UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

ANDERSON IGNACIO DA SILVA

IPSoCGen platform - Framework for MP/SoC generation

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

Advisor: Prof. Dr. Altamiro Amadeu Susin

Porto Alegre
April 2024

CIP — CATALOGING-IN-PUBLICATION

Silva, Anderson Ignacio da

IPSoCGen platform - Framework for MP/SoC generation / Anderson Ignacio da Silva. – Porto Alegre: PGMICRO da UFRGS, 2024.

108 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2024. Advisor: Altamiro Amadeu Susin.

1. Design generation, Network-on-Chip, Parallel processing, VLSI. I. Susin, Altamiro Amadeu. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Prof^{a}. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof^{a}. Ione Borcelli Gonçalves
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do PGMICRO: Prof. Cláudio Radtke
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

ABSTRACT

System-on-Chip (SoC) architectures encompass multiple processing elements and a communication fabric on a single integrated circuit, offering substantial parallelism and a high communication bandwidth. This arrangement yields significant performance benefits while maintaining low power consumption. Nevertheless, designing and verifying complex VLSI systems presents challenges and often involves employing pre-defined and certified functional building blocks referred to as Intellectual Property (IP). The state of the art in design generation focuses more on high-level abstraction through Scale-based language, limiting the flexibility of the generated hardware. This work signifies a notable advancement by introducing a platform for constructing configurable systems that streamline the interconnection of pre-designed IPs through a Network on Chip (NoC). The nodes within this network comprise processors or any self-governing data handling systems that adhere to industry-standard protocols. This enables the mapping of multiple independent or interconnected processes onto various node clusters, allowing them to function autonomously. The platform's user-friendly interface permits the specification of global parameters, simulation, debugging, RTL generation, synthesis, and the uploading of FPGA-based application code. Additionally, the communication interface protocol enables the integration of Special Purpose Cores with the internal bus or NoC Interface, thus enhancing the system's adaptability and extensibility. The principal application domains envisioned for this platform include Image Processing/Computer Vision and Artificial Intelligence engines. To demonstrate the feasibility of the design flow and explore the performance benefits derived from parallelism, basic image processing algorithms were implemented as proof of concept applications. Given the platform's capacity to generate both SoCs and MPSoCs, both designs were produced and benchmarked using the image processing application. These benchmarks revealed the advantages and limitations of each system configuration. While this initial implementation provides valuable insights, further endeavors are necessary to enrich the hardware directory of the nodes and enhance security and reliability aspects. The final platform presented is capable of generating multiple design topologies, providing flexibility to test these systems to the emulation and prototyping stages, where the performance and correctness can be evaluated.

Keywords: Design generation, Network-on-Chip, Parallel processing, VLSI.

Plataforma IPSoCGen - Framework para geração MP/SoC

RESUMO

Arquiteturas System-on-Chip (SoC) abrangem múltiplos elementos de processamento e um barramento de comunicação em um único circuito integrado, oferecendo substancial paralelismo e uma alta largura de banda de comunicação. Essa disposição resulta em significativos benefícios de desempenho ao mesmo tempo em que mantém um baixo consumo de energia. No entanto, projetar e verificar sistemas complexos de VLSI apresenta desafios e frequentemente envolve a utilização de blocos de construção funcionais predefinidos e certificados, referidos como Propriedade Intelectual (IP). O estado da arte na geração de design foca mais na abstração de alto nível por meio de bibliotecas baseadas em linguagem Scala, limitando a flexibilidade do hardware gerado. Este trabalho representa um notável avanço ao introduzir uma plataforma para construir sistemas configuráveis que facilitam a interconexão de IPs pré-projetados por meio de uma Rede em Chip (NoC). Os nós dentro dessa rede incluem processadores ou qualquer sistema autônomo de manipulação de dados que aderem a protocolos padronizados pela indústria. Isso possibilita o mapeamento de múltiplos processos independentes ou interconectados em vários aglomerados de nós, permitindo que eles funcionem autonomamente. A interface amigável dessa plataforma permite a especificação de parâmetros globais, simulação, depuração, geração de RTL, síntese e o carregamento de código de aplicação baseado em FPGA. Além disso, o protocolo de interface de comunicação possibilita a integração de Núcleos de Propósito Especial ao barramento interno ou Interface NoC, aprimorando assim a adaptabilidade e extensibilidade do sistema. Os principais domínios de aplicação idealizados para essa plataforma incluem Processamento de Imagem/Visão Computacional e motores de Inteligência Artificial. Para demonstrar a viabilidade do fluxo de design e explorar os benefícios de desempenho derivados do paralelismo, algoritmos básicos de processamento de imagem foram implementados como aplicações de prova de conceito. Dada a capacidade da plataforma de gerar tanto SoCs quanto MPSoCs, ambos os projetos foram produzidos e avaliados usando a aplicação de processamento de imagem. Essas avaliações revelaram as vantagens e limitações de cada configuração de sistema. Embora essa implementação inicial forneça insights valiosos, esforços adicionais são necessários para enriquecer o diretório de hardware dos nós e aprimorar aspectos de segurança e confiabilidade. A plataforma final apresentada é capaz de gerar múltiplas topologias de projeto, proporcionando flexibilidade para testar esses sistemas até as etapas de emulação e prototipagem, onde o desempenho e a correção podem ser avaliados.

Palavras-chave: Geração de design, Redes-em-Chip, Processamento paralelo, VLSI.

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF ABBREVIATIONS AND ACRONYMS

3PIP	Third-party intellectual property
AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
AXIL	AXI-Lite protocol
AXIS	AXI-Stream protocol
BRAM	Block RAM
CAD	Computer-aided design
CDC	Clock Domain Crossing
CSR	Control and Status Register
CV	Computer Vision
DFT	Design for Testability
DMA	Direct Memory Access
DSE	Design Space Exploration
DUT	Device Under Test
DVFS	Dynamic Voltage and Frequency Scaling
EDA	Electronic Design Automation
ELF	Executable and Linkable Format
FIFO	First-in First-out
FPS	Frames-per-second
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

GPP	General Purpose Processor
HDL	Hardware Description Language
HTML	HyperText Markup Language
I2C	Inter-Integrated Circuit protocol
IO	Input and Output
ISA	Instruction Set Architecture
KPI	Key Performance Indicator
LMA	Loaded Memory Address
LSU	Load-and-Store Unit
LUT	Look-Up Table
MAC	Media Access Control
MMCM	Mixed-Mode Clock Manager
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
MPSoC	Multi-Processor System-on-Chip
MTU	Maximum Transmission Unit
NI	Network Interface
NoC	Network-on-Chip
OS	Operating System
PHY	Physical Layer
PLL	Phase-locked loop
PMCA	Programmable many-core accelerators
PPA	Power, Performance and Area
QEMU	Quick Emulator
RAM	Random Access Memory
RI	Random Instruction

RGMII	Reduced Gigabit Media Independent Interface
RISC-V	Reduced Instruction Set Computer (RISC) Five
RISCOF	RISC-V Compatibility Framework
RLNC	Random Linear Network Coding
RTOS	Real-time operating System
SoC	System-on-Chip
SPMD	Single Program Multiple Data
TLM	Transaction Level Modeling
TEE	Trusted Execution Environment
UART	Universal Asynchronous Receiver / Transmitter
UVM	Universal Verification Methodology
UDP	User Datagram Protocol
UEFI	Unified Extensible Firmware Interface
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
VMA	Virtual Memory Address
VC	Virtual Channel
WFI	Wait for Interrupt
WNS	Worst Negative Slack
YAML	Yet Another Markup Language

CONTENTS

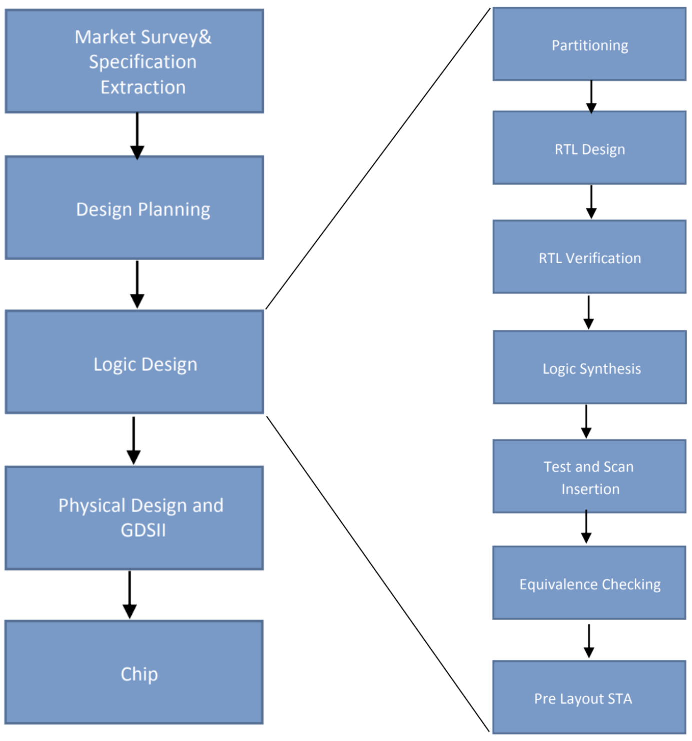
1 INTRODUCTION

Over the past few years, the number of devices integrating Systems-on-Chip (SoCs) has been massively increasing. From simple networking modems to advanced embedded AI controllers, the number of different usages aims to grow as much as possible. As initially mentioned by Lee (2003), the advance of the technology enables the integration of multiple processing elements, memory cells, and analog macros, which converges to the *SoC era*. The versatility of these systems is evident in their capacity to be applied across multiple domains, including mobile processing (BRIGGS; ZARKESH-HA, 2014), Edge server (XU; ZHANG; WANG, 2022) or even implant for in-body strain sensing (ABDELHAMID et al., 2023). Thus, the number of different requirements is analogous to the SoC growth, with applications requiring General- Purpose Processor (GPP) and hardware accelerators.

To meet an extensive array of requirements, Multi-Processor System-on-Chip (MPSoC) architectures emerge as a highly suitable solution. These MPSoCs find utility across a diverse spectrum of applications, spanning from automotive to healthcare. They typically exhibit a tile-based architectural structure and facilitate intercommunication through a Network-on-Chip (NoC), as elaborated in (AZAD et al., 2019). Differently from standard SoCs, MPSoCs are highly influenced by their application programming model, at the first one the CPU can run fairly independent programming model and in the last it is build to share workloads considering its application in general (WOLF; JERRAYA; MARTIN, 2008). Henceforth, applications encompassing various forms of data processing, including but not limited to image, audio, and video processing, can experience substantial acceleration when tailored for execution on a MPSoC platform.

When it comes to manufacturing, both SoC and MPSoC are implemented through the same custom ASIC (Application-Specific Integrated Circuit) flow. The general VLSI (Very Large Scale Integration) flow, as represented in Figure 1.1, illustrates the various steps involved in the entire process, starting from product scoping to the foundry where the chip is built. According to (TARAATE, 2021), the initial step involves product scoping and specification extraction, where the requirements are defined with a focus on the end product. Subsequently, the design planning phase is primarily characterized by architecture and micro-architectural definition. In this phase, high-level models can be implemented to describe the product at a system level, abstracting the low-level details. These models can also serve as a reference for parallel software development.

Figure 1.1: ASIC Custom flow



Source: Modified from (TARAATE, 2021)

The logic design phase encompasses the conversion of the high-level specifications into tangible design blocks described in a commercially available Hardware Description Language (HDL). This phase runs in parallel with the verification process. Once the design achieves a specific level of quality, logic synthesis takes place, initiating the technology mapping process that transforms the RTL (Register Transfer Level) into a netlist. Within this stage, various sub-stages occur, including DFT (Design for Testability) and Scan-chain insertion, which are assessed using logical equivalence checking tools. Subsequently, the design is transformed into geometric structures using a range of CAD (Computer-Aided Design) tools during the Physical Design step. This step encompasses power planning, clock tree synthesis, and place-and-route, among others.

Throughout each stage of the ASIC development process, multiple checkpoints are established to evaluate the project's constraints and the PPA (Power, Performance, and Area) characteristics. As highlighted by sources such as (LI et al., 2018), (HASSAN et al., 2021), and (ESMAEILZADEH et al., 2022), metrics related to power consumption, performance, and chip area are invaluable for comprehending the system's behavior. These metrics aid in assessing whether the various levels of abstractions are aligned with the ultimate chip target.

Evaluating these metrics throughout the development flow is a complex task that typically necessitates a flexible methodology. In such a methodology, different teams are required to support multiple inputs and outputs regarding project planning, result compilation, synchronization barriers, and other factors. For instance, when the system is modeled using a high-level language like SystemC (commonly used in UVM verification environments), bus transactions are executed atomically through TLM (Transaction-Level Modeling) (XU; POLLITT-SMITH, 2005). However, when aiming to calculate the throughput of an architectural specification model against an RTL (Register Transfer Level) design that provides cycle-accurate waveforms, the comparison becomes significantly challenging. The lack of a clear method to turn the specification into an emulated circuit can result in significant differences between the initial proposal and the actual circuit design throughout various stages.

From an industry standpoint, the general workflow involves multiple levels of integration with a substantial number of engineers involved at each stage of the chip development cycle. Consequently, the comparison of the aforementioned metrics is usually conducted through sign-off reviews, which progress through each team responsible for taking the design forward in the flow. Typically, the architectural specification provides a rough estimate or a "*ballpark*" idea of the final PPA numbers that will be attained after the *tapeout*. Considering the inherent risks involved in the process, the team can evaluate whether the worst-case scenario is still acceptable from a product perspective.

In conclusion, the implementation of a platform that facilitates rapid and precise design development serves to bridge the gap existing between the various levels of abstraction within the ASIC design flow. This can ultimately contribute to achieving a more favorable Time-to-Market (TTM) by eliminating the need for iterative exchanges between architectural and design teams. By providing a snapshot of the final metrics based on a cycle-accurate approach, such a platform enables a more efficient and streamlined design process.

1.1 Dissertation Goal

The primary objective of this dissertation is to develop a framework capable of effectively addressing the previously mentioned challenge of facilitating rapid and flexible digital design prototyping, starting from an architecture specification and proceeding to the emulation stage. In addition to the framework, a comprehensive workflow will be presented to enable the utilization of the platform, encompassing all the necessary steps, including software development. To accomplish this objective, the following contributions can be emphasized:

1. Conduct a review of the existing literature on MP/SoC design frameworks, industry standard protocols, and various IP designs.
2. Define a collection of essential components required for constructing a framework, which includes pre-verified IPs.
3. Establish a methodology for constructing and generating designs, based on a standardized file format utilized within the EDA tools domain.
4. Develop and demonstrate the practical application of the auto-generated design, by comparing different approaches between SoC and MPSoC designs, along with their respective capabilities.

1.2 Dissertation Outline

The presented work is structured as follows. Chapter 2 provides a comprehensive literature review of various frameworks for design generation, encompassing both general-purpose and system-specific generation frameworks. This chapter also examines open-source and industry standards about each individual IP.

Chapter 3 focuses on the base IP set chosen to construct the framework. The chapter highlights the significance of these IPs within the framework while delving into their features and limitations in detail.

Chapter 4 presents the proposed methodology flow, elucidating the different stages involved. Each stage is explained, along with its connection to the overall proposal. Additionally, this chapter demonstrates the chosen configuration file and its role in driving the design generation process. It also provides insight into the background of platform development, including the various structures implemented within the proposal.

Chapter 5 describes the two template architectures developed using the framework and outlines the implementation of their respective software stacks. This chapter also addresses the distinct requirements for each system.

Moving on to Chapter 6, the associated experimental results are presented, considering the developments described in the previous chapter. A thorough comparison of the systems is conducted, focusing on PPA metrics, with an extended analysis of their performance.

Finally, the conclusion summarizes the accomplishments of the research and suggests potential avenues for future enhancements to the framework, paving the way for further work in this field.

2 BACKGROUND AND RELATED WORK

The objective of this chapter is to offer an in-depth examination of existing literature, with a specific emphasis on diverse hardware generators. This review seeks to provide a broad perspective on this field, outline various problem-solving approaches, and facilitate comparisons among them. Additionally, each solution will be evaluated and analyzed, highlighting their respective advantages, drawbacks, and potential areas for improvement within a comprehensive framework.

The criteria for choosing the following related works were based on their mention in published articles and their attempts to address the common problem of enabling fast design prototyping. Additionally, among those selected, only works that presented systems generated using their frameworks were described.

2.1 Bastl - EDA tool

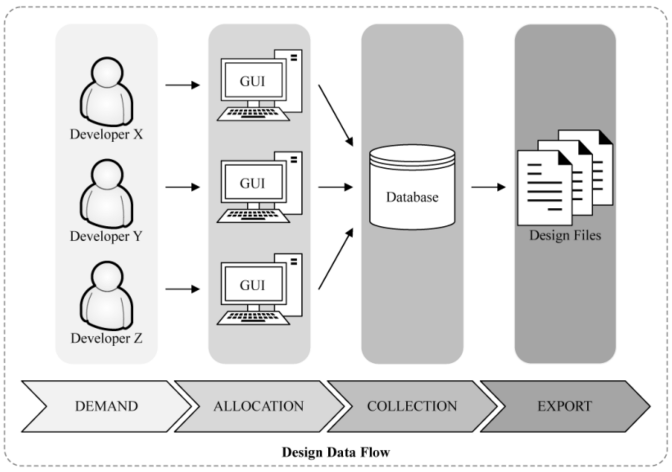
To begin, the work by (BASTL et al., 2022) introduces a register-based configuration tool for ASIC development, coupled with auto-generated pre-verified code. This research demonstrates the value of auxiliary tools within the EDA domain, as they contribute to error avoidance and enhance repeatability during the synthesis and subsequent *tapeout* processes. Using pre-verified RTL in the auto-generated design offers a substantial assurance of quality; nevertheless, it does not replace the need for system-level verification scenarios. Its flow is presented in the image 2.1 below.

However, it is important to address several drawbacks associated with this work. Firstly, the lack of industry support HDL languages is a notable limitation. Additionally, the absence of support for industry-standard interfaces, such as the AMBA set, raises compatibility concerns. Moreover, the lack of support for MPSoC generation and the omission of discussions on the flexibility for custom designs are noteworthy aspects that require further attention.

2.2 LiteX

In the works by Nguyen-Hoang et al. (2022) and Mosanu et al. (2022), both authors utilize the LiteX framework (KERMARREC et al., 2019) for the generation of their

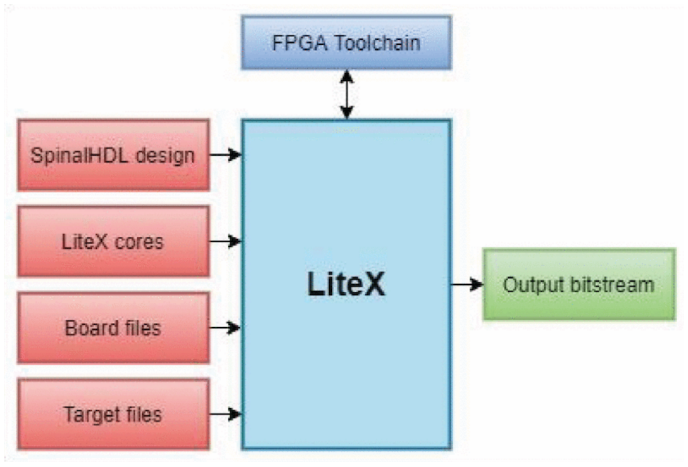
Figure 2.1: Design data flow



Source: (BASTL et al., 2022)

final designs. LiteX is a SoC builder framework developed in Python language aiming FPGA based systems supporting industry standard protocols and different sets of cores. Although it lacks support of MPSoC system generation.

Figure 2.2: The design flow of the LiteX framework



Source: (NGUYEN-HOANG et al., 2022)

In the first one previously mentioned, the author develops a prototype SoC for a Trusted Execution Environment (TEE) (NGUYEN-HOANG et al., 2022). The design is constructed using Spinal HDL (PAPON, 2023a), which is a domain-specific language (DSL) implemented as a library atop the Scala programming language, similar to Chisel (BACHRACH et al., 2012). Once the design is created, Spinal HDL generates a Verilog output, which is then integrated with the LiteX framework (Figure 2.2).

The second work presents an open-source emulation framework for Processing-in-Memory, implemented in System Verilog (MOSANU et al., 2022). This framework is integrated with LiteX, enabling communication between various IPs, such as DRAM and a RISC-V processor. Both works indicate that the framework used is flexible supporting different types of systems and input languages.

The framework's commendable flexibility in supporting various applications is apparent. However, it comes with an implicit requirement of comprehending the underlying Python toolbox Migen (SETETEMELA et al., 2019), which can result in a relatively steep learning curve, particularly for newcomers to hardware description languages (HDLs) and FPGA development. This challenge also extends to the integration of new IPs into the framework, as clear instructions for this process are lacking. Additionally, the generated design undergoes post-processing by the tool, even when incorporating a black-box IP wrapper, thereby limiting the user's ability to modify the RTL (Register-Transfer Level) by adding or removing components before commencing the synthesis flow.

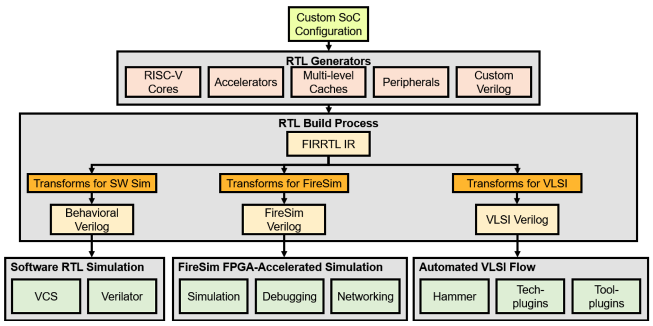
2.3 Chipyard

An additional noteworthy framework for consideration is Chipyard (2.3), as presented by Amid et al. (2020). This framework is characterized by its adaptable and open-source IP blocks, which are generated through a configurable and composable approach, making them applicable in various stages of hardware development while ensuring design intent and integration consistency. Chipyard further distinguishes itself by employing cloud-hosted FPGA (Firesim) accelerated simulation and rapid ASIC implementation, enabling seamless validation of custom systems with physical feasibility.

Unlike the previously mentioned works, Chipyard adopts a distinct approach in its front end, leveraging the Rocket Chip SoC generator (ASANOVIć et al., 2016) Figure 2.3, entirely developed in Chisel which is a library of Scala language. Its core methodology follows a hardware generator flow, where the design input is not a Verilog module but a description in a DSL. As elucidated by SiFive (2017), Chipyard incorporates the "Diplomacy" feature, a Scala-based parameter-negotiation framework that allows multiple IPs to negotiate compatible presets of configurations.

This framework provides a wide range of options for the users to configure and tweak the final design intention. Such flexibility comes at the cost of increased complexity. Developers who are new to the framework might find it challenging to navigate through the intricate configurations and integration options. Similar to LiteX, the learning curve associated with Chipyard can be a significant hurdle, especially for individuals with limited experience in hardware design or those accustomed to more straightforward SoC development methodologies.

Figure 2.3: Chipyard flow



Source: Adapted from Chipyard online documentation

Chipyard, being a highly configurable framework, offers the significant advantage of tailoring SoCs to specific requirements. However, this level of flexibility can also result in the inefficient allocation of hardware resources. In intricate designs, the resulting SoCs may consume more resources than necessary, thereby impacting performance and area utilization, and ultimately increasing the implementation costs.

Furthermore, Chipyard's extensive features and configurability contribute to a verbose and time-consuming generation process. Creating complex SoCs with numerous customizations may entail prolonged compilation times, impeding the development workflow and hindering rapid prototyping.

Additionally, akin to LiteX, the final design generated by Chipyard is obfuscated, often necessitating its consideration as a black box, comparable to a *netlist* produced by the FIRRTL tool (IZRAELEVITZ et al., 2017). This limitation restricts the extensibility of the final RTL and may require repeated hardware-flow generations when making design updates. Moreover, it can complicate the user's description of synthesis constraints, as the hierarchical path of the design may undergo complete changes with each iteration of the tool. For instance, if a sub-block needs to be set as a multi-cycle path, the user must delve into the generated RTL to discern how the tool produced such code, adding complexity to updating the constraints accordingly.

2.4 Comparison between frameworks

Table 2.1 presents a comprehensive comparison of three distinct frameworks discussed in this chapter, along with the proposed framework and its intended achievements. Among the frameworks analyzed, only (BASTL et al., 2022) currently demonstrates the capability to generate documentation, a feature yet to be achieved by the other works.

Table 2.1: Comparison between the different frameworks and their features

Framework	(BASTL et al., 2022)	LiteX	Chipyard	This work
Doc. generation	Yes	No	No	No
HDL language	VHDL	VHDL/ System/Verilog/ nMigen/ Spinal-HDL	Verilog/Chisel	System/Verilog
IP I/F	Custom	Industry + Open source	Industry + Open source	Industry
Base language	C++ / Qt	Python	Scala	Python
Custom designs	Not mentioned	Yes	Yes	Yes
MPSoC support	No	No	Yes	Yes
SW support	Yes	No	Partial	Yes
Design model generation	Not mentioned	Yes	Yes	Yes

Source: The Author

Regarding the hardware description language (HDL) support, three out of the four works provide compatibility with Verilog, while (BASTL et al., 2022) is limited to VHDL. Moreover, both the proposed framework and LiteX support System Verilog, are widely used in the ASIC industry.

Concerning IP interfaces, in this work, LiteX, and Chipyard accommodate industry-standard interfaces, while the open-source frameworks mentioned, such as LiteX with Wishbone and Chipyard with TileLink, are limited to specific interfaces. This work primarily focuses on the AXI4 standard, given its prevalence in commercially available IPs, allowing interchangeability with AHB or APB if needed.

In terms of the base language, only this work and LiteX were developed in Python, whereas Chipyard employs Scala as its base language. Nearly all frameworks facilitate custom designs, except for (BASTL et al., 2022), where such support remains undisclosed. Evaluating MPSoC support, both this work and Chipyard provide solutions, with software support offered through headers, APIs, or libraries also available in (BASTL et al., 2022), except memory map generation in Chipyard. However, most of its open-source IPs are widely accessible on the internet for reuse by new developers.

Regarding model generation, the majority of frameworks can produce simulation-ready models through EDA tools, except (BASTL et al., 2022). The three frameworks supporting models adopt a similar approach where a C++ output is generated from the final set of Verilog files using Verilator.

2.5 Conclusion

Throughout this chapter, three distinct frameworks have been introduced and analyzed, delving into their unique characteristics, advantages, and drawbacks. Each framework's attributes were examined to gain a comprehensive understanding of their applicability and potential limitations while considering the design generation context. By conducting a comparative analysis, we aimed to discern how these existing frameworks stack up against the proposed solution.

3 BASE IP SET FOR THE PROPOSED PLATFORM

This chapter constitutes a critical component of this Master's work, as it details the fundamental building blocks required for constructing the proposed platform. The chapter will present each of these building blocks in detail, highlighting their significance within the overall framework. All of the intellectual properties (IPs) discussed in this section were designed by the author to achieve a superior level of flexibility, modularity, and industry compatibility, and to address the common limitations encountered with open-source IPs found on the internet like non-standard protocols, low integration support, and domain-specific design languages.

Additionally, the design choices made for each sub-block and its respective micro-architecture will be thoroughly explained, to provide a comprehensive understanding of the platform's underlying structure. Finally, a comprehensive list of all available designs that comprise the framework will be presented, serving to demonstrate the building blocks of the system in its entirety.

3.1 RaveNoC Network-on-Chip

Network-on-Chip (NoC) is one of the fundamental blocks of the MPSoC, as it provides a communication infrastructure that is flexible, extensible, and can be also energy-efficient (if enabled by DVFS). It can support a variety of communication patterns, including *multicast*, *broadcast*, and point-to-point communication.

As mentioned by Wolf, Jerraya and Martin (2008), the concept of a Network-on-Chip (NoC) involves the utilization of a hierarchical network consisting of routers to enhance the efficiency of packet transfer between initiators and targets. NoC employs a communication infrastructure that differs from traditional shared-bus systems, by providing additional resources to enable concurrent communication across multiple channels. This approach aims to alleviate the energy and performance inefficiencies commonly associated with shared-medium bus-based communication methodologies.

As a result, the adoption of NoC has become increasingly prevalent in the design of MPSoC as it provides an efficient and scalable solution for managing the communication traffic within complex and heterogeneous SoC, the benefits of utilizing an NoC also includes increased performance, lower energy consumption. Furthermore, it introduces support for heterogeneous communication architectures, where distinct Network Interfaces (NIs) can incorporate bus bridges to facilitate communication between high-speed tiles (AXI) and low-speed ones (APB).

In summary, NoC has emerged as an essential component of MPSoC design due to its ability to provide efficient and *versatile* interconnects. Its modular and symmetric architecture supports a variety of communication patterns, which makes it an attractive solution for multi-processor systems. Furthermore, NoC offers better performance and scalability than traditional bus-based interconnects, which suffer from limited bandwidth and high latency as the number of processing elements increases.

The NoC design presented in this work is called RaveNoC, this is a configurable HDL for mesh NoC topology that allows the designer to change different parameters and setup many configurations. The list of features of the RaveNoC are listed down below here.

- Mesh topology (2D-XY)
- Valid/ready flow control
- Switching: Pipelined wormhole
- Virtual channel flow control
- Slave I/F AMBA AXIv4
- Different IRQs that can be muxed/masked individually

As the chosen language by the author, the RaveNoC design was written using the System Verilog IEEE 1800 standard (IEEE. . . , 2018b), which is highly adopted in the ASIC industry for Digital Design and UVM Verification (HOSNY, 2022). Thanks to its flexibility, it enables the usage of **packages**, which allows the inclusion of *structs* and *parameters* to easily change the hardware behavior across different modules without re-written the interfaces between each hierarchy *top-down*.

