FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automated Integration of High-Level Simulators for RISC-V SoC Co-Simulation

Pedro Miguel Moreira Ramalho

Master in Informatics and Computer Engineering

Supervisor: Dr. João Carlos Viegas Martins Bispo

Second Supervisor: Dr. Nuno Miguel Cardanha Paulino

July 28, 2025

© Pedro Miguel Moreira Ramalho, 2025

Automated Integration of High-Level Simulators for RISC-V SoC Co-Simulation

Pedro Miguel Moreira Ramalho

Master in Informatics and Computer Engineering

July 28, 2025

Resumo

A desacelaração da Lei de Moore, que previa que o número de transístores num *chip* duplicaria aproximadamente a cada dois anos, juntamente com a quebra do escalonamento de Dennard, que anteriormente garantia uma densidade de potência constante à medida que os transístores reduziam de tamanho, tem conduzido a retornos decrescentes no desempenho das arquiteturas tradicionais de von Neumann. Para dar resposta a esta tendência, a comunidade científica tem vindo a explorar paradigmas alternativos que oferaçam algum tipo de reconfigurabilidade ao nível do *hardware*.

Entre estas alternativas encontram-se os *Field-Programmable Gate Arrays* (FPGAs), que permitem uma reconfiguração ao nível do *bit*, e os *Coarse-Grained Reconfigurable Arrays* (CGRAs), que sacrificam granularidade em troca de tempos de compilação mais rápidos e maior facilidade de programação. A integração destes dispositivos reconfiguráveis com processadores genéricos tornou-se uma área de investigação de grande interesse, especialmente no contexto do desenho de plataformas *System-on-Chip* (SoC).

Contudo, a incorporação de nova funcionalidade, como periféricos ou aceleradores de *hardware*, numa plataforma SoC continua a ser uma tarefa demorada e propensa a erros. A complexidade do processo vai frequentemente para além do desenho do bloco de *hardware* em si, e acaba por se estender à sua integração no ecossistema de *hardware* e *software*.

A co-simulação surgiu como uma ferramenta valiosa para enfrentar estes desafios, permitindo a simulação conjunta de components de *hardware* e *software*. Esta técnica oferece uma visão crítica sobre o comportamento e interações do sistema antes da fase mais dispendiosa de implementação em *hardware*.

Esta dissertação apresenta a implementação de uma *framework* de co-simulação baseada em *software* que automatiza a integração de simuladores de alto nível numa plataforma SoC. Ao invés de exigir que os programadores alterem diretamente descrições de hardware ou escrevam código de comunicação de baixo nível, a framework recorre a um simples ficheiro de configuração JSON para especificar de forma declarativa os parâmetros de simulação e a interface dos periféricos.

A *framework* foi avaliada experimentalmente através da plataforma de hardware *X-HEEP* e a biblioteca de comunicação *ZeroMQ*. Foi replicado um acelerador da suite oficial do *X-HEEP* e observou-se um comportamento de simulação, em número de ciclos, consistente para todos os tamanhos de *buffer*, com uma única exceção, e registou-se um *overhead* máximo de 11% no tempo de simulação de relógio de parede, face à implementação original. Apesar deste *overhead*, a *framework* oferece um ambiente flexível e acessível ao programador, e reduz significativamente as barreiras à prototipagem rápida de simuladores.

Abstract

The slowdown of Moore's Law, which predicted that the number of transistors on a chip would double roughly every two years, combined with the breakdown of Dennard scaling, which once ensured constant power density as transistors shrank, has led to diminishing returns in the performance of traditional von Neumann architectures. In response, researchers have turned to alternative computing paradigms that offer some form of hardware reconfigurability.

Among these alternatives are Field-Programmable Gate Arrays (FPGAs), which support fine-grained, bit-level reconfiguration, and Coarse-Grained Reconfigurable Arrays (CGRAs), which sacrifice granularity in favor of faster compilation times and greater ease of programmability. The integration of such reconfigurable architectures with general-purpose processors has become a prominent area of research, especially in the context of System-on-Chip (SoC) design.

However, incorporating new functionality, such as custom peripherals or hardware accelerators, into a complex SoC platform remains a time-consuming and error-prone task. The complexity often extends beyond the design of the hardware block itself, but also in integrating it into the broader hardware and software environment.

Co-simulation has emerged as a powerful tool to address these challenges. By enabling hardware and software components to be simulated together, it provides critical insight into system behavior and interactions prior to committing to hardware implementation.

This thesis presents the design and implementation of a software-based co-simulation framework that automates the integration of high-level simulators into a SoC platform. Instead of requiring developers to manually alter hardware descriptions or write low-level communication code, the framework uses a simple JSON configuration file to declaratively specify simulation parameters and peripheral interfaces.

The framework was evaluated experimentally using X-HEEP as the target hardware platform and ZeroMQ as the communication layer. When replicating an accelerator from X-HEEP's official suite, the framework exhibited consistent simulation behavior, in clock cycles, across all buffer sizes, with a single exception, and incurred a maximum wall-clock simulation overhead of 11% compared to the baseline. However, this overhead is largely offset by the ability to develop and iterate entirely in software, without requiring repeated recompilation of SystemVerilog components. As a result, the framework provides a flexible and developer-friendly environment that significantly reduces the effort required for rapid simulator prototyping.

Acknowledgments

I would like to begin by expressing my deepest gratitude to my supervisor, Dr. João Carlos Viegas Martins Bispo, and my co-supervisor, Dr. Nuno Miguel Cardanha Paulino. Your guidance helped shape this work and gave me confidence during the moments I needed it most. Thank you for your patience and support.

To the friends who've walked this path with me over the last five years: thank you for the laughs, the long nights spent working on projects, and the shared victories and struggles. I'm eternally grateful for the friendships we've built and the memories we carry now together. You made this journey possible to endure.

To my family, thank you for keeping me together. To my father, Carlos, thank you for your strength and silent sacrifices. To my mother, Sandra, thank you for your love and unwavering belief in me. To my godmother, Ana, thank you for nurturing my curiosity and love for learning. To my grandmother, Arminda, thank you for your care, your cooking, and your soft prayers that have always been a gentle source of comfort.

You have all given so much of yourselves so that I could be here, writing these words. I hope I've made you proud, in some way.

And finally, to my younger brother, Henrique, my closest friend and companion - a special thank you for always being there for me with open arms and an open heart. Your presence means more to me than words can say. This dissertation is dedicated to you.

Pedro Ramalho

“However ruined this world has become, however mired in torment and despair...
Life endures. Births continue. There is beauty in that, is there not?”

Melina, Elden Ring

Contents

1 Introduction 1
1.1 Context and Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Hypothesis . . . 3
1.4 Document Structure . . . 3

2 State of the Art 4
2.1 Background . . . 4
2.1.1 Reconfigurable Computing Architectures . . . 4
2.1.2 RISC-V Instruction Set Architecture . . . 6
2.1.3 Hardware/Software Co-Design . . . 7
2.2 Related Work . . . 9
2.2.1 Simulation Frameworks for Accelerators . . . 9
2.2.2 Hardware/Software Co-Design and Integration . . . 14
2.2.3 Optimization of Simulation Techniques . . . 17
2.2.4 Hybrid and Configurable Architectures . . . 19
2.3 Summary . . . 21
2.3.1 Overview of State-of-the-Art Frameworks and Methodologies . . . 21
2.3.2 Trends and Open Challenges . . . 22

3 Co-Simulation Framework Design 23
3.1 Architectural Overview . . . 23
3.2 Peripheral Architecture . . . 24
3.2.1 Register Interface . . . 25
3.2.2 Memory Access Interface . . . 27
3.3 Communication Subsystem . . . 28
3.3.1 Abstraction Layers . . . 30
3.4 Code Generation Pipeline . . . 31

4 Multi-Language Co-Simulation API 32
4.1 API Functionality . . . 32
4.2 Core Components . . . 33
4.2.1 Simulation Data Abstraction . . . 33
4.2.2 Simulator Execution Model . . . 34
4.2.3 Communication Subsystem . . . 36
4.2.4 Memory Transactions . . . 37
4.2.5 Issuing Memory Reads and Writes . . . 40
4.3 Simulator Communication Loop . . . 43

5 Automated Simulator Integration 44
5.1 Expanding a Hardware Platform . . . 44
5.2 Code Generator Implementation . . . 45
5.2.1 Architecture Overview . . . 45
5.2.2 Configuration File Structure . . . 46
5.2.3 Template Infrastructure . . . 48

6 Framework Evaluation 53
6.1 Functionality Assessment . . . 53
6.2 Experimental Setup . . . 53
6.2.1 Hardware and Software Environment . . . 53
6.2.2 Case Study: X-HEEP's Simple Accelerator . . . 54
6.3 Experiment 1: Replicating the Simple Accelerator . . . 55
6.3.1 Baseline: Pure SystemVerilog Accelerator . . . 55
6.3.2 Framework-enabled: Co-Simulated Accelerator . . . 56
6.3.3 Host Program . . . 60
6.3.4 Performance Benchmarks . . . 60
6.4 Experiment 2: Scaling for Multiple Accelerators . . . 60
6.5 Analysis of Results . . . 61
6.5.1 Experiment 1: Results Discussion . . . 61
6.5.2 Experiment 2: Results Discussion . . . 62
6.6 Overall Evaluation and Takeaways . . . 64

7 Conclusion and Future Work 67
7.1 Summary of Research . . . 67
7.2 Future Work . . . 68
7.2.1 Limitations . . . 68
7.2.2 Performance Enhancements . . . 68

References 70

List of Figures

2.1 An early Field-Programmable Gate Array (FPGA) architecture with programmable logic compared to a modern heterogeneous Field-Programmable Gate Array (FPGA) featuring a memory controller and other hard blocks [3]. 5
2.2 Illustration of a simple Coarse-Grained Reconfigurable Array (CGRA), showing the mesh topology (a), the internal architecture of the Reconfigurable Cell (RC) (b), and an example of the configuration register (c) [27]. 6
2.3 Taxonomy and functionalities of Instruction Set Architecture (ISA) extensions of Reduced Instruction Set Computer - V (RISC-V) [10]. 8
2.4 NNSim's proposed work flow of the Convolutional Neural Network (CNN) accelerator simulator [19]. 10
2.5 GVSoC's components [5]. 10
2.6 GVSoC's communication mechanism [5]. 11
2.7 SECDA's methodology (from [14]). 11
2.8 FLECSim-SoC's SystemC simulator setup [17]. 12
2.9 SimPyler's illustration of a general Machine Learning (ML) system [4]. 13
2.10 An overview of eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP)'s architecture [22]. 14
2.11 TaPaSCo's interface components [15]. 15
2.12 TaPaSCo's System-on-Chip (SoC) template [15]. 15
2.13 ERAS' framework place for architectural simulations [24]. 16
2.14 RoSÉ's components [25]. 17
2.15 The proposed emulator [18]. 18
2.16 Execution flow illustrating the transition from software emulation to Register Transfer Level (RTL) model simulation during execution (from [18]). 19
2.17 4×4 ADRES Coarse-Grained Reconfigurable Array (CGRA) architecture [20]. . 20
2.18 Block diagram for the resulting hybrid system [20]. 20
2.19 μRV32's Reduced Instruction Set Computer - V (RISC-V) core [1]. 21
2.20 μRV32's System-on-Chip (SoC) diagram [1]. 21

3.1 High-level block diagram of the framework's generic architecture. 24
3.2 High-level block diagram of the framework when considering the use of eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) as the hardware platform System-on-Chip (SoC). 25

4.1 High-level block diagram of the `step` method Finite State-Machine (FSM). . . . 34
4.2 High-level block diagram of a single memory read operation, depicted in a Finite State-Machine (FSM). 37

4.3 High-level block diagram of a single memory write operation, depicted in a Finite State-Machine (FSM). 38
4.4 High-level block diagram of a burst memory read operation, depicted in a Finite State-Machine (FSM). 39
4.5 High-level block diagram of a burst memory write operation, depicted in a Finite State-Machine (FSM). 39

5.1 High-level block diagram of the code generator's architecture. 45

6.1 Simulation times, measured in clock cycles, for different data sizes and implementations, when dealing with a single accelerator instance. 61
6.2 Wall-clock simulation times, measured in seconds, for different data sizes and implementations, when dealing with a single accelerator instance. 62
6.3 Ratio between wall-clock simulation times for different data sizes and implementations, when dealing with a single accelerator instance. 63
6.4 Simulation times, measured in clock cycles, for different numbers of accelerator instances. 64
6.5 Wall-clock simulation times, measured in seconds, for different numbers of accelerator instances. 65
6.6 Ratio between wall-clock simulation times for different numbers of accelerator instances. 65

List of Tables

5.1 Summary of generated eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) artifacts and their output locations. 52

6.1 Experimental setup hardware specifications. 54
6.2 Experimental setup software specifications. 54
6.3 Experiment's 1 performance summary for varying data sizes and implementations, when dealing with a single accelerator instance. 63
6.4 Experiment's 2 performance summary when using buffer sizes of 256 words for different numbers of accelerator instances. 64

Abbreviations and Acronyms

AI Artificial Intelligence

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

CDFG Control Data Flow Graph

CGRA Coarse-Grained Reconfigurable Array

CLINT Core-Local Interrupt Controller

CNN Convolutional Neural Network

CPU Central Processing Unit

DFG Dataflow Graph

DMA Direct Memory Access

DMI Direct Memory Interface

DNN Deep Neural Network

DPI Direct Programming Interface

DRAM Dynamic Random Access Memory

DSE Design Space Exploration

DSP Digital Signal Processing

ESL Electronic System Level

FPGA Field-Programmable Gate Array

FSM Finite State-Machine

GPU Graphics Processing Unit

HDL Hardware Description Language

HLL High-Level Language

HLS High-Level Synthesis

HPC High-Performance Computing

I/O Input/Output

IC Integrated Circuit

ISA Instruction Set Architecture

IoT Internet of Things

IP Intellectual Property

JSON JavaScript Object Notation

MAC Multiply-and-Accumulate

MCU Microcontroller Unit

ML Machine Learning

MMIO Memory-Mapped Input/Output

MQTT Message Queuing Telemetry Transport

OBI Open Bus Interface

OS Operating System

PC Processing Cluster

PE Processing Element

PULP Parallel Ultra-Low Power

RC Reconfigurable Cell

REQ-REP Request-Reply

RISC-V Reduced Instruction Set Computer - V

RTL Register Transfer Level

SRAM Static Random-Access Memory

SoC System-on-Chip

SV SystemVerilog

TCP Transmission Control Protocol

TLM Transaction-Level Modeling

UART Universal Asynchronous Receiver/Transmitter

VP Virtual Prototype

X-HEEP eXtendable Heterogeneous Energy-Efficient Platform

YAML Yet Another Markup Language

Chapter 1

Introduction

1.1 Context and Motivation

The increasing demand for computational power has driven significant changes in how modern computing systems are designed. Traditionally, general-purpose processors have delivered steady performance improvements, guided by Moore's Law and Dennard scaling [33]. However, as these trends have approached their physical limits, performance gains in conventional von Neumann architectures have slowed dramatically [27]. This has led researchers and engineers to explore alternative computing paradigms, especially reconfigurable and heterogeneous architectures, to meet the demands of domains like embedded systems, Internet of Things (IoT), and High-Performance Computing (HPC).

Among reconfigurable platforms, FPGAs are widely recognized for their flexibility. They enable fine-grained bit-level hardware customization and are supported by established design flows using Hardware Description Languages (HDLs) or High-Level Synthesis (HLS). However, these flows are often general-purpose, not optimized for specific application needs. Moreover, FPGAs suffer from long compilation times, higher power consumption, and lower clock frequencies when compared to Application-Specific Integrated Circuits (ASICs).

For many use cases, a more effective approach is to integrate custom hardware accelerators tailored to specific tasks. These accelerators, whether implemented as FPGA overlays or synthesized as ASIC components, can provide significant performance and energy efficiency improvements without requiring a complete system redesign.

The integration of custom accelerators alongside general-purpose cores, such as open-source RISC-V processors, has become a central theme in modern SoC design [30]. RISC-V's extensible and open architecture makes it particularly well-suited for this kind of heterogeneous integration. However, incorporating new components into a system introduces challenges, particularly in managing inter-component communication, ensuring correct timing synchronization, and validating functional correctness across subsystems.

Co-simulation has emerged as a valuable tool for addressing these challenges. It enables the simultaneous simulation of hardware and software components, allowing developers to analyze

interactions and verify system behavior before committing to the more costly phase of hardware development. Co-simulation can refer to a variety of techniques, such as running hardware and software components simultaneously or simulating hardware at different levels of abstraction [2]. In this work, we focus on software-based co-simulation where high-level software models interact with cycle-accurate RTL hardware models, simulated via tools like Verilator [29].

As hardware systems grow more complex, the need for agile development workflows and rapid prototyping becomes increasingly important. Fast iteration is particularly critical for the design of custom peripherals and hardware accelerators, where integration and validation within a realistic system context can quickly become a bottleneck. Although this work does not perform Design Space Exploration (DSE), it lays essential groundwork for it by proposing a co-simulation framework that enables the integration and testing of hardware and software components in a unified environment.

1.2 Problem Statement

The primary goal of this work is twofold: first, to enable system-level simulation of select hardware components as functional software processes rather than relying solely on traditional RTL implementations, and second, to provide a declarative methodology for automating their integration into established SoC platforms. This paradigm shift aims to accelerate both simulation and development cycles, making it easier to explore different design configurations. This approach is particularly advantageous for DSE, where testing various configurations through High-Level Languages (HLLs) is often more efficient than generating separate RTL for each variation.

However, many of the existing co-simulation frameworks are limited by their focus on unidirectional communication, where the processor dictates operations to the hardware. In these setups, feedback from hardware accelerators back to the processor is often overly simplified, not fully captured, or requires manual intervention and setup. These limitations can lead to bottlenecks during performance validation, making it challenging to ensure the accuracy and reliability of the simulated system in real-world conditions.

