The bus reads the incoming request, particularly the address, which is compared with all the addresses available in memory space using a C language if-statement that utilizes information contained in the global parameters file. When the address corresponds to a certain memory space, the functional bus extracts the sub-address, removing the sensor's base address, which is a specific register address of the selected sensor. At this point, the bus prepares all the fields of the request by reading the `flag_in_M` and `data_in_M`. These signals specify the nature of the





**Figure 4.3:** Functional diagram

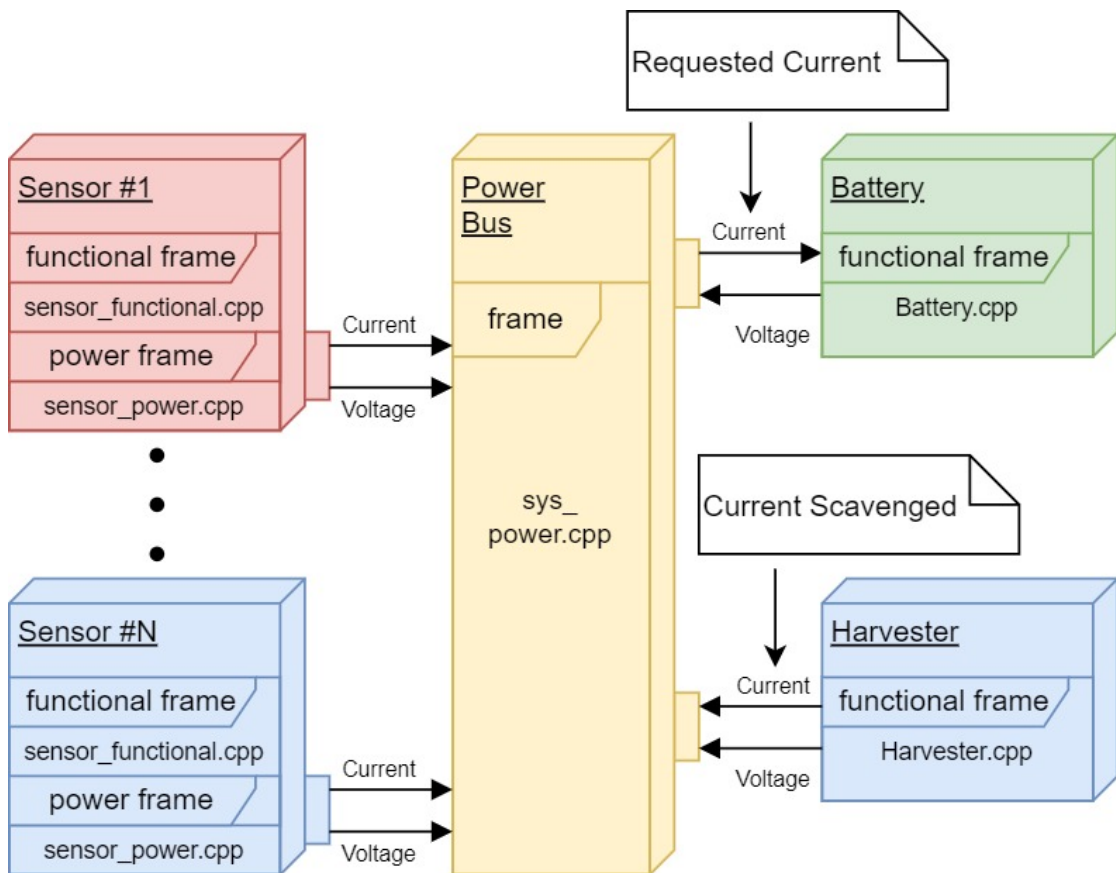
request and its data, if any. The bus forwards the request and raises a **READY** signal for the sensor, ensuring that all other signals are correctly set. After that, the bus enters in a waiting state until the sensor completes its internal procedures. When the sensor raises its **GO** signal, the functional bus wakes up and prepares the answer for the core, sending commands to the slave to return to the **IDLE** state, and waits for another signal from the sensor. In the end, when everything is done, the bus forwards the data to the core and raises a **GO** signal. Section 4.1.6 provides a detailed explanation of this flow.

In SystemC, the bus is modeled with an **SC\_MODULE** because its purpose is only to manage data without the necessity of operating in the time domain. It is composed of different functions, each with a part of the above procedures:

- **processing\_data()**: This function opens the request, reads it, and forwards it to the sensor.
- **response()**: The response is the first function called when the sensor raises the **GO** signal and is responsible for preparing the data for the core.
- **set\_GO()**: This is the last function called, which sets the **GO** signal for the core, indicating that all operations have been executed correctly.
- **set\_slave()**: It prepares the channel of the sensor with the correct values.

### 4.1.3 Power Bus

In the same way of functional bus, there is the power bus. However its function is completely different from the previous one, in fact, it manages all the aspects concerned power management neglecting all the data behind every request. Power bus exposes like functional an array of channels where, this time, sensor's power instances can connect themselves. In this case, the information shared are the voltage state and the current state. Since the simulator supports battery and harvester, the bus also takes and gets information from those components. In the figure 4.4 an overview of connections is provided.



**Figure 4.4:** Power diagram

As it is possible to see in the figure 4.4 the bus gets from all component, except for the battery, current information. Since there are converters in between every component and the bus, all the currents must be at the same voltage, so the bus is in charge to sum all currents coming from sensors and subtract, eventually, the current scavenged by the solar panel. In the end the result is the current demanded

to the battery in a certain time step, the formula for the requested current is:

$$Current_{request} = \sum_{i=1}^N Current_{sensor-i} - Current_{scavenged}$$

$$\begin{cases} Current_{request} \geq 0 & \text{Discharging} \\ Current_{request} < 0 & \text{Charging} \end{cases}$$

The power bus is represented in the model as an SCA\_TDF\_MODULE, which is consistent with the modeling approach used for the power components of sensors. Since the power bus deals with dynamic measures that change over time, it is essential to define its behavior within the time environment. One notable feature of the power bus is its capability to measure the current demanded by the battery in a step-by-step manner, allowing for precise monitoring. Additionally, it can separately track the current requested by individual sensors, ensuring accuracy down to the level of each sensor's requirements. Furthermore, the power bus can also monitor and track the current produced by the solar panel as part of its functionality.