3.1.1 Parameters to tune

During the design development, some of the project requirements can change as part of the DSE, thus having a configurable architecture is likely an advantage to the Digital RTL engineers who can tune and test different sets of combinations of configurations. The design of the presented NoC was thought to be used in the same fashion with enough flexibility to drive different, workloads, hence the list of configurable parameters of the design is presented in the table 3.1.

Table 3.1: Configurable parameters of the RaveNoC HDL design.

Parameter	Min	Max	Default
Flit/AXI data width	32	128	32
Number of buffers in the input module	1	4096	3
Number of virtual channels	1	32	3
Order of priority in the VCs	0H	0L	0H
Dimensions of the NoC (Rows_X_Cols)	1x2	1000x1000	2x2
Routing algorithm	XY	YX	XY
Maximum size of packets	1	256	256

Source: The Author

3.1.2 Network Architecture

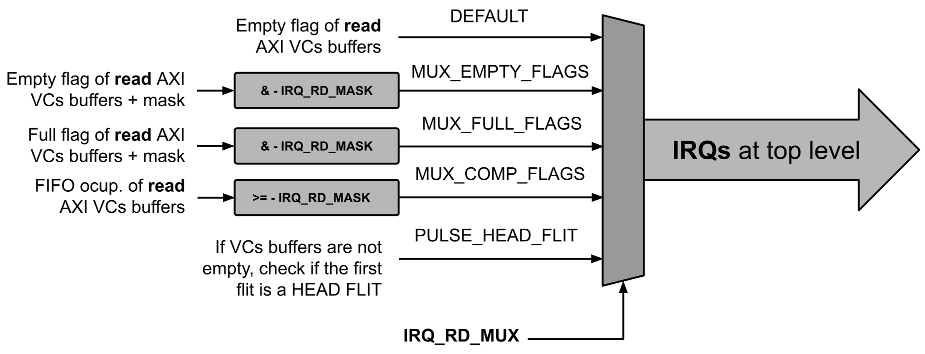
The network architecture of RaveNoC follows the classical structure as presented in Zeferino and Susin (2003) and Micheli et al. (2006), Zhang et al. (2010), with a 2-dimensional network of tiles. Each network router is composed of an input controller with different buffers for multiple virtual channels (VC), and a routing algorithm (XY) design, also we have the output controller with one round-robin arbiter per VC.

Like what was described by Zhang et al. (2010), the switching mechanism implemented by RaveNoC is wormhole, where each packet has a head, body, and tail *flit*. The head flit contains information about the X and Y destination router and the packet width aside from reserved bits for sort messages i.e. smaller than a single *flit*. As soon as a head *flit* reaches a router it blocks its path (through the correspondent VC) till a tail *flit* crosses the router.

When the *flits* arrive at the destination router, they are placed into the correspondent VC channel buffer. As part of the network interface, an AXI4 Slave interface is used to read and write all packets from and to the network correspondingly, this design module is shown in Figure 3.2. Besides the previously defined function, the AXI interface is also used to control the CSRs such as the ones responsible for handling the IRQ of the NI. The NI module has a single interrupt and we have multiple ways of masking and multiplexing its source with different options such as the ones presented in Figure 3.1.

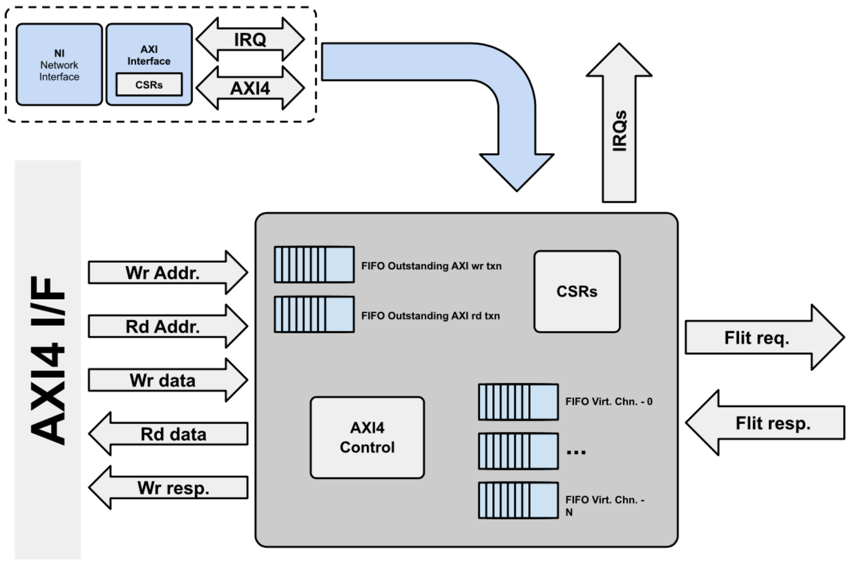
In Figure 3.3, we have a *2x2* Mesh 2D RaveNoC example design with the connections between the routers only. For a single router, there are four input and four output interfaces with an additional one for the local *flits* that are transferred to the NI which

Figure 3.1: NI IRQ logic


Empty flag of read AXI VCs buffers
DEFAULT
Empty flag of read AXI VCs buffers + mask
& - IRQ_RD_MASK
MUX_EMPTY_FLAGS
Full flag of read AXI VCs buffers + mask
& - IRQ_RD_MASK
MUX_FULL_FLAGS
FIFO ocup. of read AXI VCs buffers
>= - IRQ_RD_MASK
MUX_COMP_FLAGS
If VCs buffers are not empty, check if the first flit is a HEAD FLIT
PULSE_HEAD_FLIT
IRQ_RD_MUX
IRQs at top level

Source: The Author

Figure 3.2: Network Interface with AXI I/F



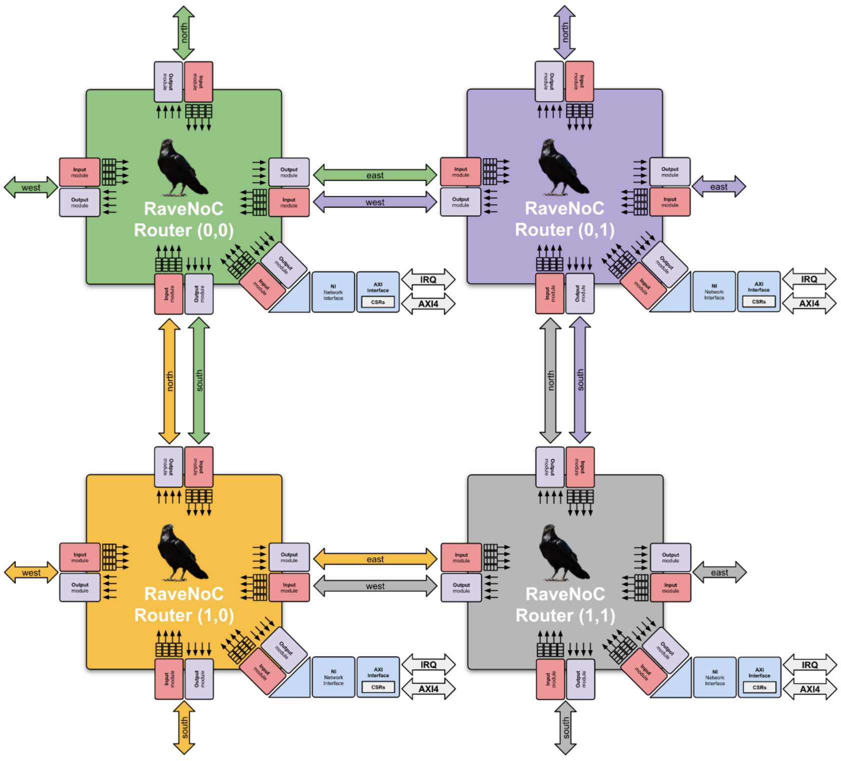
Source: The Author

has configurable buffers (FIFO) to store and forwards packets that will be further processed. The RaveNoC design is available on the internet at the author's GitHub repository (SILVA, 2023f).

3.2 NoX RISC-V core

Considering that, in general, all PEs (Processing Elements) of homogeneous MPSoCs will share the same design, a compact but fast processor can be an interesting choice for the *tile* architecture of the system. Furthermore, the workload usually executed by PEs

Figure 3.3: RaveNoC Mesh 2x2 example



Source: The Author

(*slave tiles*) tends to be small in comparison with the workload running on the *master tile*, as the slave tiles do not need to be responsible for splitting or merging the data that is being processed as mentioned by (AZAD et al., 2019).

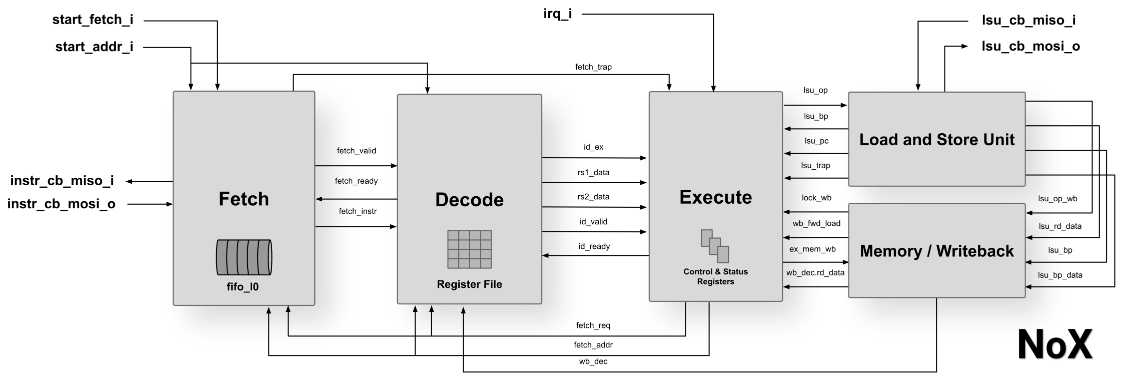
Numerous frameworks and platforms in the literature have been using open source ISAs (Instruction Set Architectures) to build the core tile of their MPSoC, such as Open-RISC by (CARARA et al., 2009), MIPS by (AGUIAR et al., 2014) and RISC-V by (EL-MOHR et al., 2018), (KAMALELDIN et al., 2019). Among these, RISC-V has emerged as the dominant open source ISA, with adoption by a vast majority of the open source community, whereas there are over hundreds of diverse implementations of this architecture freely available on the internet. The ISAs are open-source, which, as described by Asanović and Patterson (2014), confers certain benefits:

> - Greater innovation via free-market competition from many more designers, including open vs. proprietary implementations of the ISA.
> - Shared open core designs, which would mean shorter time to market, lower cost from reuse, fewer errors given many more eyeballs3, and transparency that would make it hard, for example, for government agencies to add secret trap doors.
> - Processors becoming affordable for more devices, which helps expand the Internet of Things (IoTs), which could cost as little as $1.

As noted in (WATERMAN et al., 2014), RISC-V is a versatile open-source instruction set architecture (ISA) that supports a range of extensions, including vector processing, single/double floating-point precision, atomic instructions, and other features. According to Kurth et al. (2017), due to its modularity, a RISC-V processor can also be very small using only the base subset of integer instructions which leads to a good candidate for a PMCA (programmable many-core accelerators).

Building on the aforementioned features of RISC-V, this study employs the ISA to develop a compact central processing unit (CPU) that supports the base I (Integer) extension of the standard. The resulting core is referred to as the NoX RISC-V RV32I CPU and is characterized by a small area footprint, with industry-standard interfaces such as AMBA AXI4 on both instruction fetching and load-store unit. The design was written using the System Verilog IEEE 1800 standard (IEEE. . . , 2018b), with a few set parameters to configure its design. Basic macros can change the behavior of the design by selecting between *synchronous* vs *asynchronous* and *active-low* vs *active-high* reset, also it is possible to enable a verbose debug log in case the user wants to extract more information of the different stages of processing within the core.

Figure 3.4: NoX CPU diagram



Source: The Author

The NoX core is a four-stage, single-issue, in-order pipeline with full bypassing, indicating that any data hazards will not result in penalty-delaying cycles. The only circumstance in which a stall may occur is if the core experiences back-pressure from the LSU (Load-and-Store Unit) or Instruction Fetching as a result of an ongoing on-the-fly operation. This design follows a micro-architecture similar to that of the classic five-stage pipeline CPU, as described by Patterson and Hennessy (2017), except the last two stages (memory and write-back), which have been consolidated into a single stage, as seen in Figure 3.4. The NoX CPU design is available on the internet at the author's GitHub repository (SILVA, 2023e).

The current version of this CPU does not provide caches. This decision is based on the class of applications foreseen. The application program will fit in the available program memory while the data are transferred through the communication channel over the NoC. Another rationale for not implementing caches is to restrict the scope of development to reduce its complexity and to fit on the prototyping platform.

3.2.1 NoX area

In the context of MPSoC design, having a small CPU design can provide significant benefits. Firstly, a smaller CPU design typically has lower complexity, which translates to reduced design time and lower development costs. Additionally, smaller digital designs generally consume less power, which is a crucial factor in embedded systems where power consumption is a significant concern. This can lead to reduced heat dissipation requirements and longer battery life in portable devices. A breakdown of the core's size is presented in Table 3.2 while synthesized for the FPGA XC7K325TFFG676-1@100MHz using Vivado 2022.1.

Table 3.2: NoX breakdown area report.

Name	Slice LUTs	Slice Registers
u_nox (nox)	2517	1873
u_wb (wb)	32	33
u_reset_sync (reset_sync)	1	2
u_lsu (lsu)	538	105
u_fetch (fetch)	276	134
u_fifo_l0 (fifo)	259	68
u_execute (execute)	229	359
u_csr (csr)	187	255
u_decode (decode)	1445	1240
u_register_file (register_file)	615	1056

Source: The Author

As for comparison, the VeX RISC-V processor (PAPON, 2023b) area in FPGA is around 1935 LUTs and 1216 FFs while the Ibex (KREMER et al., 2023) base design the base Ibex architecture uses 4645 LUTs and 3138 FFs.

Moreover, the utilization of multiple small CPU designs within an MPSoC can provide a significant degree of flexibility and scalability in system design. Employing different core designs for distinct processing tasks allows for more efficient resource allocation and improved performance. Furthermore, the integration of a compact CPU design can facilitate the creation of specialized tiles, where the majority of the tile's area size is attributed to the set of accelerators embedded within it. This approach enables optimization for specific applications, resulting in improved system performance and efficiency.

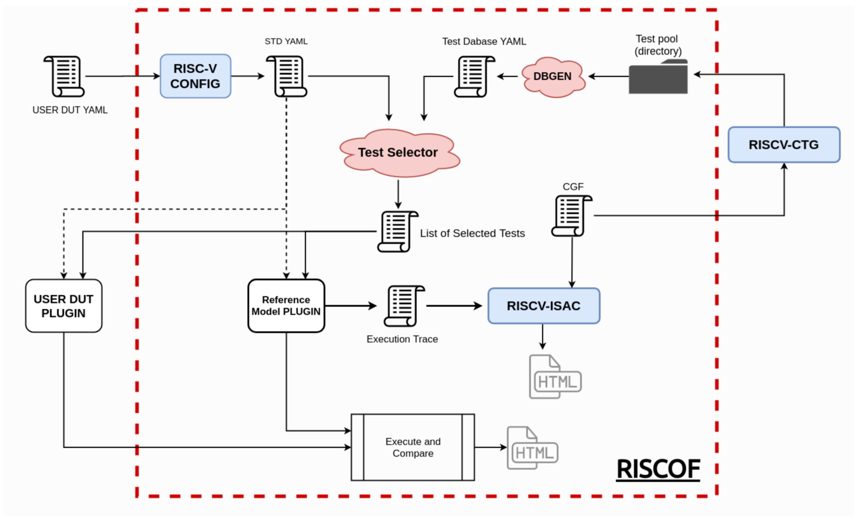
The rationale behind incorporating a RISC-V compliant core into this work was also driven by the author's appreciation of the benefits derived from the author's knowledge of the subject. This familiarity empowers the author to meticulously refine the core's design by selectively incorporating or omitting additional features in alignment with the specific requisites of the platform. For instance, the inclusion of supplementary Control and Status Registers (CSRs) can be undertaken, the pre-fetch buffer size can be modulated based on *benchmarking* outcomes, and provisions can be made for the integration of custom opcodes available within the ISA. This degree of adaptability ensures the core's capacity to be tailored to accommodate a spectrum of demands and to optimize its performance by divergent requirements.

3.2.2 RISC-V Compliance tests

To attain RISC-V CPU compliance, each core must complete a suite of tests documented in Gala (2023a), one of the official RISC-V repositories. The RISC-V Architectural Tests constitute a dynamic set of tests that aim to ensure that software developed for a specific RISC-V Profile/Specification is compatible with all implementations that comply with that profile. Moreover, these tests serve to verify that the implementer has comprehended and executed the specification accurately. It is noteworthy that the RISC-V Architectural Test suite serves as a basic filter, and obtaining approval by RISC-V International for the test results is a prerequisite for licensing the RISC-V trademarks related to the design.

Beneath the scope of the compliance tests, lies the RISC-V Compatibility Framework, referred to as RISCOF. It is a Python-based software framework designed to streamline the testing process for RISC-V targets, comprising different types of implementa-

Figure 3.5: RISCOF framework diagram



Source: (GALA, 2023b)

tions, using a standard RISC-V golden reference model, utilizing a comprehensive suite of RISC-V architectural assembly tests. As illustrated in Diagram 3.5, the framework necessitates two specific inputs from the user: a RISC-V-CONFIG based YAML specification of the ISA choices made by the user, and a Python plugin that can be employed by the framework to compile, simulate and extract the signature of each test.

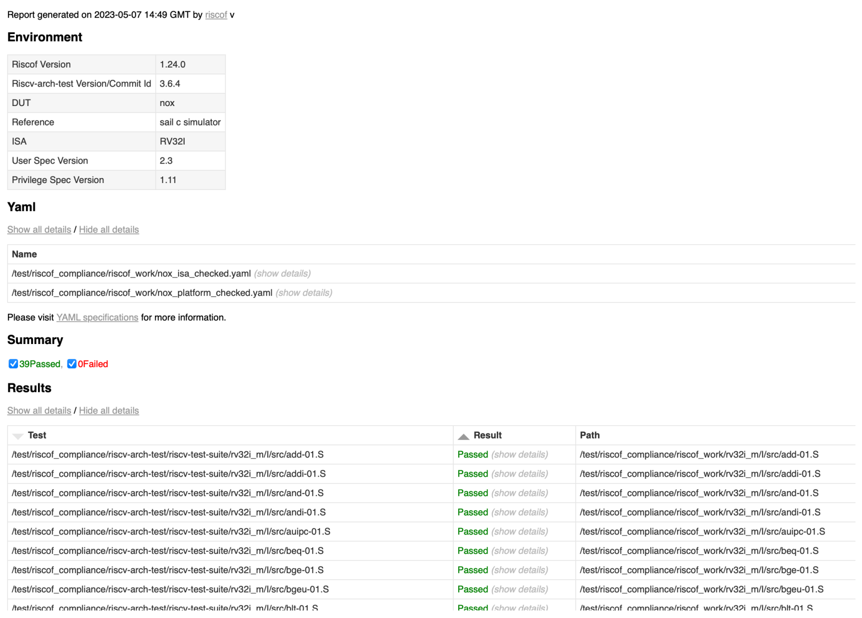
In the case of the NoX CPU, both sets of files were supplied, and all compliance tests were completed. After the compliance run, a report in HTML format was generated. The report depicting all successful tests for the NoX CPU can be observed in Figure 3.6.

3.2.3 FreeRTOS port

FreeRTOS recognized as one of the most widely used open-source Real-Time Operating Systems (RTOS), is specifically tailored for embedded systems. Its real-time kernel incorporates task scheduling, intertask communication, and resource management functionalities, rendering it an optimal choice for applications that demand precise timing and efficient resource allocation.

Acknowledging the existence of numerous applications and APIs already developed for FreeRTOS, a strategic decision was made to generate the essential set of files required for porting the NoX CPU to support FreeRTOS (GCC_RISC_V). Consequently, a repository was established, encompassing NoX running FreeRTOS with multiple tasks,

Figure 3.6: RISC-V compliance tests (RISCOF)



Report generated on 2023-05-07 14:49 GMT by riscof v

Environment

Riscof Version	1.24.0
Riscv-arch-test Version/Commit Id	3.6.4
DUT	nox
Reference	sail c simulator
ISA	RV32I
User Spec Version	2.3
Privilege Spec Version	1.11

Yaml

Show all details / Hide all details

Name
/test/riscof_compliance/riscof_work/nox_isa_checked.yaml *(show details)*
/test/riscof_compliance/riscof_work/nox_platform_checked.yaml *(show details)*

Please visit YAML specifications for more information.

Summary

✓ 39Passed, ✓ 0Failed

Results

Show all details / Hide all details

Test	Result	Path
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/add-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/add-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/addi-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/addi-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/and-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/and-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/andi-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/andi-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/auipc-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/auipc-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/beq-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/beq-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/bge-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/bge-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/bgeu-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/bgeu-01.S
/test/riscof_compliance/riscv-arch-test/riscv-test-suite/rv32i_m/I/src/blt-01.S	Passed *(show details)*	/test/riscof_compliance/riscof_work/rv32i_m/I/src/blt-01.S

Source: The Author

serving as the foundational template for subsequent firmware development on this platform. The repository is conveniently accessible on the GitHub website (SILVA, 2023d).

3.3 AXI DMA

DMA (Direct Memory Access) is an essential component presented in different MPSoC designs. DMA provides a mechanism for high-speed data transfer between memory and peripheral devices without requiring CPU intervention (MA; HE, 2009). DMA allows for efficient data transfer between peripherals and memory, freeing up CPU resources for other processing tasks. This is particularly important for multi-processor systems, where each tile needs to move data through the NoC and such a task can be deployed to the engine.

In addition to offloading data transfer tasks from the CPU, DMA can also improve system performance by reducing latency and improving data throughput. By allowing peripheral devices to directly access memory, such design can enable faster data transfers compared to traditional CPU-mediated data transfer methods. Likewise, DMA can play a key role in reducing power consumption in MPSoC designs. By reducing the number of CPU cycles required to perform data transfer operations, it can reduce overall system power consumption, which is a significant concern in embedded systems where power consumption is a critical factor.

A crucial aspect of incorporating DMA within the MPSoC relates to its ability to facilitate the transfer of data from NoC through the network interface (NI) to local memory for processing. This requirement is particularly common in streaming applications which are often processed by MPSoCs. A dedicated DMA design can be utilized to perform this data transfer task, relieving the tile CPU from the responsibility of data movement. This results in improved efficiency as the Tile CPU can focus on performing tasks related to image processing while the DMA engine is responsible for managing data transfer. Ultimately, by programming the DMA descriptors, the CPU can delegate the responsibility of all master access to the DMA engine, thereby streamlining the data transfer process.

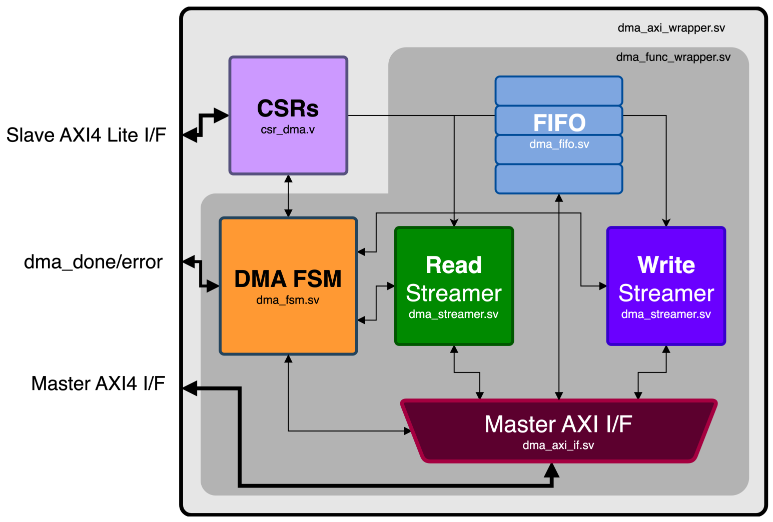
Building upon the prior discussion, the design developed for this platform exhibits the following characteristics:

- AXI4-Lite Slave interface to program the *CSRs* (Control and Status registers).
- AXI4 Master interface to fetch/read and write data.
- Support for unaligned *xfers*.
- Configurable number of descriptors (default to 2).
- Abort processing available.
- Transfers up to 4GB of data per descriptor.
- Two modes of data access, Fixed [FIFO] and Incremental.
- Programmable number of bursts, to support simpler slaves.
- Status of error during DMA operation.
- Configurable bus width option between 32-bit / 64-bit.

The micro-architecture of the AXI DMA is founded on conventional engines, devoid of superfluous features that would lead to increased area without benefit. The design is separated into two primary data flow paths: the read and write data paths.

The DMA FSM is responsible for managing the processing of all descriptors by the streamers and dispatching new ones as soon as each descriptor finishes the transfer. The streamers are tasked with breaking transactions into multiple AXI transfers, adhering to the protocol, such as the 4KB address boundary or the number of bursts per burst type (i.e., 16-FIFO/256-INCR). On the Master AXI I/F, the DMA waits for requests from the streamers and, if available, dispatches them through the corresponding address channel until the maximum of outstanding transfers, defined in the configuration files (read/write),

Figure 3.7: AXI DMA diagram



Source: The Author

is reached.

The FIFO serves as the primary buffer for the data that is read and written through the DMA and can be configured to any size in depth with a width equal to the AXI data bus width.

Regarding unaligned access, the DMA can be programmed with unaligned source and/or destination addresses. However, the support for unaligned access is limited to the bus width. The smallest transfer through the rd/wr channel must match the defined AXI bus width, and internal masking will be applied to match the source/destination addresses. Therefore, it is the user's responsibility to program the descriptors correctly considering symmetric alignment in the source/destination address. This design decision simplifies the interface between the modules, and fine-grained memory moving is typically performed by CPU instead of a DMA. The AXI DMA design is available on the internet at the author's GitHub repository (SILVA, 2023a).

3.3.1 Parameters to tune

Table 3.3 summarizes the parameters available for this engine.

Table 3.3: AXI DMA parameters

Parameter	Min	Max	Default
DMA_NUM_DESC	2	4096	2
DMA_ADDR_WIDTH	32	64	32
DMA_DATA_WIDTH	32	64	32
DMA_BYTES_WIDTH	32	64	32
DMA_RD_TXN_BUFF	2	64	8
DMA_WR_TXN_BUFF	2	64	8
DMA_FIFO_DEPTH	2	128	16
DMA_ID_WIDTH	1	256	1
DMA_MAX_BEAT_BURST	1	256	256
DMA_EN_UNALIGNED	0	1	1
DMA_MAX_BURST_EN	0	1	1

Source: The Author

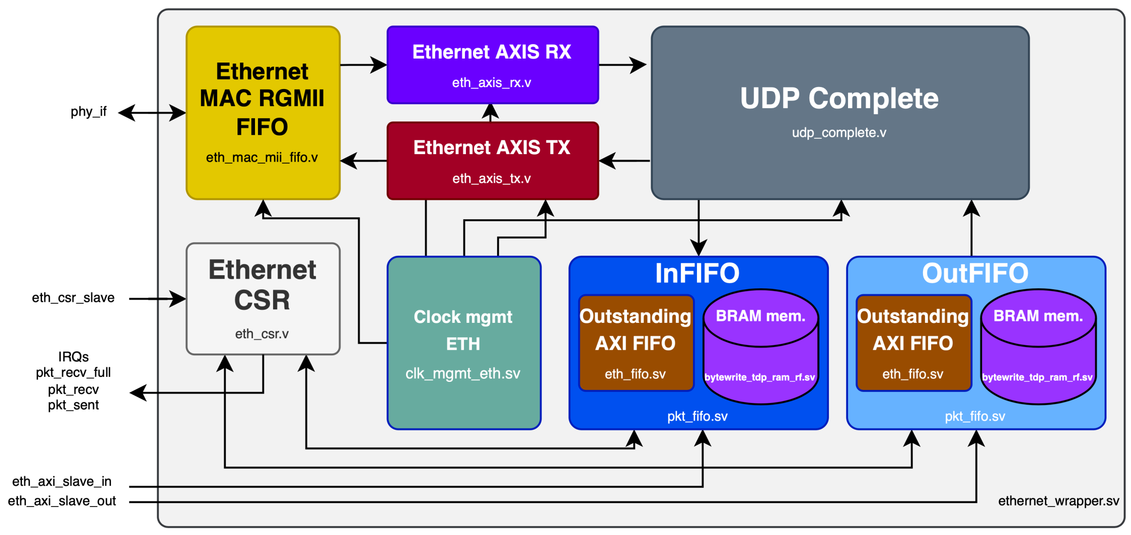
3.4 Ethernet AXI

Ethernet has become a crucial communication standard in contemporary electronic systems due to its high-speed and reliability capabilities, which can reach up to 100 Gigabit Ethernet (100 GbE). Incorporating Ethernet as a *gateway* in MPSoC architectures has proven advantageous in terms of cost-effectiveness, scalability, and performance. The integration of it provides the MPSoC with the ability to interact with other systems and devices, allowing seamless communication and data transfer. Its use also offers flexibility to connect with multiple devices through local area networks (LANs), providing the capability to operate in a diverse range of applications, including remote processing, and audio and video streaming, among others. In the study conducted Hasler et al. (2022), the authors introduced an MPSoC that was tailored and optimized to operate as an RLNC (Random Linear Network Coding) accelerator. The specific design of the MPSoC was customized to meet the unique requirements of an Ethernet router application, which serves as evidence that an MPSoC can be utilized to enhance networking processing tasks as well.