Furthermore, the process of integrating new hardware functionality, such as custom peripherals or accelerators, into complex SoC platforms is notoriously challenging and time-consuming. This integration often involves intricate manual steps beyond the scope of the hardware block design itself, which increases the risk of errors and hinders productivity.

To address these challenges, this thesis proposes a co-simulation framework that automates the integration of high-level hardware simulators into a SoC environment. Communication between components is managed via messaging protocols, like ZeroMQ. Beyond addressing communication challenges, the framework must also be adaptable enough to support DSE, enabling the evaluation of multiple hardware configurations.

With this problem in mind, we can formulate the following research questions:

RQ1 Can the developed co-simulation framework enable bidirectional communication between a Verilated RTL design (e.g., C++-based simulator) and a hardware component modeled as a higher-level software process?

RQ2 Is a messaging protocol (e.g., Message Queuing Telemetry Transport (MQTT) or ZeroMQ) effective for managing inter-process communication between heterogeneous components?

RQ3 Can high-level simulators, when integrated through this framework, independently issue memory transactions such as reads and writes to the SoC?

1.3 Hypothesis

Considering the previously identified problem, this work formulates the following hypothesis:

A co-simulation framework that makes use of messaging protocols such as ZeroMQ to manage bidirectional communication between high-level simulators and Verilated RTL components can achieve efficient and flexible integration within complex SoC platforms. By automating the integration process and enabling simulators to independently initiate memory read/write requests to the SoC, the framework will allow for prototyping with shorter feedback loops without sacrificing functional correctness. This approach will create an adaptable co-simulation environment, capable of supporting DSE across various hardware configurations.

1.4 Document Structure

The remainder of this document is organized as follows: Chapter 2 discusses the state-of-the-art technologies and methodologies, focusing on simulation frameworks. Chapter 3 presents the framework's architecture. Chapter 4 and Chapter 5 detail the main components of the framework. Chapter 6 provides a series of experiments designed to evaluate the framework's performance.

Chapter 2

State of the Art

This chapter provides an overview of foundational concepts and related works that underpin the advancements in the fields of reconfigurable architectures and co-design methodologies.

2.1 Background

This section introduces essential concepts and technologies that form the foundation of this work. It covers reconfigurable computing architectures, focusing on Field-Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs). The section also discusses the Reduced Instruction Set Computer - V (RISC-V) Instruction Set Architecture (ISA), its extensions and applications, as well as methodologies for hardware/software co-design, including simulation frameworks and High-Level Synthesis (HLS).

2.1.1 Reconfigurable Computing Architectures

In recent decades, computing architectures have made use of advances in Integrated Circuit (IC) technology to enhance performance. However, as Moore's Law and Dennard scaling are slowing down or even coming to an end [33], this approach has become less effective. Power constraints in ICs have tightened, and energy efficiency gains from newer technologies are diminishing, limiting further improvements in computational power. As a result, architectural design priorities have shifted from maximizing speed and performance to optimizing energy efficiency. In [21], L. Liu *et al.* explain that flexibility has become critical, as hardware incapable of adapting to rapidly evolving software risks obsolescence and high development costs. Today, energy efficiency and flexibility are the primary criteria for modern computing architectures.

The concept of reconfigurable computing dates back to the 1960s when Gerald Estrin proposed a hybrid system combining a general-purpose processor with an array of reconfigurable hardware [13]. The general-purpose processor would manage the reconfigurable array, which could be tailored to perform specific tasks like image processing or pattern recognition. Once a task was completed, the hardware could be reconfigured for a different operation.

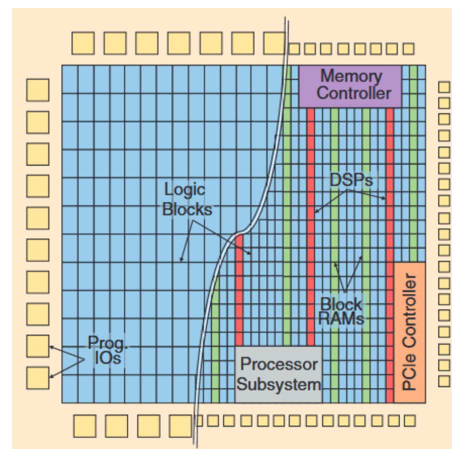


Figure 2.1: An early Field-Programmable Gate Array (FPGA) architecture with programmable logic compared to a modern heterogeneous Field-Programmable Gate Array (FPGA) featuring a memory controller and other hard blocks [3].

FPGAs embody this reconfigurable computing paradigm. They were developed to reduce the cost of simulation and developing Application-Specific Integrated Circuits (ASICs) by enabling post-manufacturing programmability [27]. An FPGA is composed of an array of programmable logic blocks interconnected via configurable routing networks, all controlled by on-chip Static Random-Access Memory (SRAM) cells programmed at runtime. Figure 2.1 shows an illustration of an FPGA architecture. Although the programmability of FPGAs makes them highly versatile, they are not without limitations. Slow configuration and compilation times have hindered their adoption [27]. These challenges have pushed researchers to pursue alternatives, leading to the development of CGRAs.

CGRAs extend the principles of FPGAs but adopt a coarser level of reconfigurability. Instead of fine-grained logic blocks, CGRAs use larger, specialized Processing Elements (PEs) or Reconfigurable Cells (RCs), typically organized in a mesh structure and interconnected via data paths. Each RC includes an Arithmetic Logic Unit (ALU) for integer or floating-point operations, multiplexers, and local SRAM for temporary data storage. This generic architecture of a CGRA can be seen in Figure 2.2.

The authors also explain that CGRAs can bring data in and out of the fabric in several ways: memory-mapping the device to a host processor, using configurable address generators to fetch data from external memory, or enabling the RCs to handle both computation and data fetching.

Although both FPGAs and CGRAs offer flexibility and adaptability, their design and configuration involve inherent trade-offs between performance, energy efficiency, and resource utilization. For instance, as previously stated, the fine-grained reconfigurability of FPGAs allows for highly versatile designs but comes at the cost of longer compilation times and higher overheads. In comparison, CGRAs tend to be more computationally efficient but are also less flexible.

Both FPGAs and CGRAs have found their uses in computing fields like embedded systems, Internet of Things (IoT), and High-Performance Computing (HPC) [27]. The first is of particular

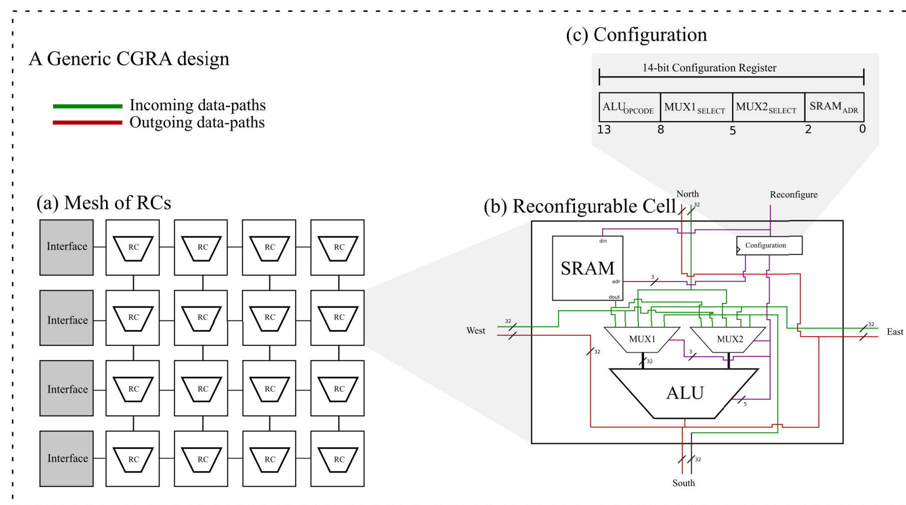


Figure 2.2: Illustration of a simple Coarse-Grained Reconfigurable Array (CGRA), showing the mesh topology (a), the internal architecture of the Reconfigurable Cell (RC) (b), and an example of the configuration register (c) [27].

interest. Embedded devices often need to balance multiple conflicting goals, such as real-time performance, power efficiency, and adaptability for future applications. As many embedded systems are deployed in mass production or battery-operated environments, they face tight energy budgets and constraints on active cooling.

Andy D. Pimentel explains that as the complexity of modern embedded systems grows, and their requirements become more stringent, the need to configure the underlying hardware of these systems optimally arises [26]. To this end, Design Space Exploration (DSE) has become an essential methodology for embedded system design. This process involves exploring numerous parameters like the number and types of processors, interconnection networks, and task mappings. For embedded systems, DSE guides architectural decisions as early design choices often determine the success or failure of the final product. However, achieving the most optimal configuration may be infeasible, since the design space can become extremely vast. DSE is, by nature, an extremely computationally intensive and complex problem. For instance, mapping application tasks to processing resources while optimizing for both performance and power consumption is an NP-hard problem [26].

2.1.2 RISC-V Instruction Set Architecture

RISC-V is an open source ISA which has grown rapidly since its inception due to its benefits of no license fee, flexibility, and scalability. It has become a driver of global chip innovation. However, although RISC-V has achieved significant success in recent years, its ISA standard is still under development and needs to be further enhanced for various scenarios. RISC-V is reaching maturity in the field of embedded processors, but it is still in its infancy in other areas such as cloud computing and HPC [10].

RISC-V is widely adopted in IoT applications such as smart homes, wearable devices, and edge computing, thanks to its modularity and ability to customize instruction sets. Embedded processors often perform tasks like Digital Signal Processing (DSP), encryption, and Artificial Intelligence (AI) computing, which demand low-power, efficient instruction sets to meet real-time performance and energy constraints.

RISC-V's extensibility also makes it a viable alternative for domain-specific processors tailored to specialized applications. Examples include Vortex [12], a Graphics Processing Unit (GPU) for rendering and graphics, Manticore [32], a floating-point processor with a chiplet architecture, and EdgeQ [28], a System-on-Chip (SoC) designed for 5G communication with over 50 custom instruction extensions.

In the realm of HPC, RISC-V is used in processors for data centers, laptops, supercomputers, and smartphones. Notable examples are Alibaba's XuanTie C910 [8], capable of running Android 12, MIPS' eVocore processors, scalable to 512 cores and 1024 threads, and open-source designs like Berkeley's BOOM [7] and China's Xiangshan [31], which rival commercial cores in performance.

The RISC-V ISA specification consists of a mandatory set of basic instructions, extensible with modular sets for domain-specific use. The basic instruction set only contains the basic integer instruction set for general integer operations, like addition and subtraction. The ISA extensions contain a variety of modular extensions to meet different computing needs. The RISC-V standard organization has constantly introduced new instruction set extensions to enhance the functionality of RISC-V, such as the V-extension for vector calculation and the H-extension for virtualization.

In [10], E. Cui *et al.* have also illustrated RISC-V's ISA, which can be seen in Figure 2.3. It consists of two parts: a non-privileged ISA and a privileged ISA. The first includes the basic integer instruction set and several non-privileged ISA extensions. The privileged ISA extension thus enables RISC-V architectures to also host Operating Systems (OSs), placing the ISA on par with well-established architectures, such as x86.

2.1.3 Hardware/Software Co-Design

The design of hardware/software systems involves three steps: modeling, validation, and implementation. According to Giovanni de Micheli *et al.* in [23], modeling entails conceptualizing and refining specifications to produce hardware and software models. Validation ensures confidence that the system will work as designed. Finally, implementation involves the physical realization of hardware through synthesis and the generation of executable software through compilation.

Hardware/software co-design techniques primarily target System-on-Chip (SoC) and embedded core designs that integrate general-purpose microprocessors, DSP structures, programmable logic, ASIC cores, memory peripherals, and interconnection buses onto a single chip. Traditionally, hardware and software were designed independently, requiring compatibility standards. However, as systems have grown more complex and power constraints have become critical, modern co-design methodologies aim to partition tasks between hardware and software from the early design stages [11].

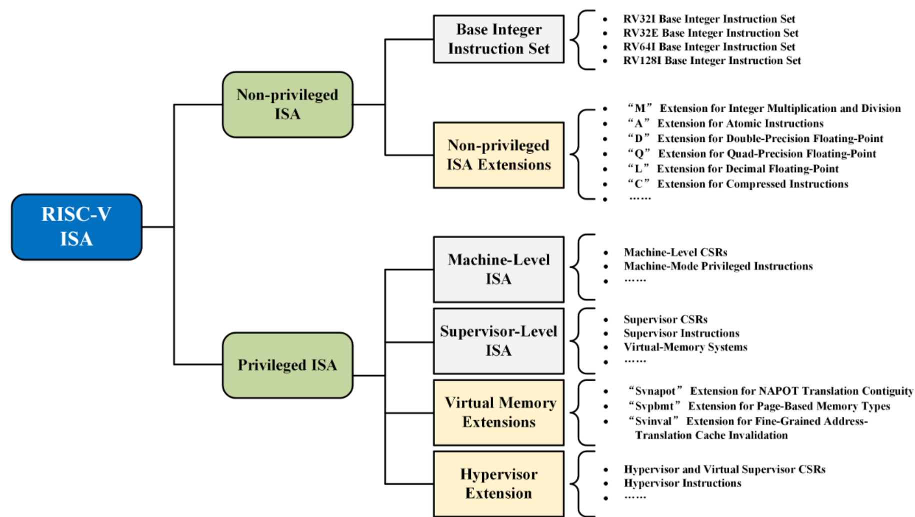


Figure 2.3: Taxonomy and functionalities of Instruction Set Architecture (ISA) extensions of Reduced Instruction Set Computer - V (RISC-V) [10].

In software development, the evolution from machine code to assembly language and then to High-Level Languages (HLLs) has significantly improved productivity. In [9], Coussy *et al.* summarize it perfectly: *"No one today would even think of programming a complex software application solely by using an assembly language"*. Similarly, in hardware design, early ICs were manually optimized and laid out, while advancements in the 1970s introduced gate-level simulation. HDLs like Verilog and VHDL later enabled automated logic synthesis, making a major leap in hardware design efficiency. By the 1990s, commercial HLS tools began automating the generation of Register Transfer Level (RTL) code from high-level software specifications, a trend further supported by the emergence of Electronic System Level (ESL) paradigms like SystemC in the 2000s.

High-Level Synthesis (HLS) aims to automate RTL generation from high-level code, typically C, C++, or OpenCL. Coussy *et al.* describe in great detail the steps involved in HLS [9]. It begins with compilation, where the input specification is transformed into a formal representation, such as a Control Data Flow Graph (CDFG), which exposes data and control dependencies. During allocation, hardware resources like functional units, storage, and interconnects are selected from a library based on design constraints. Next, scheduling assigns operations to clock cycles while respecting dependencies and resource availability. In binding, variables are mapped to storage elements, operations to functional units, and data transfers to buses or multiplexers. Finally, RTL generation produces a detailed architecture comprising a data path and a controller. This RTL design, written in HDL, is then ready for validation through simulation, followed by synthesis to hardware.

Simulation is used to verify that the generated RTL meets functional and performance requirements. It allows designers to test hardware designs at various abstraction levels before committing to physical implementation. According to Nazareno Bruschi *et al.*, simulators can be categorized based on their abstraction level and specialization [5].

Functional simulators replicate a system's high-level behavior without modeling internal architectural details. For example, Spike[1] models the ISA of a RISC-V core but does not account for pipelines or micro-architectural features, making it fast but unsuitable for performance analysis, which relies on cycle-accurate simulation [5].

Timing simulators, on the other hand, model micro-architectural details such as pipelines, caches, and Direct Memory Access (DMA). This category requires a further distinction among cycle-accurate, instruction-level, and event-driven. Cycle-accurate simulators perform their simulation in a cycle-by-cycle manner for precise timing, often using HDL tools like ModelSim or Verilator. Instruction-level simulators model the hardware with a coarser granularity, providing faster but less detailed results. Lastly, event-driven simulators abstract away cycles and focus on state changes. By jumping between events in time, these simulators, such as SystemC with Transaction-Level Modeling (TLM), achieve faster performance while maintaining sufficient accuracy for system-level validation [5].

2.2 Related Work

This section reviews recent research contributions in simulation frameworks, hardware/software co-design, and hybrid architectures. These works address challenges in designing, simulating, and optimizing accelerators and SoCs.

2.2.1 Simulation Frameworks for Accelerators

The design and optimization of accelerators, such as CNN processors, Deep Neural Network (DNN) accelerators, and heterogeneous platforms, require accurate and efficient simulation frameworks. These tools must ensure both functional correctness and timing accuracy with simulation speed to enable DSE while accounting for real-world hardware constraints, such as interconnect bandwidth, processing latency, and memory bottlenecks. Existing simulators often fall short by either being too slow (cycle-accurate models) or lacking in detail (functional simulators).

2.2.1.1 A Simulator for DCNN Accelerators

In 2019, Yi-Che Lee *et al.* proposed NNSim [19]. This simulator replicates the operation of CNN accelerators by modeling their core components as SystemC modules connected via a TLM network.

NNSim is made up of three essential components: a controller, which manages operations and sends configuration commands to other modules; a global buffer, which serves as a shared memory space for intermediate data, modeled as a TLM target with constant access latency; and a two-dimensional array of PEs responsible for computation, where each PE includes TLM sockets for data input and output. The workflow of the proposed CNN accelerator simulator is illustrated in Figure 2.4.

[1] https://github.com/riscv-software-src/riscv-isa-sim

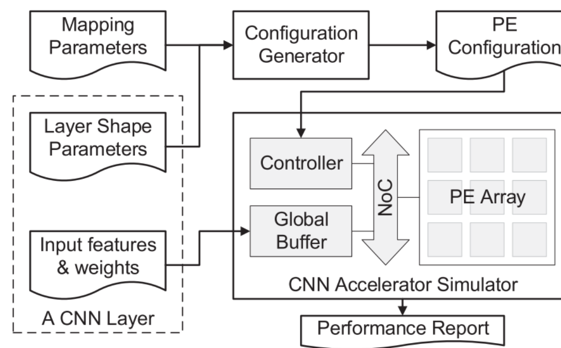


Figure 2.4: NNSim's proposed work flow of the Convolutional Neural Network (CNN) accelerator simulator [19].