#### 4.1.4 Battery, harvester and converters

In this section there are the general descriptions of all the side components, because of the possibility of customization certain components can be different from one configuration to another. A good example is the battery that can have circuit model or more high-level ones such as Peukert representation. What is important in this case are the interfaces exposed, ensuring, a general compatibility with other components. Battery must have:

- **Voltage OUT port** Used to keep track of the voltage state.
- **Current IN port** Used by the power bus to request the current.
- **State of Charge (SOC) OUT port** Used to keep track of the SOC of the battery.

About what concerns the harvesters the reasoning is the same of the battery, no matter what is the physical phenomena used to scavenge the energy or the internal process used, what is important for the simulator is the presence of the following interfaces:

- **Voltage OUT port** Used to keep track of the voltage state and make conversions where it is needed.

- **Current OUT port** Used by the power bus to compute the requested current.

The last component is the converter, it manages incoming voltage and current and make them compliant with the power bus. This kind of component can easily splitted in multiple sub categories. In the power simulator are present three different type of converters, starting from the simplest one, the load converter is a converter placed in between sensors and bus, specifically between each power instance of every sensor and the power bus. It is designed with a static behavior and scales the power of sensor by a constant value that can be set at start of the simulation. the result is the current sent to the power bus adjusted by a efficiency factor  $K < 1$ :

$$Current_{Bus} = \frac{K * (Current_{sensor} * Voltage_{sensor})}{Voltage_{Bus}} \quad (4.1)$$

However, it is possible to define more complex converters. In the simulator the harvester used is a solar panel, for example. This just one of the possible harvesters that can be modeled with SystemC. It has a converter where the efficiency strongly depends on the voltage of the panel. In particular the converter aims to scavenge the current near to the maximum power point, ensuring the best efficiency every time. Therefore, the formula is much more complex compared to the previous one. In the end, what is important from the simulator's point of view is the forwarding of data without manipulation of data types.

Before converter	After converter	
SC_IN <double>	SC_IN <double>	Ok!
SC_IN <double>	SC_OUT <int>	Not Ok!

**Table 4.1:** Examples of converter, the data integrity must be ensured

### 4.1.5 Core

The SystemC core is the entity that interacts with the functional simulator, here in after GvSoC, enabling the forwarding of AXI requests from one simulator to another. This module is unique in that it inherits from two different classes: the standard SC\_MODULE and gv::Io\_user. The former is mandatory for coordinating all the components within the SystemC simulator, ensuring the correct request is sent to the functional bus and retrieving the correct responses. The latter is necessary for intercepting requests coming from the GvSoC simulation. Additionally, the core is responsible for controlling both simulators, ensuring temporal alignment between the counters.

Regarding GvSoC management, the core is equipped with three methods: access(), grant(), and reply(). The most crucial of these is access(), as it is invoked

when a GvSoC's side access attempt aims to reach an external service, in this case, SystemC. The other two methods, instead, are used when an external service attempts to query the GvSoC.

In terms of query management, the core utilizes four types of information to categorize the requests:

- **Type** (Read/Write)
- **Address**
- **Size**
- **Data**

Each time it's necessary to send a request, GvSoC creates a pointer to a custom C struct of type `gv::Io_request`, and this pointer contains the aforementioned data.

The coordination between SystemC and GvSoC is accomplished with a while-statement. First, the next event in GvSoC is detected, and after its execution, the SystemC timestamp is updated accordingly. A more detailed view of GvSoC integration is provided in 4.2.

#### 4.1.6 Acknowledgement protocol

This subsection has been added to clarify the flow of a request within the SystemC simulator, ensuring coherence and data updating. To begin, the core is triggered by its `access()` method. Inside this method, the necessary signals for the functional bus: Type, Address, and Data are set up. Additionally, a signal named 'Ready' is established, which the functional bus uses to verify the availability and correctness of the other fields. When the ready signal is set to 'True,' the functional bus is invoked. It reads the address and compares it with all the known sensors. If there is a match between the requested address and a known address, the functional bus forwards the information, splitting the base address from the register address, to the sensor. It also raises a 'Ready\_S' signal used by the selected sensor to awaken. If no match is found, an error is raised, indicating that the requested sensor is not recognized. The functional bus keeps track of the selected sensor throughout the process.

The sensor wakes up when it recognizes the 'Ready\_S' signal from the bus. It begins by reading the information and performing the computation. The first step is to prepare the data by reading/writing the corresponding internal register. After that, the corresponding power state of the sensor is activated through the power signal. A SystemC `wait()` is performed to simulate the power consumption for that particular operation. Subsequently, the information is sent to the functional

bus, and a 'GO\_S' signal is raised to acknowledge the functional bus. Finally, the sensor performs a second wait().

At this point, the functional bus receives the response from the sensor and executes the response() method. This method forwards the data to the core and sets the 'Ready\_S' signal to 'False' and the 'GO\_M' signal to 'True.' These actions are used to reset the sensor signals. In fact, the sensor resets its 'GO' signal to 'False' and 'GO\_M' wakes up the core. At this point, the core can send the information received from the functional bus to GvSoC.