Given the context of the current study, incorporating an Ethernet accelerator serves as a proficient approach to facilitate communication with the MPSoC and to transmit data through the network for further processing. The block diagram depicted in Figure 3.8

illustrates all sub-blocks of the Ethernet AXI slave responsible for executing the lower portion of the Data Link layer, commonly known as the Media Access Control (MAC) ((IEEE. . . , 2018a)). This implementation enables the design to communicate with an external PHY physical layer Integrated Circuit usually available in development boards and connected to FPGA IOs such as Nexys Video from *Digilent*.

Figure 3.8: Ethernet AXI diagram



Source: The Author

The present design leveraged the Ethernet MAC RGMII FIFO, Ethernet AXIS RX/TX, and UDP complete designs from an open source project named *verilog-ethernet*, which is available on the GitHub repository website (FORENCICH, 2023a). While these blocks constitute a significant portion of the design, they cannot be directly connected to an AXI interconnect to be accessed as a slave by a tile CPU. To be controlled by a processor, the author implemented a configurable sub-block called *pkt_fifo* that can communicate through the standard AXI4 and handle the CDC (Clock Domain Crossing) between the interconnect and the UDP Complete that runs on different clock speeds. The two instances of *pkt_fifo* (InFIFO and OutFIFO) communicate through AXIS (AXI-Stream protocol) with the UDP Complete module to either send or receive UDP (User Datagram Protocol) packets over the Ethernet. The same sub-block acts as a local temporary buffer to store the packets as they are being sent or received through configurable BRAM (Block RAM) memories. As a design choice, the UDP transport protocol was chosen due to its lack of a handshake process and error checking that allows it to transmit data more quickly, making it a better choice for applications that prioritize speed over reliability.

Also, the design has three signals that can be used as interrupts by a processor that indicates when a packet has been received, a packet has been sent and when the FIFOs are full. As the user interface, a set of CSRs are available through an AXIL [AXI-Lite] connection, where different features of the Ethernet IP can be configured, such list includes:

- Source and Destination Ethernet MAC Address (6-bytes)
- Source and Destination IPv4 and gateway address (4-bytes)
- Sub-net mask (4 bytes)
- UDP packet length
- Source and Destination UDP port
- Read individual FIFOs pointers and its flags
- Clear and read interrupts
- Clear ARP table
- Filter for UDP port and IPv4 address

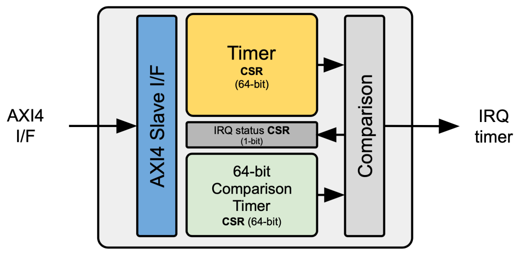
The previously mentioned features were achieved through the implementation of the Ethernet AXI design. The original design files did not provide the desired level of flexibility or include an AXI4 slave interface, nor did they incorporate the two sets of FIFOs. It is worth noting that the availability of open-source Ethernet IP is limited, with the majority of options being proprietary offerings from vendors such as *Xilinx* or *Intel/Altera*, which often require licensing for full access. Hence, the presented design emerges as a favorable choice owing to its extensive range of features and lack of dependency on proprietary licenses. Additionally, its open-source nature further strengthens its appeal, ensuring accessibility and fostering collaboration among developers. The Ethernet AXI design is available on the internet at the author's GitHub repository (SILVA, 2023b).

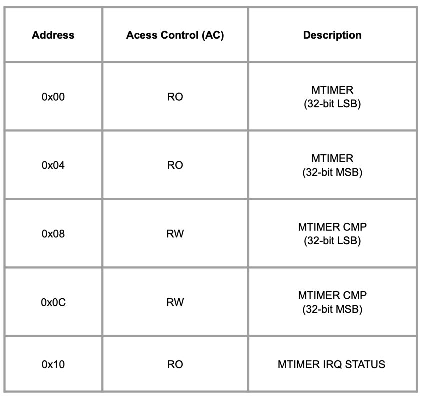
3.5 Programmable AXI Machine Timer

As part of the RISC-V privileged specification (Andrew Waterman; Krste Asanović; John Hauser, 2021), a platform has to provide a memory-mapped real-time counter denoted as *mtimer* that wraparound behavior in the event of count overflow. This counter is accessible through a memory-mapped machine-mode read-write register. It is noteworthy that the *mtimer* register must increment at a constant frequency, and the platform must incorporate a mechanism to determine the duration of an *mtimer* tick.

The significance of this counter is fundamental in the context of operating-system execution, as demonstrated by its utilization in FreeRTOS. Specifically, the standard RISC-V port implemented by FreeRTOS employs the aforementioned counter to facilitate the operation of the OS (Operating System) *tick* timer. The following block diagram presented in Figure 3.9 represents the design implemented by the author. An AXI4 Interface allows the RISC-V core to access the *mtimer* to read its value and to set the comparison CSR.

Figure 3.9: Programmable AXI Machine Timer





Address	Acess Control (AC)	Description
0x00	RO	MTIMER (32-bit LSB)
0x04	RO	MTIMER (32-bit MSB)
0x08	RW	MTIMER CMP (32-bit LSB)
0x0C	RW	MTIMER CMP (32-bit MSB)
0x10	RO	MTIMER IRQ STATUS

Source: The Author

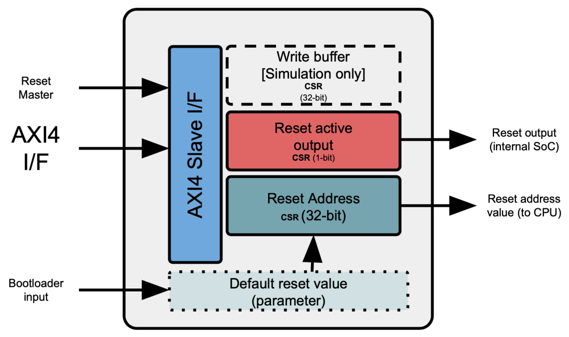
The design behavior is characterized by its simplicity, as it entails the continuous incrementation of a real-time counter, which is subsequently compared against a comparison timer CSR. Upon the *mtimer* counter value equating to that of the comparison CSR, an interrupt is activated, and subsequently linked to the RISC-V core as the machine timer interrupt. This AXI programmable timer design is part of the SoC components used in the IPSoCGen platform available on the author's GitHub repository (SILVA, 2023g).

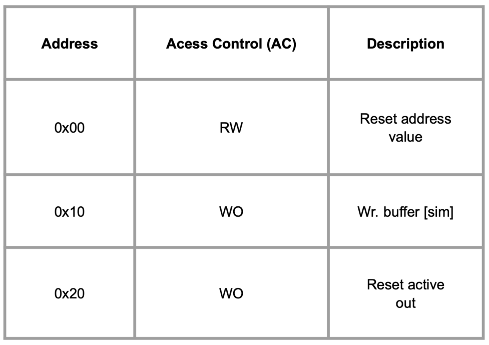
3.6 AXI Reset Controller

To facilitate the transfer of different programs to the RISC-V CPU, it is a widely adopted practice to incorporate a *bootloader* program within a ROM image. Once the program is successfully loaded into local memory, there is a need to modify the CPU's reset vector to initiate execution from the loaded program rather than the original boot ROM. Additionally, it is imperative to have control over the reset of the entire System-on-Chip (SoC) in the event of a detected hang, which may be triggered by a *WatchDog* or any other monitoring mechanism. In this section, it is presented a designed AXI Reset

Controller aimed at enabling the aforementioned functionalities.

Figure 3.10: Programmable AXI Reset Controller


Reset Master
AXI4 I/F
Bootloader input
AXI4 Slave I/F
Write buffer [Simulation only] CSR (32-bit)
Reset active output CSR (1-bit)
Reset Address CSR (32-bit)
Default reset value (parameter)
Reset output (internal SoC)
Reset address value (to CPU)



Address	Acess Control (AC)	Description
0x00	RW	Reset address value
0x10	WO	Wr. buffer [sim]
0x20	WO	Reset active out

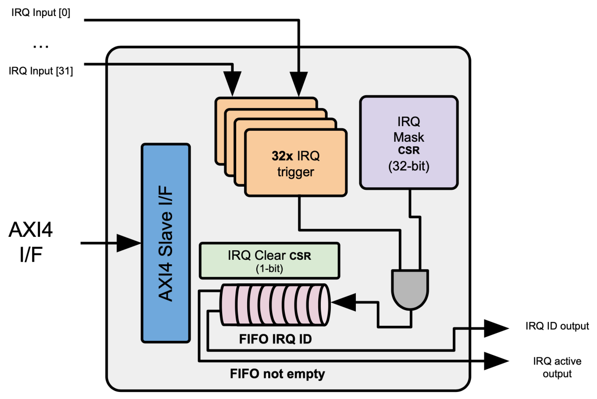
Source: The Author

As illustrated in Figure 3.10, the presented design comprises a reset master input responsible for resetting the internal registers of the controller. Moreover, it incorporates a *bootloader* input that configures the reset address register to its default value, which is determined by a parameter specified during its instantiation. Regarding the outputs, the design includes the reset address, which is routed to the RISC-V CPU, and the reset output, which is utilized for resetting the connected designs. The write buffer mentioned in the diagram is a *simulation-only* register added to the design to serve as a temporary buffer for printing characters into the terminal while simulating the RTL. The AXI reset controller design is part of the SoC components used in the IPSoCGen platform available on the author's GitHub repository (SILVA, 2023g).

3.7 AXI Interrupt Controller

Another essential design in SoCs is the interrupt controller, due to its significance in managing and prioritizing interrupt signals from different accelerators within complex integrated systems. By providing a centralized mechanism for interrupt management, IRQ controllers facilitate the timely and accurate processing of critical events, thereby enhancing the overall functionality, responsiveness, and real-time performance of SoCs as mentioned by Bargholz, Dietrich and Lohmann (2022). The design, implementation, and optimization of IRQ controllers are crucial research areas in modern SoCs development, as they directly impact the system's ability to handle interrupts effectively and meet the stringent requirements of diverse applications and industries.

Figure 3.11: AXI Interrupt Controller



Address	Acess Control (AC)	Description
0x00	RO	IRQ ID FIFO read
0x04	RW	IRQ Mask
0x08	WO	IRQ FIFO Clear

Source: The Author

Given that a majority of the IRQ controllers accessible on the internet are typically designed for specific platforms, lacking the necessary generic compatibility to cater to different architectures and their implementations, the decision was made to undertake the implementation of an IRQ controller that could effectively address this demand.

The design presented in Figure 3.11 shows the micro-architecture in a simplified version of the proposal. Such a design is composed of a configurable IRQ trigger module (defined at instance level as a parameter) that pre-processes the interrupts (level vs edge type). The output of all pre-processed interrupts is then filtered by a register mask, masking unwanted interrupts. Once masked, the interrupts are stored into a configurable depth FIFO to be then read by the CPU later. The FIFO stores the interrupt ID corresponding to the interrupt input vector position, with priorities fixed such that the first entry has the highest priority.

At its output, this FIFO exposes the latest unprocessed interrupt ID and provides a summary trigger output when it is not empty (usually connected to the CPU), indicating the presence of external interrupts that need to be processed by the CPU. When the CPU or any other bus master reads the ID from the Interrupt Controller, the FIFO increments its read pointer, thereby pointing to the subsequent ID for processing, if applicable. The AXI Interrupt controller design is part of the SoC components used in the IPSoCGen platform available on the author's GitHub repository (SILVA, 2023g).

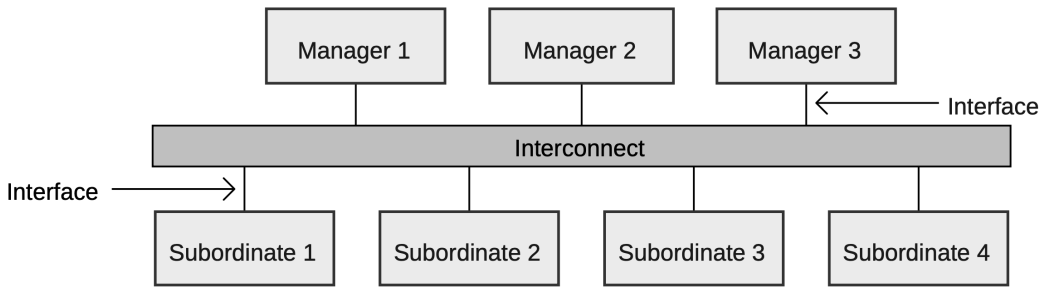
3.8 Additional design elements

Up to this juncture of Chapter 3, the presented work was based on specific designs that were created entirely by the author (with an expectation on 3.4). Nonetheless, to construct the ultimate platform, further design components are necessary and some of them are not entirely developed by the author, with the design being integrated from a different open-source contributor. Instead, the core of such designs is a 3PIP (third-party intellectual property) and a wrapper is employed to ensure complete compatibility of the design with the platform. All the following designs are part of the SoC components used in the IPSoCGen platform available on the author's GitHub repository (SILVA, 2023g).

3.8.1 AXI Crossbar/Interconnect wrapper

The AXI4 crossbar/interconnect is responsible for handling AXI transactions from different masters (managers) and slaves (subordinates) within the SoC. AXI4 was chosen because it is widely used in the industry for high-speed data transfer protocol as well as in various academic research areas, such as high-speed memory interface in (NOAMI et al., 2021), QoS for NoCs in (WANG; LU, 2020), and low-power firewall in (LIUBAVIN et al., 2022).

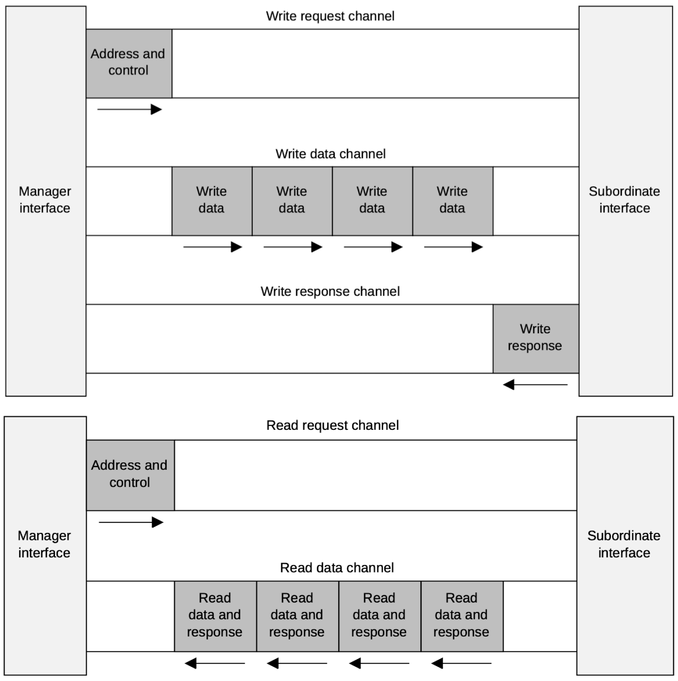
Figure 3.12: AXI Interface and interconnect



Source: (ARM, 2023)

For this work, it was implemented a *wrapper* using System Verilog constructions on top of the open-source AXI4 Crossbar design *verilog-axi*, which is available on the GitHub repository website (FORENCICH, 2023b). This design implements the architecture described in the Figure 3.12. Such AXI crossbar interconnect is characterized by its configurable data and address interface widths with support for all burst types. It operates in a fully *nonblocking* manner with separate read and write paths. The interconnect includes transaction ordering protection logic based on ID, along with per-port address decode, admission control, and decode error handling mechanisms.

Figure 3.13: Channel architecture of writes/reads



Source: (ARM, 2023)

The inclusion of the wrapper aims to simplify the integration of the crossbar with the aforementioned set of masters and slaves, particularly due to the AXI's complex nature with multiple channels and numerous interconnecting wires between the masters and slaves (3.13). By utilizing a System Verilog wrapper, a notable benefit is the ability to condense the multiple channels and their corresponding sets of wires into two simplified interfaces, namely MOSI (Master Output Slave Input) and MISO (Master Input Slave Output). This abstraction provides a more streamlined and manageable code perspective and it is less error-prone while editing the design.

3.8.2 AXI UART Slave

To enable debugging and program transfer to the RISC-V CPU, the addition of the AXI UART Slave serves as a significant accelerator within the SoC for the context of the platform discussed in this work. Building upon the *wbuart32* design available on the GitHub repository website (GISSELQUIST, 2023), a System Verilog wrapper was implemented. Since the design follows the AXIL (AXI Lite) protocol instead of AXI4, a bus-bridge design was also added to ensure protocol compatibility with the rest of the Interconnect.

3.9 Final considerations

In Chapter 3, a comprehensive overview of various designs was provided to facilitate a thorough comprehension of each element's functionality and their integration to assemble the proposed platform. Notably, the majority of these intricate designs were exclusively developed by the author, rendering them highly valuable assets for the platform. The author meticulously tailored each design element to seamlessly integrate and collaborate with the other components, further enhancing the overall cohesiveness and effectiveness of the proposed platform.

4 IPSOCGEN - SOC GENERATION METHODOLOGY

Given the objective of this research, which is to develop a platform capable of generating System-on-Chip (SoC) and Multi-Processor System-on-Chip (MPSoC) architectures generic for different application scenarios, the forthcoming chapter will provide a comprehensive overview of the construction process. Furthermore, it will elucidate the platform's inherent capabilities in terms of flexibility and extensibility, ensuring its adaptability to diverse requirements and potential future expansions.

4.1 Proposed flow

This work focuses specifically on a defined scope, encompassing the flow from system specification to the completion of RTL (Register Transfer Level) generation and its synthesis into a digital circuit. The presented IPSoCGen outlines a flow that is organized into three distinct phases, which are presented as follows:

1. Phase 1 (P1) - Application: specification, requirements, and profiling
2. Phase 2 (P2.a) - Design setup and generation (Hardware)
3. Phase 2 (P2.b) - Software/Firmware development
4. Phase 3 (P3) - System's validation

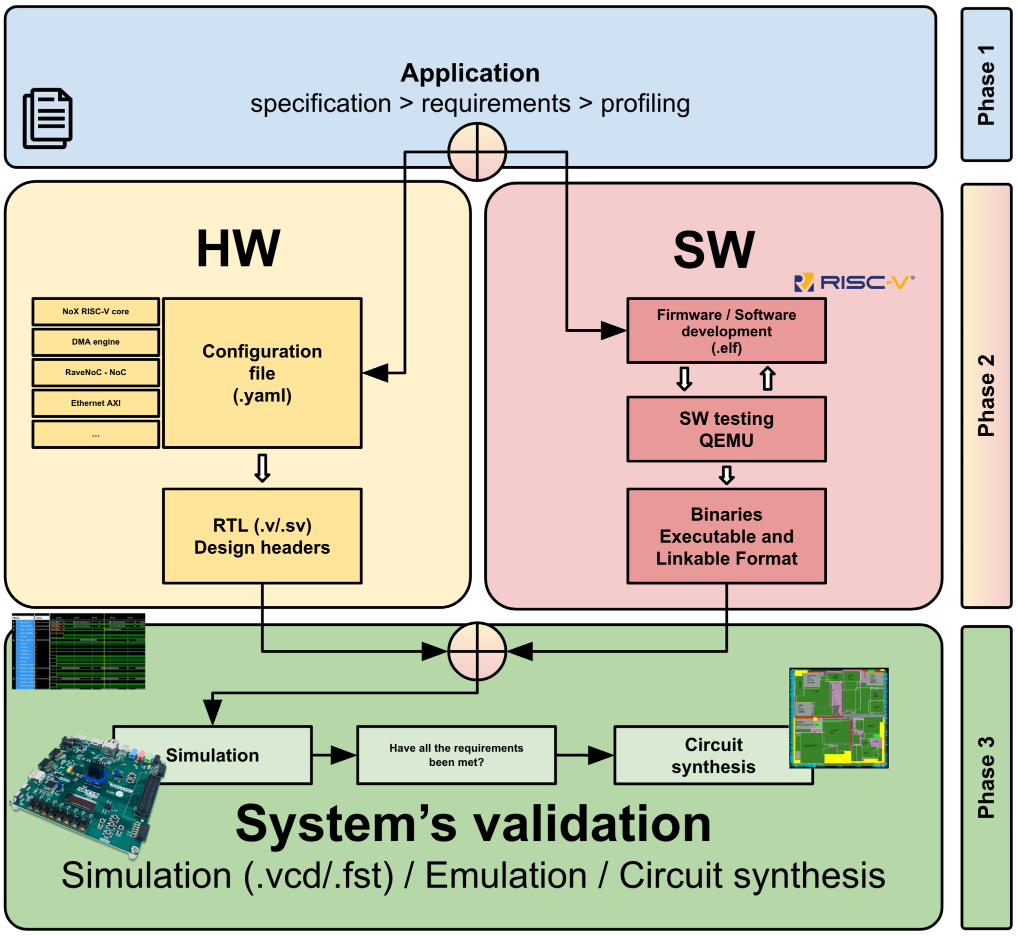
The diagram depicted in Figure 4.1 demonstrates the general idea of the flow while Figure 4.2 provides a comprehensive visual and detailed representation of the previously mentioned phases.

4.1.1 Phase 1 - Application: specification, requirements, and profiling

In the first phase (Phase 1), a thorough analysis of the application is conducted. This phase encompasses the definition of the application's specifications, requirements, and profiling. Its primary goal is to extract all the necessary information that will shape the completeness of the system.

Following the diagram, the "Application Specification" refers to a detailed description or set of requirements that outlines the specific functionality, behavior, and performance expectations of a software application or system. It serves as a blueprint or

Figure 4.1: IPSoCGen block diagram


Application
specification > requirements > profiling
Phase 1
HW
NoX RISC-V core
DMA engine
RaveNoC - NoC
Ethernet AXI
...
Configuration file (.yaml)
RTL (.v/.sv) Design headers
SW
RISC-V
Firmware / Software development (.elf)
SW testing QEMU
Binaries Executable and Linkable Format
Phase 2
Simulation
Have all the requirements been met?
Circuit synthesis
System's validation
Simulation (.vcd/.fst) / Emulation / Circuit synthesis
Phase 3

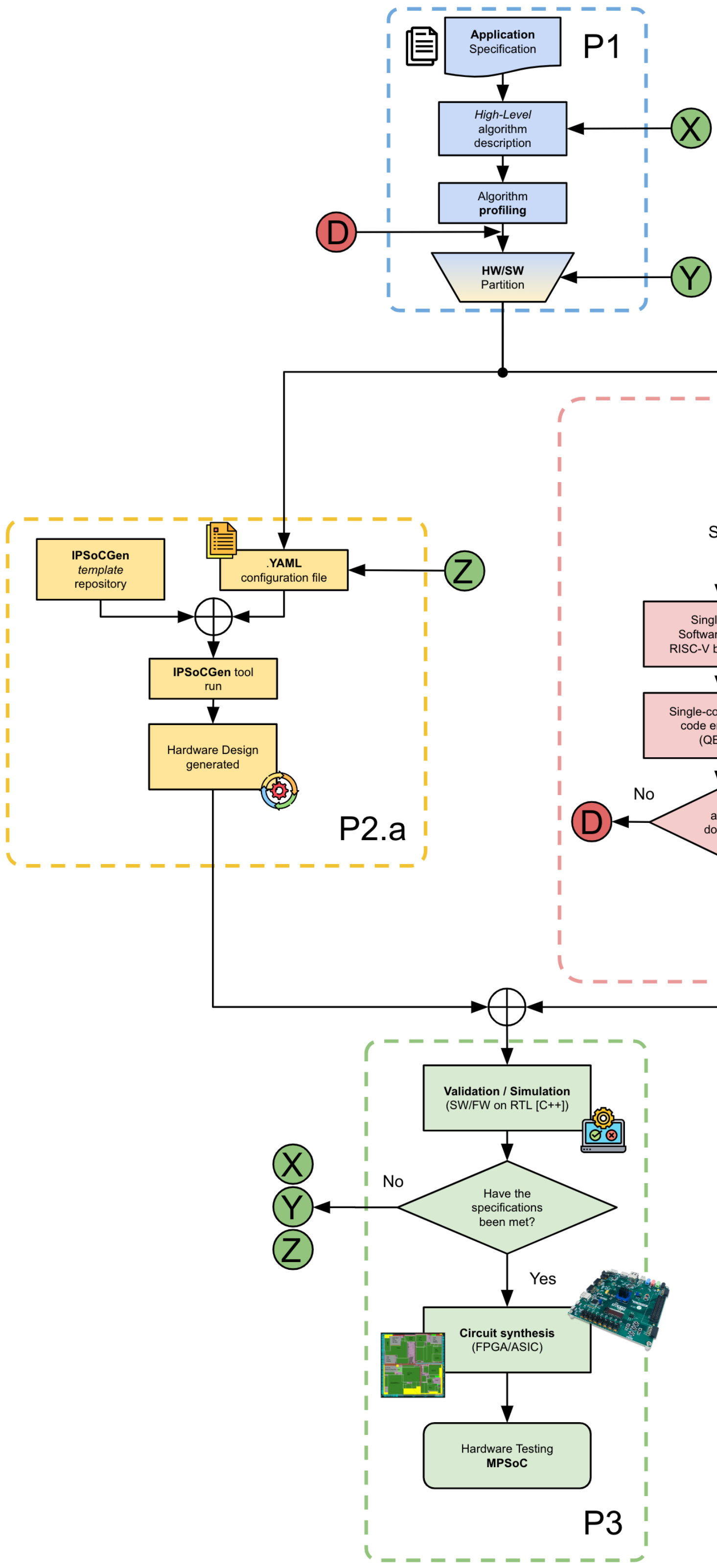
Source: The Author

guide for developers, designers, and stakeholders involved in the development process, for instance, it can be proposed as a flowchart with inputs and outputs in terms of what needs to be processed and distributed between different tiles (each unit of the MPSoC).

By the application specification, a "High-Level algorithm description" is required. This description can be defined in any high-level abstraction language, such as Python or Go, for instance. It serves as an initial modeling of the data flow, processing the inputs and generating the expected output after being processed by the MPSoC. The same algorithm can be employed to evaluate the alignment of key performance indicators (KPIs) with the application specification through the profiling of various inputs.

Taking into consideration the aforementioned example, during the "Algorithm profiling" phase, it would be utilized to assess the flow of splitting and merging image segments via a Python script, examining the potential impact of processing overhead on the desired final throughput.

Figure 4.2: IPSoCGen general flow


Application Specification
P1
High-Level algorithm description
X
Algorithm profiling
D
HW/SW Partition
Y
SW flow
SoC
MPSoC
SoC type
IPSoCGen template repository
.YAML configuration file
Z
Single-core Software design RISC-V binary (.elf)
Multi-core Software design RISC-V binary (.elf)
IPSoCGen tool run
Single-core RISC-V code emulation (QEMU)
Multi-core RISC-V code emulation (QEMU)
Hardware Design generated
No
No
app done?
app done?
D
D
P2.a
Yes
App (binary) ready for validation (.elf)
P2.b
Validation / Simulation (SW/FW on RTL [C++])
X
No
Y
Have the specifications been met?
Z
Yes
Circuit synthesis (FPGA/ASIC)
Hardware Testing MPSoC
P3

Source: The Author

Finally, after the application has been thoroughly scoped, the hardware and software partitioning process determines what should be offloaded to hardware as an accelerator and what should be processed in software, aligning with the initial specification and requirements.

Phase 1 (P1) forms the basis for the concurrent hardware phase (P2.a) and software development (P2.b). In the hardware domain, it molds design aspects like memory size, the count of master/slave components, bus width, and more. Meanwhile, in the software domain, P1 serves as the benchmark for defining the expected system behavior, input and output formats, and the software's responsibilities in processing the application.

4.1.2 Phase 2.a - Design setup and generation (Hardware)

Phase 2.a (P2) constitutes the core of this project, focusing on the conversion of specifications into a tangible design. In Phase 1, users were tasked with delineating which tasks would be executed on hardware (HW) and which would be designated for software (SW). Having established the hardware obligations, the next step involves translating these hardware requirements into a configuration file in YAML format. This translation process must consider the array of available IPs provided by IPSoCGen, along with their associated configuration options. Since most of these IPs offer a range of configurations, users must carefully evaluate the requirements to select the most suitable combination. This selection should aim to optimize the resulting hardware design in terms of performance, area, and power efficiency.

The hardware generation process offers significant flexibility, enabling the creation of diverse configurations through the use of multiple YAML files that cater to various development requirements. As an illustration, in the early stages of system development, debug flags are activated when compiling all *firmwares*, leading to increased memory demands. Hence, the ability to tweak a parameter or two or multiple YAMLs for obtaining a fresh design proves to be exceptionally advantageous.

After the user creates the configuration file in YAML format, the subsequent step, referred to as the "IPSoCGen tool run," involves the tool's operation to generate the hardware design. In conjunction with this process, the "IPSoCGen *template* repository" serves as a crucial component. It provides the necessary templates and resources essential for the tool to create the design based on the IPs outlined in Chapter 3.

In case a new IP that is not part of the library set is required, the user has the flexibility to indicate through the configuration file, a *custom* IP master or slave, that later will be translated into a *wrapper*. This wrapper will be the starting point for the user to start designing its new IP or to copy or adapt from the ones available on the internet.

As the last step in this phase, the "Hardware Design Generated" defines the outputs that are created by the IPSoCGen tool. The output generated by the tool encompasses System Verilog Register Transfer Level (RTL) designs and header files (.h) utilized by software components. The RTL files provide a comprehensive hardware description of the entire design in a format that is easily understandable to humans. This facilitates the ability of users to make edits to the files if desired, including the addition of custom modules or integration of new intellectual properties (IPs) within another wrapper, should the need arise. In the end, all these newly generated files will be transferred to the user's development repository, which can either be a fork or a copy of the template repository provided by IPSoCGen.