PE operation is abstracted through the use of a processing pass model. This model simulates the computation of data instead of individual clock cycles, which requires the use of timestamp tables. An input table is used to track the arrival of input pixels for processing, whereas an output table records when output pixels are ready.

2.2.1.2 A Full-Platform Simulator for RISC-V based IoT Processors

In 2021, Nazareno Bruschi *et al.* proposed GVSoC [5], an open-source, event-driven simulation framework for Parallel Ultra-Low Power (PULP) platforms. It balances accuracy and speed by modeling multicore processors, hierarchical memory, Input/Output (I/O) peripherals, and accelerators while supporting DSE with <10% error when compared to physical implementations.

The framework consists of three main components (see Figure 2.5): C++ models to simulate system behavior (like cores, memory, DMA), JavaScript Object Notation (JSON) configuration files for customizing architecture parameters (like the bandwidth and latency of the interconnect), and Python generators to automate platform instantiation.

Interactions between components are modeled using a request-based mechanism, illustrated in Figure 2.6. Requests encapsulate information like memory addresses, payloads, and latency.

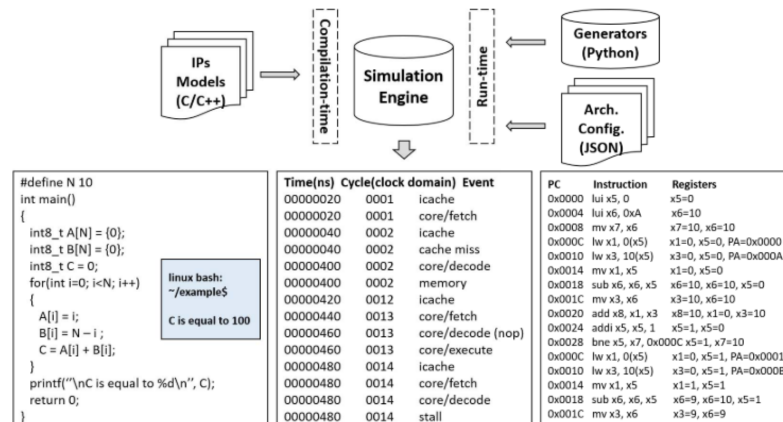


Figure 2.5: GVSoC's components [5].

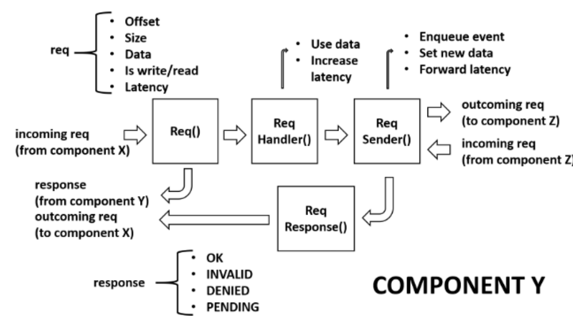


Figure 2.6: GVSoC's communication mechanism [5].

Components forward requests as needed, adding latency at each stage to simulate the behavior of a dynamic system.

2.2.1.3 A Framework for the Optimization of DNN Accelerators

In 2021, Jude Haris *et al.* proposed SECDA, a high-level framework designed to simplify the exploration and optimization of DNN accelerators [14].

The framework abstracts critical hardware components into configurable and reusable models. This enables designers to evaluate a wide range of architectural configurations without the need for detailed low-level implementation, which accelerates the prototyping process and allows for a systematic exploration of design trade-offs. An overview of SECDA's methodology is illustrated in Figure 2.7.

SECDA can be integrated with performance and power estimation tools that provide insights into metrics like energy consumption and area utilization. It models the typical components of an accelerator such as PE arrays, on-chip memory hierarchies, and interconnect systems. For instance, SECDA allows designers to simulate parallel compute operations on PE arrays, evaluating how array size and workload characteristics influence throughput and energy efficiency. Similarly, it provides detailed simulations of on-chip memory, capturing data reuse patterns, access latency, and bandwidth requirements. The interconnects between these components are also modeled.

SECDA is particularly well-suited for edge-specific DNN scenarios. This is because in these scenarios balancing power consumption and hardware area with computational performance is

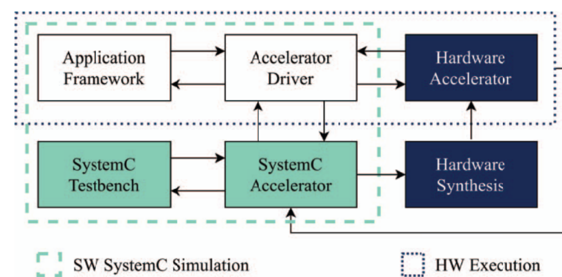


Figure 2.7: SECDA's methodology (from [14]).

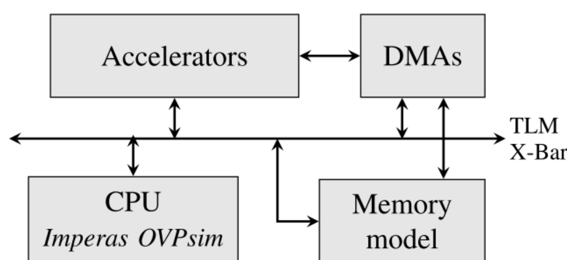


Figure 2.8: FLECSim-SoC's SystemC simulator setup [17].

critical. For instance, the framework can analyze the trade-offs between memory capacity and processing efficiency or evaluate the impact of interconnect topology on system throughput.

2.2.1.4 A Simulation Framework for System-on-Chips

Tim Hotfilter *et al.* proposed FLECSim-Soc in 2021 [17], a simulation framework designed to evaluate accelerators embedded in SoCs, with a focus on enabling co-design for DNN applications. The framework is capable of integrating many different components like accelerators, Central Processing Units (CPUs), and memories, all modeled in SystemC and connected via a TLM-based crossbar. This setup is illustrated in Figure 2.8.

Accelerators modeled in SystemC or at the RTL using tools like Verilator can be integrated into the platform. The framework allows the inclusion of cycle-accurate simulated peripherals and instruction-accurate processor simulations. This is particularly useful for scenarios involving DNN accelerators, where the interplay between CPU and hardware accelerators is very important to optimize performance.

Configuration is done through JSON files. Simulation has adjustable parameters like the memory size or the accelerator design. FLECSim also incorporates a memory model with configurable Dynamic Random Access Memory (DRAM) and DMA engines. For energy and area measurements, the framework uses tools like Accelergy.

2.2.1.5 A Simulation Framework for Machine Learning Accelerators

In 2023, Yannick Braatz *et al.* proposed SimPyler [4], a simulation-based framework designed for estimating the latency of CNN workloads on ML accelerators. It automates the process of generating simulation models from dataflow-modified CNN operators.

SimPyler makes use of Virtual Prototypes (VPs) to model the hardware components of ML accelerators, like PEs, DMA units, and memory buffers. This architecture is illustrated in Figure 2.9. Each VP encapsulates an analytical model that calculates the latency of specific operations, such as Multiply-and-Accumulate (MAC) or data transfers, based on the hardware parameters and workload characteristics.

CNN operators are transformed into Dataflow Graph (DFG) using the TVM machine learning compiler. The graphs represent the operations performed on the accelerator, with nodes corresponding to hardware operations and edges representing data dependencies. DFGs are derived

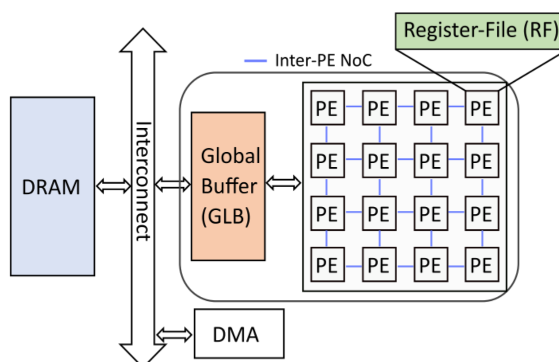


Figure 2.9: SimPyler's illustration of a general Machine Learning (ML) system [4].

from operator loop nests, which are modified using techniques like tiling, reordering, and unrolling to align with the accelerator's hardware architecture. For instance, DMA transfers and PE operations are marked with specific pragmas that guide their mapping onto the accelerator's hardware blocks.

To simulate the execution of DFGs, SimPyler uses Petri Nets, which are a graph-based methodology that models concurrent hardware processes. The graphs orchestrate the activation of VPs in the specified order and capture execution patterns like pipelining.

2.2.1.6 A RISC-V Microcontroller

eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) is a configurable RISC-V microcontroller described in SystemVerilog (SV), designed to target both small and ultra-low-power platforms. It is particularly well-suited for extensibility, enabling the integration of hardware accelerators without requiring modifications to the core Microcontroller Unit (MCU) [22].

The platform provides a modular architecture, where components such as the CPU, memory, and peripherals are organized in a way that simplifies integration. Custom accelerators can be instantiated directly in the top-level design without altering the existing microcontroller structure. Simulation is supported through various tools, including Verilator.

Architecturally, X-HEEP is divided into four main domains, as illustrated in Figure 2.10. First, the CPU subsystem domain, which hosts one of several embedded-class 32-bit open-source RISC-V cores developed by the OpenHW Group, such as the CVE2, CV32E40P, and its X-IF variant CV32E40PX, as well as the CV32E40X core. Second, the memory banks domain contains multiple memory banks that can vary in size and are used for storing both instructions and data. Each bank connects to the system through a dedicated interface, enabling parallel access without contention between banks. Third, the peripheral subsystem domain is composed of general-purpose peripherals that, while not essential during computation or data acquisition phases, provide useful features for broader microcontroller functionality. And lastly, the always-on peripheral domain, which encompasses all peripherals that must remain powered at all times.

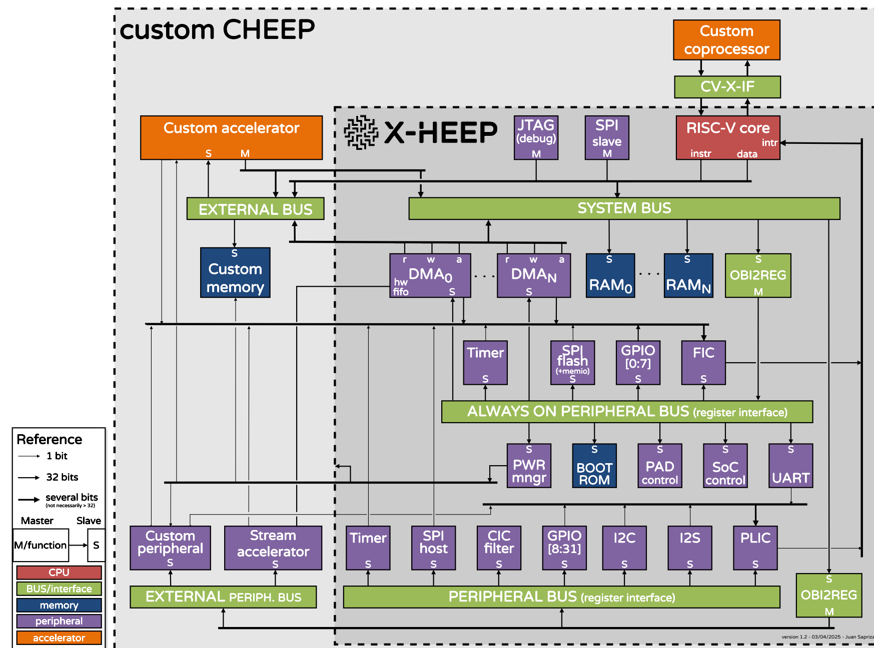


Figure 2.10: An overview of X-HEEP's architecture [22].

2.2.2 Hardware/Software Co-Design and Integration

The integration of hardware accelerators into heterogeneous systems poses significant challenges, particularly in ensuring communication and synchronization between hardware components and software. The design process must balance performance, energy efficiency, and scalability, while also addressing complex interdependencies between hardware and software layers. Traditional approaches often separate hardware and software design. This leads to inefficiencies and longer development cycles. Co-design methodologies seek to bridge this gap by allowing for the development of both hardware and software to be done simultaneously. This section explores frameworks designed to simplify and improve hardware/software co-design and integration.

2.2.2.1 TaPaSCo: Simplifying FPGA Integration in Heterogeneous Systems

In 2021, Carsten Heinz *et al.* proposed TaPaSCo [16], an open-source framework that automates FPGA design and eases their integration into heterogeneous systems. To this end, TaPaSCo offers two main components: a software runtime library to interact with FPGA accelerators and a tool flow to compose complete FPGA-based SoC designs.

The runtime Application Programming Interface (API) can be used from C/C++. Listing 2.1 provides an example code of a program that instantiates a TaPaSCo object, prepares buffers for input and output data, and finally creates a TaPaSCo job and launches it.

Listing 2.1: Adapted example code using TaPaSCo runtime API [15]

```
1 #include <tapasco.hpp>
2 #include <vector>
```

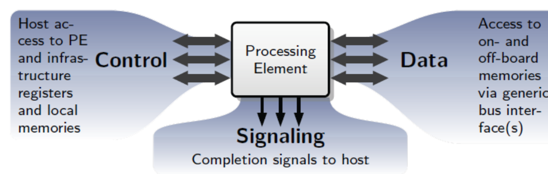


Figure 2.11: TaPaSCo's interface components [15].

```
3
4 Tapasco tapasco;
5
6 std::vector<int> v = {0, 1, ...};
7 auto buf = make_wrapped_ptr(v.data(), v.size() * sizeof(int));
8 auto job = tapasco.launch(KERNEL_ID, buf);
9
10 job();
```

In [15], Carsten Heinz *et al.* proposed an extension to this runtime: a Rust-based implementation for safer concurrency and optimized interrupt signaling. The authors also introduce Cascabel, a hardware dispatcher that offloads job management to the FPGA.

TaPaSCo abstracts accelerators as PEs, which adhere to a standardized interface composed of control, signaling, and data components (see Figure 2.11).

The interface is utilized to enforce compatibility across PEs implemented in HLS, HDL, or even imported softcore CPUs like RISC-V. Architectures composed of multiple PEs are grouped into Processing Clusters (PCs) and interconnected through signal aggregators to form a complete FPGA-based SoC, as illustrated in Figure 2.12.

2.2.2.2 ERAS: A Framework for Integrating RTL Models with Structural Simulation

ERAS was proposed by Shubham Nema *et al.* in 2023 [24]. The authors address the challenge of maintaining signal-level accuracy in RTL simulations, which is often difficult when interfacing with abstract event-based simulators. Traditional architectural simulators, which operate at the event granularity, simplify the detailed signal-level interactions. This process may cause a loss of

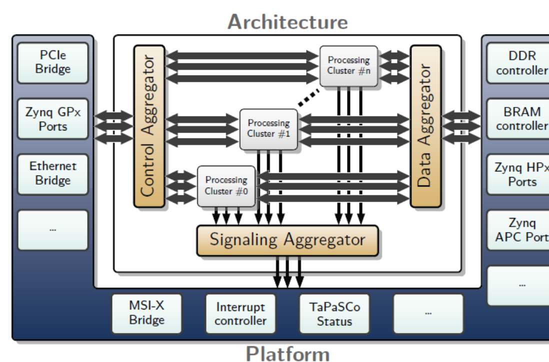


Figure 2.12: TaPaSCo's System-on-Chip (SoC) template [15].

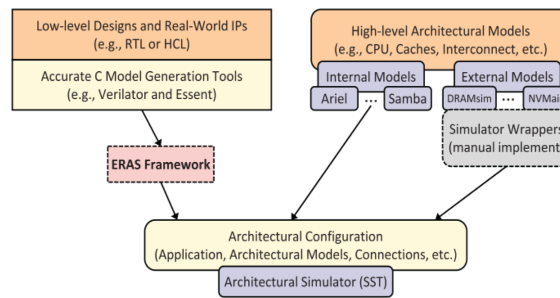


Figure 2.13: ERAS' framework place for architectural simulations [24].

precision. On the other hand, directly integrating RTL models with architectural simulators often results in complex and time-consuming design processes. Figure 2.13 illustrates ERAS's role in the architectural simulation workflow.

The framework uses ESSENT to generate cycle-accurate C-models from RTL netlists. These models are then integrated into high-level simulators like SST or gem5. This integration process involves using a parser that creates dynamically loadable components within the simulation toolkit, which will allow the interaction between the RTL and the architectural simulation layers. The generated models are not only functionally correct but also flexible.

It's important to note that ERAS has the ability to auto-generate wrappers for the C-models and integrate them into existing architectural simulators without requiring manual intervention. This allows developers to focus on higher-level design exploration while ensuring that the low-level RTL design maintains its accuracy.

2.2.2.3 RoSÉ: A Co-Simulation Infrastructure Enabling Pre-Silicon SoC Evaluation

In 2023, Dimi Nikiforov *et al.* proposed RoSÉ [25], a hardware/software co-simulation infrastructure that enables robotics developers to evaluate the end-to-end performance of robotics hardware and software in closed-loop environments. The infrastructure contains three main components, as depicted in Figure 2.14.

The environment simulation in RoSÉ is managed using AirSim, an open-source robotics simulator that generates sensor data and processes actuation commands. AirSim simulates the physical environment with discrete time steps which allows synchronization with the simulated hardware.

For hardware simulation, RoSÉ uses FireSim, an FPGA-accelerated RTL simulator capable of cycle-accurate hardware modeling. FireSim evaluates pre-silicon SoCs at high fidelity by deploying their RTL designs on FPGAs. To bridge the gap between the two simulations, the authors introduce the RoSÉ Bridge, which synchronizes data transfer and models I/O between the SoC and the environment. The bridge exposes memory-mapped I/O registers for communication.