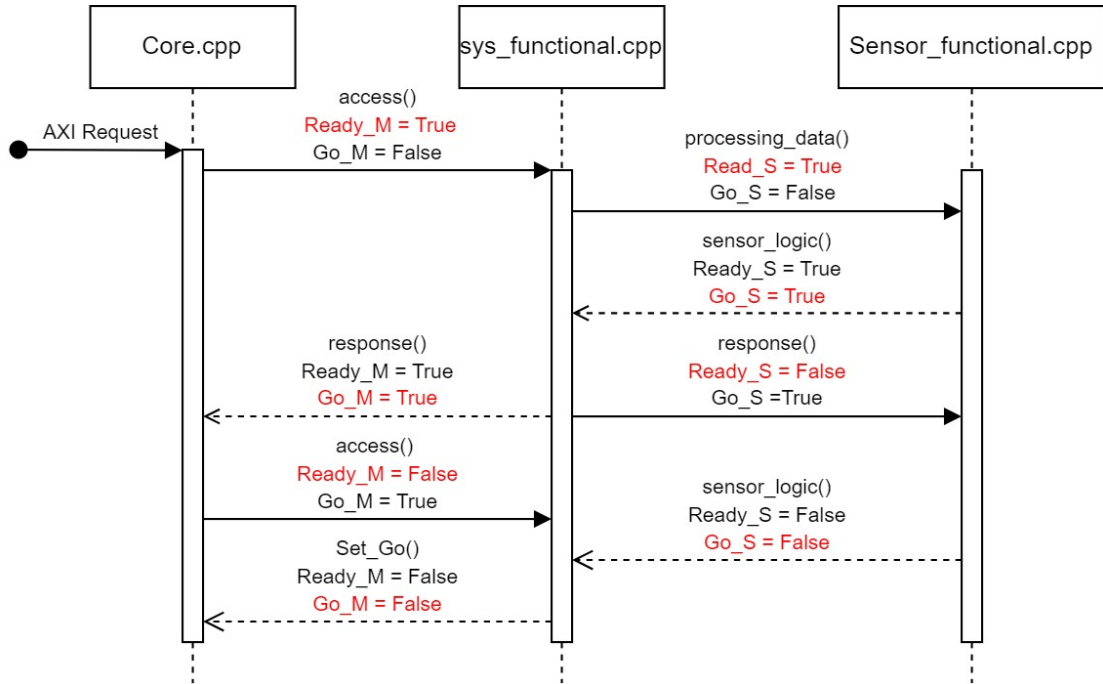


Figure 4.5: Acknowledgement protocol conceptual

## 4.2 GvSoC integration

In this section, we analyze two primary issues: the interaction between GvSoC and SystemC and the time synchronization between them. It is essential to provide a comprehensive description of the simulation flow. The simulation commences by executing the *main* function, which declares and instantiates all the components. Following this, the SystemC simulation is initiated, transferring control to the core module. As mentioned earlier, the core interacts with GvSoC in the following manner:

First and foremost, it loads the configuration of GAP9 into a variable of type `gv::Gvsoc`. Subsequently, it establishes a connection to the main SoC AXI to capture

external read/write operations. From this point onward, the core cyclically executes instructions. It begins by verifying the alignment of the SystemC timestamp and the GvSoC timestamp. If they are not aligned, it takes corrective measures to align them.

Next, through the `gvsoc->step_until()` instruction, the core performs a specific time interval of GvSoC simulation, executing instruction from the code inside the `.bin` file, and retrieves the timestamp of the next important event. After the `step_until()` instruction, two scenarios may arise:

1. When GvSoC does not require data from SystemC, the GvSoC timestamp exceeds the SystemC timestamp. In this case, no further actions are necessary, as the core will realign the timestamps on the next iteration.
2. In the scenario where GvSoC requires data from SystemC, adjustments to the GvSoC timestamp are needed, as it does not account for the sensor's computation time. Therefore, if the SystemC timestamp surpasses the GvSoC timestamp, the core executes the `gvsoc->step_until(int64 time)` instruction, used to increase GvSoC timestamp, where *time* represents the time difference between the two timestamps.

The simulation can end in two distinct conditions, when the battery life time expires or when it is interrupted from the code inside the `.bin` file. When it ends is first stopped the GvSoC simulation and sequentially stopped also the SystemC simulation.

### 4.2.1 GvSoC Requests

Now, let's examine the simulation flow when GvSoC requires data from SystemC. In this scenario, the core handles the exception to ensure that GvSoC obtains the updated information for reading or updates the information in case of writing. The code inside the memory of the SoC may include these types of instructions:

```

1 //Creation of memory pointer to the external sensor
2
3 int* sensor_name = (volatile int *)0x80000000;
4
5 //Write Operation
6 *sensor_name = /*Value*/
7
8 //Read Operation
9 printf("Sensor Measure %x \n", *sensor_name);

```

During the execution of `gvsoc->step_until()`, if the line of code to be executed belongs to that types, SystemC is invoked through the `access()` method present in the core class. In this method, a pointer is passed as a parameter, and this pointer contains all the necessary information.

First, the method reads the 'type' field inside the pointer to distinguish between incoming Read and Write requests. Second, it retrieves the 'address' field and compares it with the addresses known by SystemC using a switch statement. Finally, the request is forwarded to the functional bus. The code that emulates this process is as follows:

```

1   if (req->type == gv::Io_request_read)
2   {
3       printf("Received request (is_read: %d, addr: 0x%lx, size: 0←
      x%lx, data: %d)\n", req->type == gv::Io_request_read, req->addr, ←
      req->size , *(req->data));
4
5       switch (req->addr)
6       {
7           case 0x00:
8               A_Out.write(101);
9               break;
10          ...
11          }
12          D_Out.write(1);
13          F_Out.write(true);
14          Ready.write(true);
15          ...
16          }
17          wait();
18          Ready.write(false);
19          wait();

```

After completing the request, SystemC writes the new data inside the request pointer and sends it back to GvSoC. At this point, GvSoC resumes its job and continues with the simulation until it reaches the end of the `gvsoc->step_until()` instruction. If another request is present, the entire flow is re-executed.

```

1   //Update the data field in the request pointer
2   *((uint32_t*)req->data) = Data_in.read();
3   //Reply to GvSoC
4   this->axi->reply(req);

```



It is easy to notice that during the reply process, SystemC performs computations where it is required to wait for some time. Meanwhile, GvSoC is frozen. After the end of the reply, inevitably, GvSoC's timestamp mismatches SystemC's timestamp. For this reason, as mentioned above, after the `gvsoc->step_until()`, it is necessary to realign the timestamps before continuing the simulation. The following pseudo-algorithm illustrates the synchronization process.