4.1.3 Phase 2.b - Software/Firmware development

For the Phase P2.b (P2), it is presented the software flow. In this phase, we have the development of the different binaries either SoC or MPSoC depending on the user's project.

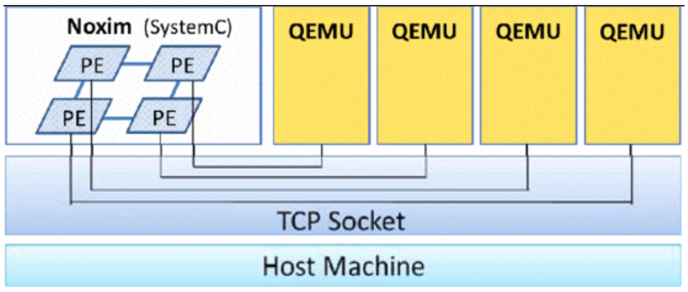
In both project types, whether it's a System-on-Chip (SoC) or a Multi-Processor System-on-Chip (MPSoC), this phase comprises two distinct steps. The first step involves software design, typically implemented in C, C++, or Rust, depending on the chosen framework. The output of this step is an executable binary, often in the format .ELF file. Following the software design, the second step involves emulation through the use of QEMU.

To test the firmware, it is typically built and evaluated on the QEMU platform. QEMU is known for its dynamic binary translation capabilities and its support for various architectures, including RISC-V. Furthermore, it holds wide-ranging utility in both industry and academia, serving as a tool for fault effect analysis on RISC-V targets (as demonstrated in (ADELT et al., 2020)) and as a platform for debugging UEFI bootup issues in the context of RISC-V (as illustrated in (ZHANG, 2022)).

Although not covered in this work, the described SW flow can be extended to a multi-core platform by integrating QEMU with different simulators such as presented in the work (KURIMOTO et al., 2013) through a TCP socket as shown in the Figure 4.3.

After tested, the application binary can be ready for validation in phase (P3).

Figure 4.3: QEMU integrated with Noxim simulator



Source: (KURIMOTO et al., 2013)

While QEMU cannot achieve cycle-accurate precision, the primary objective of this step is to ensure that the algorithm, described at a lower level of abstraction, executes its operations as intended. This readiness serves as a precursor for testing the algorithm on real hardware, as suggested in Phase 3.

If the application is prepared for Phase P3 of validation and simulation, no further actions are necessary. However, if the application is not yet ready, as indicated by arrow connector D, there is an option to revisit the hardware/software partitioning. This allows for the possibility of transferring some tasks back to the hardware or vice versa if needed.

4.1.4 Phase 3 - System's validation

In Phase 3 (P3), two essential inputs are required: the hardware design files generated by IPSoCGen and the software binaries. These inputs will be utilized by the open-source *Verilator* tool to generate a cycle-accurate executable file representing the entire platform. Such a step requires the user to follow a *template* repository with the required files like *Makefiles*, *Docker images* and the source code of the IPSoCGen IPs.

The RTL files will initially undergo conversion by the tool into C++ classes, which will be instantiated in a *testbench*. Within this *testbench*, a function will facilitate the loading of software binaries (.ELF) into the cycle-accurate simulator of the platform. This simulator possesses the capability to generate *waveforms*, demonstrating the execution of various applications within the entire design. This setup gives high visibility to the inner structures of the design and can be used to *benchmark* the achieved results in terms of clock cycles.

By examining the *waveform* dump, the user can verify if the performance specifications are met and subsequently proceed with the workflow. If the specifications are not satisfied, the user has the option to iterate through Phase 1 or Phase 2, allowing for modifications to the application specification or hardware configuration as necessary.

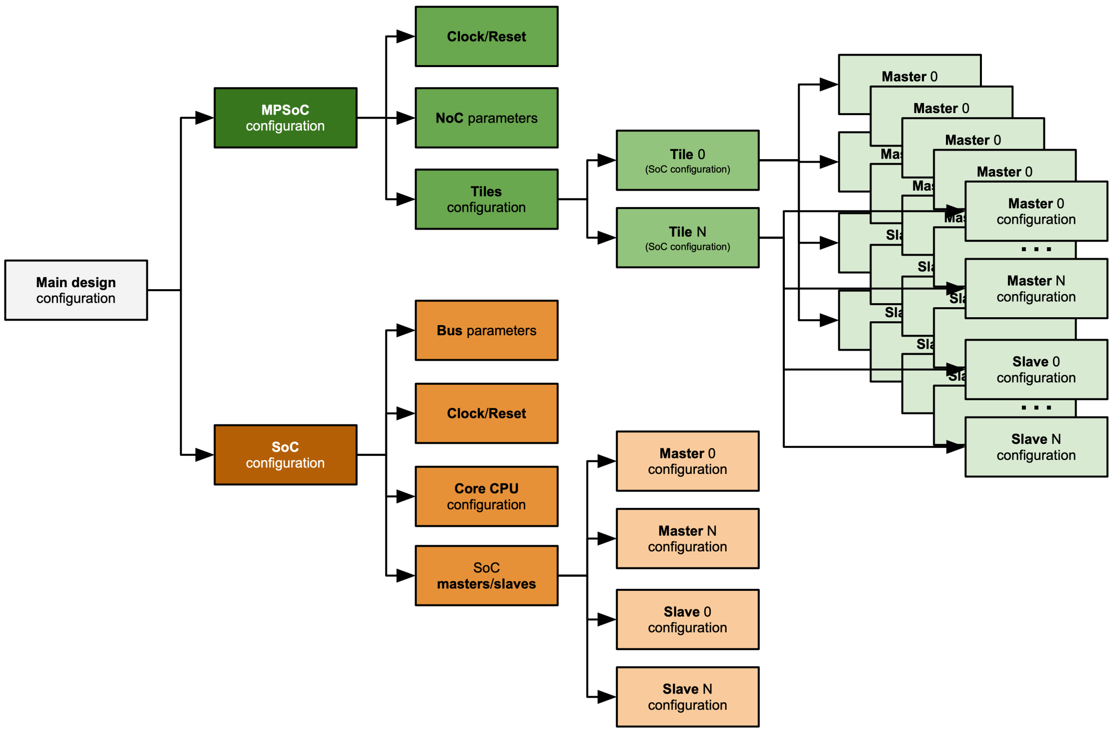
Provided that the design performance aligns with the application specification, the subsequent stage of Phase 3 entails circuit synthesis. This involves following an emulation route through either the FPGA or ASIC flow (as mentioned in Section 1). Upon successful synthesis of the final design, the flow is deemed complete, allowing for the acquisition of results through circuit testing.

4.2 Configuration files

In terms of file format for the configuration file, IPSoCGen makes usage of YAML (Yet Another Markup Language). The YAML format is regarded as a highly suitable format for configuration files due to several compelling reasons. Firstly, YAML offers a human-readable and intuitive syntax, making it easier for users to understand and modify configuration settings. The format utilizes indentation and whitespace conventions, facilitating clear organization and visual hierarchy. Secondly, YAML supports a wide range of data types, including scalar values, lists, and associative arrays, enabling complex and structured configurations, also the *anchor* and the *alias* features allow the user to define tags and replicate designs that were previously defined, simplifying the effort while creating the MPSoC configuration file. This flexibility allows for the representation of diverse configuration parameters and their relationships. Overall, the readability, versatility, and widespread adoption of YAML by the EDA industry make it an excellent choice for configuration files in various software applications. Through the following Section 4.2, an example of the configurable parameters in the configuration file will be demonstrated along with the available options.

To elucidate the configuration file, a diagram denoted as Figure 4.4 has been included. This diagram serves as an illustrative representation of the configuration input file employed by IPSoCGen. Positioned at the far left of the aforementioned figure, there exists an initial configuration section where the user is required to specify essential details such as the project's name, description, and the designated design type, which can either be a System-on-Chip (SoC) or a Multi-Processor System-on-Chip (MPSoC) as shown in Listing 1.

Figure 4.4: IPSoCGen configuration tree diagram



Source: The Author

Listing 1: Initial information on configuration file

```
proj_name: #Project Name Example
desc: #Short description of this project
type: #System type (soc or mpsoc)
```

Source: The Author

4.2.1 SoC configuration

The diagram is partitioned into two branches to accommodate the distinct input requirements associated with each design type. When selecting the SoC branch, it is essential to provide various parameters (as exemplified in Listing 2), including bus parameters such as width and addressing type. In the subsequent section, the definition of clock and reset is presented, affording the user the option to include a PLL if deemed necessary. Furthermore, the reset mechanism can assume the form of a simple direct connection or a reset controller (which, in turn, necessitates a reset controller AXI slave at a later stage). Lastly, the CPU core configuration allows the user to specify the reset type and value, as well as the interrupt mapping.

Following the same branch, the next set refers to the number of slaves and masters

Listing 2: SoC base configuration

```
soc_desc:
  bus_name: #Name for the bus Interconnect
  bus_type: #Type of the bus
  addr_width: #Bus address width
  data_width: #Bus data width
  txn_id_width: #Bus transaction ID width
  num_masters: #Number of master in the bus (min == 1)
  num_slaves: #Number of slave in the bus (min == 1)
  proc_required: #Set if a processor is required (y or n)
  mmap_type: #Memory map type (manual or auto)
  clk:
    name: #Clock name
    clk_int: #Internal signal name for the clock
    io_in_clk: #Wrapper input pin for the clock
    type: #Type of clock (pll or direct)
    pll: #Only applicable if type == pll
      divclk_divide: #PLL parameters
      clkfbout_mult: #PLL parameters
      clkout_divide: #PLL parameters
      io_rst_pin: #Wrapper input pin for the PLL reset
      rst_in_type: #Type of reset for the PLL (act_l or act_h)
      clkin_period: #PLL parameters
  rst:
    name: #Reset name
    rst_int: #Internal signal name for the reset
    io_in_rst: #Wrapper input pin for the reset
    rst_in_type: #Type of reset for the PLL (act_l or act_h)
    type: #Type of reset (acc_rst or direct)
  proc:
    name: #Name of the main CPU that will be used
    type: #Type of processor (nox, vex...)
    clk: #Describe the clock of the processor
    rst: #Describe the reset of the processor
    boot: #Describe the boot for this processor
      type: #Type of boot (slave, value or signal)
      signal: #Only applicable if type == signal
      slave: #Only applicable if type == slave
      value: #Only applicable if type == value
    irq_mapping:
      timer: #Internal signal name for RV timer interrupts
      software: #Internal signal name for RV software interrupts
      external: #Internal signal name for RV external interrupts
```

Source: The Author

in the interconnect. A detailed list of all masters and slaves needs to be described, detailing their types, with the required specific inputs depending on the selected master or slave (Listing 3).

Listing 3: Masters and Slaves configuration

```
masters: #Describe all masters within the system
  x: #Master ID, start always from 0
    name: #Unique name for the instance
    desc: #Brief description of the master I/F
    type: #Master type
    if: #Interface name (must be a valid one supported)
  ...
slaves:
  x: #Slave ID, start always from 0
    name: #Unique name for the instance
    desc: #Brief description of the slave
    type: #Slave type
    base_addr: #Base address if mmap_type == manual
    addr_width: #Base address if mmap_type == manual
    [Multiple different tags depending on the slave]
  ...
```

Source: The Author

4.2.2 MPSoC configuration

For the MPSoC design (Listing 4), a distinct set of inputs is necessary. These inputs include the clock and reset parameters (as mentioned in Section 4.2.1), the NoC (Network-on-Chip) parameters such as size and number of buffers, and finally, the configuration for each tile within the design. By default, IPSoCGen mandates the description of each tile, even if the configuration precisely matches a previously defined one. To address this requirement, the YAML file format offers the *anchor* and *alias* features as mentioned earlier. These features simplify the design description process, making it more concise and efficient. The description of each tile follows the same format as the SoC configuration which gives enough flexibility for each tile the assemble the MPSoC. One limitation of the MPSoC configuration is the fact that the NI (AXI slave in the SoC) of the NoC needs to have the same local address within the tile memory map, this exist to reduce the complexity of the final design.

Once the configuration file is defined, the tool parses the YAML file to verify its syntax correctness and ensure that all the required inputs are provided. Additionally, it checks for parameter consistency before proceeding to the generation stage using internal scripts. For instance, if the user specifies the parameter *num_slaves* as five, but only four slaves are detailed in the configuration, the tool identifies this inconsistency and reports an error stopping the continuation of the flow. Also, the tool will check that the range of parameters (width, size, ...) are valid for each of the sub-components that are provided.

One of the significant advantages of employing the YAML file format, as previ-

Listing 4: MPSoC configuration

```yaml
mpsoc_desc:
  clk:
    name: #Clock name
    clk_int: #Internal signal name for the clock
    io_in_clk: #Wrapper input pin for the clock
    type: #Type of clock (pll or direct)
    pll: #Only applicable if type == pll
      divclk_divide: #PLL parameters
      clkfbout_mult: #PLL parameters
      clkout_divide: #PLL parameters
      io_rst_pin: #Wrapper input pin for the PLL reset
      rst_in_type: #Type of reset for the PLL (act_l or act_h)
      clkin_period: #PLL parameters
  rst:
    name: #Reset name
    rst_int: #Internal signal name for the reset
    io_in_rst: #Wrapper input pin for the reset
    rst_in_type: #Type of reset for the PLL (act_l or act_h)
    type: #Type of reset (acc_rst or direct)
  noc:
    type: #Type of the noc (currently only ravenoc is available)
    name: #Unique name for the NoC instance
    size_x: #Num. of rows
    size_y: #Num. of columns
    flit_data_width: #Data width of the flit
    flit_buff: #Num. of flit buffers
    h_priority: #Priority of the virtual channels
    n_virt_chn: #Num. of virtual channels
    routing_alg: #Routing algorithm (XYAlg or YXAlg)
    max_sz_pkt: #Max. size of packet in flits (usually 256)
    base_addr: #NoC common base address across all Tiles
  tiles:
    x: #Tile ID, always start from zero
      ... #Same description pattern as soc_desc
```

Source: The Author

ously mentioned, is the inclusion of the anchor and alias features. This usage benefit can be effectively demonstrated in the subsequent Listing 5. In this snippet, an example of an MPSoC configuration with nine tiles (such as an NoC 3x3) is provided. Since eight out of the nine tiles require the same design configuration, instead of repetitively writing the same tags, the user can create an alias, denoted as *std_tile*, and use the *anchor* link to replicate the fields. The YAML parser will interpret this as a copy, reducing the number of lines and making the final configuration file less verbose.

Listing 5: *anchor* and *alias* used in MPSoC configuration

```
...
#Start definition of standard Tile
1: &std_tile #Alias to the Tile 1 config.
  bus_name: axi4_crossbar
  bus_type: axi4
  addr_width: 32
  data_width: 32
  txn_id_width: 8
  num_masters: 3
  num_slaves: 9
  mmap_type: manual
  proc_required: yes
  clk:
    name: clk
    clk_int: clk_int
    ... #Several other tags
2: *std_tile #Anchor to the Tile 1 config.
3: *std_tile
4: *std_tile
5: *std_tile
6: *std_tile
7: *std_tile
8: *std_tile
```

Source: The Author

4.3 Classes and modules

IPSoCGen was developed using Python as the primary programming language and is packaged for ease of installation and maintenance using pip, a package manager for Python packages. This packaging approach facilitates straightforward installation and updating of the platform, with automatic management of any required dependencies. Consequently, users can effortlessly install and utilize the platform without the burden of complex installation procedures or manual management of dependencies.

The architecture of the IPSoCGen framework was structured around classes representing various types of masters and slaves within the overall design. Each class serves as a blueprint for instantiating objects that represent specific components of the design. These classes are derived from a base class that encompasses a range of methods. These methods are utilized by scripts to assemble the final HDL and its associated artifacts. After the script has traversed all the tags specified in the configuration file, a compilation of objects will be employed to extract signals, inputs, and outputs, as well as the necessary System Verilog packages. Additionally, the memory map and other options will be checked across different requirements.

One crucial feature of the tool is its utilization of templates for efficient design generation. Within the framework of IPSoCGen, templates are *text* files that depict the final hardware description language, including variables or expressions that are substituted with values during rendering. For instance, in Listing 6, the provided code serves as the template for the AXI Interrupt Controller, while the bottom section illustrates the output in System Verilog after being processed by the tool.

4.4 Extension and flexibility

Following what was described at the end of Section 4.3, the inclusion of new modules (masters or slaves) can be achieved through the creation of new templates and their corresponding classes that would represent a new design. Due to its simplicity, it does not require an extensive modification of the framework to support new additions. As the tool is part of a Python package, the open-source repository can be forked to add or change new tags and include new design plugins.

Moreover, the tool also offers support for customized AXI masters and slaves. A standardized template is available that facilitates the connection of inputs and outputs related to the bus, as well as the allocation of space within the SoC memory map. Within this custom module description, users can specify the allocated size using a dedicated tag (mem_size_kib) through the configuration file. This feature allows for the seamless integration of various accelerators, as the provided wrapper can be easily extended based on the user's requirements.

Different types of parameters and options can be configured within the framework, the following list enumerates each of them and provides an overview of its capabilities.

1. Project type: SoC vs MPSoC
2. Network-on-chip and its parameters
3. Number of masters and slaves
4. Type of memory map and its parameters (width)
5. If a processor is required
6. Type of processor and its boot/IRQs
7. Type of clock: PLL or direct
8. Type of reset: Through a reset controller or direct
9. Configuration of masters

   1. cpu_nox: CPU NoX
   2. cpu_vex: CPU Vex
   3. acc_dma: DMA master
   4. acc_custom_master: Custom AXI master wrapper

5. Configuration of slaves

   1. ram_mem: RAM memory
   2. rom_mem: ROM memory
   3. acc_uart: Configurable UART
   4. acc_timer: Configurable TIMER
   5. acc_dma: Configurable DMA
   6. acc_irq: Configurable IRQ
   7. acc_rst: Programmable Reset Controller
   8. acc_custom_slave: Custom AXI slave wrapper
   9. acc_noc: Configurable Network-on-chip
   10. acc_eth: Configurable Ethernet Controller

11. Configuration of tiles

4.5 Conclusion

Chapter 4 introduced the IPSoCGen flow, outlining its phases and providing a comprehensive description of each step until the final hardware testing stage. The required input for each phase and the corresponding expected outputs during the design exploration development were also presented. The following sections explain the configuration file, detailing its development process to achieve a balance between flexibility and configurability. Finally, the tool's inner workings were explored, including the rationale behind its class-based organization and how the final design is constructed using the objects generated during the process.

Listing 6: Template of the AXI interrupt controller and the output

```
####### IRQ template #######
  logic [31:0] irq_vector_mapping;

  {%- for irq in tmpl.vec_mapping %}
  assign irq_vector_mapping[{{ loop.index0 }}] = {{ irq }};
  {%- endfor %}
  {% if tmpl.all_irq_filled == 0 %}
  assign irq_vector_mapping[31:{{ tmpl.max_irq }}] = '0;
  {% endif %}

  //
  // {{ tmpl.desc }}
  //
  axi_irq_ctrl #(
    .BASE_ADDR          ({{ tmpl.base_addr }}),
    .TYPE_OF_IRQ        ({{ tmpl.irq_type }})
  ) u_{{ tmpl.name }} (
    .clk                ({{ tmpl.clk }}),
    .rst                ({{ tmpl.rst }}),
    .irq_i              (irq_vector_mapping),
    .irq_summary_o      ({{ tmpl.irq_summary }}),
    .axi_mosi           (slaves_axi_mosi[{{ tmpl.slv_id }}]),
    .axi_miso           (slaves_axi_miso[{{ tmpl.slv_id }}])
  );
####### Rendered output #######
  logic [31:0] irq_vector_mapping;
  assign irq_vector_mapping[0] = irq_ravenoc.irq_trig;
  assign irq_vector_mapping[1] = dma_error;
  assign irq_vector_mapping[2] = dma_done;
  assign irq_vector_mapping[3] = 1'b0;

  assign irq_vector_mapping[31:4] = '0; // TIE-L not used IRQs

  //
  // IRQ Controller
  //
  axi_irq_ctrl #(
    .BASE_ADDR          ('h72000),
    .TYPE_OF_IRQ        ('hffffffff)
  ) u_irq_ctrl (
    .clk                (clk_int),
    .rst                (rst_int),
    .irq_i              (irq_vector_mapping),
    .irq_summary_o      (irq_ctrl_ext),
    .axi_mosi           (slaves_axi_mosi[8]),
    .axi_miso           (slaves_axi_miso[8])
  );
```

Source: The Author

5 SOC AND MPSOC PLATFORM

Chapter 5 will provide an exposition of the outcomes achieved by the developed platform, IPSoCGen, as described in Chapter 4. In the pursuit of testing and demonstrating its applicability, two distinct types of Systems-on-Chip (SoC) were formulated and evaluated. The first one encompasses a straightforward SoC, while the second one embodies a more complex Multi-Processor System-on-Chip (MPSoC). A comprehensive analysis of both will be presented, encompassing intricate aspects such as *firmware* development, *hardware* development, and testing.

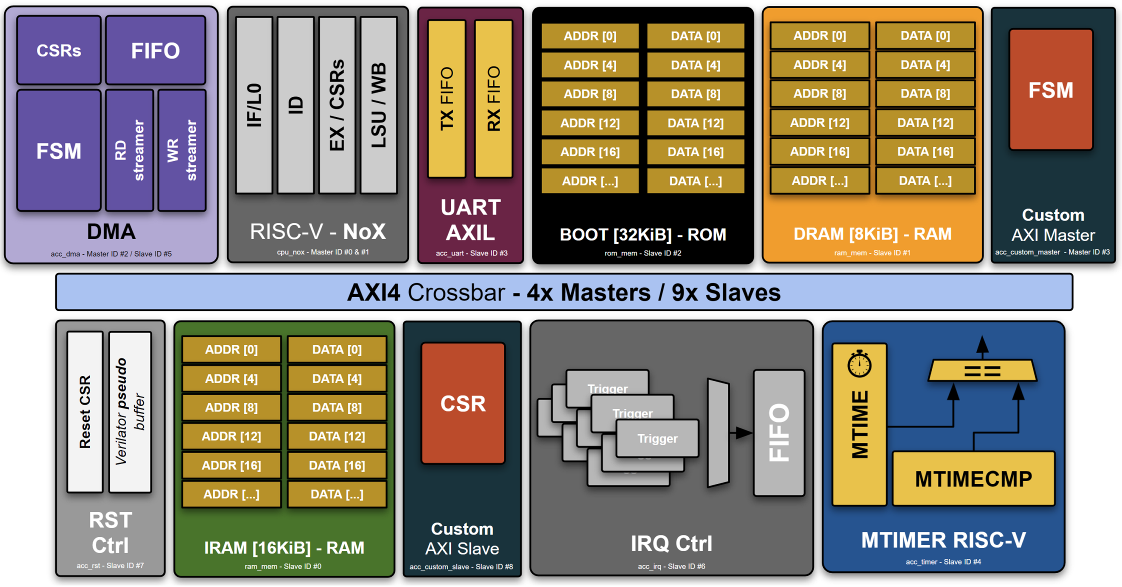
5.1 SoC template

5.1.1 Architecture

In Figure 5.1, a block diagram is presented, illustrating the various masters and slaves along with their corresponding identifiers (IDs) and types. The main CPU, denoted as NoX type 3.2, is depicted with two master interfaces: LSU and Fetch. The reset controller and interrupt controller, indicated by the gray background color, are closely associated with the core. The former provides the reset vector, while the latter handles the mapping of interrupt vectors. To enable the execution of a basic program on the CPU, three memories with distinct types were included: an Instruction and a Data RAM, as well as a Boot ROM. These memories store the program instructions, variables, and a UART bootloader for testing the initial boot code sequence. Furthermore, additional slaves were incorporated, including a DMA, MTIMER, and UART peripheral. To evaluate the custom AXI wrappers, a *custom* slave and master were added, both featuring simple designs aimed at testing their integration within the SoC.

The design was entirely compiled using *Verilator*, which converts all RTL design files into C++ classes. Once the design is converted, it is instantiated into a *testbench* responsible for loading the ELF files into the RAM memories mentioned earlier. The primary objective of this SoC was to test the infrastructure and the complete *firmware* development flow, including simulation testing and *hardware* behavior debugging using *waveforms*. After loading the program, it was possible to observe that once completed the boot flow, the RISC-V processor prints some information in the output terminal as can be seen in the Listing 7.

Figure 5.1: SoC template diagram



Source: The Author

To emulate and test this SoC generated using IPSoCGen, additional files such as the *firmware* and the set of *Makefiles* and IP database were merged into a single repository called IPSoCGen template, available on author's GitHub website (SILVA, 2023c). Due to that, IPSoCGen outputs are limited to the design generation and the header files consumed by the *firmware*.

Listing 7: SoC Hello World boot

```
 _   _       __  __
| \ | |  ___ \ \/ /
|  \| | / _ \ \  /
| |\  || (_) |/  \
|_| \_| \___//_/\_\
NoX RISC-V Core RV32I

CSRs:
mstatus     0x1880
misa        0x40000100
mhartid     0x0
mie         0x0
mip         0x0
mtvec       0x101
mepc        0x0
mscratch    0x0
mtval       0x0
mcause      0x0
cycle       358
...
```

Source: The Author

5.1.2 Program loading through testbench

As previously mentioned, the *testbench* implemented in C++ plays a crucial role in managing the Device Under Test (DUT), which in this case is the SoC. The *testbench* provides the necessary functionality to interpret ELF files and correctly load each program segment into the appropriate memory location, while also verifying their corresponding *hardware* physical addresses. Additionally, the *testbench* enables emulation of various peripherals such as the UART and different interrupts within the IRQ controller. In Listing 8, it is possible to observe the function that implements the loading of the ELF program into the memories using the ELFIO C++ library. Through the same listing, several important aspects of this loading are represented. Firstly, the program is parsed (line 7) with the path received by command line arguments (*argc/argv*), then it gets split into different segments, each with its particular Virtual Memory Address (VMA) and Loaded Memory Address (LMA) (from line 10 to 14).

Listing 8: ELF loader function in *testbench*

```
 1  bool loadELF(testbench<Vtest> *sim,
 2              string program_path,
 3              s_tile_t tile,
 4              const bool en_print){
 5    ELFIO::elfio program;
 6
 7    program.load(program_path);
 8  ...
 9    for (uint8_t i = 0; i<seg_num; i++){
10      const ELFIO::segment *p_seg = program.segments[i];
11      const ELFIO::Elf64_Addr lma_addr =
12                      (uint32_t)p_seg->get_physical_address();
13      const ELFIO::Elf64_Addr vma_addr =
14                      (uint32_t)p_seg->get_virtual_address();
15      const uint32_t mem_size = (uint32_t)p_seg->get_memory_size();
16      const uint32_t file_size = (uint32_t)p_seg->get_file_size();
17  ...
18        // IRAM Address
19        for (uint32_t p = 0; p < file_size; p+=4){
20          uint32_t word_line =
21              ((uint8_t)p_seg->get_data()[p+3]<<24)+
22              ((uint8_t)p_seg->get_data()[p+2]<<16)+
23              ((uint8_t)p_seg->get_data()[p+1]<<8)+
24              (uint8_t)p_seg->get_data()[p];
25          if (!(word_line == 0x00)) {
26            sim->core->test->writeWordRAM___05Firam(
27                                    (p+init_addr)/4,word_line);
28          }
29  ...
```

Source: The Author

The VMA is the memory address of the program while in execution, whereas the LMA is the location when it is loaded into the memory. Typically, this means that some segments that contain initialized (*.data*) and uninitialized variables (*.bss*) used during the program execution will have their VMA addresses differing from the LMA, as the usual initialization code (*crt0.S*) will be responsible for moving the data before the program jumping to the main function. The link between host emulation and the RTL occurs through the write xRAM reference (line 26), where the *testbench* calls a method that is part of the DUT class (converted by *Verilator* previously). This converted method is a System Verilog function (Listing 9) in the original design, which performs the inner loading of the memory. It should be noted that this function should not be translated into *hardware* during synthesis.

Listing 9: SV function to load Instruction RAM used by the *testbench*

```
// synthesis translate_off
function automatic void writeWordRAM__iram(addr_val, word_val);
  /*verilator public*/
  logic [31:0] addr_val;
  logic [31:0] word_val;
  u_dram.mem_loading[addr_val] = word_val;
endfunction
// synthesis translate_on
```

Source: The Author

Given that the ELF header provides access to the *entry point address*, it is feasible to extract this information utilizing the aforementioned library. By utilizing the obtained *entry point*, it becomes possible to configure the reset vector of the CPU or the reset controller, which is also a component within the design (Listing 10). Consequently, if desired, the user has the option to commence the simulation with the loaded program at the initial timestamp.

Listing 10: SV function to set the reset vector used by the *testbench*

```
// synthesis translate_off
function automatic void writeRstAddr__rst_ctrl(rst_addr);
  /*verilator public*/
  logic [31:0] rst_addr;
  u_rst_ctrl.rst_loading = rst_addr;
endfunction
// synthesis translate_on
```

Source: The Author

In embedded platforms, the loading of a program into RAMs or manipulation of the reset vector among other tasks is commonly managed by a *bootloader* program. With the developed *testbench*, all these tasks can be easily integrated without requiring any real *hardware*. This advantage stems from the flexibility offered by C++, which is not limited to the final target binary but rather by the system where it is emulated. Once the program is loaded, the same *testbench* will simulate as many clock cycles as specified through command line arguments. Additionally, the simulation has the option to enable or disable *waveform* dumping based on the user's choice.