It is crucial that both simulations stay aligned in time during the process. The synchronization process works in lockstep, meaning both the environment and the hardware simulators advance their simulations at a predefined rate. However, since these simulators operate with different time

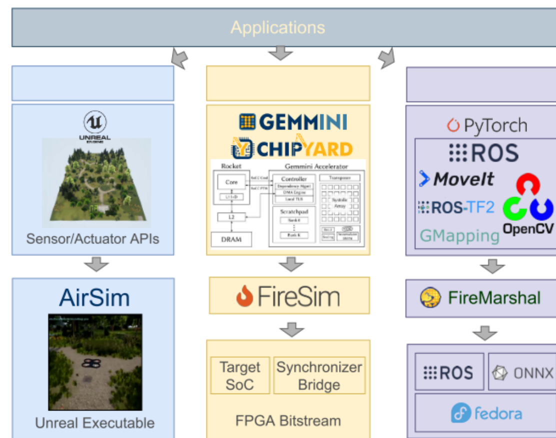


Figure 2.14: RoSÉ's components [25].

intervals (AirSim uses frames, while FireSim operates in clock cycles), there is a need for a synchronization mechanism capable of ensuring that events in one simulator are correctly reflected in the other. The RoSÉ synchronizer is the software component that coordinates the communication between the two simulators. It ensures that both simulators update their states in sync by defining a synchronization period based on the rate of each simulator. The data required for synchronization is transferred between the two simulators using Transmission Control Protocol (TCP) packets. The packets are structured with a header that identifies the type of data being transferred and a payload containing the actual data. The synchronizer first sends synchronization packets between the simulators, which convey the current simulation time, cycle counts, and other synchronization-related information. Once synchronization is confirmed, data packets are then transmitted to update both simulators on the relevant state changes. For instance, if FireSim needs sensor data from AirSim, the synchronizer sends a request for this data via TCP. Once AirSim processes the request and sends the data back, the synchronizer ensures that this data is injected into the FireSim hardware model through the I/O registers, where the simulated SoC can access it.

2.2.3 Optimization of Simulation Techniques

The growing complexity of modern computing systems, including multicore processors and specialized accelerators, demands simulation techniques that balance accuracy, speed, and reliability. Traditional simulation methods such as full RTL simulations or sequential SystemC-based simulations often struggle to meet these requirements or end up failing entirely. This section explores approaches that aim to optimize simulation frameworks.

2.2.3.1 Parallel SystemC Simulation for Loosely-Timed TLM Models

In 2020, Gabriel Busnot *et al.* proposed a parallel SystemC kernel that supports loosely-timed TLM models while maintaining compliance with IEEE SystemC standards [6].

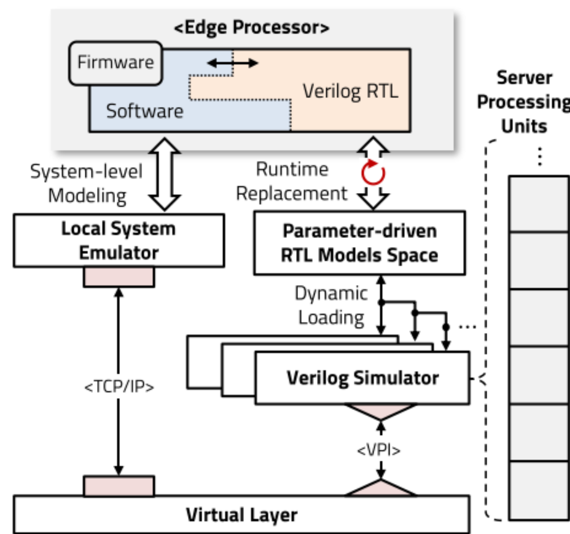


Figure 2.15: The proposed emulator [18].

The proposed kernel implements a parallel process evaluation mechanism, which ensures atomicity and reproducibility. This makes it possible to evaluate SystemC processes concurrently on multicore host platforms without introducing race conditions or inconsistencies. The kernel supports Direct Memory Interface (DMI) so that simulated hardware components can directly access the host system's memory. This is valuable in loosely-timed simulations since it reduces the overhead associated with higher-level message-passing systems.

Loosely-timed TLM models are characterized by coarse-grained timing abstractions and are particularly suited for this parallel approach. Although the models abstract away fine-grained synchronization between components, they allow for faster simulations while still capturing performance metrics. The integration of DMI further enhances the simulation by enabling direct memory interactions between the simulation environment and external processes, like software applications running on a simulated processor.

2.2.3.2 Flexible Runtime Replacement of Verilog RTL Models

In 2021, Jisu Kwon *et al.* proposed a simulation framework for edge processors that combines parameter-driven flexibility with real-time RTL model integration, as illustrated in Figure 2.15. The framework introduces a virtual layer that interfaces between embedded software running on QEMU and Verilog RTL models [18].

The interaction between QEMU, the system emulator, and the Verilog RTL simulators is done through the virtual layer (see Figure 2.16). QEMU generates virtual register memory based on processor specifications, which are stored in a JSON format. Modifications to QEMU allow it to emulate peripheral operations and communicate register activities to the Verilog simulation via IEEE Verilog PLI 2.0. A network socket module provides real-time data exchange and also application-level debugging through the GNU Debugger over TCP.

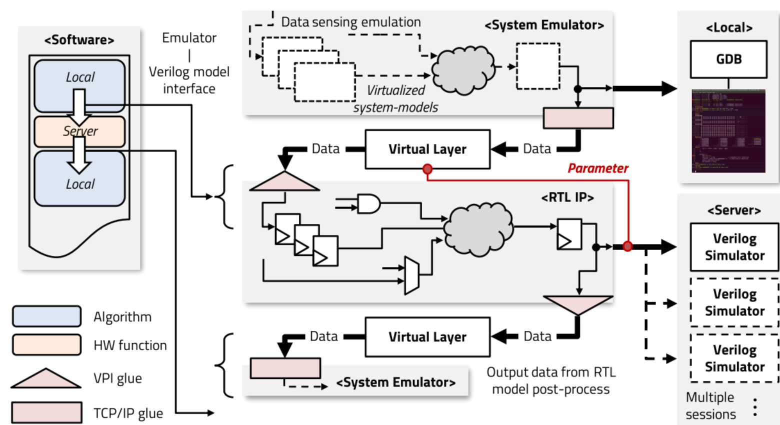


Figure 2.16: Execution flow illustrating the transition from software emulation to Register Transfer Level (RTL) model simulation during execution (from [18]).

The framework also supports runtime dynamic loading of simulation sessions. Workloads can be distributed across multiple cores or cloud servers since RTL simulations are segmented based on parameter sets. A shared library mechanism links the system emulator and simulator to specific parameter sets without requiring application code modifications.

2.2.4 Hybrid and Configurable Architectures

As computing systems become increasingly diverse, hybrid and configurable architectures are gaining prominence in research and development. These architectures typically combine general-purpose processors with specialized accelerators or peripherals.

2.2.4.1 RISC-V + CGRA Hybrid Systems

In 2021, Xiaoyi Ling *et al.* proposed a software framework that automates the generation of hybrid systems combining a RISC-V processor with a CGRA. The selected RISC-V implementation is the CV32E40P core, which supports the RV32IMC ISA alongside XPULP custom extensions, while the CGRA is modeled using the ADRES architecture.

The ADRES CGRA is composed of a 4×4 grid of PEs, where each PE features an ALU capable of basic arithmetic and logical operations. The architecture integrates register files for the top row of PEs and memory ports for each row (see Figure 2.17). The communication within the CGRA occurs via connections to the neighboring and diagonal PEs. The hybrid system uses a Harvard memory architecture, which separates instruction and data memories. A dual-port banked memory enables concurrent access by the processor and the CGRA.

The RISC-V core and the CGRA communicate via two interfaces: memory-mapped I/O and shared memory. The I/O ports act as pointers to shared memory or direct data channels. This reduces the need for frequent shared memory access. The RISC-V processor is responsible for managing the CGRA, invoking it for computational hotspots, and monitoring its execution status.

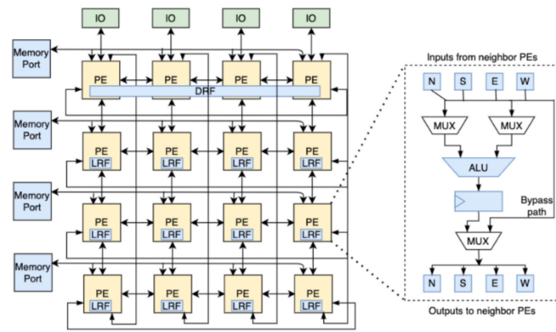


Figure 2.17: 4×4 ADRES Coarse-Grained Reconfigurable Array (CGRA) architecture [20].

The framework generates the RTL for the hybrid system, including the RISC-V/CGRA interface, dual-port memory, and a top-level Verilog model. The CGRA Verilog RTL is generated using CGRA-ME, which is then compiled and simulated with ModelSim. Figure 2.18 shows a block diagram of the resulting hybrid system.

2.2.4.2 RISC-V Platform for Education and Research

In 2022, S. Ahmadi-Pour *et al.* proposed μRV32, an open-source RISC-V platform aimed at education and research. The platform integrates a 32-bit RISC-V core with a set of peripherals using a generic bus system. The μRV32 implementation is available in two forms: an RTL version written in SpinalHDL for FPGA deployment and a VP for high-level simulation. Both are binary-compatible and support custom instruction extensions [1].

The μRV32 core implements the RV32I base instruction set, along with optional extensions like RV32M for multiplication/division and RV32C for compressed instructions. Four core variants (RV32I, RV32IC, RV32IM, and RV32IMC) can be generated depending on configuration needs. The multiple components of this core are represented in Figure 2.19. The core operates with a valid-ready handshake protocol. Requests to peripherals are routed based on address mappings.

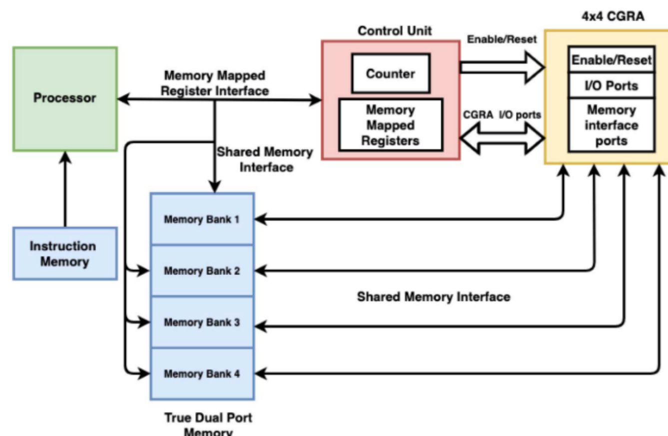


Figure 2.18: Block diagram for the resulting hybrid system [20].

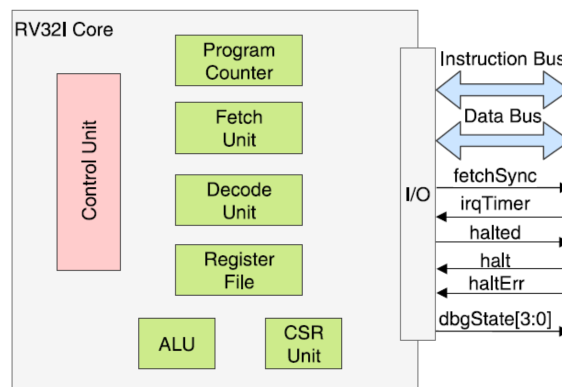


Figure 2.19: μRV32's Reduced Instruction Set Computer - V (RISC-V) core [1].

The SoC platform integrates the μRV32 core with peripherals through the SimpleBus interface. For instance, the Universal Asynchronous Receiver/Transmitter (UART) peripheral is used for serial communication, while the Core-Local Interrupt Controller (CLINT) provides timer-based interrupts. A block diagram of this SoC platform is illustrated in Figure 2.20.

The RTL version is designed to be FPGA-friendly and compatible with open-source toolchains like IceStorm. The VP provides a high-level abstraction for simulation, suitable for development and testing.

2.3 Summary

This section summarizes the related work discussed previously, identifying trends and challenges.

2.3.1 Overview of State-of-the-Art Frameworks and Methodologies

Several simulation frameworks address the challenges of designing and evaluating hardware accelerators, particularly for deep learning and edge applications. NNSim [19] models CNN accelerators by abstracting hardware components using SystemC and TLM. Similarly, GVSoC [5] focuses on PULP platforms, offering an event-driven simulation framework that models multicore

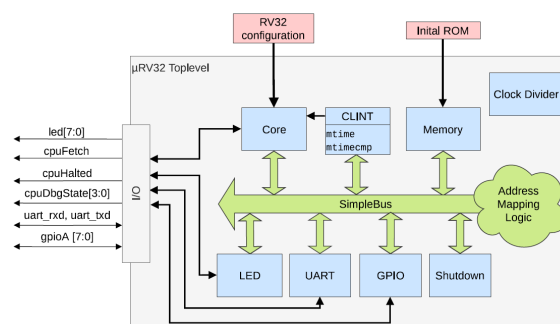


Figure 2.20: μRV32's System-on-Chip (SoC) diagram [1].

processors, memory hierarchies, and peripherals. Its request-based communication mechanism captures dynamic interactions in heterogeneous systems and supports DSE.

Frameworks like SECDA [14] and FLECSim-SoC [17] target DNN accelerator optimization. SECDA abstracts hardware components into reusable models while integrating with performance and power estimation tools for edge-specific DNN scenarios. FLECSim-SoC provides a SystemC-based simulation environment for SoCs, integrating accelerators, CPUs, and memory while supporting cycle-accurate peripherals. SimPyler [4] further enhances ML accelerator design by automating the generation of simulation models from modified CNN dataflows.

In hardware/software co-design, TaPaSCo [16] simplifies the integration of FPGAs into heterogeneous systems by automating accelerator design and offering a runtime library for execution management. ERAS [24] builds on this by integrating RTL models with architectural simulators, providing signal-level accuracy alongside design flexibility.

In robotics and pre-silicon SoC evaluation, RoSÉ [25] combines hardware and environment simulations using AirSim for environment modeling and FireSim for cycle-accurate hardware simulation.

Finally, optimization techniques improve simulation performance. A parallel SystemC kernel [6] by Gabriel Busnot *et al.* enhances the speed of loosely-timed TLM models through concurrent process evaluation on multicore hosts. Jisu Kwon *et al.'s* metamorphic framework [18] combines RTL flexibility with real-time simulation, integrating QEMU-based emulation with Verilog simulations.

2.3.2 Trends and Open Challenges

A clear trend in the reviewed frameworks is the use of abstraction. NNSim [19] and GVSoC [5] prioritize abstracting hardware components to enable more flexible simulations. Another common approach is the incorporation of performance and power estimation tools for optimized DSE, as seen in SECDA [14] and FLECSim-SoC [17].

In hardware/software co-design, TaPaSCo [16] and ERAS [24] focus on bridging the gap between high-level architectural models and low-level implementations.

Despite these advancements, challenges remain. The main issue seems to be balancing simulation accuracy with speed. While loosely-timed models improve performance, they sacrifice detail, whereas cycle-accurate models are computationally expensive. The integration of heterogeneous components also faces challenges in communication.

Another gap is the lack of generalization across frameworks. Tools like GVSoC [5] and SimPyler [4] are tailored to specific domains, limiting their applicability to other systems. Additionally, the absence of standardized benchmarks and metrics makes cross-framework evaluation difficult.

Chapter 3

Co-Simulation Framework Design

This chapter presents the architecture and design of the proposed co-simulation framework. The framework enables the bidirectional communication between high-level software simulators and simulated hardware peripherals. A key feature of the framework is its lock-step execution, ensuring that both sides advance in perfect synchrony at the cycle level. To the best of our knowledge, this level of tightly coupled, bidirectional co-simulation is not supported by existing state-of-the-art solutions.

3.1 Architectural Overview

The co-simulation framework developed in this work is composed of three main components, as illustrated in Figure 3.1.

The first component is the hardware platform. This represents the SoC environment. It typically includes a processing core executing a host application and a collection of peripheral modules. While peripheral designs can be arbitrarily complex, featuring control pipelines, FSMs, or tightly integrated logic, the framework deliberately imposes a constraint: peripheral modules generated by the framework are kept simple and lightweight. Their primary purpose is to expose internal state via memory-mapped registers. All functional behavior is delegated to the corresponding high-level simulators.

The second component includes high-level simulators. For each peripheral instance P_i, there is a corresponding simulator S_i, written in a HLL. Simulators encapsulate the peripheral's behavior and are connected to the hardware platform through a communication channel. Users are only responsible for implementing the simulator's domain-specific logic. Communication and data transfer between the hardware and each simulator are handled transparently by the framework.

The last component is the code generation pipeline that automates the integration process. The entry point is a user-defined JSON configuration file, where the peripheral layout is declared, including register definitions and communication properties. This file is parsed into an internal

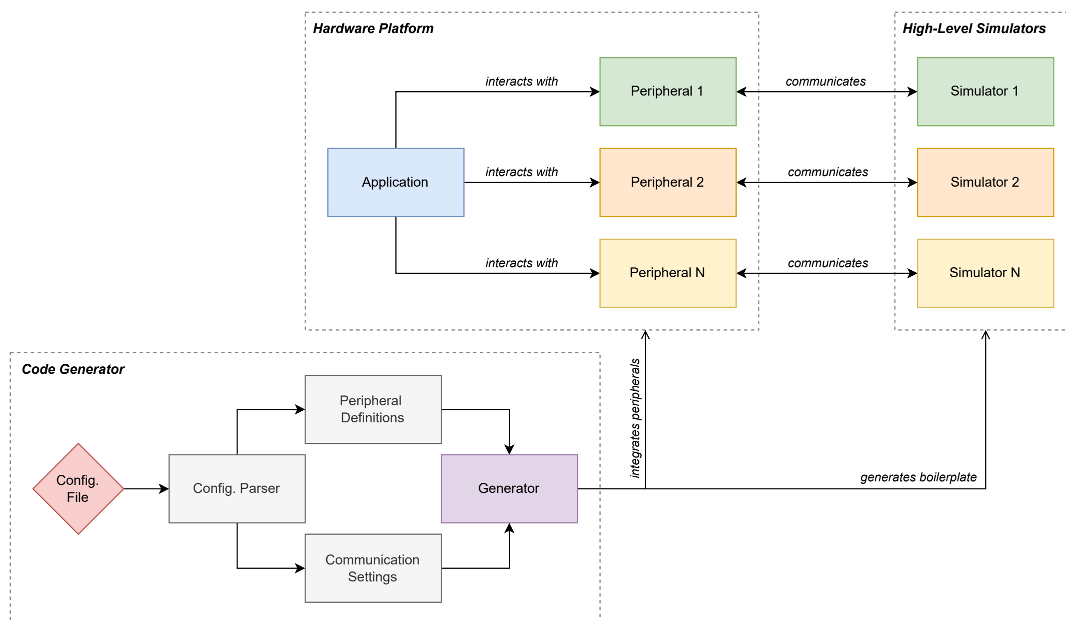


Figure 3.1: High-level block diagram of the framework's generic architecture.

data model and used to generate all the necessary artifacts required to integrate the defined peripherals into the hardware platform, as well as base simulator projects that communicate with said peripherals.