---

**Algorithm 1** Synchronization, Pseudo-Algorithm.

---

```

1: procedure CORE::RUN()
2:   ▷ old_Time is the time of SystemC before GvSoC step
3:   ▷ next_timestamp is the next important event in GvSoC
4:   ▷ time is the time of SystemC after GvSoC step
5:   ▷ Execution
6:   while Simulation ends do
7:     old_time = sc_time_stamp().to_double()
8:     next_timestamp = gvsoc->step_until(old_time)
9:     ▷ Execute until next important event
10:    time = sc_time_stamp().to_double()
11:    if old_time != time then
12:      ▷ If different means SystemC was used during step_until instruction
13:      next_timestamp = gvsoc->step_until(time)
14:      this->axi-reply() ▷ Reply GvSoC request
15:    else
16:      ▷ If equals means no use of SystemC during step_until instruction
17:      wait(next_timestamp - time) ▷ Synchronize SystemC with GvSoC
18:    end if
19:  end while
20: end procedure

```

---

### 4.3 Python code generation

The second part of the work is interested in customization, it means that every one can introduce new component or make the existent ones more complex. Customization is necessary because in real application the environment can quickly change, so it is important to guarantee that the simulator can be adapted in an agile way. First important thing is the possibility of adding a variable number of sensor, this maybe useful in case of the application requires more data form the environment or maybe is necessary to compare two different sensors acquiring the same measure from an energy efficiency point of view. Moreover, is important to

ensure the possibility of changing on the fly some parameter of existing sensor like: technical parameters or maybe internal processes that manage data. To develop these features, a Python script was created using the Jinja framework, which is a Python library that facilitates template creation. Starting from "skeletons" that are *.txt* files the script is able to recreate the complete simulator. What is needed precisely is: templates of all components used and a JSON that defines all parameters. Jinja provides a proper syntax to enrich the templates with dynamic sections, these sections are modeled with the parameters inside the JSON file. Next a python script exposes a class that utilizes jinja methods to create from templates the actual file in *.ccp* and *.h* extensions. the script, containing the class definition, is finally used in a general script that read the source and destination folder and creates the whole simulator.

### 4.3.1 Template syntax

The first structure introduced is the template, it is in most part similar to the final file, except for the some syntax-structures used by jinja to add non-static information. The main three jinja syntax-structures are:

- **{%....%}**: This syntax is used to add statements like: for-loops or if-else. (Ex. `{% if loop.previtem is defined and value > loop.previtem %}` ).
- **{{....}}**: Double curly bracket is used when is necessary to insert a regular expression inside the template. (Ex. `{{ sensor['voltage'] - 3 }}`, it prints the value inside the sensor dictionary diminished by 3).
- **{#....#}**: Hashtags are used to indicate a Jinja comment, is important to notice that this kind of lines are ignored also in the final rendering of the file, they are more useful to explain what others instructions do inside the template. (Ex. `{# This line is a comment! #}` ).

The template will result in a mix of jinja structures and plain text. Following an example of power sensor instance template from the simulator.

```

1 #include "{{sensor['name']}}_power.h"
2
3 void {{sensor['name']}}_power::set_attributes()
4 {
5     func_signal.set_rate(1);
6     func_signal.set_timestep(1, sc_core::SC_SEC);
7 }
8
9 void {{sensor['name']}}_power::initialize() {}
10
11 void {{sensor['name']}}_power::processing()
12 {
13     {% for state in sensor['states'] -%} //Starting of jinja for-←
loop
14     if(func_signal.read() == {{state['number']}} ){
15         //std::cout << "{{sensor['name']}} in {{state['name']}} ←
state!" << std::endl;
16         voltage_state.write(VREF_BUS);
17         current_state.write({{sensor['name'].upper()}}_I_{{state['←
name']}});
18         return;
19     }
20
21     {% endfor -%} //Ending of jinja for-loop
22     if(func_signal.read() == 0){
23         //std::cout << "{{sensor['name']}} in OFF state" << std::←
endl;
24         voltage_state.write(0.0);
25         current_state.write(0.0);
26         return;
27     }
28     std::cout << "{{sensor['name']}} in an Unknown state!" << std::←
endl;
29 }
30

```

### 4.3.2 JSON file

The second element introduced is the JSON file, it as was written above, contains all the parameter of each component. This file is useful for two main reason: the first is the wide support from most popular programming languages, thus it manipulation is really easy to do, if in the futures other software wants to generate the JSON file for the simulator, the main issue will be implement JSON's methods that is not a big deal. Second is that jinja allows dictionaries as parameters of generation, that means every template can be customized just calling the generating method fed

with different JSONs. Recalling the code above, it is possible to create as many power instance as required from the same code, what's matter is just changing the input. The JSON file used for the simulator contains a detailed description of every sensor in particular each sensor object is composed following this structure:

```
1 {
2     "power"   : true,
3     "name"    : "air_quality_sensor",
4     "reg"     : 50,
5     "voltage" : 3.3,
6     "states" : [
7         {
8             "name"      : "ON_READ",
9             "current"   : "49.2",
10            "time_on"  : "30",
11        },
12        {
13            "name"      : "ON_WRITE",
14            "current"   : "48.2",
15            "time_on"  : "30",
16        },
17        {
18            "name"      : "IDLE",
19            "current"   : "0.002",
20        }
21    ]
22 }
```

The *power* field indicates the presence of the power instance, *name* simply indicates the name of the sensor, *reg* defines the number of sensor's internal registers, *voltage* define the operation voltage of the sensor and in the end *states* is a vector of objects where every object is a power state, it is useful to describe in detail every possible action performed by the sensor. Taking the previous JSON, the sensor has three different states: read, write and idle each one has, besides the name, two fundamental parameters: the current and the operation time them are used by the simulator to set appropriate *wait()* statement in systemC and set proper current to send to the power bus. Also here is clear that there is an high degree of freedom in customization, because every sensor can have multiple sub states that refine its energy profile making the simulation more and more closer to the reality.

# Chapter 5

## Experimental Results

In the following section, we will present reports generated by the simulator in both graphical and textual formats. Specifically, the simulator can produce two different .vcd files: one is the Power report, which contains power flow data throughout the entire simulation and provides specific parameters for every component in the system. The other is the Functional report, which contains data flow information through all the components in the system, enabling the verification of the correctness and alignment of GvSoC requests and responses. Following this, we will present a code running on two different computer configurations to observe how available resources affect the simulation speed. The code utilized has been optimized with minimal system calls to ensure that the running time accurately reflects only the time effectively used by the simulator. Furthermore, we will include a simulation of a resource-intensive program that stresses both GvSoC and SystemC. The chapter begins with a setup section that outlines all the necessary steps before using the simulator.