5.1.3 Custom AXI master and slave

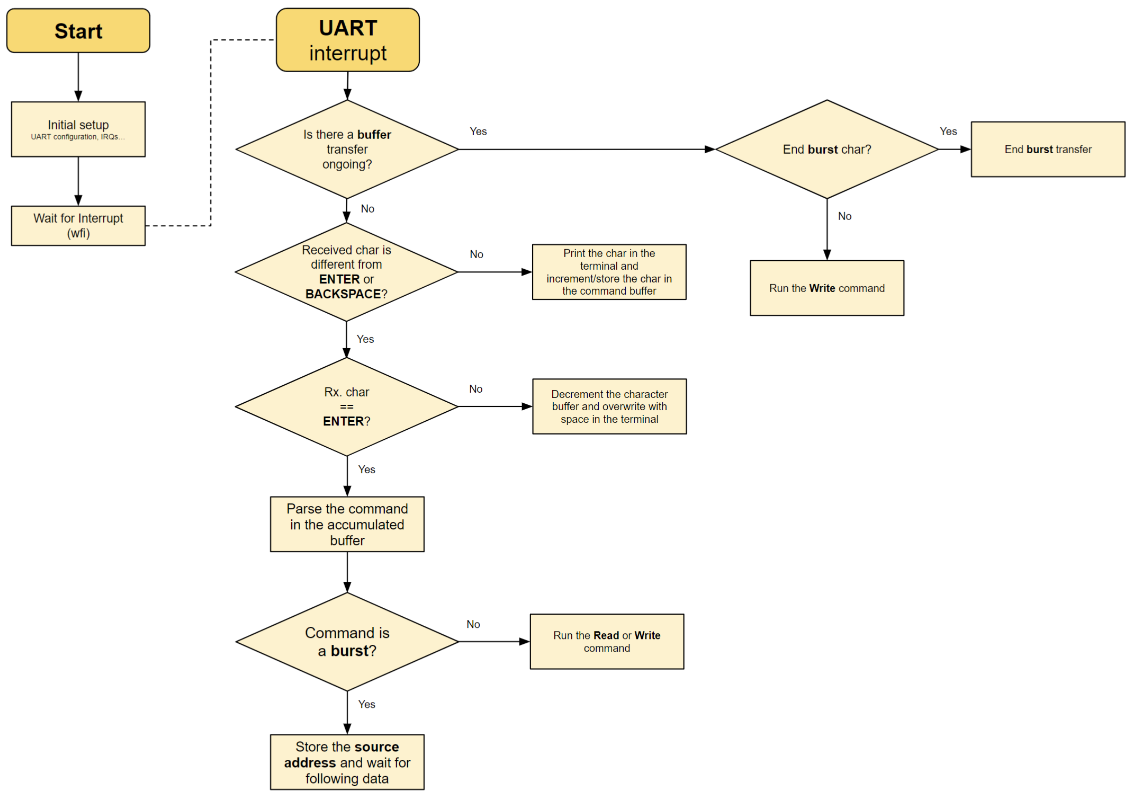
As depicted in Figure 5.1, a *Custom* AXI master and slave components were incorporated to evaluate their integration with the remaining parts of the SoC. The aim was to verify the accuracy of the address mapping allocation and to assess the feasibility of seamlessly integrating *custom* accelerators. On the slave side, twenty different CSRs were integrated into the wrapper to test if the protocol communication was working properly which was then read by the CPU. These CSRs could be part of a memory-mapped accelerator that runs a particular task such as matrix multiplication. Regarding the AXI master, a straightforward FSM was implemented to perform polling on a fixed address within the SoC. Despite its low complexity, the task executed by this master represents a typical design scenario such as DMA.

5.1.4 Bootloader for FPGA/ASIC testing

Another motivation behind creating this SoC template was to develop and test bootloader code capable of receiving read and write commands from a peripheral device (in this case, the UART) and executing them within the SoC. The importance of this bootloader is fundamental for FPGA prototyping and ASIC development, as it provides an easy way to update the content of the memories that contain programs or data used by the different entities within the design.

In Figure 5.2, the bootloader program flow developed using this design is presented. The program first configures the interrupts and the UART peripheral, and then it stalls its execution by entering wfi mode. The UART peripheral is configured to generate an IRQ to the main core (NoX) whenever the received buffer reaches a certain threshold of characters in its internal FIFO. Once the processor receives an interrupt due to a received character, it jumps to the interrupt handler, where it checks the type of character.

Figure 5.2: Bootloader flowchart


Start
Initial setup
UART configuration, IRQs...
Wait for Interrupt (wfi)
UART interrupt
Is there a buffer transfer ongoing?
Yes
No
End burst char?
Yes
End burst transfer
No
Run the Write command
Received char is different from ENTER or BACKSPACE?
No
Print the char in the terminal and increment/store the char in the command buffer
Yes
Rx. char == ENTER?
No
Decrement the character buffer and overwrite with space in the terminal
Yes
Parse the command in the accumulated buffer
Command is a burst?
No
Run the Read or Write command
Yes
Store the source address and wait for following data

Source: The Author

A simple protocol was defined between the *firmware* and the user who sends characters over the UART. This protocol includes the type of operation (read or write), an address, and optional data (in the case of a write operation followed by the "enter" character). Through this protocol, users can potentially read from and write to all peripherals within the SoC. An additional option called *burst* has been added. In this mode, the user sends a single address followed by a burst of data. The processor automatically performs write operations and increments the address. This mode is particularly useful when loading programs through the terminal, as it allows for updating the entire memory at once.

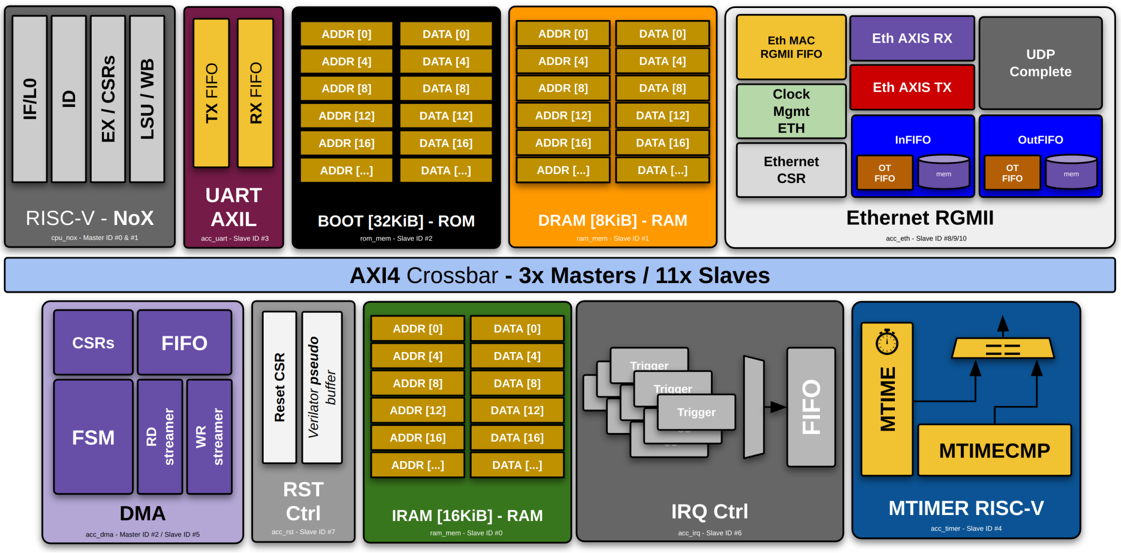
Subsequently, the bootloader was converted into a Boot ROM image, as depicted in Figure 5.1. To simplify its usage, a Python *script* was developed. The *script* expects a single input, which is the ELF file. It automatically transfers the entire program to the design, parsing the file to filter the type of segment before sending the commands over the UART. Additionally, the *script* can update the reset vector to match the *entry point address*. If the user has integrated the reset controller provided by IPSoCGen in their design configuration, the *script* can also perform a local reset through the bootloader.

The complete development of this bootloader was carried out through simulation, as the *waveforms* proved to be valuable for debugging various scenarios based on different user input. To emulate the UART reception (rx), the input keyboard captured the characters and sent them through the *testbench*. For the transmission (tx), a specific register in the reset controller was utilized to capture characters written by the program. Whenever the register was updated, the 8-bit character was captured and forwarded to the standard output in the terminal. Ultimately, the design was tested initially in the Xilinx Arty A7-35T FPGA to confirm its correct execution with real FPGA *hardware*.

5.1.5 Application testing - Histogram

To facilitate the development of an application for testing the System-on-Chip (SoC) and conducting a comparative analysis with the Multi-Processor System-on-Chip (MPSoC), three supplementary AXI slaves have been incorporated into the existing design configuration, with the removal of the *custom* AXI slaves (as there was no need to test anymore). These additional AXI slaves correspond to the Ethernet AXI slave (CSRs, Input FIFO [InFIFO] and Output FIFO [OutFIFO]). Figure 5.3 represents the final design with all masters and slaves considering the changes to run an application.

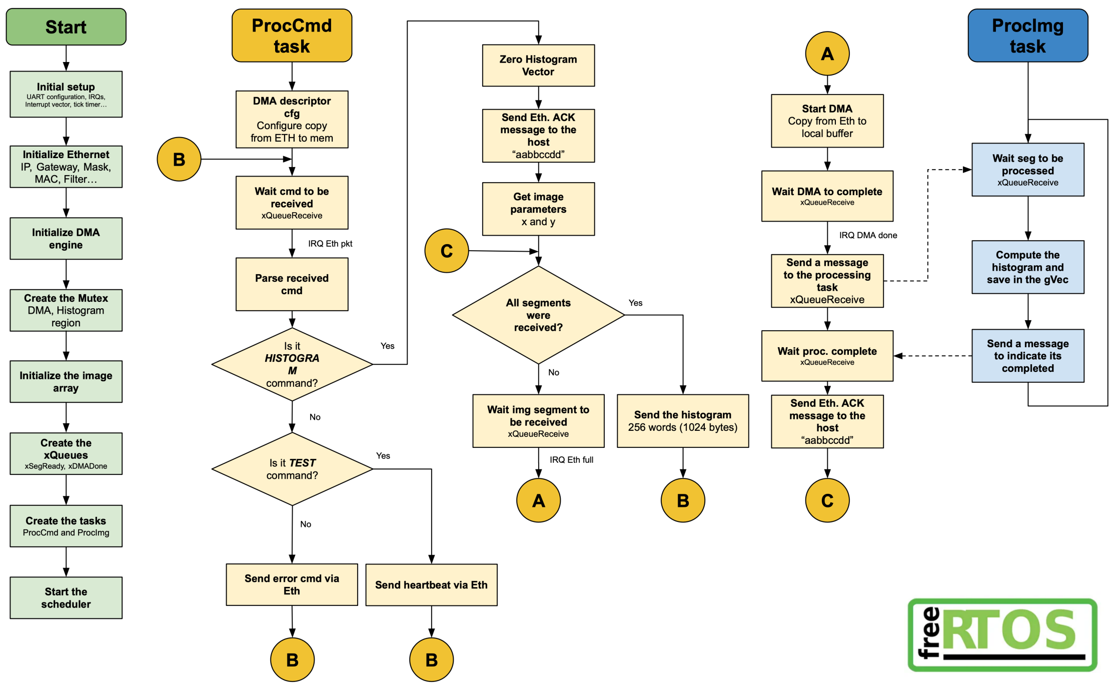
Figure 5.3: SoC architecture with additional slaves (Ethernet)



Source: The Author

The selected benchmark application for evaluating the two template designs, namely the System-on-Chip (SoC) and Multi-Processor System-on-Chip (MPSoC), involves the computation of image histograms. The fundamental concept entails a program running on a host computer, transmitting images via Ethernet to the SoC deployed on an FPGA. Considering the design previously mentioned, a *firmware* was developed to enable the communication between the host PC and the target design, using the operational system for embedded platforms FreeRTOS version v10.4.4+ with a tick timer of 500 Hz. Figure 5.4, it is presented the program execution flow while running the histogram application.

Figure 5.4: SoC Histogram app flowchart



Source: The Author

For the program execution flow depicted in Figure 5.4, the flowchart employs three distinct colors to represent different meanings. The leftmost section, highlighted in green, represents the initialization routine, which is executed each time the System on Chip (SoC) restarts. Following the "*start*" step, the initial setup involves configuring various peripherals, such as the Universal Asynchronous Receiver-Transmitter (UART), interrupts, the "*tick*" timer, Ethernet, and Direct Memory Access (DMA) peripherals.

Upon completing this initial setup, two tasks are created: *ProcCmd* and *ProcImg*. The *ProcCmd* task is responsible for parsing commands transmitted from the host computer to the FPGA. Once a command is received, a specific action is executed accordingly. In the case of a *histogram* command, the program waits for all image segments to be transferred from the host. Each segment is then sent to the *ProcImg* task for processing, which accumulates the final histogram result. After all segments are processed, the *ProcCmd* task sends the histogram to the host computer and returns to a waiting state. At this point, it can receive other commands, such as a *test* command that sends a *heartbeat* signal to the host.

5.2 MPSoC template

As one of the primary objectives of the IPSoCGen framework, the generation of MPSoCs represents one of the most intricate tasks within design generation frameworks. This section will provide a comprehensive overview and explanation of the MPSoC template that has been constructed utilizing the IPSoCGen framework. It will delve into the intricacies of both the *hardware* and *firmware* aspects, with a particular focus on the integration for the final *histogram* application testing.

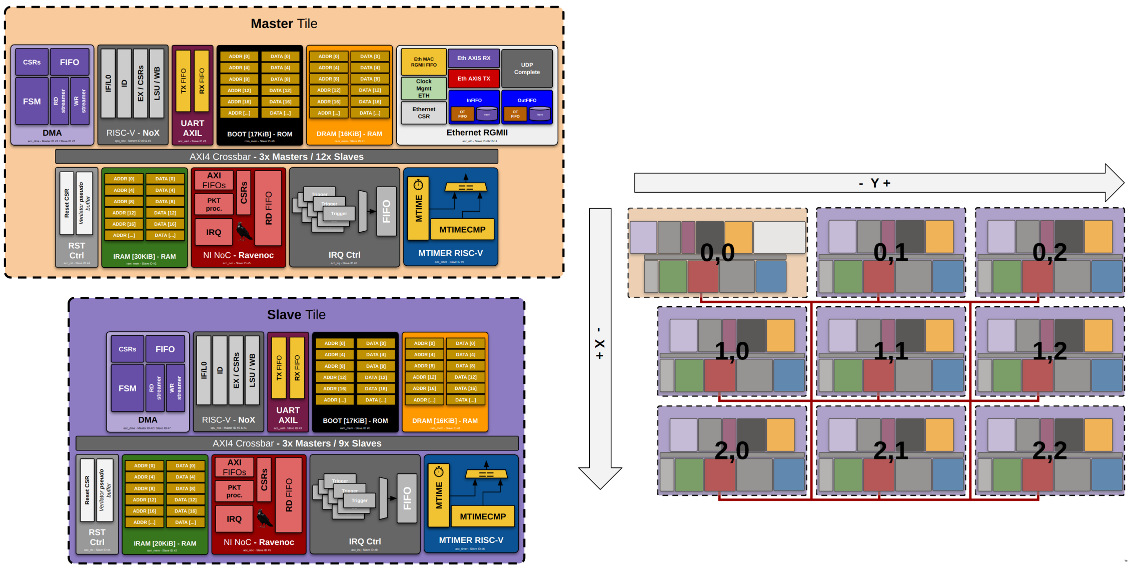
5.2.1 Architecture

Figure 5.5 illustrates the structure of the MPSoC, which encompasses two distinct designs: a single master tile and a set of eight slave tiles. This segregation into two tile types was implemented to streamline the *firmware* development process, necessitating only two distinct *firmware* implementations. The master tile serves as the communication hub with the host PC and dispatches instructions to the slave tiles through the NoC. In contrast, the slave tiles predominantly undertake the computational workload, receiving commands and transmitting results back to the master tile.

The master tile consists of several components, including a DMA unit, a NoX RISC-V Core, a boot ROM with a UART bootloader, a peripheral UART interface, 16 KiB of data RAM memory, an Ethernet AXI interface, a reset controller, a 30 KiB instruction RAM memory, the NoC NI (Network Interface), an Interrupt Controller, and a timer utilized by the FreeRTOS running on the core. On the other hand, the slave tile exhibits a similar configuration but lacks the Ethernet controller and possesses a smaller instruction RAM memory. Notably, the NoC is configured as a 3x3 grid, consisting of nine PE (Processing Elements).

In addition to the MPSoC wrapper, a manual inclusion was made in the final generated RTL design. This inclusion comprises a multiplexer controlled by an external switch, which allows for the selection of the serial FTDI interface to establish a connection with

Figure 5.5: MPSoC 3x3 architecture



Source: The Author

either the master tile or all the slave tiles for the UART peripheral. This enhancement facilitates the simultaneous download of the *firmware* to all slave tiles at the same time.

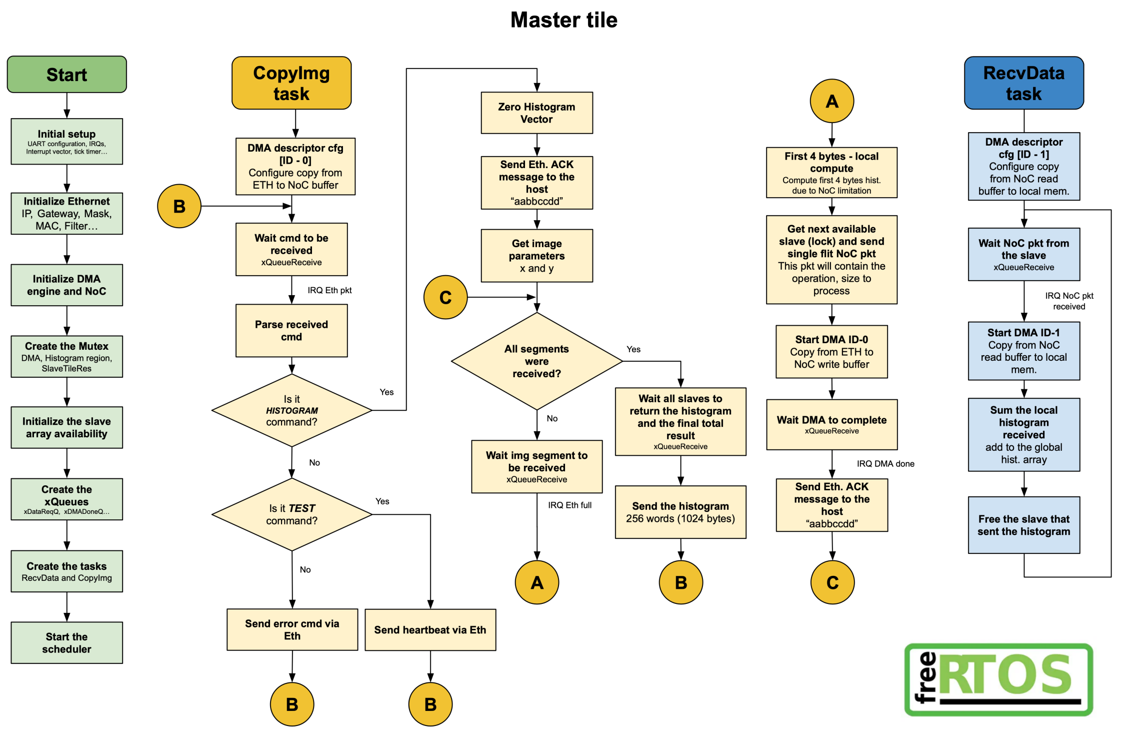
5.2.2 Histogram MPSoC flow

Following a similar approach as the one described for the SoC, a FreeRTOS application was developed for the MPSoC with the difference that two distinct apps are necessary, one for the master tile and another one for the slave. Figure 5.6 details the program flow developed by the master tile which is responsible for receiving image segments and forwarding them to the slave tiles.

Certain intricate details, such as mutex control, task synchronization, and interrupt handling, have been intentionally omitted to simplify the understanding of the program flow. However, it is crucial to acknowledge that these aspects were indeed implemented to ensure the accurate execution of the program.

The master tile program flow, as depicted in Figure 5.6, follows a similar approach to the SoC diagram, with different stages denoted by distinct colors representing the execution context. On the leftmost side (green), the initialization of APIs, mutexes, queues, and task creation is illustrated, along with the array responsible for tracking the availability of slave tiles. On the right side (yellow), the *CopyImg* task is described, which receives

Figure 5.6: Master Tile Histogram app flowchart

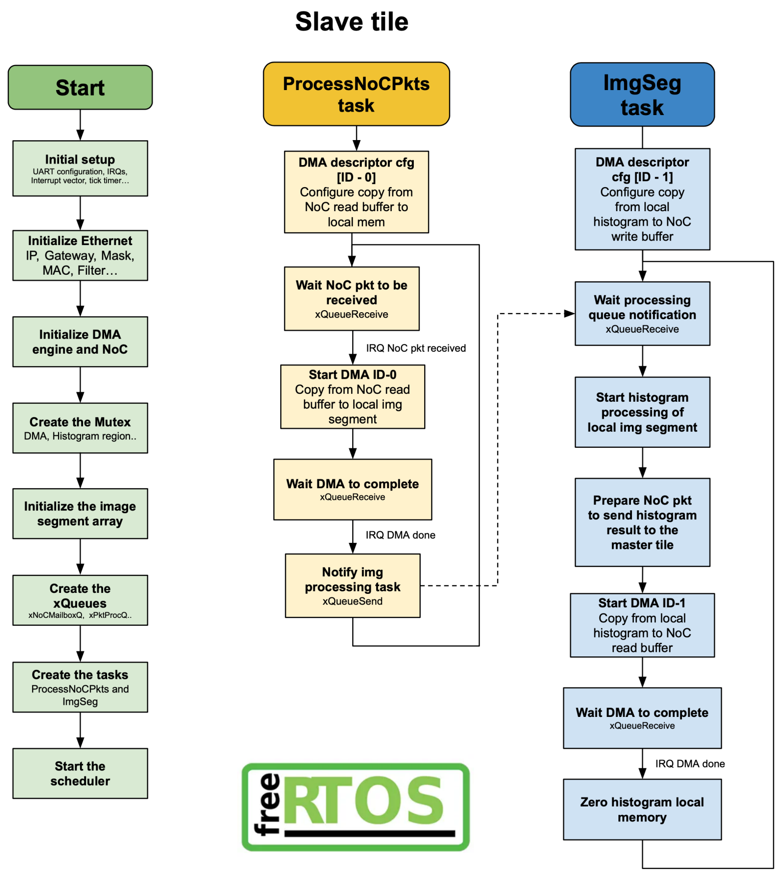

Master tile
Start
Initial setup
UART configuration, IRQs, Interrupt vector, tick timer…
Initialize Ethernet
IP, Gateway, Mask, MAC, Filter…
Initialize DMA engine and NoC
Create the Mutex
DMA, Histogram region, SlaveTileRes
Initialize the slave array availability
Create the xQueues
xDataReqQ, xDMADoneQ…
Create the tasks
RecvData and CopyImg
Start the scheduler
CopyImg task
DMA descriptor cfg [ID - 0]
Configure copy from ETH to NoC buffer
B
Wait cmd to be received
xQueueReceive
IRQ Eth pkt
Parse received cmd
Is it HISTOGRAM command?
Yes
No
Is it TEST command?
Yes
No
Send error cmd via Eth
Send heartbeat via Eth
B
B
Zero Histogram Vector
Send Eth. ACK message to the host "aabbccdd"
Get image parameters x and y
C
All segments were received?
Yes
No
Wait img segment to be received
xQueueReceive
IRQ Eth full
A
Wait all slaves to return the histogram and the final total result
xQueueReceive
Send the histogram
256 words (1024 bytes)
B
A
First 4 bytes - local compute
Compute first 4 bytes hist. due to NoC limitation
Get next available slave (lock) and send single flit NoC pkt
This pkt will contain the operation, size to process
Start DMA ID-0
Copy from ETH to NoC write buffer
Wait DMA to complete
xQueueReceive
IRQ DMA done
Send Eth. ACK message to the host "aabbccdd"
C
RecvData task
DMA descriptor cfg [ID - 1]
Configure copy from NoC read buffer to local mem.
Wait NoC pkt from the slave
xQueueReceive
IRQ NoC pkt received
Start DMA ID-1
Copy from NoC read buffer to local mem.
Sum the local histogram received
add to the global hist. array
Free the slave that sent the histogram
freeRTOS

Source: The Author

Ethernet packets, parses them, and forwards the image segments through the NoC to the respective slave tiles as they are received. Once all segments have been processed and returned by the slave tiles, the final histogram vector is sent to the host. On the rightmost side (blue) of the diagram, the parallel *RecvData* task responsible for receiving the partial histogram results from each slave tile is shown.

Figure 5.7 illustrates the program flow for the slave tile, which is comparatively simpler than that of the master tile. The slave tile program follows a sequence of tasks. Initially, in the *ProcessNoCPkts* task, the slave tile awaits NoC packets. Upon receiving a packet, it initiates a DMA operation to locally copy the image segment and subsequently signals the *ImgSeg* task. The *ImgSeg* task receives the image segment, processes it to compute the histogram, and then utilizes a DMA operation to send the resulting histogram back to the slave tile.

Figure 5.7: Slave Tile Histogram app flowchart


Slave tile
Start
Initial setup
UART configuration, IRQs, Interrupt vector, tick timer…
Initialize Ethernet
IP, Gateway, Mask, MAC, Filter…
Initialize DMA engine and NoC
Create the Mutex
DMA, Histogram region..
Initialize the image segment array
Create the xQueues
xNoCMailboxQ, xPktProcQ..
Create the tasks
ProcessNoCPkts and ImgSeg
Start the scheduler
ProcessNoCPkts task
DMA descriptor cfg [ID - 0]
Configure copy from NoC read buffer to local mem
Wait NoC pkt to be received
xQueueReceive
IRQ NoC pkt received
Start DMA ID-0
Copy from NoC read buffer to local img segment
Wait DMA to complete
xQueueReceive
IRQ DMA done
Notify img processing task
xQueueSend
ImgSeg task
DMA descriptor cfg [ID - 1]
Configure copy from local histogram to NoC write buffer
Wait processing queue notification
xQueueReceive
Start histogram processing of local img segment
Prepare NoC pkt to send histogram result to the master tile
Start DMA ID-1
Copy from local histogram to NoC read buffer
Wait DMA to complete
xQueueReceive
IRQ DMA done
Zero histogram local memory
freeRTOS

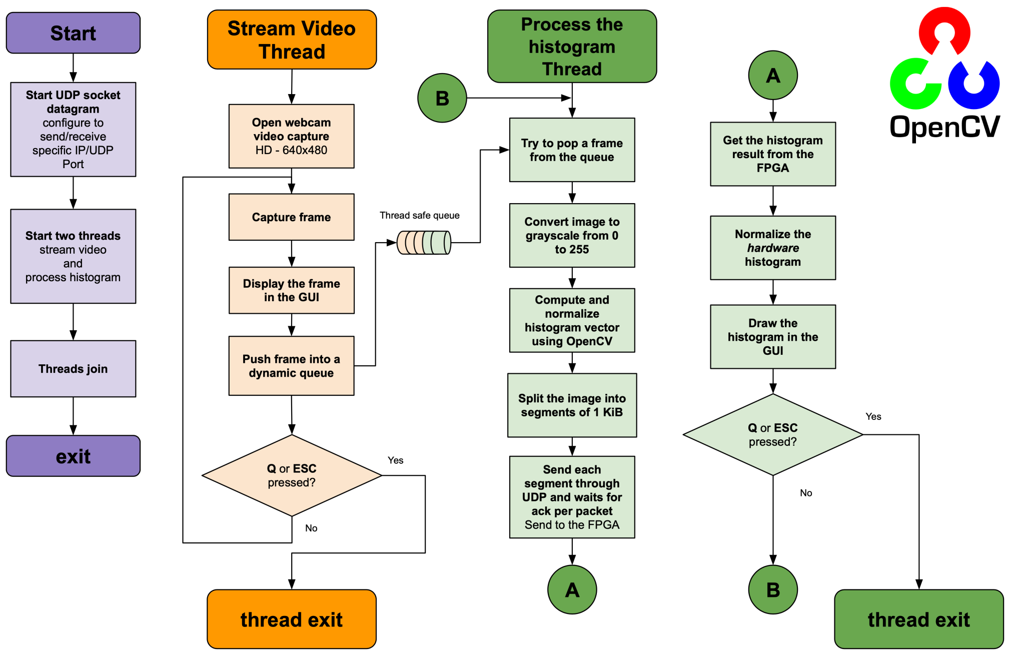
Source: The Author

5.3 Host histogram application

Considering that both designs have to communicate with the host (PC) application for the histogram *benchmark*, an application was developed to capture images from a webcam, split and send through the local Ethernet network to the FPGA. Figure 5.8 shows the program flow for the final application that runs on the host machine.

Beginning with the leftmost (purple) region, the program initiates two threads: one dedicated to video streaming/image capturing, and another responsible for histogram processing. The thread handling video streaming captures a frame and adds it to a dynamic thread-safe queue. If the queue reaches its maximum capacity, the oldest element is removed to prevent memory overflow. The thread assigned to histogram processing retrieves a frame from the queue, converts it to grayscale, and utilizes OpenCV to compute the histogram. Subsequently, the frame is divided into multiple 1 KiB segments, which are then transmitted to the FPGA. The FPGA performs the final histogram computation in

Figure 5.8: Host Histogram app flowchart


Start
Start UDP socket datagram configure to send/receive specific IP/UDP Port
Start two threads stream video and process histogram
Threads join
exit
Stream Video Thread
Open webcam video capture HD - 640x480
Capture frame
Display the frame in the GUI
Push frame into a dynamic queue
Q or ESC pressed?
Yes
No
thread exit
Thread safe queue
Process the histogram Thread
B
Try to pop a frame from the queue
Convert image to grayscale from 0 to 255
Compute and normalize histogram vector using OpenCV
Split the image into segments of 1 KiB
Send each segment through UDP and waits for ack per packet Send to the FPGA
A
A
Get the histogram result from the FPGA
Normalize the hardware histogram
Draw the histogram in the GUI
Q or ESC pressed?
Yes
No
B
thread exit
OpenCV

Source: The Author

the SoC or MPSoC and returns the *hardware* computed result. Once the final histogram is received from the FPGA, it is normalized, and both results—computed by OpenCV and the hardware implementation—are displayed.