Figure 3.2 shows the concrete instantiation of this architecture using X-HEEP, a RISC-V based open-source SoC platform, as the target hardware environment. In this configuration, the host application runs on a simulated RISC-V processor within X-HEEP and interacts with peripherals using Memory-Mapped Input/Output (MMIO). Peripherals are SV modules connected to the CPU and allocated distinct memory regions, defined in the platform's testbench via address offsets and sizes. The host application can simultaneously interact with multiple peripherals, provided their memory locations are known in advance. To enable co-simulation, each peripheral is equipped with a C-based Direct Programming Interface (DPI) wrapper, which handles communication between the SV module and its external simulator. These wrappers use ZeroMQ as the communication backend.

3.2 Peripheral Architecture

Each peripheral in this framework follows a standardized hardware structure, automatically generated from user-provided configuration files. While the internal register layout may vary between peripherals, their core responsibilities remain the same: manage a set of internal registers accessible by both the processor and the corresponding high-level simulator and expose an interface that enables the simulator to initiate memory transactions via the hardware platform's memory bus protocol.

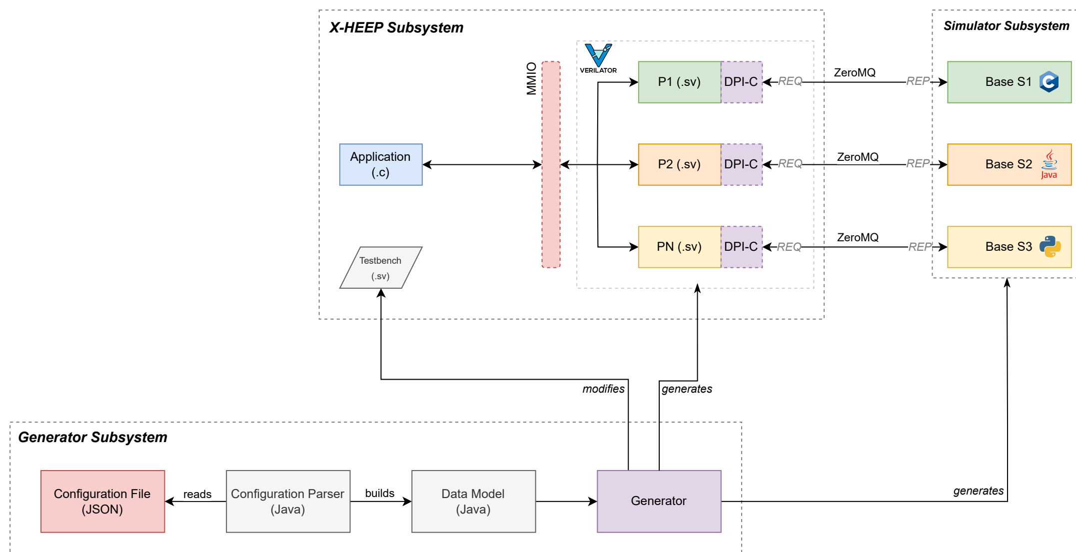


Figure 3.2: High-level block diagram of the framework when considering the use of eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) as the hardware platform System-on-Chip (SoC).

The specific memory interface protocol can vary depending on the chosen SoC platform. In this work, all peripherals are designed to integrate with X-HEEP, which employs the Open Bus Interface (OBI) protocol for communication with memory. As such, the generated peripherals implement an OBI-compatible interface to initiate memory transactions.

At a structural level, each peripheral connects to the rest of the system through a fixed set of signals. These include a clock signal to synchronize all logic, an active-low reset signal for initialization and fault recovery, an MMIO interface for register-level access by the CPU, and two dedicated OBI channels, one for issuing read transactions and another for writes.

3.2.1 Register Interface

To allow the SoC's CPU to communicate with a peripheral, each module includes an MMIO register interface. The CPU sends a request through `reg_req_t`, and the peripheral responds through `reg_rsp_t`. The definition of these types is presented in Listing 3.1.

Listing 3.1: Fields of the `reg_req_t` and `reg_rsp_t` packed struct types.

```
typedef struct packed {
    logic        valid;
    logic        write;
    logic [3:0]  wstrb;
    logic [31:0] addr;
    logic [31:0] wdata;
} reg_req_t;

```

```
 9 typedef struct packed {
10     logic        error;
11     logic        ready;
12     logic [31:0] rdata;
13 } reg_rsp_t;
```

When the CPU initiates a transaction, it populates `reg_req_t` with a request packet:

- `valid`: is asserted high to indicate that a request is active and should be processed by the peripheral in the current cycle.
- `write`: is asserted high in case of a write, and low in case of a read.
- `wstrb`: a byte-level write strobe, indicating which bytes of `wdata` should be written.
- `addr`: specifies the address of the register being accessed.
- `wdata`: contains the 32-bit data payload to be written to the specified register, valid only during a write operation.

Upon receiving and decoding the incoming request, the peripheral provides a response through `reg_rsp_t`:

- `ready`: is asserted high once the peripheral has finished processing the received request.
- `error`: is asserted high if the request targets an invalid address or performs an unsupported operation.
- `rdata`: holds the 32-bit data result of a read operation and is only meaningful when `write` is low and `ready` is high.

The implementation of this MMIO request-response protocol within every peripheral is handled by two distinct SV `always` blocks, which are automatically generated by the framework based on the set of input and output registers.

Listing 3.2 shows the simplified logic used to handle register write requests from the CPU.

Listing 3.2: Simplified register-write block.

```
 1 always_ff @(posedge clk_i or negedge rst_ni) begin
 2     if (!rst_ni) begin
 3         /* assign defaults... */
 4     end else if (reg_req_i.valid && reg_req_i.write) begin
 5         case (reg_req_i.addr[7:2])
 6             0: r1_i <= reg_req_i.wdata;
 7             1: r2_i <= reg_req_i.wdata
 8             /* (...) */
 9             default: ;
10         endcase
11     end
12 end
```

Conversely, the register read logic, depicted in Listing 3.3, is implemented using a combinatorial block. If a valid request is detected, the `case` statement assigns the value of the requested internal register to `reg_rsp_o.rdata`.

Listing 3.3: Simplified register-read block.

```
always_comb begin
    reg_rsp_o = '{ready: 1'b1, error: 1'b0, rdata: '0};

    if (reg_req_i.valid && !reg_req_i.write) begin
        case (reg_req_i.addr[7:2])
            2: reg_rsp_o.rdata = r3_o;
            /* (...) */
            default: reg_rsp_o.error = 1'b1;
        endcase
    end
end
```

3.2.2 Memory Access Interface

In addition to registers, each peripheral can directly access system memory using the OBI protocol, which in turn will allow corresponding external high-level simulators to access the SoC's memory.

In our framework, each peripheral acts as an OBI master, issuing memory transactions. The memory system on X-HEEP acts as the OBI slave, responding to these requests. Listing 3.4 provides the definitions of the `obi_req_t` and `obi_resp_t` structures, used during OBI transactions.

Listing 3.4: Fields of the `obi_req_t` and `obi_resp_t` packed struct types.

```
typedef struct packed {
    logic        req;
    logic        we;
    logic [3:0]  be;
    logic [31:0] addr;
    logic [31:0] wdata;
} obi_req_t;

typedef struct packed {
    logic        gnt;
    logic        rvalid;
    logic [31:0] rdata;
} obi_resp_t;
```

Theoretically, a single OBI channel could conduct both memory read and write operations. However, this imposes a considerable constraint: within a single clock cycle, only one of the two (either a read or a write) would be possible. To overcome this limitation, each generated peripheral is equipped with two OBI channels, one dedicated to reads and another to writes.

When peripherals conduct a memory request:

- `req`: is asserted high to indicate that a valid request is present and should be processed by the memory.
- `we`: is asserted high in case of a write request and low in case of a read request.
- `be`: the byte enables, typically asserted as `4'b1111`.
- `addr`: contains the memory address from which data is to be read from or written to.
- `wdata`: contains the data to write to the specified address, ignored for read operations.

The response to this request is conveyed through the `resp` ports:

- `gnt`: is asserted high by the memory to acknowledge and grant the request.
- `rvalid`: is asserted high by the memory one or more cycles after `gnt` to indicate that valid read data is available on `rdata`, only meaningful during reads.
- `rdata`: contains the 32-bit data retrieved from the specified address, only meaningful during reads.

3.3 Communication Subsystem

In our framework, the SoC platform and the external high-level simulators operate as independent software processes running on the same OS. This enables the use of standard inter-process communication libraries, like ZeroMQ, to mediate co-simulation between the hardware design and software models.

In essence, communication between any two software processes, P_i and P_j, follows a standardized sequence:

1. Initially, both processes configure the communication channel, specifying socket roles, addresses, timeouts, and other relevant parameters.
2. Once initialized, processes engage in bidirectional data exchange using repeated sequences of `send` and `recv` operations.
3. When communication is no longer needed, resources are released, and internal structures are cleaned up.

Because SV does not natively support ZeroMQ or external networking libraries, we rely on the DPI to call C-defined functions from within SV modules. Each peripheral is paired with a DPI-C wrapper composed of a header and source file. The header declares four core communication functions, as shown in Listing 3.5.

Listing 3.5: DPI header function declarations.

```
void communication_init();

void communication_send(/* input registers... */);

void communication_recv(/* output registers... */);

void communication_exit();
```

As the simulated SoC is responsible for initiating communication, the framework adopts a Request-Reply (REQ-REP) pattern, where X-HEEP acts as a client, requesting data, and simulators act as servers.

The `communication_init` function is invoked when the peripheral is instantiated. It sets up the required communication context, including ZeroMQ sockets, serialization buffers, and supporting structures.

Peripherals send their internal state to simulators using `communication_send`, on every clock cycle. Since different peripherals expose different sets of input registers, this function does not operate on a fixed argument list. Instead, all input registers are packed into a single serialized message, sent once per cycle. Several serialization formats were considered. A rudimentary approach involves sending a raw, formatted buffer. This technique is simple but error-prone and inflexible. Human-readable formats, like JSON or Yet Another Markup Language (YAML), are easier to debug but are very inefficient due to verbose key-value syntax. Ideally, serialization should produce compact messages while retaining lookup convenience methods. This is precisely what MessagePack [1] offers: efficient binary serialization and automatic integer encoding. Consequently, all input registers, arguments of `communcation_send`, are serialized into a byte array using MessagePack and transmitted over ZeroMQ.

After processing the received input, the simulator may compute response data meant to influence the peripheral's state. To retrieve this data, the peripheral invokes `communication_recv`, also once per clock cycle. As with the `communication_send` function, this operation cannot rely on a fixed argument list, as each peripheral may have a different set of output registers. This operation starts by receiving a serialized buffer from a simulator, which it deserializes using MessagePack. Then, it extracts and updates relevant output registers by name. If deserialization fails, the peripheral assigns user-defined default values to its outputs.

Once communication is no longer required, the peripheral calls `communication_exit`, which deallocates internal structures and shuts down the ZeroMQ socket.

As an example, consider a peripheral wrapper defined by the user, containing two signed 32-bit input registers, r_1 and r_2, and one signed 32-bit output register r_3. These registers form part of the peripheral's internal state and are made accessible to the corresponding simulator, as illustrated in Listing 3.6.

[1] https://github.com/msgpack/msgpack-c

Listing 3.6: A brief overview of the communication flow employed in generated SV peripheral wrappers.

```
always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        /* assign defaults... */
    end else begin
        /* temporary variable... */
        int signed _r3;

        /* send inputs... */
        communication_send(r1, r2, ...);

        /* recv outputs... */
        communication_recv(_r3);

        r3 <= _r3;
    end
end
```

3.3.1 Abstraction Layers

Direct integration of ZeroMQ and MessagePack into every DPI-C wrapper would lead to bloated and difficult-to-write code. To manage this complexity, the framework employs two lightweight abstraction libraries: `cipc` for communication and `cmsg` for serialization.

The `cipc` library defines a generic interface for communication backends. Its key abstraction is the `cipc` struct, shown in Listing 3.7.

Listing 3.7: The `cipc` struct fields.

```
typedef struct cipc {
  cipc_err (*init) (void **context, const void *config);
  cipc_err (*send) (void *context, const char *data, size_t length);
  cipc_err (*recv) (void *context, char *buffer, size_t length);
  void (*free) (void *context);

  void *context;
} cipc;
```

This structure abstracts away specific communication details through function pointers. The default implementation, used by the framework, uses ZeroMQ as its backend. However, the design is protocol-agnostic. Alternative backends can be implemented by conforming to this interface.

The `cmsg` library simplifies the encoding and decoding of register states into binary messages. It replaces boilerplate code in the DPI-C wrappers with generic map-based operators, allowing the framework to construct key-value maps of register data, serialize these maps into byte arrays using MessagePack, and deserialize received messages and extract values by field name.

3.4 Code Generation Pipeline

The integration of custom simulators into the co-simulation framework environment is automated through a code generation pipeline. This pipeline produces an extensive set of artifacts required for the entire system to function.

The pipeline is driven by a user-defined configuration file written in JSON. This file describes a *project*, which, within the context of our framework, consists of a host C-based program running on X-HEEP's core interacting with one or more high-level simulators connected to custom peripheral interfaces.

Each simulator entry in the configuration must specify the target programming language and the desired class name for the generated simulator scaffold. Additionally, users define peripheral properties, such as input and output registers. Every register must contain a unique identifier, used as both a variable name and serialization key, a default value, which acts as a fallback, a fixed bit width and signedness, the corresponding SV type, and an optional description field, useful for documentation.

The configuration also captures communication parameters, like timeout thresholds for responses and retry policies in case of communication failures.

Chapter 4

Multi-Language Co-Simulation API

This chapter presents the high-level API used by simulators to interact with X-HEEP from within their programming environments. We begin by outlining the API's purpose and functionality, then explore its core components in detail. Finally, we examine the structure and behavior of the main communication loop from the simulator's perspective.

4.1 API Functionality

A primary motivation for developing this framework was to allow developers to interact with a simulated SoC platform directly from within their simulator's native programming environment. This functionality enables users to prototype and iterate the intended simulator functionality in software, rather than navigating the complex process of manually extending a hardware platform.

Early integration experiments, which involved manually creating high-level simulators communicating with X-HEEP, revealed a recurring pattern: while peripheral-specific logic varied between projects, the supporting infrastructure code remained largely identical. This observation prompted the design of a reusable, modular, and extensible API layer, which has since been implemented in multiple programming languages, in order to support simulators written in different languages.

Regardless of the target language, the API must fulfill three key responsibilities. First, it must enable the simulator to read the current state of its peripheral inputs at each simulation step. Second, it must allow the simulator to respond to this state and issue updates accordingly. Lastly, it must provide a mechanism to issue memory transactions to X-HEEP on demand. This third capability presents challenges regarding race conditions that will be discussed in later sections.

In the simulator's programming environment, observing and modifying a peripheral's state is performed through a pair of `recv` and `send` operations, conceptually similar to the DPI-C block used in SV modules presented in Chapter 3.

4.2 Core Components

At its core, the API is made up of four distinct components: a data class used to abstract messages being exchanged between the simulator and the peripheral wrapper within the verilated SoC, a simulator interface containing a method responsible for advancing the simulation, a communication module which deserializes incoming messages from the peripherals and serializes the simulator's outputs into an appropriate format, and, lastly, a memory controller that allows the simulator to issue memory requests on demand. The implementation of each of these components is explained in the upcoming sections.

4.2.1 Simulation Data Abstraction

One of the main benefits of using a HLL is the ability to define expressive abstractions and structured data types. Instead of adhering to low-level conventions, we can model our system using constructs that better represent its semantics.

In our API, the data exchanged between a simulator and a peripheral interface during a simulation cycle is encapsulated in the `simulation_data` class. Rather than handling each register as a raw, untyped value, this class maintains a typed mapping from register names to their corresponding values. Internally, it uses a hash map where string keys (register names) map to values stored using a type-erased `any`, capable of holding any standard data type.

The class also offers type-safe access through a set of helper methods that retrieve and cast register values to specific types. This preserves crucial information like signedness and bit width, allowing simulators to interface with registers in a semantically meaningful way.

Beyond its key-value storage, the class tracks a `direction` flag that indicates whether the data instance represents values being set to or received from a peripheral.

Lastly, `simulation_data` provides higher-level utilities for interacting with memory transaction metadata. These methods do not implement transaction logic themselves but support their analysis. This topic is clarified in Section 4.2.4. A simplified pseudocode representation of this class is shown in Listing 4.1. Naturally, this pseudocode needs to be adapted for the simulator's target language. Currently, we support the generation of this pseudocode functionality in three target languages: C++, Python, and Java.