### 5.1 Setup

Throughout this thesis, several software tools and programs played a pivotal role in facilitating analysis and simulation. These tools not only served as the foundation for experimentation but also provided the means to analyze and visualize data, thus contributing significantly to the overall research process. The simulator is executed on a virtual machine running Ubuntu LTS 20.04.6, hosted by VirtualBox Version 6.1.40 r154048 (Qt5.6.2). The following programs are installed on Ubuntu:

- SystemC 2.3.3
- SystemC-AMS 2.3.0
- SDK of RISC-V Toolchain

- GAP SDK of GAP9
- g++ 11.4.0
- Python 3.8.10

To prepare for the simulation, it is necessary to follow specific steps. Assuming it is the first time the simulator is being used, the steps are as follows:

1. Create the SystemC simulator, using the JSON file to specify the configuration.
2. Build the GAP project with the GAP SDK to obtain simulation files.
3. Copy simulation files in the same folder of SystemC simulator.
4. Launch the simulation and create the reports.

Starting from the first step, it is necessary to compile and build the SystemC simulator. To do this, one needs to create the JSON file mentioned in 4.3.2. Once the JSON file is ready, the 'make' command should be executed inside the root folder of the SystemC simulator. This procedure will generate an executable file, which constitutes the complete simulator. However, for running the simulator, it is also necessary to build the GAP project. This step is required to create certain files necessary for mimicking the behavior of the board throughout the entire simulation. The instructions for building the project are as follows:

```
1 cd *GAP_SDK_DIRECTORY*
2 source sourceme.sh
3 *Select the Board (Ex. 1 - GAP9_EVK_AUDIO)*
4 cd *GAP_PROJECT_DIRECTORY*
5 make all run platform=gvsoc
```

After executing these commands, GvSoC will build the project and create several files, with the most important ones being:

- gvsoc\_config.json, the configuration of the board
- chip.soc.mram.bin, the content of the board's memory
- efuse\_preload.data, the stimuli applied on the board

These are the three files that need to be copied into the same folder as the previous executable file. In the end, it is possible to run the program to initiate the simulation and generate reports. If any changes are made, it is necessary to repeat the steps corresponding to the modified part (e.g., GvSoC Project).

## 5.2 Simulator Results

The example simulation consists of a board equipped with five sensors, each with different characteristics. The JSON configuration of this setup can be found in the appendix. Meanwhile, the C program running on the board is a simple while loop that reads the values from a fixed internal register of each sensor during each iteration. The code for the board is also included in the appendix A.

At this point, after the simulation, three types of results are available: the functional output of the board, the corresponding functional report, and the power report.

**Functional Output** The functional output is the data that the board transmits to a potential output peripheral capable of receiving data, such as a monitor or a communication channel like UART. In practical applications, this is how data is retrieved, stored, and analyzed when the board is operating in a real-world environment. In reference to the code in the appendix, the output corresponds to the 'printf' instructions, and the result is:

```
1 Air_quality Read 61
2 Temperature Read 13
3 Methane Read 7
4 MicroPhone Read 70
5 Radio Frequency Transmission OK!
```

This output is printed multiple times throughout the entire simulation to continuously engage with the sensors attached to the board. This serves two purposes: first, it facilitates interactions between SystemC and GvSoC; second, it enables the sensors to consistently switch between active and idle power states, resulting in a more comprehensive power report.

**Functional Report** The functional report is generated following the VCD trace format, which includes time and other pertinent details for each step. It records the functional flow within the SystemC simulator, encompassing the sequence of requests. More specifically, it displays the core's provided address and data, along with the bus data retrieved. An example of the functional report is structured as follows.

```
1 %time Core_Address Core_Data Bus_Data
2 0 0 0 0
3 0.001737550452 101 1 0
4 0.031737550452 101 1 61
5 0.031791620254 201 1 61
6 0.061791620254 201 1 13
7 0.09789394302 301 1 13
8 0.12789394302 301 1 7
9 0.127943934367 401 1 7
10 0.139943934367 401 1 70
11 0.139993376078 501 1 70
```

When reviewing the report, it becomes possible to comprehend the logic of each line. Starting from the second line, there are two lines with identical addresses. The first line represents the moment when the core sends the request to the bus, while the second line shows the moment when the bus responds to the core. By comparing the timestamps between these two lines, it is feasible to calculate the delay of the sensor connected to a specific address. In most instances, this delay corresponds to one of the potential power states outlined in the JSON sensor specification. If the sensor performs a reading between the two lines, the delay will be associated with the reading status.



**Power Report** The power report contains all energy information of the system, with one of the most important aspects being the current profile of each off-chip component. These profiles are used to monitor the current trends during the simulation. An essential field is the SoC (State of Charge), which indicates the system’s battery charge level. In applications where power consumption is a critical constraint, this report is highly valuable for identifying moments when the system consumes more power. When used in conjunction with the functional report, it becomes possible to correlate high power consumption with the specific instructions executed. This correlation allows for a focus on optimizing those particular situations when feasible. The report maintains the same structure as the functional report, which follows the general .vcd format. Each line is timestamped, followed by the corresponding measurements for that time. Below is a summarized version of the report.