5.4 Software and APIs

Multiple C APIs were developed to facilitate the utilization of peripherals in both SoC and MPSoC, considering the presence of various types of slave devices. These libraries make use of the set of header files, which are among the outputs generated by IPSoCGen. The coding style employed adheres to the guidelines followed by FreeRTOS, including conventions for type names such as prefixing *uint32_t* with "ul" (unsigned long) and using "us" or "uc" for unsigned short/char (*uint16_t* and *uint8_t*). Additionally, a *CamelCase* naming convention is adopted, among other conventions. For instance, a small excerpt of the DMA API is presented in Listing 11.

The development of APIs was carried out for various components, including DMA, UART, IRQ controller, NoC, and Ethernet. Since most of the designs were primarily developed by the author, it became necessary to establish this layer of abstraction to ensure system usability. The majority of the API development was executed within the C++

Listing 11: DMA embedded C API

```c
...
#include "dma.h"
...
static volatile uint32_t* const pulDMADesc1DstAddr  = (uint32_t*)dmaDESC_1_DST_ADDR;
static volatile uint32_t* const pulDMADesc1NumBytes = (uint32_t*)dmaDESC_1_NUM_BYTES;
static volatile uint32_t* const pulDMADesc1Cfg      = (uint32_t*)dmaDESC_1_DESC_CFG;

void vDMAInit (void) {
  DMAStatus_t xDMAStatus = xDMAGetStatus();

  dbg("\n\r");
  dbg("\n\rDMA init:");
  dbg("\n\rVersion: %x", xDMAStatus.Version);
  dbg("\n\r");
}

void vDMASetDescCfg (uint8_t ucDescID, DMADesc_t xDesc) {
  switch (ucDescID) {
    case 0:
      *pulDMADesc0SrcAddr  = (uint32_t)xDesc.SrcAddr;
      *pulDMADesc0DstAddr  = (uint32_t)xDesc.DstAddr;
      *pulDMADesc0NumBytes = xDesc.NumBytes;
      *pulDMADesc0Cfg      = (((uint8_t)xDesc.Cfg.WrMode & 0x1) << dmaCFG_WRITE_MODE) |
                             (((uint8_t)xDesc.Cfg.RdMode & 0x1) << dmaCFG_READ_MODE)  |
                             (((uint8_t)xDesc.Cfg.Enable & 0x1) << dmaCFG_ENABLE);
      break;
...
      break;
    default:
      masterCRASH_DBG_INFO("Unexpected DMA Descriptor ID");
      break;
  }
}
```

Source: The Author

model generated from the two designs (SoC and MPSoC) throughout *Verilator* tool.

5.5 FPGA prototyping

The FPGA platform selected to facilitate the implementation of both the SoC and MPSoC designs was the Digilent Nexys Video, which can be found at Digilent's website. This board encompasses a Xilinx/AMD FPGA boasting a high number of logic slices, exceeding 200,000 (specifically, 215,360), as well as a substantial amount of block RAM, surpassing 13 million bits (equivalent to 13,140 kilobits). Notably, this FPGA belongs to the Artix 7 family, and it represents the largest member of this family in terms of resource capacity.

To facilitate rapid FPGA prototyping, a configuration file in the required format for the tool *FuseSoC* was authored and integrated. The core objective of *FuseSoC* is to enhance the reusability of IP cores and provide support for the creation, building, and simulation of diverse designs. Typically, generating a bitstream file for the target device involves creating a project and configuring the vendor-specific toolchain to facilitate the final compilation and bitstream generation. However, with *FuseSoC*, these steps are abstracted, thereby rendering the RTL project nearly vendor-independent. One of the features utilized in this work is the capability of *FuseSoC* to generate FPGA images.

5.6 Conclusion

Within this chapter, two systems generated using the IPSoCGen tool were introduced. Furthermore, an extensive examination of the histogram hardware, firmware, and software host applications was provided. Additionally, key aspects of the implemented set of APIs and the FPGA prototyping process were elucidated, thereby offering a comprehensive overview of the entire workflow.

6 CASE STUDIES

This chapter presents examples of the utilization of the IPSoCGen framework. The primary aim is to elucidate the complete process of generating a System-on-Chip, starting from a high-level plan to the generation of a digital system. Specifically chosen applications are not overly intricate, serving to demonstrate that the process engenders functional and efficient systems. The examples highlight that the process yields operational hardware with outcomes akin to reliable implementations found in open-source IPs (developed by the author). Despite inherent measurement imprecision, the obtained results fall within the expected range.

While not the central focus of this study, it becomes apparent that the performance of the synthesized system is heavily contingent on the initial algorithm. The selection of a suitable algorithm and intelligent task specification and management, particularly concerning the exploration of parallelism, stands out as fundamental for enhancing performance.

Given the widespread interest in systems with multiple processors, attempts were made to ascertain whether this architecture genuinely manifests a performance increase in line with the augmented available resources. To achieve this, some experiments involved repeating the processing to demand more computational effort.

Additionally, it compares the power, performance, and area (PPA) of different systems, elucidating the design choices, bottlenecks encountered, and potential avenues for future improvements.

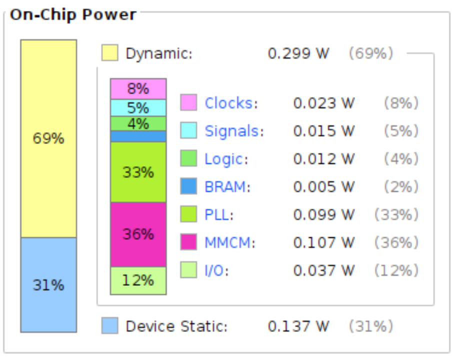
6.1 SoC - Single Processor SoC

The next section will detail each of the three metrics (power, performance, and area) for the SoC running the histogram application.

6.1.1 Power

Starting with the power numbers, the results obtained from the implementation on Vivado (2022.2) demonstrate a total power number estimation of 436mW between dynamic and static, with its breakdown demonstrated in Figure 6.1. It is relevant to highlight that both PLL and the Mixed-Mode Clock Manager (MMCM) together represent 69% of the total estimated power value as no simulation activity file (.saif) was supplied to the tool and it is executed in *vectorless* mode. According to Xilinx/AMD, *vectorless* mode analysis is done based on default switching activity (usually low) specification on the primary ports and the design clocks.

Figure 6.1: SoC power breakdown numbers



On-Chip Power

Dynamic:	0.299 W	(69%)
Clocks:	0.023 W	(8%)
Signals:	0.015 W	(5%)
Logic:	0.012 W	(4%)
BRAM:	0.005 W	(2%)
PLL:	0.099 W	(33%)
MMCM:	0.107 W	(36%)
I/O:	0.037 W	(12%)
Device Static:	0.137 W	(31%)

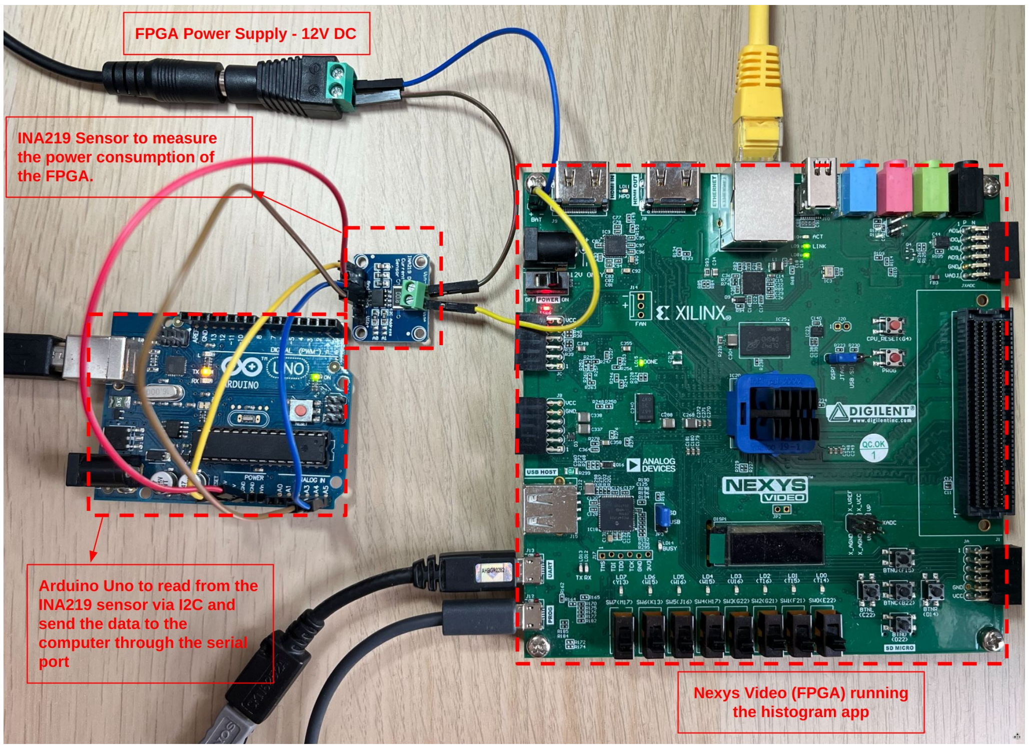
Source: The Author

To obtain a more accurate estimation of the power consumed by the SoC in conjunction with the running application on the target device, the setup shown in Figure 6.2 was assembled. This setup consists of an Arduino Uno and the I2C sensor INA219. The sensor functions by measuring the voltage at its input terminal using a precision amplifier across a 0.1-ohm shunt resistor (1% sense resistor), as well as the bus supply voltage. It then converts both measurements into values that are stored in its internal registers, allowing it to provide voltage, current, and consequently, the instantaneous power value.

A simple firmware was developed for the Arduino Uno to read the sensor and print through its serial port, an average value of the last 10 samples of instant power. An average of 300 hundred measurements was captured and presented in Figure 6.3 for different modes: Running the histogram application, in idle mode waiting to start, and with the reset asserted. The total number of instantaneous power measurements is 3000, as each of the 300 samples represents the average of the last 10 instantaneous power samples.

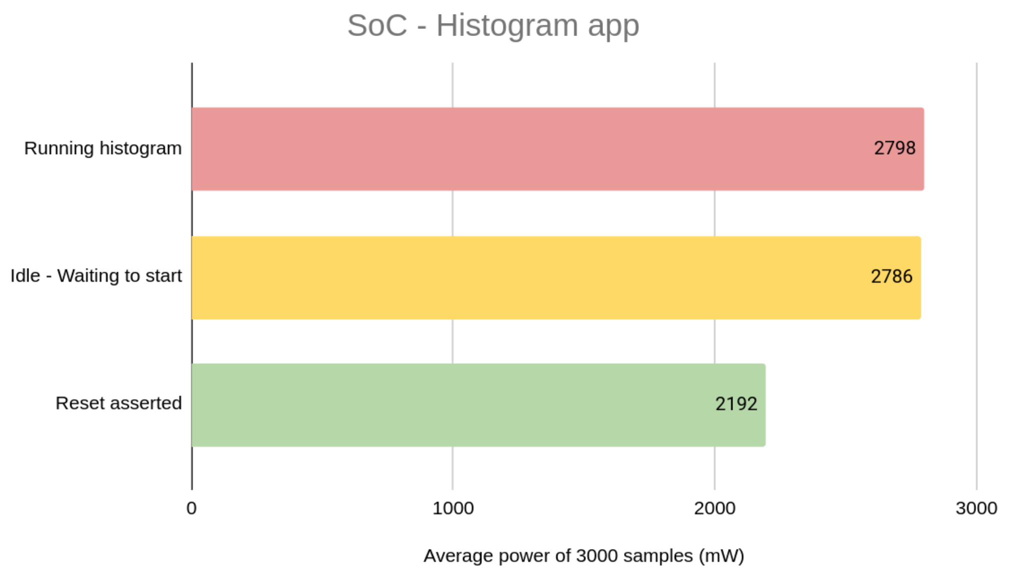
Since the power values are measured at the supply of the FPGA board, several other components (USB-UART, DDR memory, etc) including the efficiency of the DC-DC converters on the board contribute to a total of more than two watts. Based on the obtained power measurements, it is possible to observe that the total power consumption of the design is approximately 2192 milliwatts when reset. Subtracting the idle power

Figure 6.2: Power setup


FPGA Power Supply - 12V DC
INA219 Sensor to measure the power consumption of the FPGA.
Arduino Uno to read from the INA219 sensor via I2C and send the data to the computer through the serial port
Nexys Video (FPGA) running the histogram app

Source: The Author

Figure 6.3: SoC Power comparison chart



Source: The Author

consumption, which occurs when there is no histogram being processed, it is obtained a delta value of around 594 milliwatts. Lastly, the difference between idle power and the power consumed during the execution of the histogram application is 12 milliwatts, indicating that the execution of the application does not significantly increase power consumption.

6.1.2 Performance

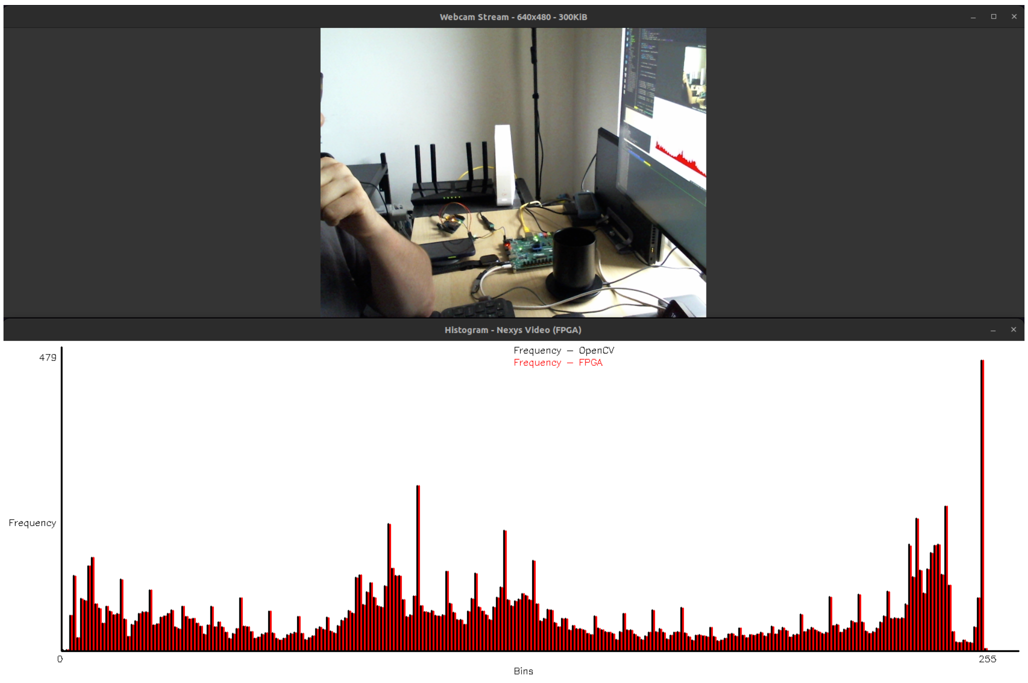
An application written in C++ was used to capture frames from an HD webcam at a resolution of 640x480 (VGA), following the program flow proposed in section 5.3. The frames were captured using the OpenCV library and then converted from a 3-channel (RGB) format to a single grayscale channel ranging from 0 to 255. All the 256 bins of the histogram are normalized from values between 0 to 479 (higher frequency). The total memory size for each frame was 300 KiB (1 byte x 640 x 480 pixels).

Each 1 KiB segment was then sent to the FPGA, which accumulated the total histogram data. Once all segments were completed, the SoC returned the histogram in the format of 4 bytes per bin, totaling 1024 bytes (4 bytes x 256 bins). The choice of a 1 KiB segment was made by the author due to a limitation of an internal design within the Ethernet AXI, which does not support UDP packet fragmentation. Fragmentation occurs when the packet size exceeds 1500 bytes (the maximum MTU size), and the frames are sent separately.

The performance analysis starts by measuring the total time taken to process each frame of the 300 KiB sent from the Webcam Stream to the FPGA. It was observed that, on average of 300 frames, it took 590.30 ms per frame. This time is measured from the moment the host program sends the *histogram* command, through the image being split into 300 UDP packets, to the return of the histogram vector (1 KiB).

On an AMD Ryzen 9 5900X with 64 GiB DDR4 RAM, OpenCV computed the histogram in an average time of 0.35 ms, this time taken is not used for comparison with the MPSoC but to indicate that the host computation does not affect the time taken to transmit the frames to the FPGA in the host application. As it can be observed in Figure 6.4, the histogram reference computed by OpenCV (black bars) is similar to the one computed by the FPGA (red bars), with a correlation of 99.94% (average of 300 frames). The correlation was computed using the *cv::compareHist* function from the OpenCV library that follows the math expression 6.1/6.2.

Figure 6.4: SoC running histogram app


Webcam Stream - 640x480 - 300KiB
Histogram - Nexys Video (FPGA)
479
Frequency
Frequency – OpenCV
Frequency – FPGA
0
255
Bins

Source: The Author

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2)}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}} \tag{6.1}$$

where

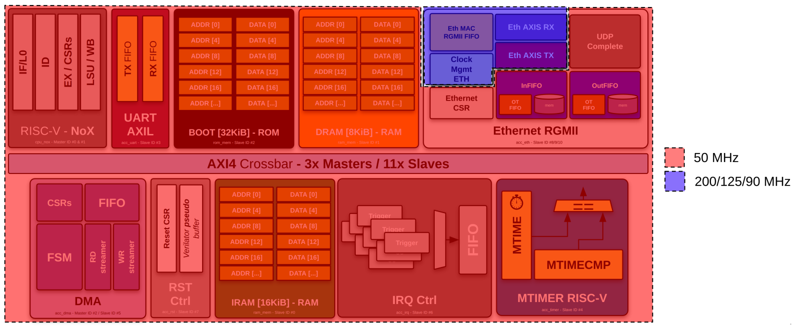
$$H1/H2 = Histograms, \bar{H}_k = \frac{1}{N} \sum_J H_k(J) \tag{6.2}$$

6.1.3 Area

The design was implemented with a frequency plan consisting of four distinct clock domains. Figure 6.5 illustrates the mapping of these clocks within the design modules. The majority of the design operates at a frequency of 50 MHz, which is generated by a Phase-Locked Loop (PLL) (not depicted in Figure 6.5). This PLL is responsible for deriving the 50 MHz clock from the main clock input source of 100 MHz, provided by the Nexys Video board.

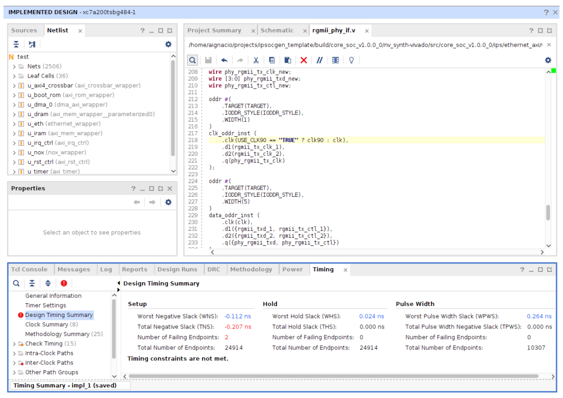
Due to the utilization of the RGMII interface in the Ethernet AXI design for communication with the on-board PHY IC, a minimum of three distinct clock frequencies is necessary. These frequencies are specifically set at 200 MHz, 125 MHz, and 90 MHz

Figure 6.5: SoC Frequency plan



Source: The Author

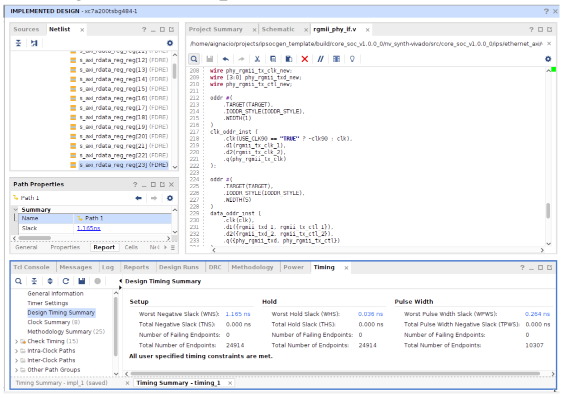
Figure 6.6: Timing report issue during FPGA implementation



```
wire phy_rgmii_tx_clk_new;
wire [3:0] phy_rgmii_txd_new;
wire phy_rgmii_tx_ctl_new;

oddr #(
    .TARGET(TARGET),
    .IODDR_STYLE(IODDR_STYLE),
    .WIDTH(1)
)
clk_oddr_inst (
    .clk(USE_CLK90 == "TRUE" ? clk90 : clk),
    .d1(rgmii_tx_clk_1),
    .d2(rgmii_tx_clk_2),
    .q(phy_rgmii_tx_clk)
);

oddr #(
    .TARGET(TARGET),
    .IODDR_STYLE(IODDR_STYLE),
    .WIDTH(5)
)
data_oddr_inst (
    .clk(clk),
    .d1({rgmii_txd_1, rgmii_tx_ctl_1}),
    .d2({rgmii_txd_2, rgmii_tx_ctl_2}),
    .q({phy_rgmii_txd, phy_rgmii_tx_ctl})
```

```
Design Timing Summary

Setup                                        Hold                                       Pulse Width
Worst Negative Slack (WNS):    -0.112 ns     Worst Hold Slack (WHS):       0.024 ns     Worst Pulse Width Slack (WPWS):            0.264 ns
Total Negative Slack (TNS):    -0.207 ns     Total Hold Slack (THS):       0.000 ns     Total Pulse Width Negative Slack (TPWS):   0.000 ns
Number of Failing Endpoints:   2             Number of Failing Endpoints:  0            Number of Failing Endpoints:               0
Total Number of Endpoints:     24914         Total Number of Endpoints:    24914        Total Number of Endpoints:                 10307
Timing constraints are not met.
```

(a) Setup violation on phy_if



```
wire phy_rgmii_tx_clk_new;
wire [3:0] phy_rgmii_txd_new;
wire phy_rgmii_tx_ctl_new;

oddr #(
    .TARGET(TARGET),
    .IODDR_STYLE(IODDR_STYLE),
    .WIDTH(1)
)
clk_oddr_inst (
    .clk(USE_CLK90 == "TRUE" ? ~clk90 : clk),
    .d1(rgmii_tx_clk_1),
    .d2(rgmii_tx_clk_2),
    .q(phy_rgmii_tx_clk)
);

oddr #(
    .TARGET(TARGET),
    .IODDR_STYLE(IODDR_STYLE),
    .WIDTH(5)
)
data_oddr_inst (
    .clk(clk),
    .d1({rgmii_txd_1, rgmii_tx_ctl_1}),
    .d2({rgmii_txd_2, rgmii_tx_ctl_2}),
    .q({phy_rgmii_txd, phy_rgmii_tx_ctl})
```

Summary	
Name	Path 1
Slack	1.165ns

```
Design Timing Summary

Setup                                        Hold                                       Pulse Width
Worst Negative Slack (WNS):    1.165 ns      Worst Hold Slack (WHS):       0.036 ns     Worst Pulse Width Slack (WPWS):            0.264 ns
Total Negative Slack (TNS):    0.000 ns      Total Hold Slack (THS):       0.000 ns     Total Pulse Width Negative Slack (TPWS):   0.000 ns
Number of Failing Endpoints:   0             Number of Failing Endpoints:  0            Number of Failing Endpoints:               0
Total Number of Endpoints:     24914         Total Number of Endpoints:    24914        Total Number of Endpoints:                 10307
All user specified timing constraints are met.
```

IMPLEMENTED DESIGN - xc7a200tsbg484-1
Sources
Netlist
s_axi_rdata_reg_reg[12] (FDRE)
s_axi_rdata_reg_reg[13] (FDRE)
s_axi_rdata_reg_reg[14] (FDRE)
s_axi_rdata_reg_reg[15] (FDRE)
s_axi_rdata_reg_reg[16] (FDRE)
s_axi_rdata_reg_reg[17] (FDRE)
s_axi_rdata_reg_reg[18] (FDRE)
s_axi_rdata_reg_reg[19] (FDRE)
s_axi_rdata_reg_reg[20] (FDRE)
s_axi_rdata_reg_reg[21] (FDRE)
s_axi_rdata_reg_reg[22] (FDRE)
s_axi_rdata_reg_reg[23] (FDRE)
Path Properties
Path 1
General
Properties
Report
Cells
Project Summary
Schematic
rgmii_phy_if.v
/home/aignacio/projects/ipsocgen_template/build/core_soc_v1.0.0_0/nv_synth-vivado/src/core_soc_v1.0.0_0/ips/ethernet_axi/
Tcl Console
Messages
Log
Reports
Design Runs
DRC
Methodology
Power
Timing
General Information
Timer Settings
Design Timing Summary
Clock Summary (8)
Methodology Summary (25)
Check Timing (15)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
Timing Summary - impl_1 (saved)
Timing Summary - timing_1

(b) Positive WNS after the fix

Source: The Author

generated by the *Clock Mgmt Eth* block (MMCM). The primary purpose of these clock frequencies is to effectively meet the timing requirements of the RGMII interface while guaranteeing reliable communication between the Ethernet PHY (Physical Layer) and MAC (Media Access Control) layers.

A problem that arose during the synthesis process for the FPGA in question involved the identification of two setup violations (refer to WNS in the Figure 6.6a) noted in the final report. It is worth mentioning that the author of the open-source project known as *verilog-ethernet* has acknowledged this known issue specifically for this particular board, attributing it to the stringent timing requirements inherent in the Ethernet design. However, a resolution was provided by an external individual via the Internet, which effectively addressed the timing concerns and facilitated the attainment of the desired timing closure (refer to 6.6b). Apart from this particular issue, no additional timing violations were detected during the design synthesis phase.

Upon completion of the implementation stage, the comprehensive FPGA area utilization has been summarized in Table 6.1. The employment of Look-Up Tables (LUTs) by the System-on-Chip (SoC) amounts to 10452, accounting for 7.81% of the overall resources. In terms of flip-flops, a total of 10012 units have been utilized, representing 3.72% of the available resources. Regarding Block RAM (BRAM), there are 27 instances employed, consisting of two instances of RAMB18 (18 Kbits) and twenty-five instances of RAMB36 (36 Kbits).

Table 6.1: Nexys Video FPGA total utilization (SoC)

Resource	**Utilization**	**Available**	**Utilization %**
LUT	10452	133800	7.81
LUTRAM	116	46200	0.25
FF	10012	269200	3.72
BRAM	27	365	7.40
IO	19	285	6.67
BUFG	5	32	15.63
MMCM	1	10	10.00
PLL	1	10	10.00

Source: The Author

Furthermore, a hierarchical report of the design was generated, providing a breakdown of resources per module, as detailed in Table 6.2. Notably, the Ethernet AXI design occupies the largest area, accounting for 29.45% of the total. Following closely is the NoX RISC-V core, which utilizes 2685 LUTs, representing 25.68% of the overall resources. Additionally, the AXI4 Crossbar contributes significantly to the design, occupying 20.83% of the total area in terms of LUTs.

Table 6.2: SoC - Hierarchical area breakdown

Name	**Slice LUTs**	**Slice Registers**	**Block RAM**
SoC	10452	10012	27
u_eth (ethernet_wrapper)	3078	3895	7
u_nox (nox_wrapper)	2685	1888	0
u_axi4_crossbar (axi_crossbar_wrapper)	2177	1681	0
u_dma_0 (dma_axi_wrapper)	1590	1688	0
u_uart (axi_uart_wrapper)	452	351	0
u_timer (axi_timer)	183	169	0
u_dram (axi_mem_wrapper)	100	84	4
u_iram (axi_mem_wrapper)	78	87	8
u_irq_ctrl (axi_irq_ctrl)	69	103	0
u_rst_ctrl (axi_rst_ctrl)	44	49	0
u_boot_rom (axi_rom_wrapper)	12	8	8

Source: The Author

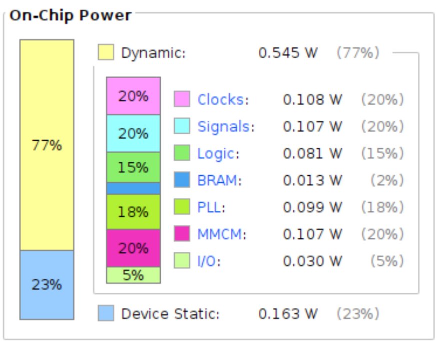
6.2 MPSoC - Multi-Processor System-on-Chip

In the following section, we will elaborate on each of the three fundamental metrics, namely power, performance, and area, in relation to the Multi-Processor System-on-Chip (MPSoC) employed for executing the histogram application.

6.2.1 Power

Starting by power, two types of results were compared, the initial estimation *vectorless* from Vivado and the measurement captured by the sensor INA219 (same setup as 6.1.1). The initial estimation from Vivado is presented in Figure 6.7, where 545 mW is the dynamic and 163 mW is the static power.