Listing 4.1: A pseudocode implementation of the `simulation_data` class.

```
1  CLASS simulation_data:
2      PRIVATE:
3          DECLARE data AS map<string, any>
4          DECLARE dir AS direction
5
6      PUBLIC
7          CONSTRUCTOR simulation_data(data_i: map<string, any>, dir_i: direction):
8              SET data = data_i
9              SET dir = dir_i
```

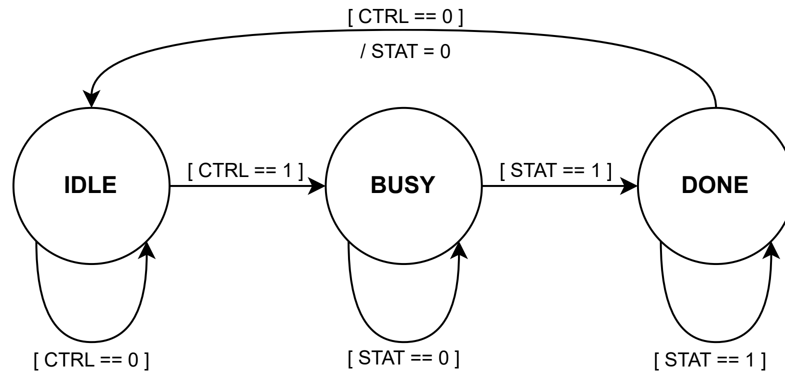


Figure 4.1: High-level block diagram of the `step` method FSM.

```
        END CONSTRUCTOR

        METHOD get_uint8(key: string) RETURNS uint8_t:
            RETURN CALL (CALL map.get(key)).cast(uint8_t)
        END METHOD

        METHOD get_int8(key: string) RETURNS int8_t:
            // (...)
        END METHOD
END CLASS
```

4.2.2 Simulator Execution Model

At the core of the simulator's communication loop lies the `step` method, which advances the simulation by one cycle. This method is part of a minimal interface, presented in pseudocode in Listing 4.2.

Listing 4.2: A pseudocode implementation of the simulator interface.

```
CLASS simulator:
    VIRTUAL step(inputs: simulation_data) RETURNS simulation_data
END CLASS
```

Each invocation of `step` conceptually represents a single hardware clock cycle. The logic behind this step is partially implemented in a class automatically generated by the framework, based on the user-defined simulator interface in the JSON configuration file. This base class implements a simple three-state FSM, illustrated in Figure 4.1, and maintains internal fields for all input and output registers defined in the peripheral specification. For every input and output register, the class provides appropriate getter and setter methods, respectively.

The FSM governs the simulator's execution across three states:

- `IDLE`: in this state, the simulator awaits a start signal from the CPU, which occurs when the software application on the X-HEEP platform asserts a common `control` input register.
- `BUSY`: once triggered, the simulator transitions to this state to carry out its core computation, remaining here until the computation concludes.
- `DONE`: upon completion, the simulator transitions to this state and signals the host by asserting a `status` output register, also common across all generated peripherals. The simulator may transition back to the `IDLE` state if the control register is de-asserted by the CPU, setting the status register low in the process.

While in the `BUSY` state, the simulator will invoke an abstract method called `busy()`, which must be implemented by a subclass. It encapsulates the simulator's custom behavior during the `BUSY` state. Each time `step` is invoked during `BUSY`, it calls the abstract method to determine if processing should continue. If so, the method returns `BUSY`; otherwise, it returns `DONE`, prompting a state transition.

Users extend the base simulator class and override `busy()` to define their simulator's domain-specific behavior. As an example, consider a user-defined peripheral wrapper which includes two 16-bit signed input registers, r_1 and r_2, and one 8-bit unsigned output register r_3, in addition to the standard `control` and `status` registers that are automatically included for all peripheral wrappers. The corresponding generated simulator base class might resemble the pseudocode in Listing 4.3.

Listing 4.3: A pseudocode implementation of a base simulator class with two input registers and one output register.

```
1  CLASS base_user_simulator:
2      PRIVATE:
3          /* other fields... */
4          DECLARE state AS simulator_state
5          DECLARE r1_i AS int16_t
6          DECLARE r2_i AS int16_t
7          DECLARE r3_o AS uint8_t
8
9      PUBLIC:
10         METHOD get_r1() RETURNS int16_t
11         METHOD get_r2() RETURNS int16_t
12         METHOD set_r3(r3: uint8_t) RETURNS void
13
14         VIRTUAL busy(in: simulation_data, out: simulation_data) RETURNS
       simulator_state
15
16         METHOD step(in: simulation_data) RETURNS simulation_data
17             DECLARE out AS simulation_data
18
19             CASE simulator_state:
```

```
20                 IDLE:
21                     IF control IS 1:
22                         SET simulator_state = BUSY
23                 BUSY:
24                     SET simulator_state = CALL busy(in)
25                 DONE:
26                     SET status = 1
27             END CASE
28
29             RETURN out
30         END METHOD
31 END CLASS
```

4.2.3 Communication Subsystem

The simulator's communication mechanism follows a structure similar to that of the DPI-C interface described earlier in Chapter 3. Fundamentally, it consists of a communication interface that defines four primary methods corresponding to distinct stages of interaction: `connect()` for initializing the connection, `send()` and `recv()` for data exchange, and `shutdown()` for gracefully closing the channel.

This interface is not intended to be implemented manually by users. Instead, the framework generates a base communication class that provides concrete implementations tailored to each simulator's specification. Importantly, the interface does not mandate a specific communication protocol. As long as the base class uses a compatible library to implement the interface, the framework remains agnostic to the underlying transport mechanism.

Currently, the framework provides native support for ZeroMQ as its communication backend. A corresponding implementation of the communication interface is already included in the API. This implementation manages protocol-specific details like context and socket objects.

The communication flow proceeds as expected: the simulator receives data from the peripheral using `recv()` and sends results back via `send()`. The `recv()` method takes no arguments and returns a `simulation_data` object. Internally, it deserializes the raw byte stream received over the socket into a structured `simulation_data` instance. Since the expected message format depends on the peripheral's input register layout, the base communication class declares an abstract method `recv_deserialize()` to perform this transformation.

Conversely, `send()` takes a `simulation_data` object as input and serializes its contents into a byte stream for transmission. This process also depends on the peripheral's output register definitions and is delegated to another abstract method, `send_serialize()`.

Critically, users are not expected to implement either of the abstract methods. Doing so would undermine the framework's core principle of reducing boilerplate and simplifying simulator development. Instead, the framework generates yet another subclass that extends the ZeroMQ-based communication class and provides concrete implementations of both serialization methods, based entirely on the simulator's peripheral specification.

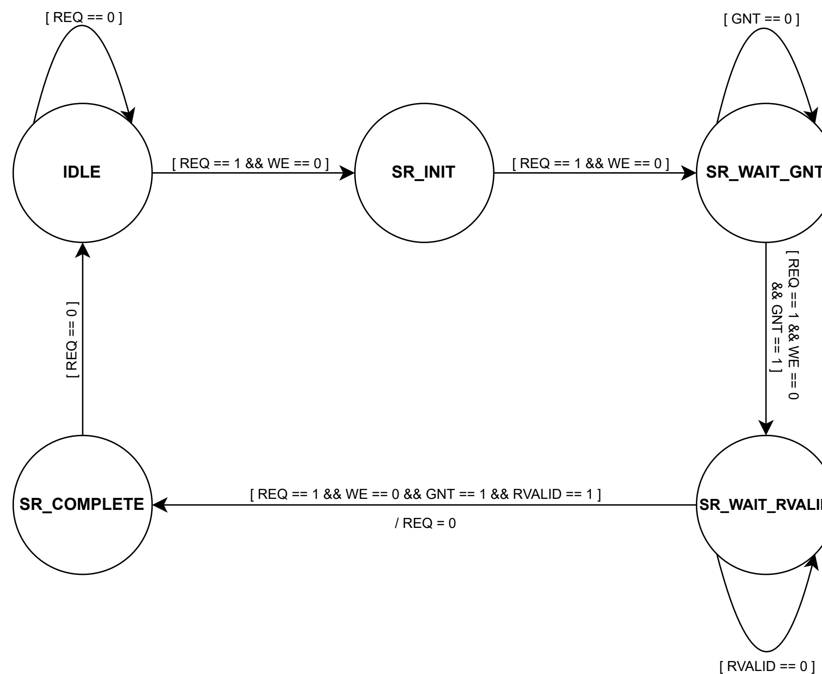


Figure 4.2: High-level block diagram of a single memory read operation, depicted in a Finite State-Machine (FSM).

4.2.4 Memory Transactions

A key feature of the high-level API is enabling simulators to directly initiate memory transactions with X-HEEP.

As described in Chapter 3, memory access within X-HEEP is managed via the OBI protocol. Each generated peripheral in the framework is equipped with two independent OBI channels: one for read operations and one for writes. The read channel uses the ports `read_channel_req_o` and `read_channel_resp_i`, whereas the write channel uses the ports `write_channel_req_o` and `write_channel_resp_i`.

The following sections present the core logic of these transactions and detail a deadlock prevention mechanism.

4.2.4.1 Memory Transaction Finite State Machines

The FSM for a single memory read operation is shown in Figure 4.2. The bus begins in the `IDLE` state, indicating that no transaction is currently in progress. When the simulator issues a read request, the FSM transitions to `SR_INIT`, where the read request is initialized by asserting the `req` signal and ensuring that `we` is low.

Next, the FSM enters `SR_WAIT_GNT`, where it waits for the memory system to grant the request. Once granted, it transitions to `SR_WAIT_RVALID`, awaiting the arrival of valid read data. When

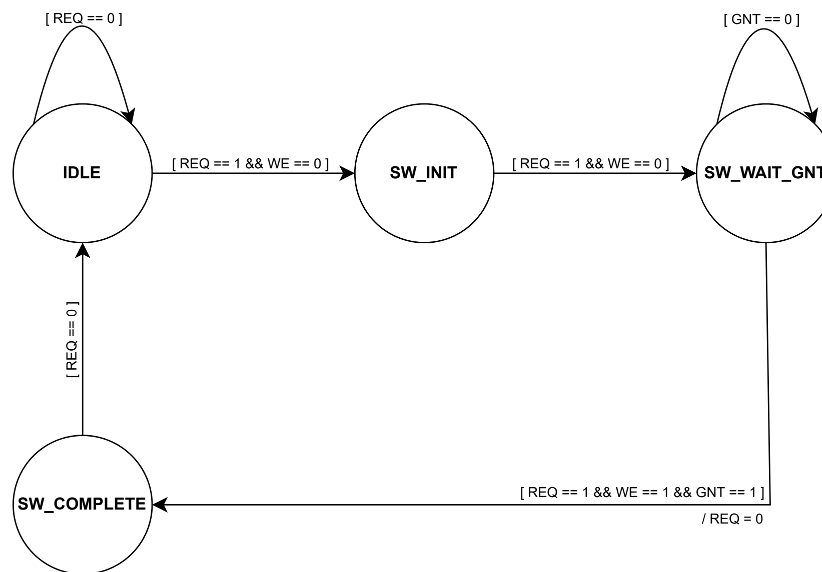


Figure 4.3: High-level block diagram of a single memory write operation, depicted in a Finite State-Machine (FSM).

`rvalid` is asserted, the FSM reaches `SR_COMPLETE`, allowing the simulator to read from the `rdata` field. Finally, the FSM returns to `IDLE`, ready for the next transaction.

The FSM for a single write operation (Figure 4.3) is structurally similar. It begins in `IDLE` and transitions to `SW_INIT` upon a write request, where both `req` and `we` are asserted. The FSM then enters `SW_WAIT_GNT`, waiting for the memory system to acknowledge the request. Once the grant is issued, it proceeds to `SW_COMPLETE` and, upon deasserting `req`, returns to `IDLE`.

The FSMs described above handle only single-word memory accesses. However, many real-world scenarios, like reading or writing buffers, require processing multiple words in sequence. These operations are known as burst transactions. While the OBI protocol does not natively support bursts, we can emulate them in high-level languages using word-by-word sequencing.

Burst operations are built by iterating over single-word transactions, each adjusted to a different memory address. As such, their FSMs include an additional state to manage per-word processing, assuming the base address and total word count are known in advance.

In the burst read FSM shown in Figure 4.4, the key difference is the addition of a `BR_WORD_REQ` state, which handles individual word requests. After each request, the FSM follows the same flow as a single read. But rather than finishing immediately once `rvalid` is asserted, it loops back to `BR_WORD_REQ`, increments the address by 4 bytes (the size of one word), and decrements the word counter. Once all words have been retrieved, the FSM transitions to `BR_COMPLETE`, signaling the end of the burst.

Burst write transitions (Figure 4.5) follow the same principle. Each word is issued sequentially, with the FSM stepping through the request and grant phases. The FSM loops until the entire burst has been written, at which point it completes and returns to `IDLE`.

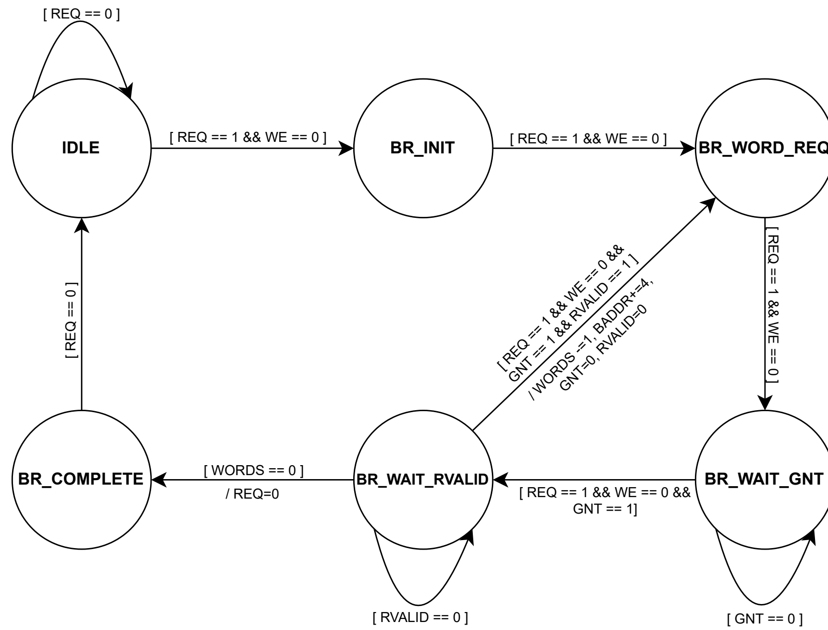


Figure 4.4: High-level block diagram of a burst memory read operation, depicted in a Finite State-Machine (FSM).

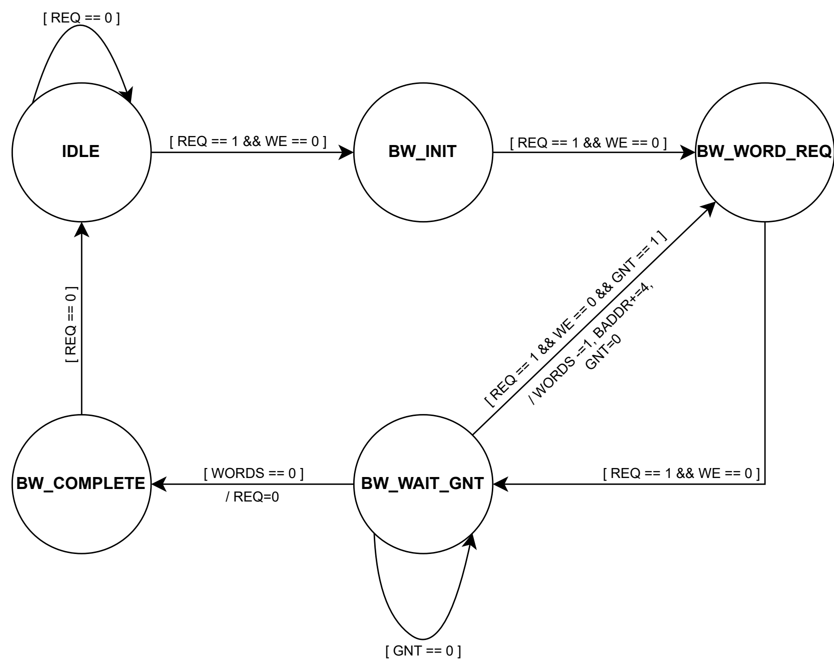


Figure 4.5: High-level block diagram of a burst memory write operation, depicted in a Finite State-Machine (FSM).

4.2.5 Issuing Memory Reads and Writes

The simulator initiates memory transactions by directly controlling the ports of the read and write OBI channels. To facilitate this, each generated peripheral includes a set of output registers that map one-to-one with the fields of these channels. These output registers are writable by the simulator and are part of the `communication_recv` argument list from the peripheral module. When the simulator issues a transaction via `send()`, the relevant data must be included in the `simulation_data` object.

Similarly, the simulator needs to observe responses from the hardware. For this purpose, each peripheral also includes a corresponding set of input registers. These registers capture the result and state of memory transactions and are made available to the simulator by the `recv()` method. They are declared as part of the `communication_send` argument list within the SV module.

Low-level protocol details are abstracted away from the user through a `memory_controller` class, which encapsulates the memory transaction behavior. A pseudocode representation of this controller is shown in Listing 4.4.

Listing 4.4: A pseudocode implementation of the memory controller class.

```
1  ENUM memory_state:
2      IDLE, SR_INIT, SR_WAIT_GNT, /* ... */
3  END ENUM
4
5  ENUM memory_operation:
6      NOP, SINGLE_READ, SINGLE_WRITE, /* ... */
7  END ENUM
8
9  CLASS memory_controller:
10     PRIVATE:
11         DECLARE state AS memory_state
12         DECLARE op AS memory_operation
13         /* ... */
14
15     PUBLIC:
16         METHOD single_read(addr: uint32_t, in: simulation_data, out:
       simulation_data) RETURNS OPTIONAL<int32_t>
17
18         METHOD burst_read(base_addr: uint32_t, words: uint16_t, in:
       simulation_data, out: simulation_data) RETURNS OPTIONAL<[int32_t]>
19
20         METHOD single_write(addr: uint32_t, data: int32_t, in: simulation_data,
       out: simulation_data) RETURNS bool
21
22         METHOD burst_write(addr: uint32_t, data: [int32_t], in: simulation_data,
       out: simulation_data) RETURNS bool
23 END CLASS
```

Because memory response latency is unknown, and communication occurs on every clock cycle, each method in the `memory_controller` class must be able to signal whether the transaction has been completed. For read operations, the return type is wrapped in an `Optional<T>`. If the value is present, the request is complete. For writes, the method returns a boolean flag.