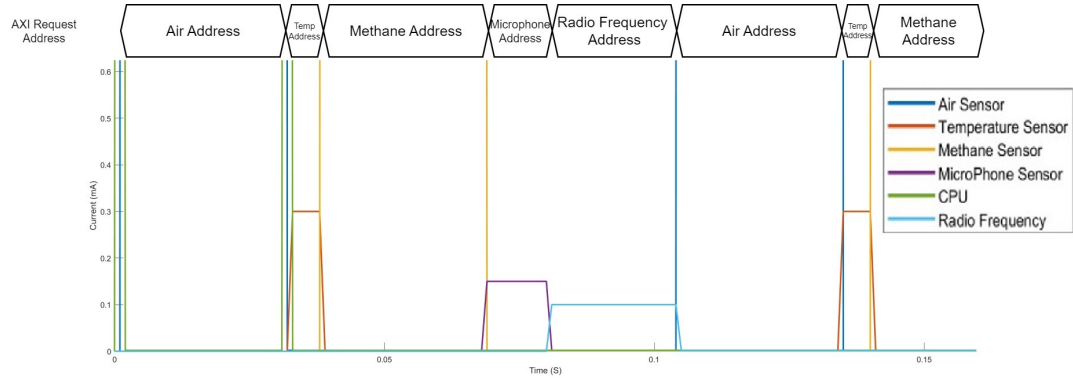
```

1 %time SoC_Batt I_Tot_Bus I_to_Batt Air Temp Meth Mic CPU RF
2 0 0 0 0 0 0 0 0 0 0
3 0.267 0.999607694469 48.207 0.065157094939 48.2 0.002 0.002 0.002 0 ←
   0.001
4 .
5 0.298 0.99951522548 0.307 0.210653052834 0.002 0.3 0.002 0.002 0 ←
   0.001
6 .
7 0.303 0.999513576036 18.007 0.210553004647 0.002 0.002 18 0.002 0 ←
   0.001
8 .
9 0.764 0.998959584653 0.108 0.0651778062117 0.002 0.002 0.002 0.002 ←
   0 0.001

```

Above is the report detailing the current demand from the system to the battery, the battery’s SoC (State of Charge), and the current demanded by each sensor. Additionally, it includes the current generated by the harvester, which is taken into account during the computation. The time scale is in the order of milliseconds, while the current is expressed in amperes.

**Plotted Results** It is possible to plot the previous reports with MATLAB to observe that the simulations indeed progress together. Using Program 1 provided in the appendix as a reference, the following graphs are obtained.



**Figure 5.1:** Plot of functional report

The upper part shows the activity from a functional perspective, specifically displaying the requests sent from the core to the bus. Concurrently, in the bottom part, the activity of each sensor is depicted. By examining a specific timeframe, it becomes evident that if a request pertains to one sensor, the others remain in the idle state, with only the activated one actively processing. Occasionally, the CPU current rises, indicating ongoing computational operations.

### 5.3 Simulator Benchmarks

This section describes the performance of the simulator in different scenarios. Two different programs will be analyzed: one involving simple sampling of each sensor and another involving complex computations between samples. The maximum simulation time will be the same for both programs. Furthermore, the same experiment will be conducted on two different computers with varying architectures to yield diverse results. These results will be described and interpreted.

Regarding the programs, the first one is a straightforward program that samples every sensor within a 'while' loop, acquiring their values without applying any computation (Program 1 in the table). The second program acquires measurements for each sensor and calculates the simple moving average. At the end of each cycle, it prints the data results and restarts (Program 2 in the table).

The computer configurations under consideration consist of virtual machines based on Intel architecture with Intel Core i5-8250U processors, featuring 2 cores and 4GB of RAM, as well as AMD architecture configured with a Ryzen 7 3700X processor, equipped with 4 cores and 8GB of RAM. In the appendix A, is possible to find the programs run on the GAP simulator, with a simulation maximum time set to 10 seconds.

	Intel Architecture		AMD Architecture
Program 1	Real	1m 30.242s	0m 19.583s
	User	1m 28.928s	0m 19.390s
	Sys	0.336s	0.186s
Program 2	Real	1m 25.719s	0m 18.524s
	User	1m 24.944s	0m 18.135s
	Sys	0.170s	0.379s

**Table 5.1:** Comparison between two C programs. The values represent the average command execution time over ten runs for the 'time command' in Linux.

It is easy to notice that the more powerful the machine used to run the simulator, the less time the simulation requires. The time savings are more than half. The crucial point to understand is that in both configurations, the time is acceptable in terms of the absolute scale, so it is possible to run extensive simulations and obtain useful results in a matter of hours at the very least.

A significant constraint lies in the resolution scale of the simulation; the smaller the measurement unit, the more time it takes to perform all the computations.

An important corner case to mention is when GvSoC doesn't require SystemC.

In such cases, the simulation is exceptionally slow because most of the time is spent realigning the timestamps for every micro-event occurring in GvSoC, which typically operates with a timescale in the picoseconds.

The following table illustrates the difference between a program that doesn't use SystemC and a program that samples only one sensor. The simulation is run on an AMD architecture with a simulation maximum time set to 10 seconds.

	No use of SystemC	Use of SystemC
Real	2m 1.228s	0m 18.901s
User	1m 57.613s	0m 18.751s
Sys	0m 3.534s	0m 0.146s

**Table 5.2:** Comparison between two C programs. The first one does not use external sensors, while the second one does. The values represent the average execution time of the 'time command' over ten runs in Linux.

In cases where there is no need to simulate extra-functional properties, it is not advisable to use the simulator due to the overhead of realigning operations. Instead, it is better to run a simple functional simulation with GvSoC, avoiding unnecessary delays.

## Chapter 6

# Conclusions and Future Works

Functional simulators are very common and widely used, but when it becomes necessary to simulate extra-functional properties, the available solutions often exhibit defects and limitations. Most of the time, one has to rely on closed companies that cannot guarantee the complete availability of what is necessary for specific cases. Moreover, most of the tools proposed are paid and receive limited updates. In response to this issue, the world is undergoing changes and proposing alternative methods to better represent and analyze different aspects such as power consumption, thermal trends, and reliability. This thesis aims to introduce one of these new solutions, positioning itself as a pioneer in this context.

SystemC is seen as a rising star, making it the best choice as the core of this work. Additionally, this thesis can serve as a starting point for various applications since the approach proposed is not limited to the explored scenario; it can be adapted to any possible extra-functional property.

The choice of the RISC-V architecture is also driven by its increasing popularity among computer engineers and its open-source nature, which aligns with SystemC's open-source philosophy. Lastly, this thesis is part of the European TRISTAN project, enhancing the set of tools available for RISC-V.

A brief overview of what this work encompasses includes guidelines on constructing a robust simulator capable of representing various properties within a given system. It emphasizes the communication flow between components and their scheduling to ensure the correct flow of information. There is also a section on making the simulator dynamic, allowing for variable and adaptive configurations, which is useful when parameters or components within the system need to be changed. Finally, a fundamental aspect is the integration with GvSoc [12], which is perhaps the most critical part as it enables the integration of multiple environments,

leveraging their strengths.