Figure 6.7: MPSoC power breakdown numbers



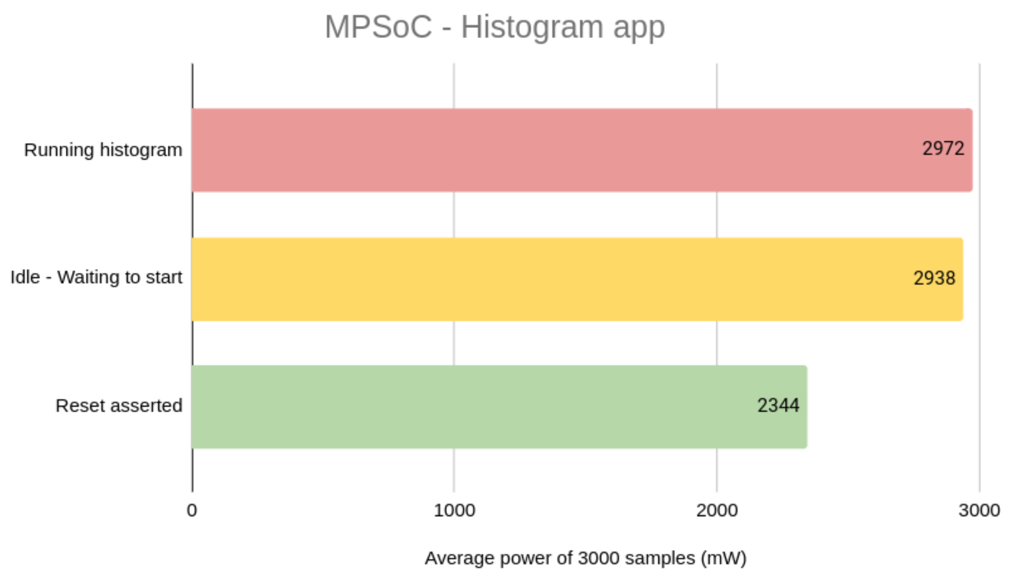
On-Chip Power

🟨	Dynamic:	0.545 W	(77%)
🟪	Clocks:	0.108 W	(20%)
🟦	Signals:	0.107 W	(20%)
🟩	Logic:	0.081 W	(15%)
🟦	BRAM:	0.013 W	(2%)
🟩	PLL:	0.099 W	(18%)
🟪	MMCM:	0.107 W	(20%)
🟩	I/O:	0.030 W	(5%)
🟦	Device Static:	0.163 W	(23%)

Source: The Author

While measuring the MPSoC using the previous setup, three distinct measurements were conducted using 300 samples (each sample is the average of the last 10 instantaneous power measurements). These measurements encompassed the execution of the histogram, the idle mode, and the reset being asserted. Figure 6.8 demonstrates that the disparity between idle mode and the running histogram application amounts to 34 mW. Additionally, the variance between idle mode and the reset being asserted is 594 mW.

Figure 6.8: MPSoC Power comparison chart



Source: The Author

6.2.2 Performance

The execution of the histogram application on the MPSoC, using the same C++ program mentioned in section 6.1.2, demonstrates an average time measurement of 339.49 ms per frame, where each frame size is 300 KiB (300 samples, same as the SoC). Additionally, the achieved average histogram correlation is 99.94%, matching that of the SoC. It is important to note that the FreeRTOS version used was v10.4.4+ with a tick timer configured at 500 Hz to handle task context switching for both SoC and MPSoC applications.

It is essential to emphasize the challenges involved in accurately measuring individual events during program execution, such as the transmission time for a 1 KiB image segment to the slave tiles for example. This complexity arises due to context switching during RTOS execution and the potential latency between each step, which may introduce deviations from the true values.

6.2.3 Area

In regards to the area utilization, the MPSoC exhibits a total LUT utilization of approximately 57.52% of the Nexys Video FPGA, which is around 7.4 times larger than the previously described SoC, according to the Table 6.3. The number of flip-flops in the MPSoC amounts to 63,289, significantly exceeding that of the reference design due to its higher complexity. A similar trend is observed for the BRAM, where the MPSoC comprises almost seven times more instances of both RAMB36 and RAMB18 compared to the reference design. As this design follows a similar frequency plan to the SoC, the clock distribution is the same for the master tile (tile_0), while the remaining slave tiles operate at 50 MHz.

Table 6.3: Nexys Video FPGA total utilization (MPSoC)

Resource	**Utilization**	**Available**	**Utilization %**
LUT	77428	134600	57.52
LUTRAM	212	46200	0.46
FF	63307	269200	23.52
BRAM	187	365	51.23
IO	20	285	7.02
MMCM	1	10	10.00
PLL	1	10	10.00

Source: The Author

Table 6.4 provides a comprehensive breakdown of each tile within the design. The majority of tiles exhibit similar area sizes, excluding the first tile (tile_0), which serves as the master tile responsible for managing communication with the host. This includes Ethernet AXI and larger memory components. Additionally, the NoC contributes significantly to the total area, surpassing the average size of the slave tile. Adjusting the parameters of the NoC can potentially reduce its size.

Table 6.4: MPSoC - Hierarchical area breakdown

Name	**Slice LUTs**	**Slice Registers**	**Block RAM**
MPSoC	77428	63307	187
u_tile_8 (tile_8)	7355	5996	20
u_tile_7 (tile_7)	7361	5996	20
u_tile_6 (tile_6)	7350	5996	20
u_tile_5 (tile_5)	7351	5996	20
u_tile_4 (tile_4)	7349	5996	20
u_tile_3 (tile_3)	7347	5996	20
u_tile_2 (tile_2)	7347	5996	20
u_tile_1 (tile_1)	7349	5996	20
u_tile_0 (tile_0)	10719	10181	27
u_ravenoc (ravenoc)	7902	5149	0

Source: The Author

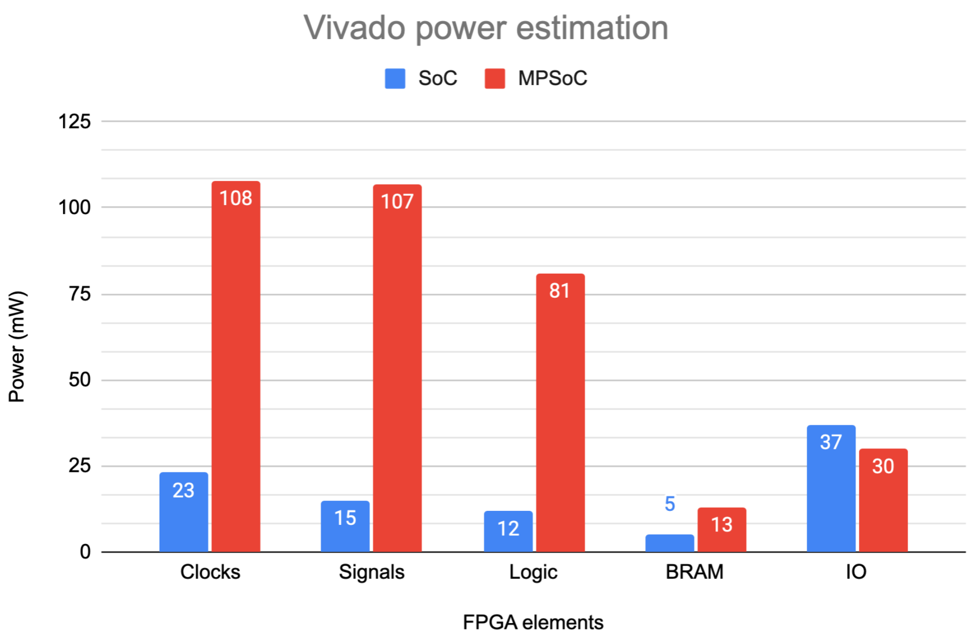
6.3 PPA evaluation

In the following section, an analysis will be conducted to compare various metrics, including power consumption, performance, and area, for the two presented systems. Furthermore, supplementary performance tests will be presented to enhance the *benchmark* of these systems.

6.3.1 Power

Upon examination of the two preceding power charts (Figure 6.7 and Figure 6.1), it becomes evident that the projected power consumption of the MPSoC (Multi-Processor System-on-Chip) is significantly greater when compared to the SoC (System-on-Chip), as reported by Vivado. The comparison of total power consumption estimates provided by the tool reveals a noteworthy finding: the MPSoC displays a 62.38% higher total power consumption in contrast to the SoC.

Figure 6.9: Dynamic power estimation per FPGA element - Vivado

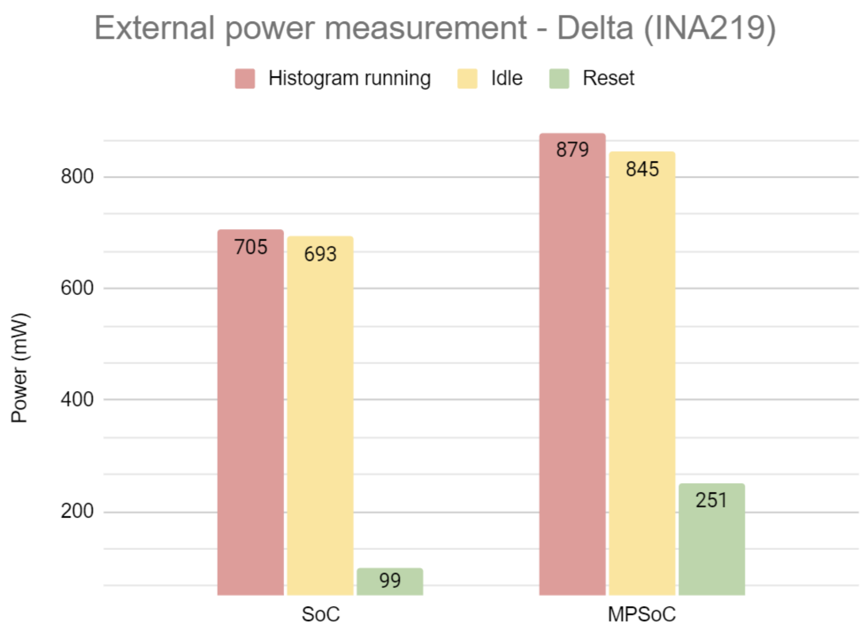


Source: The Author

Further scrutiny of the factors contributing to this power escalation, as depicted in Figure 6.9, elucidates that clocks, signals, and logic constitute the primary drivers, resulting in a respective increase of 369%, 613%, and 575% in dynamic power consumption relative to the SoC. The estimation of static power also exhibits a substantial increment, with an approximate rise of 19%.

To conduct a more thorough analysis of the measurements obtained from the INA219 sensor, a basic LED blink design was synthesized and measured on the Nexys Video FPGA using the FuseSoC tool's reference design. This design, being minimal, serves the purpose of providing insights into the underlying power consumption of the baseboard. It is challenging to accurately estimate the contribution of the entire set of chips comprising the circuitry to the previously measured power values.

Figure 6.10: External measurement (INA219) - Delta using blink design as reference



Source: The Author

Through this particular test, the power consumption recorded was 2093.24 mW. This value was subsequently subtracted from the aforementioned diagrams (Figure 6.3 and Figure 6.8), and the results are presented in Figure 6.10. The aforementioned figure reveals that the power difference between the two systems is relatively insignificant during idle or when the histogram is in operation, but becomes more noteworthy when the system's reset is asserted. This observation suggests that not all the tiles within the MPSoC may be operating concurrently, with a significant portion of them being in an idle state considering the histogram application.

6.3.2 Performance

When assessing the performance of the two systems, it is apparent that the MPSoC achieves a speedup of 73.88% over the SoC. This gain can be attributed to parallel processing facilitated by distributing image processing across multiple tiles via the NoC. However, this advantage does not exhibit a direct proportionality to the number of tiles (8). This can be attributed to the following possibilities:

1. The duration required for the master tile to delegate a certain workload to the slave tiles is highly comparable to the time taken by a slave tile to handle the image segment, perform the necessary processing, and return to the master tile.
2. The algorithm utilized by the MPSoC for load allocation inadvertently favors the first and second tiles. Additionally, thanks to the minimal processing time required for payload processing, these slave tiles promptly generate the histogram and become ready for the subsequent processing iteration. This occurs even before the master tile completes its iteration through a new slave tile.

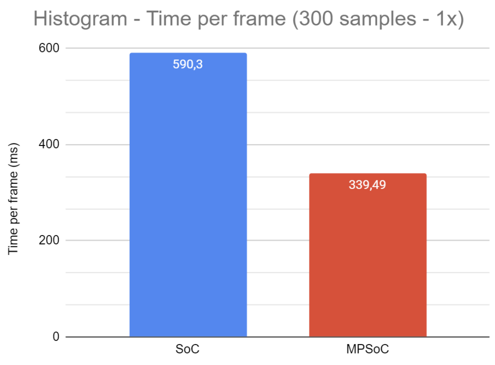
To evaluate the aforementioned hypothesis, the histogram application was modified to compute the histogram segment multiple times, specifically ten times, to increase the payload processing time. In Figure 6.11a on the leftmost side, the average time per frame for a single processing iteration is shown, while Figure 6.11b on the rightmost side represents the average time per frame when the computation of the histogram is requested ten times within each frame. As illustrated in Figure 6.11c, the speedup experiences a significant increase from 73.88% to 504.47% as the processing time per frame is extended. This observation further supports the hypothesis made earlier.

6.3.2.1 MPSoC performance exploration

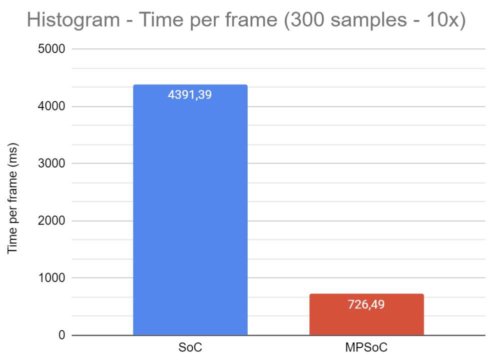
To conduct a comprehensive evaluation of the performance gain offered by the MPSoC in comparison to the SoC, three distinct *firmware* implementations were developed for the aforementioned systems. One *firmware* was specifically designed for the SoC, while two separate *firmware* versions were created for the MPSoC, targeting its master and slave tiles.

The purpose of this *firmware* is to receive a scalar value S and a number N through Ethernet communication. Subsequently, the system executes the multiplication of 8 KiB array (initialized with ones) by this scalar S value N times, accumulating the results and

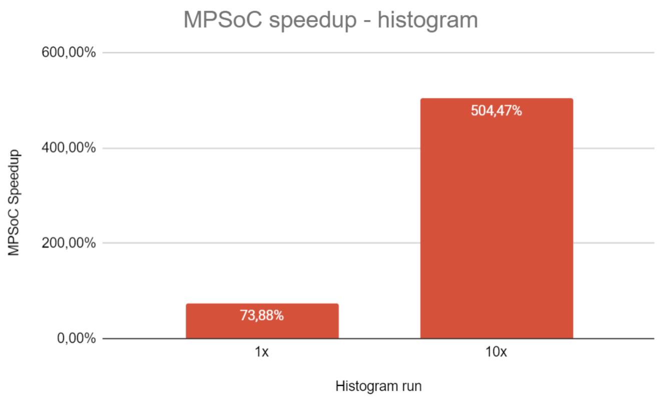
Figure 6.11: Histogram MPSoC speedup comparison



(a) Time per frame



(b) Time per frame - 10x histogram



(c) Histogram performance comparison

Source: The Author

returning the total sum to the host through Ethernet as well (Equation 6.3). The primary objective behind this design is to compare the performance gain achievable by serially contrasting the execution using the SoC, versus the parallel execution facilitated by the MPSoC. Although this firmware does not represent a specific algorithm, it emulates a general processing task that could be executed by both platforms during the processing of large blobs such as image, audio, or cipher blocks.

$$Total = \sum_{i=0}^{i=N-1} S * array\_8KiB[i] \tag{6.3}$$

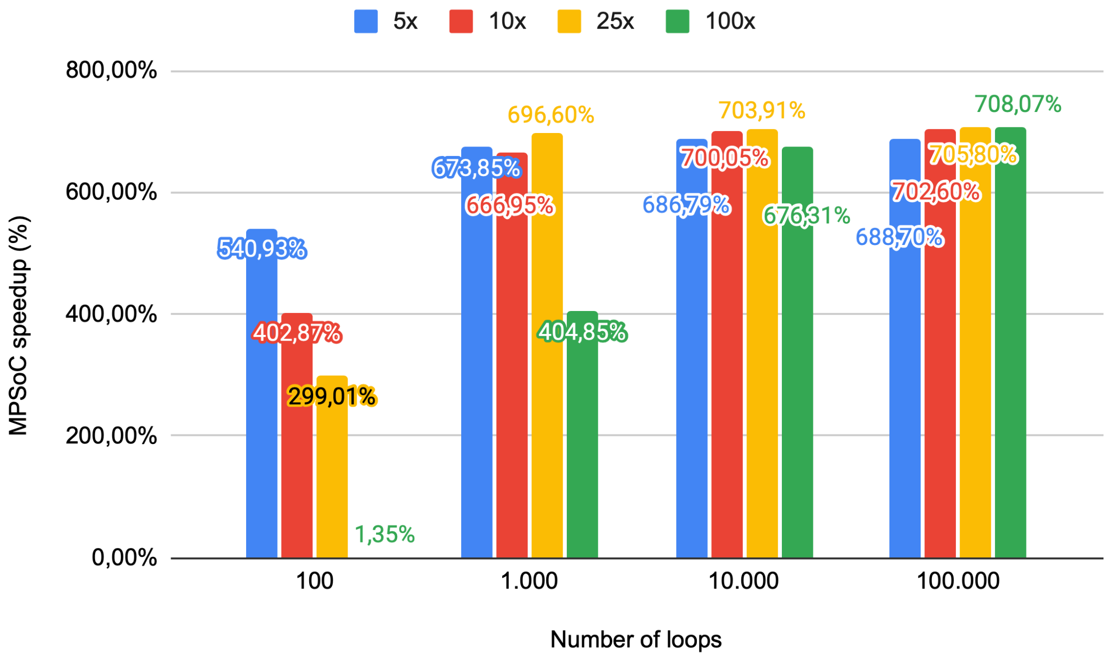
The algorithm under consideration is not excessively intricate but does entail a substantial computational load. Nonetheless, it effectively harnesses the parallel advantage of the MPSoC. The rationale behind selecting this algorithm lies in the potential impact that a more complex alternative could have on the measured throughput. Conversely, the proposed solution minimizes the significance of processing time consumed by factors such as OS context switching, Ethernet operations by the master tile, and master-slave NoC communication. These components are inherently challenging to measure accurately and are subject to considerable variation due to various factors.

Both iterations of the *firmware* were recorded, with approximately 100 samples collected for each run. The scalar factor S remained constant at a value of 123, while the number of loops N varied between 100 and 100,000. In the case of parallel execution, the evaluation also encompassed different quantities of loops processed by each slave tile, namely 5, 10, 25, and 100 loops per tile. The routing algorithm employed for task distribution among multiple slave tiles followed a round-robin approach based on loop granularity. The performance comparison between the two systems, specifically in terms of the speedup achieved through parallel execution, is graphically depicted in Figure 6.12.

Figure 6.12 illustrates the observed speedup, ranging from 1.35% (with a task size of 100 loops per tile) to 708% (with a task size of 100,000 loops and 100 loops per tile). Beginning with the smallest gain mentioned previously, a marginal speedup of 1.35% is notable in parallel execution. In this specific scenario, a single processor handles the execution of 100x loops, and the slight advantage of the MPSoC arises from the dedicated execution of interrupts and Ethernet packet handling by the master tile. In contrast, the SoC performs all tasks within a single context, and the communication overhead in the NoC is relatively low compared to the processing time.

Furthermore, when the parallel execution distributes 5x loops per tile, the speedup gain appears to be more evenly distributed across the various workloads. This is because

Figure 6.12: MPSoC speedup - Comparison of different amounts of workloads per number of loops



Source: The Author

the processing time for 5x loops (5x 123*8KiB array) is equivalent to the time taken by the master to dispatch one task (5x) to all the slave tiles through the NoC. As a result, both the master and the slaves remain occupied processing data simultaneously at any given time.

Regarding the highest speedup gain (708.07%), it is notable that it closely approaches the theoretical limit proposed by Amdahl's Law (AMDAHL, 1967) (Equation 6.4). Considering the proposed MPSoC system, where 99% of the code can be executed in parallel and a nine-slave-tile speedup factor, the enhanced speedup limit is calculated as 8.33 times, which is remarkably close to the observed experimental results (8,08 or 708,7%).

$$Speedup_{enhanced}(f, S) = \frac{1}{(1-f) + \frac{f}{S}} \tag{6.4}$$

where:

f = Proportion of overall execution time spent by the part of the task that benefits from parallel processing.

S = Performance improvement, or speedup.

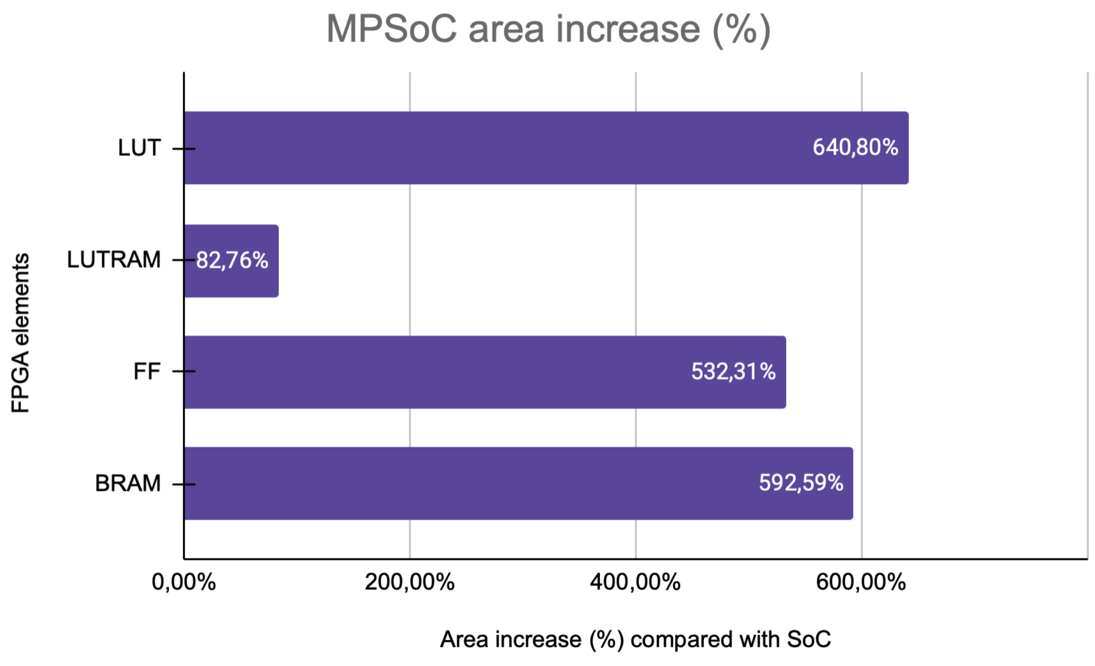
In terms of performance when considering a more complex algorithm, several features were incorporated into the framework to facilitate the acceleration of parallel processing through the MPSoC. For example, the framework allows for the configuration of a variable number of DMAs, enabling the final design to encompass multiple data movers capable of relieving the processor from the task of data movement within the system. Additionally, the system offers the flexibility to incorporate custom accelerators, which can be tailored to deploy highly specialized tasks within specific processing domains. Furthermore, the NoC design was specifically engineered to facilitate low-latency intercommunication and support AXI bursts, effectively complementing the enhancement of throughput. Rigorous tests conducted on the RaveNoC design have demonstrated a throughput of 1415.31 MiB/s (1484.06 MB/s or 1.48 GB/s) while operating at a clock frequency of 200 MHz and configured with data length of 64 bits, which is more than 92% of the theoretical limit (8 Bytes / 5 ns clock period).

6.3.3 Area

In terms of area, the multi-processor system-on-a-chip design occupies a significant area size of the Nexys Video FPGA while compared to the SoC. As demonstrated in the Figure 6.13, the LUT, FF and BRAM increase is 640.80%, 532.31% and 592.59% respectively.

This increase primarily stems from the additional tiles and the inclusion of the network-on-chip design, which accounts for approximately 10% of the MPSoC's LUT usage. The NoC design could be further optimized by reducing the number of *flit* buffers. Additionally, the size of the instruction and data RAM used by the various tiles could be optimized, as even in debug mode compilation, the overall usage does not exceed an average of 70% of its nominal value.

Figure 6.13: MPSoC design area increase in comparison to the SoC



Source: The Author

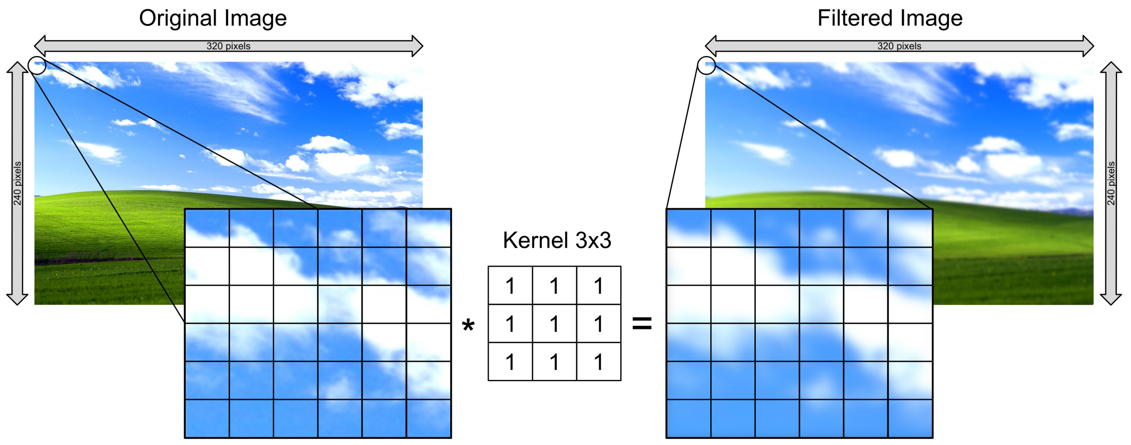
6.4 Linear mean filter application

Another application implemented for benchmarking the proposed framework involves the execution of the image linear mean filtering algorithm. The mean filter technique is fundamentally a process for noise reduction while retaining significant features within images, as described in (VALLEPALLI; RAJENDRAN, 2012). The most basic form of spatial averaging involves aggregating pixel brightness values within small regions of the image. This aggregation is achieved by summing the pixel values, dividing by the number of pixels in the neighborhood, and utilizing the resultant value to generate a new image.

The implemented application uses a box filter, achieved by applying the kernel shown in Figure 6.14 to the noisy image. This is a straightforward algorithm that provides insights into how applications can benefit from hardware parallelism. Next, we plan to implement Motion Estimation, commonly used in video compression. This is expected to demonstrate even more clearly how the algorithm can be tailored to fit the capabilities of the hardware.

To reduce the complexity of the application and the different firmware/software

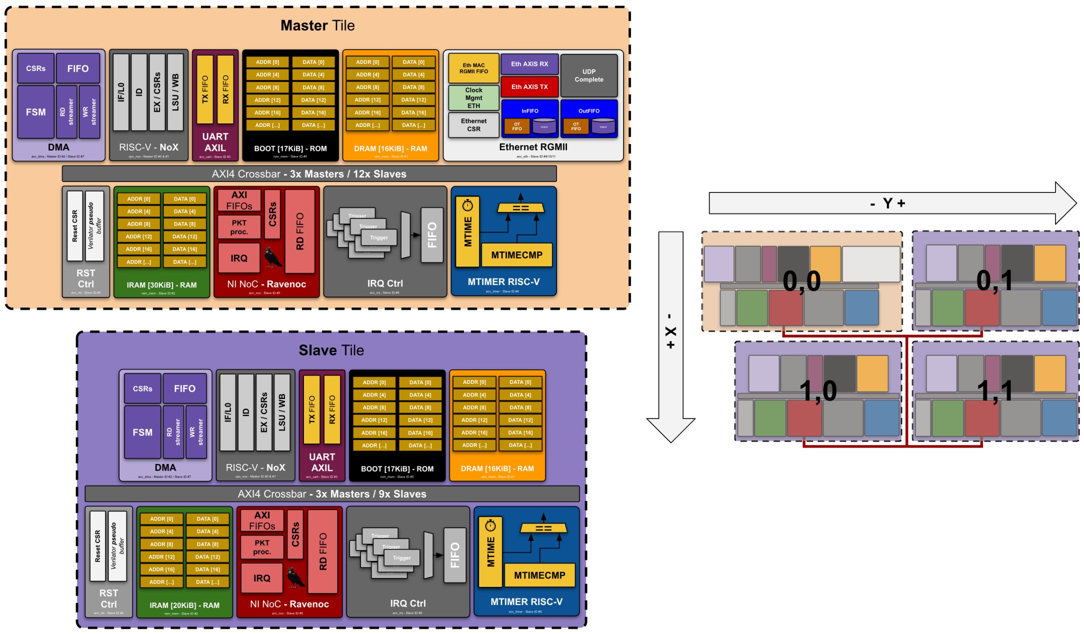
Figure 6.14: Mean filter diagram



Source: The Author

development, a set of constraints has been delineated. These constraints encompass the utilization of a 3x3 kernel and a grayscale image with dimensions of 320x240 pixels. Also, for this development, the application is compared running on a single core against a 2x2 MPSoC design as shown in the image below.