Internally, these methods implement the behavior defined by their corresponding memory transaction FSMs.

During a typical read operation, the controller prepares the `out` object of the argument list with the required fields that trigger a read. These fields correspond to the peripheral's output registers:

- `{"read_channel_req_type": 1}` signals that a memory read request is being initiated.
- `{"read_channel_req_addr": addr}` specifies the address from which data should be read, with `addr` provided as a method argument.

To monitor the transaction's progress, the controller checks values in the `in` object, which contains the peripheral's response registers:

- `{"read_req_gnt": gnt}` indicates whether the request was granted, sourced from `read_channel_resp_i.gnt`.
- `{"read_req_rvalid": rvalid}` is asserted high if the read data is valid, sourced from `read_channel_resp_i.rvalid`.
- `{"read_req_rdata": rdata}` contains the actual data read from memory, sourced from `read_channel_resp_i.rdata`.

These two `simulation_data` objects, `in` and `out`, are the inputs from the peripheral, retrieved from calling `recv()`, and the outputs to be sent to the peripheral, returning value of the `step` method, respectively.

The assignment of these values within the communication interface is illustrated in Listing 4.5. Note that every peripheral generated by our framework integrates this SV code block in order to communicate with its corresponding simulator on every clock cycle.

Listing 4.5: Assignment of OBI port fields for memory transactions.

```
always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        /* ... */
    end else begin
        communication_send(/* other input registers... */,
            read_channel_resp_i.gnt,
            read_channel_resp_i.rvalid,
            read_channel_resp_i.rdata, /* ... */
        );

```

```
        communication_recv(/* other output registers */,
            read_channel_req_type,
            read_channel_req_addr,
            /* ... */
        );

        /* ... */
    end
end
```

Finally, the SV module binds these register values to the actual OBI signal fields, as shown in Listing 4.6.

Listing 4.6: OBI read channel field assignments.

```
assign read_channel_req_o = '{
    req: read_channel_req_type | read_req_active,
    we: 1'b0,
    be: 4'b1111,
    addr: read_channel_req_addr,
    wdata: 32'b0
};
```

For example, imagine a simulator that needs to read a word from memory, perform some computation on it, and write the result back to a different address. This process begins by calling the `single_read` method, which returns an optional value. Since memory transactions take a few cycles to complete, the value may not be immediately available. Therefore, the user must first check whether the optional contains a result. If the value is present, the computation can proceed, and the result can then be written to memory using the `single_write` method. To confirm that the write was successfully issued, the user can check whether this last method returns `true`.

4.2.5.1 Deadlock Prevention Mechanism

While mapping output registers directly to their corresponding memory port fields may appear sufficient, doing so without additional safeguards can lead to race conditions and potential deadlocks. This is because memory transactions in the OBI protocol are not instantaneous. They rely on the target memory to issue a valid grant signal before progressing. If the simulator issues a new request while a previous one is still pending, multiple overlapping transactions could occur, which could result in undefined behavior.

A simple handshake mechanism is sufficient to prevent these types of issues. It introduces two internal states, one for the read channel and another for the write channel. These flags are asserted whenever a memory request is in progress but has not yet received a grant. They ensure that an issued request remains active and valid until it is acknowledged by the system, and, more importantly, block the issuance of new requests until the current transaction has been completed.

4.3 Simulator Communication Loop

The final component included in the generated simulator project is the communication loop, which controls the interaction between the simulator and its associated SV peripheral using our API. This loop serves as the driver for the simulation and is responsible for managing each iteration of the communication cycle.

The process begins by instantiating the user-defined simulator subclass and establishing a connection to the communication channel. From there, the simulator enters a loop, where it collects the current state of the peripheral by calling `recv()`, followed by `step()`, passing in the received data, and performing domain-specific computation. The computed results are passed to `send()`, which transmits the updated output register values back to the peripheral.

This communication loop executes continuously for the duration of the simulation. A high-level pseudocode implementation is shown in Listing 4.7.

Listing 4.7: A pseudocode implementation of a simulator's main communication loop.

```
FUNCTION main:
    DECLARE sim AS simulator
    DECLARE comms AS simulator_comms

    CALL comms.connect()

    WHILE TRUE DO:
        DECLARE inputs AS simulation_data = comms.recv()

        IF inputs IS NULL:
            BREAK

        DECLARE outputs AS simulation_data = sim.step(inputs)

        CALL comms.send(outputs)
    END WHILE

    CALL comms.shutdown()
END main
```

In this example, note that `simulator` is a subclass of the base simulator, automatically generated by the framework. This class also implements the `busy` method, which is called internally by `step`.

Once the simulator stops receiving requests and the return value from `recv` is null, the simulator exits the communication loop, calls `shutdown`, and exits gracefully.

Chapter 5

Automated Simulator Integration

This chapter explains how our framework automatically integrates simulators into X-HEEP. First, we dive into the challenges associated with manually expanding a hardware platform. Next, we present the configuration file, which drives our code generator, and provide a comprehensive list of all generated artifacts.

5.1 Expanding a Hardware Platform

Integrating new functionality, like custom peripherals or hardware accelerators, into a complex SoC platform is often a challenging and time-consuming process. The complexity of this task extends far beyond the design of the hardware block itself. Developers must also navigate a series of manual, error-prone steps across multiple domains, including hardware description, system-level integration, and simulation setup. This fragmented and intricate workflow can significantly slow down development.

Unlike software development, hardware design requires a completely different approach. Designers must reason about concurrency, signal timing, and hardware-specific details like logic gates and wiring. Even behaviors that are trivial to implement in software may require hundreds of lines of SV code to express accurately. This low-level nature makes hardware development particularly susceptible to errors. Debugging such errors is often tedious and time-intensive. A single overlooked signal or misaligned timing condition can cause the entire simulation to malfunction, requiring a full re-execution or a more costly re-generation, in the case of using technology such as Verilator.

Our co-simulation framework is designed to mitigate many of these difficulties. Instead of requiring users to directly modify the SoC or write verbose low-level code, the framework allows developers to declaratively specify their simulation setup and peripheral interface through a simple JSON configuration file. Users define the desired peripherals and simulation parameters, and the framework automatically handles all necessary integration steps for X-HEEP. This includes generating customized peripheral modules, updating the system testbench, and preparing a ready-to-extend simulator project. As a result, developers can concentrate entirely on implementing

domain-specific logic, reducing the time and expertise needed to expand and test new functionality on hardware platforms.

5.2 Code Generator Implementation

This section details the implementation of the code generator component of our co-simulation framework.

5.2.1 Architecture Overview

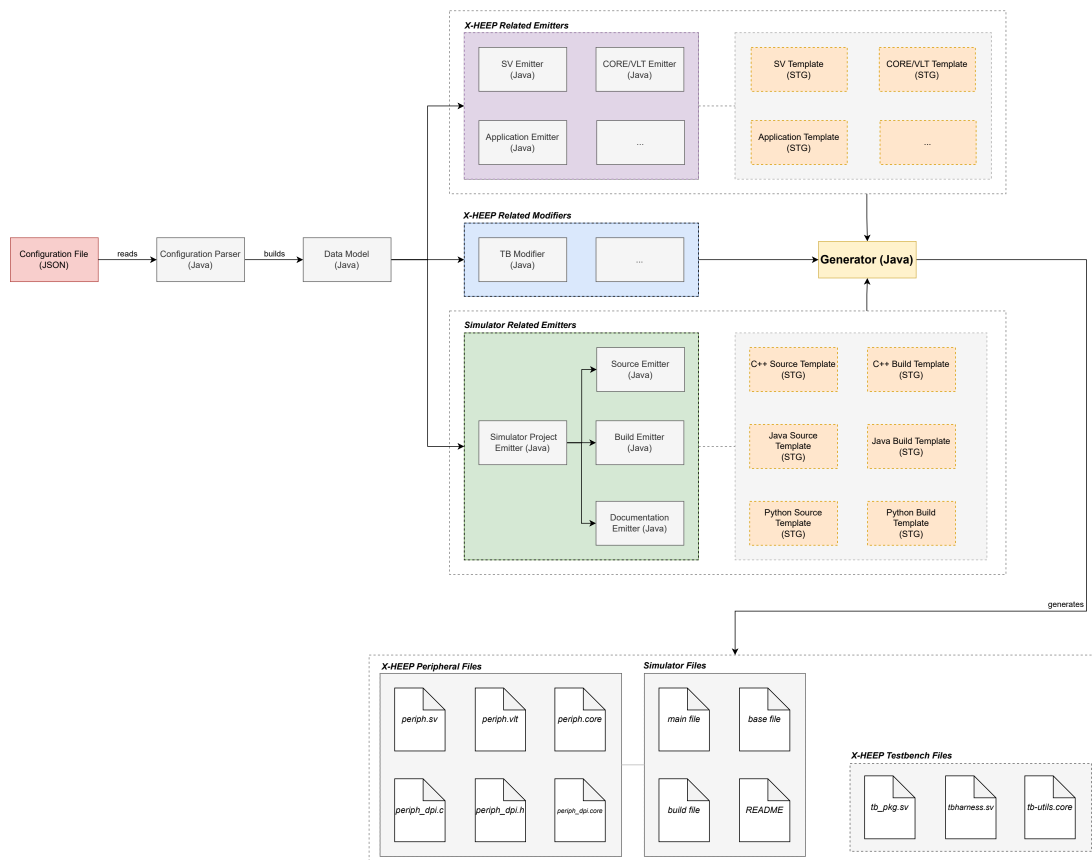


Figure 5.1: High-level block diagram of the code generator's architecture.

Figure 5.1 provides an overview of the architecture behind the code generation pipeline. The process begins with a user-defined JSON configuration file, which serves as the central specification for the simulation project.

The first stage in the pipeline is the configuration parser. This module reads the JSON input and constructs a structured data model composed of Java classes that directly mirror the contents of the configuration file. Each class corresponds to a distinct component of the simulation setup.

Once the data model has been constructed, it serves as the foundation for the generator's core functionality: emitting new files and modifying existing ones. These tasks are handled by two

primary component types, emitters and modifiers. Emitters generate source files from scratch, like SV modules or C header files. Modifiers, on the other hand, target existing source files, applying automated edits to insert or update content.

To avoid the drawbacks of manual string concatenation, which is both verbose and prone to errors, most emitters are paired with a dedicated template file. Templates define the static structure of the output code while allowing dynamic insertion of user-specific content through template parameters.

The `Generator` class acts as the orchestrator. It manages all emitters and modifiers, invoking them based on the parsed configuration model. When executed, the generator produces a comprehensive set of artifacts needed to integrate the specified simulators into X-HEEP. The outputs include custom SV modules, configuration files, updated testbench components, and base simulator projects that incorporate the API described in Chapter 4, tailored to the user's selected programming environment. Additionally, the generator also provides a method named `undo()`, which, as the name suggests, will undo the emission and modification of all affected files, returning the system to the same state as it was before generation.

5.2.2 Configuration File Structure

The structure of the configuration file is illustrated in Listing 5.1.

Listing 5.1: JSON configuration file schema.

```
1  {
2    "project_name": string,
3    "paths": {
4      "xheep_dir": string,
5      "output_dir": string
6    },
7    "peripheral_definitions": [
8      {
9        "identifier": string,
10       "description": string,
11       "class_name": string,
12       "target_language": string,
13       "hardware_interface": {
14         "memory_base_address": int,
15         "memory_size_bytes": int,
16         "registers": [
17           {
18             "name": string,
19             "direction_to_simulator": string,
20             "type": string,
21             "signed": bool,
22             "width": int,
23             "default_value": int,
```

```
24              "description": string
25            },
26            /* ... */
27          ]
28        },
29        "communication": {
30          "host_address": string,
31          "simulator_address": string,
32          "send_timeout_ms": int,
33          "recv_timeout_ms": int,
34          "retries": int
35        }
36      },
37      /* ... */
38    ]
39  }
```

The configuration begins with a top-level `project_name` field, which defines the name of the host application running on the X-HEEP CPU.

This is followed by a `paths` object, where two directories must be specified:

- `xheep_dir`: the root path of the local X-HEEP directory.
- `output_dir`: the location where the high-level simulator projects will be written to.

The primary section of the file is the `peripheral_definitions` list. Each entry in this list defines a single memory-mapped peripheral and includes the following fields:

- `identifier`: a unique name used to derive module and file names across the generated codebase.
- `description`: an optional text field for documentation purposes.
- `class_name`: the name of the simulator class associated with this peripheral, which will be used as the basis for generating the base simulator implementation.
- `target_language`: the programming language in which the simulator will be implemented. Supported values include C++, Java, and Python.

Each peripheral entry must also specify a `hardware_interface` object, which describes internal peripheral properties:

- `memory_base_address`: base address of the peripheral in the memory map.
- `memory_size_bytes`: the size of the peripheral's memory region.
- `registers`: a list of registers that make up the peripheral. Each register must include:

- `name`: the variable name of the register.
- `direction_to_simulator`: either "input" or "output", indicating the flow of data relative to the simulator.
- `type`: the SV type of the register, either `bit` or `logic`.
- `signed`: a boolean flag specifying whether the register is signed. This flag is essentially used to keep the semantic value of types across languages. If the target language does not support unsigned types, code generation does not change, and signed types are used instead.
- `width`: the bit width of the register.
- `default_value`: the initial value assigned to the register.
- `description`: a short optional text field documenting the purpose of the register.

Lastly, each peripheral entry contains a `communication` block that specifies runtime settings for the communication channel between the simulator and the host.

5.2.3 Template Infrastructure

Parameterized code generation is reliant on a template infrastructure built around the StringTemplate[1] engine.

At the root of the infrastructure is the `Emitter` interface, which defines the basic operations that any code-generating component must support (see Listing 5.2).

Listing 5.2: The `Emitter` interface from the code generator.

```
public interface Emitter {
    String emitToString();

    void emitToFile(String filepath);
}
```

The interface defines two methods: `emitToString()`, which returns the generated code as a string, and `emitToFile()`, which writes the generated output to the specified file.

A template file (that is, any file with the `.stg` extension) defines a *template group*. A template group is made up of one or more template instances, which are identified by a unique name. Hence, a single template instance is fetched based on the path to the corresponding template group and name. The `TemplateConfig` class is a simple container for this metadata. It stores the path to the template group and the name of the template instance to be used.

The same template group may be loaded multiple times for different instances. In order to avoid repeated I/O operations, the infrastructure uses a `TemplateManager` class that loads and caches template groups using a simple hash map.

[1] https://www.stringtemplate.org/

Emitters that rely on templates will extend the `TemplateEmitter` base class, which already handles common logic like loading the template, configuring and rendering it, and writing it to a file.

StringTemplate objects allow users to insert data into their output using named placeholders. Consider the simple example in Listing 5.3:

Listing 5.3: A simple example using string templates.

```
ST hello = new ST("Hello, <name>!");
hello.add("name", "World");

String output = hello.render();
```

In this case, the template string contains a placeholder `<name>`, which is later assigned the value `"World"` using the `add()` method. When `render()` is called, the final output becomes `"Hello, World"`.

To make it easier to work with structured data, especially when generating more complex outputs, the generator provides a `DataTemplateEmitter<T>` class (see Listing 5.4), which extends `TemplateEmitter` and introduces a typed `data` field. This will hold the specific input model required by the template.

Listing 5.4: The `DataTemplateEmitter<T>` abstract class.

```
public abstract class DataTemplateEmitter<T> extends TemplateEmitter {
    protected final T data;

    public DataTemplateEmitter(TemplateConfig config, T data) {
        super(config);

        this.data = data;
    }
}
```

This design is particularly helpful because different emitters often require different kinds of data. For example, a SV emitter and a host application emitter operate on distinct data models. By using generics, it's easier to bind the right kind of data to the right template in a type-safe and reusable way.

5.2.3.1 Template Data

Emitters that rely on structured input data typically follow a reusable pattern. To make this process easier to understand, consider the example shown in Listing 5.5, which defines a portion of a StringTemplate group file used to generate SV modules.

Listing 5.5: A simplified template group used to generate a SV module.

```
verilog_module(module) ::= <<
```

```
module <module.lowerCaseName> #(
    parameter type reg_req_t  = logic,
    parameter type reg_rsp_t  = logic,
    parameter type obi_req_t  = logic,
    parameter type obi_resp_t = logic
) (
    input logic clk_i,
    input logic rst_ni,

    input reg_req_t reg_req_i,
    output reg_rsp_t reg_rsp_o,

    output obi_req_t read_channel_req_o,
    input obi_resp_t read_channel_resp_i,

    output obi_req_t write_channel_req_o,
    input obi_resp_t write_channel_resp_i
);
>>
```

This template defines a `verilog_module` rule with a single placeholder: `module`. Inside the template, attributes of this placeholder, such as `lowerCaseName`, are used to populate the final output. These attributes are supplied through a structured Java object.

Listing 5.6 provides the implementation of the `VerilogEmitter` class, which makes use of the template file provided above.

Listing 5.6: Implementation of the `VerilogEmitter` class.

```java
public class VerilogEmitter extends DataTemplateEmitter<VerilogTemplateData> {
    public VerilogEmitter(SimulatorDefinitionModel simulator) {
        super(
            new TemplateConfig("path/to_template", "verilog_module"),
            new VerilogTemplateData(simulator)
        );
    }

    @Override
    protected void configureTemplate(ST template) {
        template.add("module", data);
    }
}
```

In the template file, the placeholder `module.lowerCaseName` refers to the `getLowerCaseName()` method provided by the `VerilogTemplateData` class, shown in Listing 5.7.

Listing 5.7: An excerpt from `VerilogTemplateData`, showing how placeholders are backed by method calls.

```java
public class VerilogTemplateData {
    private final PeripheralDefinitionModel peripheral;

    public VerilogTemplateData(PeripheralDefinitionModel Peripheral) {
        this.peripheral = peripheral;
    }

    public String getLowerCaseName() {
        return this.peripheral.identifier().toLowerCase();
    }

    /* (...) */
}
```

StringTemplate allows placeholders like `module.lowerCaseName` to access public getter methods of the provided object, which enables the use of structured data within templates.