The key aspects to take away from this thesis are summarized in a bullet list:

- Developing a SystemC simulator capable of self-coordinating its processes and representing a wide range of extra-functional properties.
- Providing guidelines for creating Python scripts to easily modify the developed simulator, enabling agile configuration between systems for testing purposes.
- Establishing an interfacing approach between two distinct programs, in this case, SystemC-AMS and GvSoC, which is crucial if it becomes necessary to connect and leverage the strengths of two different technologies.

**Improvements and Future Work** As an initial attempt to create this type of simulator, this work presents several aspects that can be enhanced in the future. Starting with the less significant, the interface could be made more user-friendly, perhaps by developing a dedicated GUI for instantiating simulator components and running scripts. Another significant aspect that can be improved, especially in the case of the GAP architecture, is the parallelization of operations. In this initial version of the simulator, only one of the nine cores is utilized to run the code. It is possible to make modifications to enable all nine cores to interface with SystemC, potentially generating more realistic board utilization profiles, which could be valuable for future complex implementations such as AI. Lastly, additional extra-functional simulators can be incorporated; the framework is already in place, and the only necessary steps involve creating additional extra-functional buses to manage these new sub-instances. This would be beneficial if there is a need to analyze multiple properties simultaneously under the same conditions.

# Appendix A

## Code

Listing A.1: C Program 1 inside GAP Memory

```
1  /* PMSIS includes */
2  #include "pmsis.h"
3  /* Program Entry. */
4  int main(void)
5  {
6      printf("\n\n\t *** PMSIS HelloWorld ***\n\n");
7      printf("Entering main controller\n");
8
9      uint32_t errors = 10;
10     uint32_t core_id = pi_core_id(), cluster_id = pi_cluster_id();
11     printf("[%d %d] Hello World!\n", cluster_id, core_id);
12
13     uint32_t last_value = *(volatile int *)0x1c000000;
14
15     int* air_quality = (volatile int *)0x80000000;
16     int* temperature = (volatile int *)0x80000004;
17     int* mick_click = (volatile int *)0x8000000C;
18     int* methane = (volatile int *)0x80000008;
19     int* radio_freq = (volatile int *)0x80000010;
20
21     while(1){
22         printf("Air_quality Read %d\n", *air_quality);
23         printf("Temperature Read %d\n", *temperature);
24         printf("Methane Read %d\n", *methane);
25         printf("MichroPhone Read %d\n", *mick_click);
26         printf("Radio Frequency Transmission %d\n", *radio_freq);
27     }
28
29     return errors;
30 }
```

Listing A.2: C Program 2 inside GAP Memory

```
1 /* PMSIS includes */
2 #include "pmsis.h"
3 /* Program Entry. */
4 int main(void)
5 {
6     printf("\n\n\t *** PMSIS HelloWorld ***\n\n");
7     printf("Entering main controller\n");
8
9     uint32_t errors = 10;
10    uint32_t core_id = pi_core_id(), cluster_id = pi_cluster_id();
11    printf("[%d %d] Hello World!\n", cluster_id, core_id);
12
13    int air_vector[10];
14    int temp_vector[10];
15    int methane_vector[10];
16    int mic_vector[10];
17    int flag = 0;
18
19    int* air_quality = (volatile int *)0x80000000;
20    int* temperature = (volatile int *)0x80000004;
21    int* mic_click = (volatile int *)0x8000000C;
22    int* methane = (volatile int *)0x80000008;
23    int* radio_freq = (volatile int *)0x80000010;
24
25    while(1){
26
27        for(int i = 0; i < 10; i++){
28            //Force certain values for moving average
29            if (flag == 0){
30                *air_quality = 70;
31                *temperature = 25;
32                *mic_click = 85;
33                *methane = 20;
34            } else {
35                *air_quality = 75;
36                *temperature = 22;
37                *mic_click = 95;
38                *methane = 15;
39            }
40            //printf("OK! Force");
41            //Read values for moving average
42            uint32_t air_value = *(volatile int *)0x80000000;
43            uint32_t temp_value = *(volatile int *)0x80000004;
44            uint32_t methane_value = *(volatile int *)0x80000008;
45            uint32_t mic_value = *(volatile int *)0x8000000C;
46            //Save value for moving average
47            air_vector[i] = air_value;
48            temp_vector[i] = temp_value;
```



```
49     methane_vector[i] = methane_value;
50     mic_vector[i] = mic_value;
51     if(flag == 1)
52         flag = 0;
53     else
54         flag = 1;
55 }
56 float air_avg = 0.0;
57 float temp_avg = 0.0;
58 float methane_avg = 0.0;
59 float mic_avg = 0.0;
60 //printf("OK! Read");
61 for (int i=0; i<10; i++){
62     air_avg = air_avg + air_vector[i];
63     temp_avg = temp_avg + temp_vector[i];
64     methane_avg = methane_avg + methane_vector[i];
65     mic_avg = mic_avg + mic_vector[i];
66 }
67 air_avg = air_avg/10;
68 temp_avg = temp_avg/10;
69 methane_avg = methane_avg/10;
70 mic_avg = mic_avg/10;
71 printf("Air moving average = %.2f \n", air_avg);
72 printf("Temperature moving average = %.2f \n", temp_avg);
73 printf("Methane moving average = %.2f \n", methane_avg);
74 printf("Mic Click moving average = %.2f \n", mic_avg);
75 }
76
77 return errors;
78 }
```