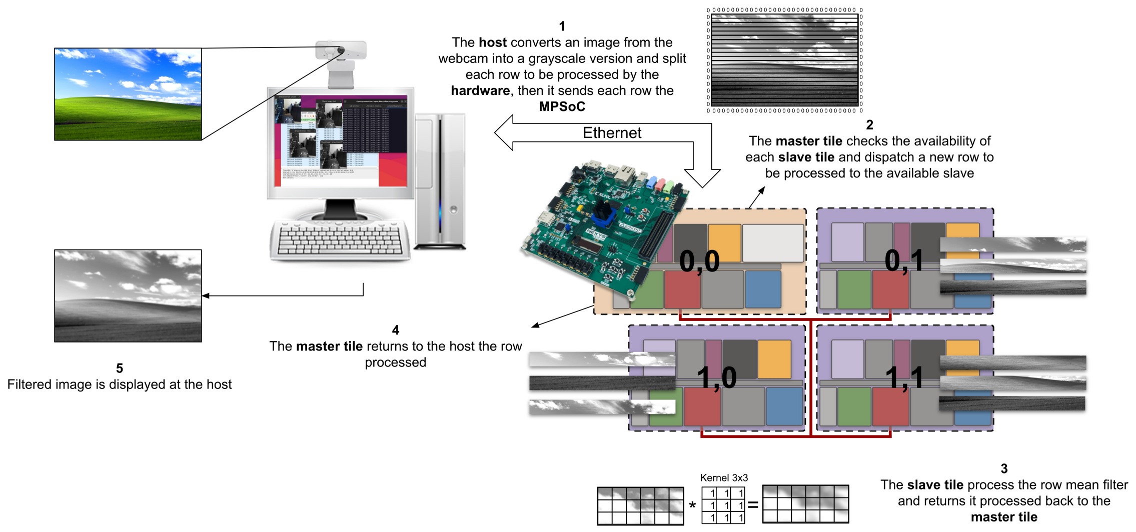
Figure 6.15: Design used to benchmark the mean filter



Source: The Author

The same design built for the MPSoC (Figure 6.15) was also the one used to run the single core application, this avoids re-flashing the bitstream into the FPGA and simplifies the general flow. While running the single-core app, the NoC AXI slave is not used and all the processing is managed locally at the core level. Also, the same host software is shared between the single core vs the MPSoC design app, which is responsible for capturing and displaying the results.

Figure 6.16: Linear mean filter running on the MPSoC


1
The host converts an image from the webcam into a grayscale version and split each row to be processed by the hardware, then it sends each row the MPSoC
Ethernet
2
The master tile checks the availability of each slave tile and dispatch a new row to be processed to the available slave
0,0
0,1
1,0
1,1
4
The master tile returns to the host the row processed
5
Filtered image is displayed at the host
Kernel 3x3
3
The slave tile process the row mean filter and returns it processed back to the master tile

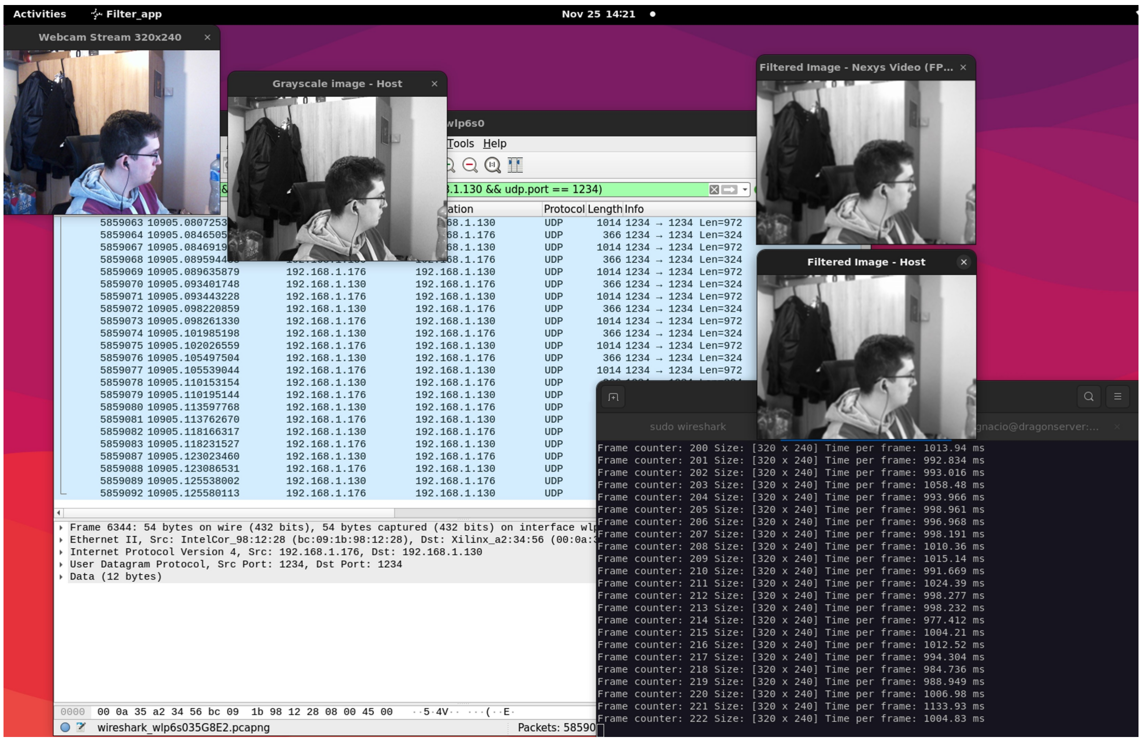
Source: The Author

In Figure 6.16, the sequential execution flow of the system is illustrated, delineating the distinct stages that each frame undergoes. Commencing on the host side, an image is captured via a webcam and subsequently converted into grayscale format (single channel). This grayscale image undergoes zero-padding and is then partitioned into rows within an array structure. Each entry within this array encapsulates the current row, its predecessor, and its successor (comprising a total of 3x rows). This simplifies the embedded software and each entry can be individually processed.

Following this arrangement, each array entry, accompanied by a small header (4 bytes) denoting its index, is transmitted via Ethernet to the FPGA. Upon reception of the array, the FPGA ascertains the availability of slave processing units and if available, dispatches the row through the Network-on-Chip (NoC) to the designated slave for mean filter processing. If no rows are currently prepared for return to the host, the master tile issues a request for additional array entries.

Upon the completion of processing by the slave tile, the processed row is transmitted back to the master tile, where it is locally stored and subsequently forwarded to the host via Ethernet. Once all rows are received by the host software, the filtered image is displayed, and the process recommences by capturing a new frame.

Figure 6.17: Host side application execution



```
.1.130 && udp.port == 1234)
                                              ation           Protocol Length Info
5859063 10905.0807253                          68.1.130        UDP      1014 1234 → 1234 Len=972
5859064 10905.0846505                          68.1.176        UDP       366 1234 → 1234 Len=324
5859067 10905.0846919                          68.1.130        UDP      1014 1234 → 1234 Len=972
5859068 10905.0895944                          68.1.176        UDP       366 1234 → 1234 Len=324
5859069 10905.089635879   192.168.1.176   192.168.1.130   UDP      1014 1234 → 1234 Len=972
5859070 10905.093401748   192.168.1.130   192.168.1.176   UDP       366 1234 → 1234 Len=324
5859071 10905.093443228   192.168.1.176   192.168.1.130   UDP      1014 1234 → 1234 Len=972
5859072 10905.098220859   192.168.1.130   192.168.1.176   UDP       366 1234 → 1234 Len=324
5859073 10905.098261330   192.168.1.176   192.168.1.130   UDP      1014 1234 → 1234 Len=972
5859074 10905.101985198   192.168.1.130   192.168.1.176   UDP       366 1234 → 1234 Len=324
5859075 10905.102026559   192.168.1.176   192.168.1.130   UDP      1014 1234 → 1234 Len=972
5859076 10905.105497504   192.168.1.130   192.168.1.176   UDP       366 1234 → 1234 Len=324
5859077 10905.105539044   192.168.1.176   192.168.1.130   UDP      1014 1234 → 1234 Len=972
5859078 10905.110153154   192.168.1.130   192.168.1.176   UDP
5859079 10905.110195144   192.168.1.176   192.168.1.130   UDP
5859080 10905.113597768   192.168.1.130   192.168.1.176   UDP
5859081 10905.113762670   192.168.1.176   192.168.1.130   UDP
5859082 10905.118166317   192.168.1.130   192.168.1.176   UDP
5859083 10905.118231527   192.168.1.176   192.168.1.130   UDP
5859087 10905.123023460   192.168.1.130   192.168.1.176   UDP
5859088 10905.123086531   192.168.1.176   192.168.1.130   UDP
5859089 10905.125538002   192.168.1.130   192.168.1.176   UDP
5859092 10905.125580113   192.168.1.176   192.168.1.130   UDP

▸ Frame 6344: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface wl
▸ Ethernet II, Src: IntelCor_98:12:28 (bc:09:1b:98:12:28), Dst: Xilinx_a2:34:56 (00:0a:3
▸ Internet Protocol Version 4, Src: 192.168.1.176, Dst: 192.168.1.130
▸ User Datagram Protocol, Src Port: 1234, Dst Port: 1234
▸ Data (12 bytes)

0000  00 0a 35 a2 34 56 bc 09  1b 98 12 28 08 00 45 00   ··5·4V·· ···(··E·
wireshark_wlp6s035G8E2.pcapng                                Packets: 58590
```

```
Frame counter: 200 Size: [320 x 240] Time per frame: 1013.94 ms
Frame counter: 201 Size: [320 x 240] Time per frame: 992.834 ms
Frame counter: 202 Size: [320 x 240] Time per frame: 993.016 ms
Frame counter: 203 Size: [320 x 240] Time per frame: 1058.48 ms
Frame counter: 204 Size: [320 x 240] Time per frame: 993.966 ms
Frame counter: 205 Size: [320 x 240] Time per frame: 998.961 ms
Frame counter: 206 Size: [320 x 240] Time per frame: 996.968 ms
Frame counter: 207 Size: [320 x 240] Time per frame: 998.191 ms
Frame counter: 208 Size: [320 x 240] Time per frame: 1010.36 ms
Frame counter: 209 Size: [320 x 240] Time per frame: 1015.14 ms
Frame counter: 210 Size: [320 x 240] Time per frame: 991.669 ms
Frame counter: 211 Size: [320 x 240] Time per frame: 1024.39 ms
Frame counter: 212 Size: [320 x 240] Time per frame: 998.277 ms
Frame counter: 213 Size: [320 x 240] Time per frame: 998.232 ms
Frame counter: 214 Size: [320 x 240] Time per frame: 977.412 ms
Frame counter: 215 Size: [320 x 240] Time per frame: 1004.21 ms
Frame counter: 216 Size: [320 x 240] Time per frame: 1012.52 ms
Frame counter: 217 Size: [320 x 240] Time per frame: 994.304 ms
Frame counter: 218 Size: [320 x 240] Time per frame: 984.736 ms
Frame counter: 219 Size: [320 x 240] Time per frame: 988.949 ms
Frame counter: 220 Size: [320 x 240] Time per frame: 1006.98 ms
Frame counter: 221 Size: [320 x 240] Time per frame: 1133.93 ms
Frame counter: 222 Size: [320 x 240] Time per frame: 1004.83 ms
```

Source: The Author

6.4.1 Performance comparison

During the execution of the identical software on the host machine and subsequent comparison of the two distinct embedded applications, the ensuing results were acquired concerning the execution of the linear mean filter.

Table 6.5: Linear mean filter average time per frame

Design	Average time per frame (ms)
Single core	3138.14
MPSoC 2x2	995.86

Source: The Author

The presented results illustrate that the MPSoC processing exhibits a speedup factor of 3.15 times when compared to the single-core version. Notably, while the MPSoC design encompasses 4 cores, only 3 cores are actively engaged in the mean filter processing. The master tile (0,0) specifically assumes the responsibility of task dispatching and data flow management within the MPSoC design. Further enhancements could be implemented at the FreeRTOS embedded firmware to speed up the processing on both designs, single and multi-core, however, as author's decision such improvement is not required to demonstrate the comparison purpose between multi-task vs single-task.

6.5 Conclusion

In Chapter 6, the focus was on presenting the outcomes of executing the histogram, MAC, and mean filter applications. This involved a detailed examination of two distinct designs: SoC and MPSoC. The presentation included a thorough description and demonstration of each PPA metric, followed by a comprehensive comparison between the two designs. Additionally, an in-depth performance analysis was carried out to justify the final numerical values obtained for each metric.

It is important to note that the results presented are approximate evaluations of the framework. For a more precise measurement, implementation with the appropriate setup and specific use case is necessary. Despite this, the results presented are sufficient to validate the initial hypothesis outlined in the first Chapter, which involves starting from an architecture specification and progressing to the emulation stage of different systems (SoC and MPSoCs).

7 CONCLUSIONS

The preliminary investigation of the design space confers substantial advantages in the conversion of architectural abstraction diagrams into tangible digital circuitry. A platform enabling designers to effortlessly generate diverse hardware configurations, while simultaneously affording sufficient flexibility for incorporating various adjustments in the final code, presents a promising solution to address the aforementioned gap.

The presented work aims to establish a connection between two domains, facilitating collaborative efforts between architectural and design teams to expeditiously assess critical trade-offs within the Very Large Scale Integration (VLSI) domain, specifically Power, Performance, and Area (PPA). By the diverse array of Intellectual Properties (IPs) designed and evaluated throughout this study, the final framework offers a substantial toolset for generating and evaluating a wide spectrum of System-on-Chips (SoCs) and Multi-Processor Systems-on-Chip (MPSoCs). Furthermore, owing to its modular nature, the framework's extensibility extends beyond the scope of the current presentation, permitting the inclusion of additional cores or IPs based on the specific preferences and requirements of different users, thereby enhancing its overall potential.

The progress of VLSI is contingent upon a collaborative effort between high-performance application designers and intellectual property (IP) developers to enrich a repository of hardware functions (accelerators, data structures, etc.). The examples presented serve as a proof of concept and can be further optimized with more efficient IPs, particularly through a more nuanced exploration of parallelism. Additional examples featuring diverse characteristics can unveil solutions for application mapping onto the Multi-Processor System-on-Chip (MPSoC), thereby optimizing performance for designers.

Regarding the methodology, this study has comprehensively presented a step-by-step approach, commencing from architectural specification and culminating in hardware testing, to elucidate the seamless integration of the framework. Furthermore, concerning the results, the investigation entailed the specification and construction of two distinct systems employing the framework, each accommodating diverse software applications. A comparative analysis of these systems was conducted, delving into their respective capabilities and limitations, with particular emphasis on the identification of bottlenecks within the Multi-Processor System-on-Chip (MPSoC). Ultimately, the findings demonstrated that the framework possesses the capability to generate the model (for simulation) and RTL (Register Transfer Level) files that are synthesizable with various configurations, while also facilitating the execution of FreeRTOS applications.

In addition to design generation and software development, the framework was implemented in the Python language, adhering to a template pattern that facilitates seamless integration of custom designs and extends the framework's life cycle. This concept exhibits scalability, as multiple YAML files and Python classes can be effectively combined to create diverse hierarchical instances of designs, thereby enhancing its adaptability and versatility.

Finally, the IPSoCgen framework, showcased throughout Chapter 6, has demonstrated its capability to quickly generate designs from architectural specifications and smoothly advance to the emulation stage, where multiple systems were created and tested.

7.1 Future Works

Throughout its development, the project has revealed numerous promising avenues for future research, encompassing both ongoing endeavors and potential opportunities.

- The enhancement of processor capabilities entails enabling cache configuration through the different master ports. This extension is expected to yield improvements in core metrics, particularly in terms of performance, while addressing systems with less stringent power or area requirements.
- The augmentation of open-source supported protocols can be achieved by incorporating conversion bridges like Wishbone or Tile-link.
- The development of a tool/script to import open-source IPs, which generates design templates and Python classes, would streamline the integration process.
- The expansion of software APIs should include support for state-of-the-art libraries in neural networks and computer vision.
- A proposed initiative involves constructing an open-source IP accelerator library that caters to various applications, such as cryptography, image/video processing, audio processing, and others.
- A new wrapper design for the NoX processor with tightly coupled memories following industry common pattern designs.
- An enhanced NoC version that converts its network interface from passive to configurable active version, this simplifies data handling within the tiles and aligns with

the industry-standard solutions.

- Improve the framework to generate documentation of the SoC / MPSoC once the design is generated. Such documentation includes a memory map in a web page of the system with the representation of the registers and their meaning.

REFERENCES

ABDELHAMID, M. R. et al. Batteryless, wireless, and secure soc for implantable strain sensing. **IEEE Open Journal of the Solid-State Circuits Society**, v. 3, p. 41–51, 2023.

ADELT, P. et al. A scalable platform for qemu based fault effect analysis for risc-v hardware architectures. In: **MBMV 2020 - Methods and Description Languages for Modelling and Verification of Circuits and Systems; GMM/ITG/GI-Workshop**. [S.l.: s.n.], 2020. p. 1–8.

AGUIAR, A. et al. Adding virtualization support in mips 4kc-based mpsocs. In: **Fifteenth International Symposium on Quality Electronic Design**. [S.l.: s.n.], 2014. p. 84–90.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1967. (AFIPS '67 (Spring)), p. 483–485. ISBN 9781450378956. Available from Internet: <https://doi.org/10.1145/1465482.1465560>.

AMID, A. et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. **IEEE Micro**, v. 40, n. 4, p. 10–21, 2020.

Andrew Waterman; Krste Asanović; John Hauser. **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203**. [S.l.], 2021. Available from Internet: <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>.

ARM, A. Axi protocol specification (rev 5.0 - arm ihi 0022). 2023. Accessed on May 15th, 2023. Available from Internet: <https://developer.arm.com/documentation/ihi0022/latest/>.

ASANOVIĆ, K.; PATTERSON, D. A. Instruction sets should be free: The case for risc-v. **EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146**, 2014.

ASANOVIć, K. et al. **The Rocket Chip Generator**. [S.l.], 2016. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.

AZAD, S. P. et al. Caesar-mpsoc: Dynamic and efficient mpsoc security zones. In: **2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2019. p. 477–482.

BACHRACH, J. et al. Chisel: Constructing hardware in a scala embedded language. In: **DAC Design Automation Conference 2012**. [S.l.: s.n.], 2012. p. 1212–1221.

BARGHOLZ, M.; DIETRICH, C.; LOHMANN, D. Psic: Priority-strict multi-core irq processing. In: **2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)**. [S.l.: s.n.], 2022. p. 1–9.

BASTL, J. et al. A design flow and eda-tool for an automated implementation of asic configuration interfaces. In: **2022 18th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)**. [S.l.: s.n.], 2022. p. 1–4.

BRIGGS, M.; ZARKESH-HA, P. Evaluating mobile socs as an energy efficient dsp platform. In: **2014 27th IEEE International System-on-Chip Conference (SOCC)**. [S.l.: s.n.], 2014. p. 293–298.

CARARA, E. A. et al. Hemps - a framework for noc-based mpsoc generation. In: **2009 IEEE International Symposium on Circuits and Systems**. [S.l.: s.n.], 2009. p. 1345–1348.

ELMOHR, M. A. et al. Rvnoc: A framework for generating risc-v noc-based mpsoc. In: **2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)**. [S.l.: s.n.], 2018. p. 617–621.

ESMAEILZADEH, H. et al. Physically accurate learning-based performance prediction of hardware-accelerated ml algorithms. In: **2022 ACM/IEEE 4th Workshop on Machine Learning for CAD (MLCAD)**. [S.l.: s.n.], 2022. p. 119–126.

FORENCICH, A. **Verilog AXI components**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/alexforencich/verilog-axi>.

FORENCICH, A. **Verilog Ethernet components**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/alexforencich/verilog-ethernet>.

GALA, M. K. N. **RISC-V Architecture Test SIG**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/riscv-non-isa/riscv-arch-test>.

GALA, M. K. N. **RISCOF Framework**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://riscof.readthedocs.io/en/latest/overview.html>.

GISSELQUIST, D. **Another Wishbone (or even AXI-lite) Controlled UART**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/ZipCPU/wbuart32>.

HASLER, M. et al. A random linear network coding platform mpsoc designed in 22nm fdsoi. In: **2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2022. p. 217–222.

HASSAN, E. et al. Asic implementation of associative memory and hamming distance for hdc application. In: **2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)**. [S.l.: s.n.], 2021. p. 1–5.

HOSNY, S. A unified uvm methodology for mpsoc hardware/software functional verification. In: **2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST)**. [S.l.: s.n.], 2022. p. 1–5.

IEEE Standard for Ethernet. **IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)**, p. 1–5600, 2018.

IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. **IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)**, p. 1–1315, 2018.

IZRAELEVITZ, A. et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In: **2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2017. p. 209–216.

KAMALELDIN, A. et al. Modular memory system for risc-v based mpsocs on xilinx fpgas. In: **2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)**. [S.l.: s.n.], 2019. p. 68–73.

KERMARREC, F. et al. Litex: An open-source soc builder and library based on migen python dsl. In: **OSDA 2019, Co-located with DATE 2019 Design Automation and Test in Europe**. [S.l.: s.n.], 2019.

KREMER, M. D. et al. Resource-constrained encryption: Extending ibex with a qarma hardware accelerator. In: **2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)**. [S.l.: s.n.], 2023. p. 147–155.

KURIMOTO, Y. et al. A hardware/software cosimulator for network-on-chip. In: **2013 International SoC Design Conference (ISOCC)**. [S.l.: s.n.], 2013. p. 172–175.

KURTH, A. et al. **HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA**. arXiv, 2017. Available from Internet: <https://arxiv.org/abs/1712.06497>.

LEE, Y.-T. Low power soc in deep-submicron era. In: **IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 421–.

LI, C. et al. Heterogeneous routing: A methodology for ppa tradeoff in network-on-chips. In: **2018 IEEE 3rd International Conference on Integrated Circuits and Microsystems (ICICM)**. [S.l.: s.n.], 2018. p. 357–361.

LIUBAVIN, K. D. et al. Design of a fully-autonomous low-power axi4 firewall for pci-express nvme ssd. In: **2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)**. [S.l.: s.n.], 2022. p. 166–169.

MA, G.; HE, H. Design and implementation of an advanced dma controller on amba-based soc. In: **2009 IEEE 8th International Conference on ASIC**. [S.l.: s.n.], 2009. p. 419–422.

MICHELI, G. D. et al. **Networks on Chips: Technology and Tools**. Elsevier Science, 2006. (ISSN). ISBN 9780080473567. Available from Internet: <https://books.google.ie/books?id=IHHTmSBcoGIC>.

MOSANU, S. et al. Pimulator: a fast and flexible processing-in-memory emulation platform. In: **2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2022. p. 1473–1478.

NGUYEN-HOANG, D.-T. et al. Implementation of a 32-bit risc-v processor with cryptography accelerators on fpga and asic. In: **2022 IEEE Ninth International Conference on Communications and Electronics (ICCE)**. [S.l.: s.n.], 2022. p. 219–224.

NOAMI, A. et al. High speed data transactions for memory controller based on axi4 interface protocol soc. In: **2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)**. [S.l.: s.n.], 2021. p. 1–7.

PAPON, C. **SpinalHDL: Scala based HDL**. 2023. Accessed on Jul 16th, 2023. Available from Internet: <https://github.com/SpinalHDL/SpinalHDL>.

PAPON, C. **VexRiscv: RISC-V processor**. 2023. Accessed on Mar 10th, 2024. Available from Internet: <https://github.com/SpinalHDL/VexRiscv>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128122757.

SETETEMELA, K. O. et al. Python-based fpga implementation of aes using migen for internet of things security. In: **2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)**. [S.l.: s.n.], 2019. p. 194–198.

SIFIVE, H. C. Diplomatic design patterns : A tilelink case study. In: . [s.n.], 2017. Available from Internet: <https://api.semanticscholar.org/CorpusID:44937251>.

SILVA, A. I. da. **Configurable AXI DMA design**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/axi_dma>.

SILVA, A. I. da. **Ethernet AXI**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/ethernet_axi>.

SILVA, A. I. da. **IPSoCGen template**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/ipsocgen_template>.

SILVA, A. I. da. **NoX FreeRTOS example port**. 2023. Accessed on May 8th, 2023. Available from Internet: <https://github.com/aignacio/nox_freertos>.

SILVA, A. I. da. **NoX RISC-V CPU RV32I**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/nox>.

SILVA, A. I. da. **RaveNoC - Configurable Network-on-Chip design**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/ravenoc>.

SILVA, A. I. da. **SoC Components**. 2023. Accessed on May 7th, 2023. Available from Internet: <https://github.com/aignacio/soc_components>.

TARAATE, V. **ASIC Design and Synthesis: RTL Design Using Verilog**. Springer Nature Singapore, 2021. ISBN 9789813346420. Available from Internet: <https://books.google.ie/books?id=qK0SEAAAQBAJ>.

VALLEPALLI, S. S.; RAJENDRAN, M. M. Image de-noising using mean pixel algorithms corrupted with photocopier noise. In: **2012 19th International Conference on Systems, Signals and Image Processing (IWSSIP)**. [S.l.: s.n.], 2012. p. 530–535.

WANG, B.; LU, Z. Supporting qos in axi4 based communication architecture. In: **2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2020. p. 548–553.

WATERMAN, A. et al. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0**. [S.l.], 2014. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html>.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor system-on-chip (mpsoc) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 10, p. 1701–1713, 2008.

XU, M.; ZHANG, L.; WANG, S. Position paper: Renovating edge servers with arm socs. In: **2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)**. [S.l.: s.n.], 2022. p. 216–223.

XU, S.; POLLITT-SMITH, H. A tlm platform for system-on-chip simulation and verification. In: **2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT).** [S.l.: s.n.], 2005. p. 220–221.

ZEFERINO, C.; SUSIN, A. Socin: a parametric and scalable network-on-chip. In: **16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.** [S.l.: s.n.], 2003. p. 169–174.

ZHANG, L. The porting and optimization of risc-v uefi boot. In: **2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)**. [S.l.: s.n.], 2022. p. 840–843.

ZHANG, W. et al. A network on chip architecture and performance evaluation. In: **2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing**. [S.l.: s.n.], 2010. v. 1, p. 370–373.

APPENDIX A — RESUMO EXPANDIDO

Plataforma IPSoCGen - Framework para geração MP/SoC

Título em inglês: IPSoCGen platform - Framework for MP/SoC generation

Palavas-chave: Geração de design, Redes-em-Chip, Processamento paralelo, VLSI

O número de dispositivos que integram Sistemas-em-Chip (SoCs) tem aumentado continuamente. Como mencionado por Lee (2003), o avanço da tecnologia possibilita a integração de múltiplos elementos de processamento, blocos de memória e módulos analógicos, convergindo para a era dos SoCs. A versatilidade desses sistemas é evidente em sua capacidade de serem aplicados em diversos domínios, incluindo processamento móvel (BRIGGS; ZARKESH-HA, 2014), servidores de borda (XU; ZHANG; WANG, 2022) ou até mesmo implantes no corpo humano (ABDELHAMID et al., 2023). Assim, o número de requisitos cresce proporcionalmente com as diferentes aplicações que requerem Processadores de Propósito Geral (GPP), aceleradores de hardware, sensores diversos incluindo circuitos de interface, etc.

O fluxo de projeto destes SoCs, além de complexo, envolve diferentes etapas (TARAATE, 2021). Módulos complexos podem ser reutilizados em aplicações futuras se projetados dentro de um padrão de reúso, criando uma biblioteca de funções (também chamadas "Intelectual Property" ou IP) a serem gerenciadas na plataforma de desenvolvimento. Durante todo o processo, diferentes pontos de observação são adicionados para garantir que projeto esteja alinhado às métricas inicialmente propostas de potência, desempenho e área. Como tal fluxo é complexo, uma plataforma que possibilite o rápido e preciso desenvolvimento de sistemas serve como uma solução para minimizar o número de iterações entre times de arquitetura e projeto, reduzindo o Time-to-Market (TTM). Além disso, esta plataforma, com ferramentas de avaliação com precisão de ciclo de *clock*, conseguiria também melhorar a eficiência dos sistemas desenvolvidos.

O objetivo principal desta dissertação é o desenvolvimento de uma plataforma capaz de abordar eficazmente o desafio mencionado anteriormente, que consiste em facilitar a prototipagem de projetos de forma rápida e flexível, partindo de uma especificação de arquitetura e avançando para a etapa de emulação. Além da plataforma, é apresentado um fluxo de projeto para permitir a utilizá-la, incluindo as etapas necessárias, assim como o desenvolvimento de software.

A plataforma desenvolvida é denominada IPSoCGen e é composta por um conjunto de IPs comuns em SoCs, além de uma metodologia para a geração do sistema onde se detalham a forma de interconectar os módulos e os parâmetros aplicáveis de acordo com os requisitos do projeto. Inicialmente, essa plataforma é comparada ao estado da arte em termos de geração de sistemas, e suas vantagens e desvantagens são analisadas em relação ao que é apresentado na literatura. Os componentes IPs dessa plataforma são detalhados neste trabalho, destacando suas características e explicando-as ao longo do Capítulo 3. Para cada um dos diferentes IPs, são apresentadas suas opções de configuração e como se integram à proposta da plataforma final. No Capítulo 4, a metodologia do fluxo de projeto de SoCs é apresentada, bem como a descrição de como a plataforma opera. A avaliação dessa plataforma ocorre nos Capítulos 5 e 6, onde dois sistemas distintos foram gerados utilizando o IPSoCGen, e diferentes aplicações foram construídas para avaliar os aspectos de potência, desempenho e área.

Este trabalho culmina com a apresentação das métricas obtidas a partir dos dois sistemas prototipados utilizando o IPSoCGen. Conclui-se que a plataforma desenvolvida é capaz de gerar sistemas com um nível específico de flexibilidade e agilidade por meio do método baseado em arquivos de configuração. Além disso, o IPSoCGen viabiliza a geração de modelos executáveis em nível de circuito RTL (Register-Transfer Level), permitindo a configuração de circuitos de prototipação e a geração de leiaute de ASICs. Nos casos estudados neste trabalho, para o desenvolvimento e a execução das aplicações foi feito o porte do FreeRTOS para o kit de prototipação. Por fim, ressalta-se que o progresso da VLSI (Very Large Scale Integration) depende de um esforço colaborativo entre projetistas de aplicativos de alto desempenho e desenvolvedores de propriedade intelectual (IP) para enriquecer o repositório de funções de hardware, como aceleradores de hardware para aplicações que exigem alto desempenho como multimídia, IA, mineração de dados e criptografia. Este trabalho apresentou um conjunto reduzido de IPs suficiente para as demonstrações e deverá crescer à medida que novos projetos aportem contribuições.