Listing A.3: JSON Configuration for SystemC

```
1 {
2     "sim_step" : 1,
3     "sim_len": 7736400,
4     "vref_bus" : 3.3,
5     "soc_init" : 1.0,
6     "selfdisch_factor" : 0.0,
7     "battery" : "circuit_model",
8     "sensors" : [
9         {
10            "power" : true,
11            "name" : "air_quality_sensor",
12            "voltage": 3.3,
```

```
13         "reg" : 50,
14         "states" : [
15             {
16                 "name" : "ON_READ",
17                 "current" : "48.2",
18                 "time_on" : "30"
19             },
20             {
21                 "name" : "ON_WRITE",
22                 "current" : "49.2",
23                 "time_on" : "30"
24             },
25             {
26                 "name" : "IDLE",
27                 "current" : "0.002"
28             }
29         ]
30     },
31     {
32         "power" : true,
33         "name" : "temperature_sensor",
34         "voltage": 3.3,
35         "reg" : 75,
36         "states" : [
37             {
38                 "name" : "ON_WRITE",
39                 "current" : "0.35",
40                 "time_on" : "6"
41             },
42             {
43                 "name" : "ON_READ",
44                 "current" : "0.3",
45                 "time_on" : "6"
46             },
47             {
48                 "name" : "IDLE",
49                 "current" : "0.002"
50             }
51         ]
52     },
53     {
```

```
54     "power"    : true,
55     "name"     : "methane_sensor",
56     "voltage"  : 3.3,
57     "reg"      : 25,
58     "states"   : [
59         {
60             "name"      : "ON_WRITE",
61             "current"   : "19",
62             "time_on"   : "30"
63         },
64         {
65             "name"      : "ON_READ",
66             "current"   : "18",
67             "time_on"   : "30"
68         },
69         {
70             "name"      : "IDLE",
71             "current"   : "0.002"
72         }
73     ]
74 },
75 {
76     "power"    : true,
77     "name"     : "mic_click_sensor",
78     "voltage"  : 3.3,
79     "reg"      : 15,
80     "states"   : [
81         {
82             "name"      : "ON_WRITE",
83             "current"   : "0.25",
84             "time_on"   : "12"
85         },
86         {
87             "name"      : "ON_READ",
88             "current"   : "0.15",
89             "time_on"   : "12"
90         },
91         {
92             "name"      : "IDLE",
93             "current"   : "0.002"
94         }
95     ]
96 }
```

---

```
95     ]
96     },
97     {
98         "power" : true,
99         "name"  : "rf_sensor",
100        "voltage": 3.3,
101        "reg"   : 10,
102        "states" : [
103            {
104                "name"      : "ON_WRITE",
105                "current"   : "0.15",
106                "time_on"  : "24"
107            },
108            {
109                "name"      : "ON_READ",
110                "current"   : "0.1",
111                "time_on"  : "24"
112            },
113            {
114                "name"      : "IDLE",
115                "current"   : "0.001"
116            }
117        ]
118     }
119 ]
120 }
```

---

# Bibliography

- [1] *About QEMU; QEMU documentation - qemu.org*. <https://www.qemu.org/docs/master/about/index.html>. [Accessed 29-09-2023] (cit. on p. 2).
- [2] *Renode — renode.io*. <https://renode.io/about/>. [Accessed 29-09-2023] (cit. on p. 2).
- [3] *RISC-V - Wikipedia — en.wikipedia.org*. <https://en.wikipedia.org/wiki/RISC-V>. [Accessed 28-09-2023] (cit. on p. 3).
- [4] *New call for developing an HPC ecosystem based on RISC-V - eurohpc-ju.europa.eu*. [https://eurohpc-ju.europa.eu/new-call-developing-hpc-ecosystem-based-risc-v-2023-02-01\\_en](https://eurohpc-ju.europa.eu/new-call-developing-hpc-ecosystem-based-risc-v-2023-02-01_en). [Accessed 28-09-2023] (cit. on p. 3).
- [5] *Together for RISC-V Technology and ApplicationS | TRISTAN Project | Fact Sheet | HORIZON | CORDIS | European Commission — cordis.europa.eu*. <https://cordis.europa.eu/project/id/101095947/it>. [Accessed 28-09-2023] (cit. on p. 4).
- [6] Amal Banerjee and Balmiki Sur. *SystemC and SystemC-AMS in practice: SystemC 2.3, 2.2 and SystemC-AMS 1.0*. July 2014, pp. 1–460. ISBN: 978-3-319-01146-2. DOI: 10.1007/978-3-319-01147-9 (cit. on pp. 6, 10).
- [7] Enrico Fraccaroli and Sara Vinco. «Modeling Cyber-Physical Production Systems With SystemC-AMS». In: *IEEE Transactions on Computers* 72.7 (2023), pp. 2039–2051. DOI: 10.1109/TC.2022.3226567 (cit. on pp. 10, 23–26).
- [8] Andrew Waterman, Krste Asanovic, et al. «The RISC-V Instruction Set Manual Volume I: Unprivileged ISA». In: *Document Version* 20191213 (2019), pp. 1–4 (cit. on pp. 14, 15, 17).
- [9] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. «Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing». In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: 10.1109/JSSC.2019.2912307 (cit. on p. 18).

- [10] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. «Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX». In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145 (cit. on pp. 19, 20).
- [11] *GAP9 Product-Brief V1.14*. GreenWaves Technologies SAS (cit. on p. 20).
- [12] Irina Sizova. *GVSOC - The Full System Simulator for profiling GAP Applications* — *greenwaves-technologies.com*. <https://greenwaves-technologies.com/gvsoc-the-full-system-simulator-for-profiling-gap-applications/>. [Accessed 09-10-2023] (cit. on pp. 21, 22, 56).
- [13] Yukai Chen, Sara Vinco, Daniele Jahier Pagliari, Paolo Montuschi, Enrico Macii, and Massimo Poncino. «Modeling and Simulation of Cyber-Physical Electrical Energy Systems With SystemC-AMS». In: *IEEE Transactions on Sustainable Computing* 5.4 (2020), pp. 552–567. DOI: 10.1109/TSUSC.2020.2973900 (cit. on pp. 23–25).
- [14] Sara Vinco, Alessandro Sassone, Franco Fummi, Enrico Macii, and Massimo Poncino. «An open-source framework for formal specification and simulation of electrical energy systems». In: *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2014, pp. 287–290. DOI: 10.1145/2627369.2627657 (cit. on pp. 23–25).
- [15] Sara Vinco, Yukai Chen, Franco Fummi, Enrico Macii, and Massimo Poncino. «A Layered Methodology for the Simulation of Extra-Functional Properties in Smart Systems». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36 (Jan. 2017), pp. 1702–1715. DOI: 10.1109/TCAD.2017.2650980 (cit. on pp. 23, 25, 26, 28).
- [16] *Jinja* — *palletsprojects.com*. <https://palletsprojects.com/p/jinja/>. [Accessed 09-10-2023] (cit. on p. 29).