This is a demonstration of how most of the emitters within our co-simulation framework are implemented.

5.2.3.2 X-HEEP Artifacts

For each peripheral definition included in the configuration file, the generator produces a number of artifacts required to integrate the peripheral with the X-HEEP platform. These artifacts include both newly generated files and modifications to existing components of the system.

First, `VerilogEmitter` generates the simulator's peripheral wrapper, which contains the specified input/output ports and internal registers. The wrapper also includes logic for memory-mapped register access via MMIO, DPI-C function invocations, a communication loop to interface with the simulator, and deadlock-prevention mechanisms using handshake protocols.

The `DpiEmitter` is responsible for generating the DPI-C interface required to bridge the peripheral with the simulator. This includes a header file with function declarations and a source file with the corresponding implementations.

`ConfigEmitter` will produce the necessary `.core` and `.vlt` metadata files required to integrate the peripheral into X-HEEP's build system.

Many files are also updated. For instance, `TestHarnessPkgModifier` will update a testbench file to add a new peripheral entry to the peripheral map. The simulation top module file is also updated by `TestHarnessModifier` to instantiate the peripheral and wire it into the simulation environment. Lastly, `CoreFileModifier` will update `x-heep-tb-utils.core` with additional source and include path entries pointing to the peripheral's files.

Table 5.1: Summary of generated eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) artifacts and their output locations.

Class	**Location of Modified/Emitted File**
`VerilogEmitter`	`hw/ip_examples/<id>/<id>.sv`
`DpiEmitter`	`hw/ip_examples/<id>/dpi/<id>_dpi.h,c`
`ConfigEmitter`	`hw/ip_examples/<id>/`
`TestHarnessPkgModifier`	`tb/testharness_pkg.sv`
`TestHarnessModifier`	`tb/testharness.sv`
`CoreFileModifier`	`x-heep-tb-utils.core`

5.2.3.3 Simulator Artifacts

For every peripheral defined in the configuration file, the generator also produces a corresponding simulator project in the specified target programming language. This project includes all the necessary source code, build scripts, and documentation required to run the simulator independently.

First, `SourceEmitter` will generate the main source file that implements the communication loop with the hardware, handling the reception and transmission of data to and from the peripheral.

`HeaderEmitter` will generate a header file (or equivalent, depending on the language) that declares a base simulator class. The source file produced by `SourceEmitter` imports this header.

Next, `BuildEmitter` generates the build configuration required to compile the project. The format of this file depends on the chosen target language: CMake for C++, Gradle for Java, and a `requirements.txt` file for Python.

Lastly, `DocumentationEmitter` generates a `README.md` file that provides instructions on how to build the project.

Each simulator project is placed within its own dedicated folder inside the output directory specified in the configuration file. Simulator code which implements functionality at a behavioural level can then be integrated with this generated code, via the previously described API which exchanges data with the verilated SoC and hosts the main execution loop.

Chapter 6

Framework Evaluation

6.1 Functionality Assessment

While setting up all the necessary components for the framework can initially seem like a lengthy process, we believe the framework itself to be very easy to use once configured.

The most time-consuming part of the setup is preparing the X-HEEP platform. To simplify this, users have the option of using a pre-built Docker image that includes all required dependencies. In this case, the setup consists of installing Docker and pulling the image. Alternatively, the users can choose to set up everything manually on their host machine. This manual setup includes installing development libraries, configuring a Python environment (using Miniconda or a virtual environment), and installing the RISC-V compiler, Verilator, and Verible. The full setup guide is available in X-HEEP's documentation[1].

The effort required to set up the framework also depends on how the user plans to interact with it. Users must install the wrapper APIs, which use the C implementation of ZeroMQ and MessagePack. Additionally, the high-level APIs provided for simulator development also rely on these libraries, so users must install language-specific bindings for both ZeroMQ and MessagePack.

The code generator is packaged as a Gradle project. This makes dependency management straightforward, as all necessary libraries can be installed using a single command.

Once setup is complete, using the framework is simple. Developers only need to write a JSON configuration file, implement the domain-specific logic for their simulator, and write the desired host program in X-HEEP. Additionally, every generated simulator project also includes a `README` file with clear build and run instructions, making it easy to get started quickly.

6.2 Experimental Setup

6.2.1 Hardware and Software Environment

The hardware and software environments are presented in Table 6.1 and Table 6.2, respectively.

[1] https://x-heep.readthedocs.io/en/latest/

Table 6.1: Experimental setup hardware specifications.

Hardware	Specification
CPU	Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
GPU	NVIDIA GeForce RTX 2060
RAM	2x8GiB DIMM DDR4 Synchronous Unbuffered 2667 MHz
Storage	ATA SDM8V512
Motherboard	H310M PRO-VDH PLUS (MS-7C09)

X-HEEP's microcontroller was generated using default settings except for the number of memory banks, which had to be increased to 12 due to the large buffer sizes used in upcoming experiments.

6.2.2 Case Study: X-HEEP's Simple Accelerator

The conducted experiments focus on an already existing hardware accelerator from the X-HEEP Intellectual Property (IP) example suite, called `simple_accelerator`.

This accelerator applies a threshold filter to a buffer of data and writes the filtered results to another memory location, making use of OBI transactions. Hence, the accelerator takes the following inputs:

- The starting address of the input data buffer in memory.
- The starting address where the filtered output will be stored.
- The threshold value used by the filter.
- The number of data words to process.

The threshold filter algorithm is straightforward. Given an input array A_i of size N and a threshold value T, the filtered array A_f is computed as:

$$A_f: \quad A_f[i] = \max(A_i[i], T), \quad \text{for } i = 0, 1, \ldots, N-1$$

The original IP example is written entirely in SV and resides in X-HEEP's peripheral subsystem as a memory-mapped peripheral. A host application controls and invokes this accelerator.

Table 6.2: Experimental setup software specifications.

Software	Version
GCC/G++	11.4.0
Python	3.8.18
Conda	23.11.0
RISC-V Toolchain	Cloned from branch "2022.01.17"
Verilator	4.210 2021-07-07 rev v4.210
Verible	v0.0-1824-ga3b5bedf

6.3 Experiment 1: Replicating the Simple Accelerator

This section presents a detailed analysis of the first experiment, which replicates the `simple_accelerator` example from X-HEEP in a C++ project.

6.3.1 Baseline: Pure SystemVerilog Accelerator

Before presenting the replicated accelerator version developed with our framework, we must first explain the details of the original SV IP implementation. This is the baseline used for comparison and highlights the complexities that our framework aims to abstract.

The original peripheral, named `simple_accelerator`, is entirely implemented in SV. This includes not only the core threshold filter algorithm but also all necessary control logic, including a dedicated FSMs for managing OBI memory transactions (both reads and writes), and other internal control structures. The module is structurally complex and serves as a development reference for the framework's design.

The `simple_accelerator` module defines an extensive set of memory-mapped input and output registers, though only a few are directly manipulated by the host CPU for the threshold filtering operation:

- `address_read_q`: a 32-bit register that represents the starting address of the input data buffer in memory.
- `address_write_q`: a 32-bit register that represents the starting address where the filtered output will be stored.
- `threshold_q`: a 32-bit register that holds the threshold value used by the filter.
- `size_q`: a 10-bit register that holds the number of data words to process.

Similarly to the peripherals generated by our framework, the original module includes standard ports that expose a compliant MMIO interface for host communication. Furthermore, it includes independent OBI channels for issuing memory read and write requests.

Additionally, the module defines a comprehensive list of local registers with varying bit widths. These internal registers are primarily used to manage the multiple control structures and state elements employed throughout the module's definition.

Memory transactions are handled within two FSMs, one for issuing read requests and another for writes.

Recall that the OBI protocol does not natively support burst transactions, which means each word must be fetched from memory individually. For every read request, the accelerator will attempt to generate a corresponding write request. It applies the filter to the retrieved word, computes the destination address for the write, and then stores the resulting write request in an internal queue.

Reads and writes can occur concurrently within the same clock cycle. The write logic monitors the queue, and if it contains entries, it begins processing the next write request. Once a write is acknowledged, the corresponding entry is popped from the queue.

To avoid overflowing the queue, the read FSM only pushes new entries when the queue is neither full nor *almost* full. In this scenario, the queue is considered *almost* full when its current size reaches `FIFO_DEPTH - 1`, where `FIFO_DEPTH` is a module parameter that defines the queue's maximum capacity.

6.3.2 Framework-enabled: Co-Simulated Accelerator

Replicating the `simple_accelerator` example using our framework is a straightforward process designed to minimize boilerplate and manual integration effort.

The first step involves writing a configuration file in JSON, conforming to the defined schema. For this accelerator, it is only necessary to specify the registers involved in the threshold filter algorithm, namely the source and destination addresses, the threshold value, and the data size. The peripheral definitions block, containing this information, consists of a single entry and would look similar to Listing 6.1.

Listing 6.1: Peripheral definitions block of the replicated `simple_accelerator` example.

```
{
  "identifier": "replicated_simple_accelerator",
  "description": "A simple accelerator",
  "class_name": "replicated_simple_accelerator",
  "target_language": "cpp",
  "hardware_interface": {
    "memory_base_address": 6000,
    "memory_size_bytes": 100,
    "registers": [
      {
        "name": "src_address",
        "direction_to_simulator": "input",
        "type": "logic",
        "signed": false,
        "width": 32,
        "default_value": 0
      },
      {
        "name": "dst_address",
        "direction_to_simulator": "input",
        "type": "logic",
        "signed": false,
        "width": 32,
        "default_value": 0
      },
```

```
26        {
27          "name": "threshold",
28          "direction_to_simulator": "input",
29          "type": "logic",
30          "signed": false,
31          "width": 16,
32          "default_value": 0
33        },
34        {
35          "name": "data_size",
36          "direction_to_simulator": "input",
37          "type": "logic",
38          "signed": false,
39          "width": 16,
40          "default_value": 0
41        }
42      ]
43    },
44  }
```

In this experiment, we chose C++ as the target language for the simulator. The same setup could be applied to other supported languages. Once the configuration file is ready, it is processed by the framework's code generation pipeline. The JSON description is parsed into a structured data model, which is then passed through the emitter infrastructure. This stage is responsible for generating all necessary artifacts, including a simplified peripheral implementation for X-HEEP and a base simulator project.

The key component from the developer's perspective is the simulator class generated by the framework. This class extends a base class and overrides the `busy()` method, which is the entry point for inserting the domain-specific computation logic required by the accelerator. Initially, this class would look similar to the implementation shown in Listing 6.2.

Listing 6.2: The replicated simple accelerator class, in C++, as generated by our co-simulation framework.

```
1  class replicated_simple_accelerator : public base_replicated_simple_accelerator
2  {
3  private:
4  public:
5    simulator_state
6    busy (
7      const socsim::simulation_data &in,
8      socsim::simulation_data &out
9    ) override {
10     /// Add your domain-specific logic here.
11     return simulator_state::BUSY;
12   }
```

```
13 };
```

The C++ implementation aims to closely mirror the behavior of the original accelerator. To do this, it handles memory transactions in a similar fashion, by using a fixed-size queue to manage write operations. A proposed version of this logic is shown in Listing 6.3.

Listing 6.3: A C++ implementation of the simple accelerator's domain-specific logic.

```
1  class reimagined_simple_accelerator
2      : public base_reimagined_simple_accelerator {
3  private:
4    static constexpr uint16_t WORD_SIZE = 4;
5    static constexpr uint16_t FIFO_DEPTH = 4;
6    static constexpr uint16_t FIFO_ALM_FULL = FIFO_DEPTH - 1;
7
8    std::queue<write_req> fifo;
9
10   uint32_t rd_count = 0, wr_count = 0;
11
12   bool fifo_full() { return fifo.size() == FIFO_DEPTH; }
13   bool fifo_alm_full() { return fifo.size() == FIFO_ALM_FULL; }
14 public:
15   simulator_state busy(const socsim::simulation_data &in,
16                        socsim::simulation_data &out) override {
17
18     /* process write requests first */
19     if (!fifo.empty()) {
20       auto &[addr, data] = fifo.front();
21
22       if (this->controller.single_write(addr, data, in, out)) {
23         fifo.pop();
24
25         wr_count++;
26       }
27     }
28
29     /* FIFO is almost full, do not issue more read requests */
30     if (fifo_full() || fifo_alm_full())
31       return simulator_state::BUSY;
32
33     /* start issuing reads */
34     if (rd_count < this->get_data_size()) {
35       uint32_t src_addr = this->get_src_address() + rd_count * WORD_SIZE;
36       uint32_t dst_addr = this->get_dst_address() + wr_count * WORD_SIZE;
37
38       std::optional<int32_t> v_opt = this->controller.single_read(src_addr, in,
    out);
```

```
39        if (v_opt.has_value()) {
40          int32_t v_actual = v_opt.value();
41          int32_t v_filtered = v_actual > this->get_threshold() ? v_actual : this->
      get_threshold();
42
43          fifo.push({dst_addr, v_filtered});
44
45          rd_count++;
46        }
47      }
48
49      /* everything has been read and written */
50      if (fifo.empty() && rd_count == this->get_data_size() && wr_count == this->
      get_data_size())
51        return simulator_state::DONE;
52
53      return simulator_state::BUSY;
54    }
55  };
```

Internally, the class maintains a queue to store pending write requests, along with counters to track the number of completed reads and writes. It also provides helper methods to check whether the queue is full or almost full.

Within the `busy()` method, we first inspect if the queue contains any write requests. If so, issue a write request on the element at the front of the queue, and upon completion, the request is popped from the queue, and the write counter is incremented.

Before issuing new read requests, we check whether the queue is either full or almost full. If so, we return `BUSY`, deferring further reads to avoid overflowing the queue.

If the queue has space and the number of issued reads is still below the expected data size, the method computes the source and destination addresses for the next transactions, followed by a read request. Once the read completes, it applies a threshold filter: if the read value is less than the threshold, the threshold value is used instead. A corresponding write request is then enqueued, and the read counter is incremented.

When all expected reads and writes are completed and the queue is empty, the method returns `DONE`, indicating that the accelerator has finished processing. Otherwise, it returns `BUSY`, signaling that further work remains and that the simulation should continue. Each return, whether `BUSY` or eventually `DONE`, corresponds to one simulation cycle.

The surrounding API is designed to encourage type safety and clarity. Inputs and outputs are explicitly declared and typed, reducing ambiguity about data direction or format. Additionally, memory accesses are abstracted through a controller interface. Developers do not manipulate memory manually, but instead use the dedicated read and write methods that encapsulate this complexity.

6.3.3 Host Program

The original host program provided with the `simple_accelerator` example and that calls the accelerator is well-suited for demonstration purposes but less ideal for simulation-based testing. This is primarily due to its extensive use of I/O operations, which can significantly degrade simulation performance.

In the original version, I/O calls serve primarily for debugging and error reporting, for example, by printing status messages. A typical verification step would involve inspecting `uart0.log` to check for the presence of a success message. While the approach is practical for manual debugging, it is inefficient.

In order to reduce simulation overhead as much as possible, we modified the host program to eliminate I/O entirely. Instead of relying on printed messages, the program now uses the error count as its exit code. After execution, it compares the contents of the output buffer against expected values, incrementing an error counter for each mismatch, just like before. The final value of this counter, however, determines the program's exit status.

If the accelerator performs correctly and all results match expectations, the program exists with a status code of 0. A nonzero exit code indicates the number of discrepancies found. This mechanism was employed to verify the functional correctness of both accelerators for all iterations of the experiments.

6.3.4 Performance Benchmarks

To evaluate the computational performance of our framework and its generated co-simulated accelerators, we selected the following metrics:

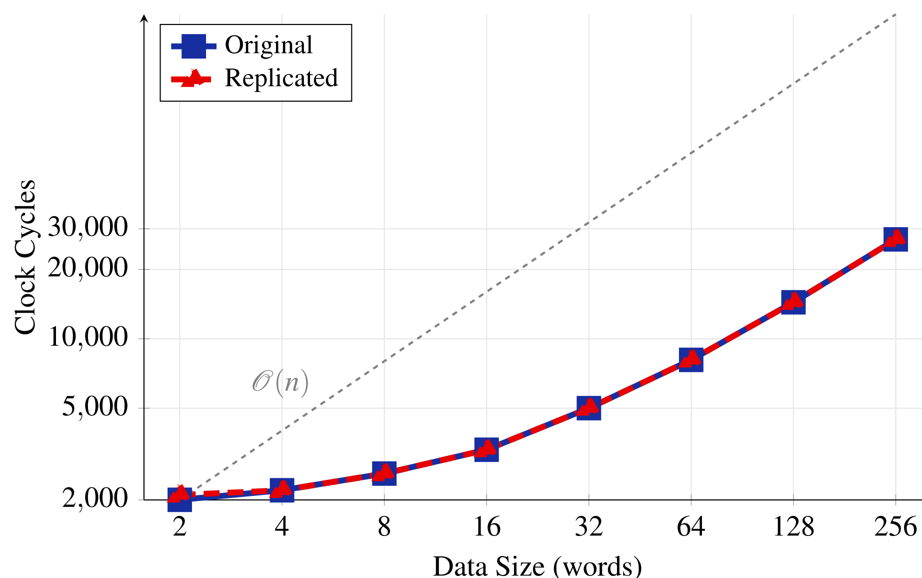
- Simulation cycles: the total number of clock cycles required to complete the simulation.
- Wall-clock simulation time: the actual elapsed real-world time it takes for the simulation to run from start to finish.

6.4 Experiment 2: Scaling for Multiple Accelerators

The second experiment evaluates how well our framework handles simulation workloads when scaling the number of instantiated accelerators within a single system. To do this, we replicate the same accelerator used in the previous experiment and instantiate multiple copies in the same simulation environment. This approach is used for both the original accelerator, written purely in SV, and the replicated `C++` version.

We report results for configurations with 2, 4, and 8 accelerators running concurrently, all using a fixed buffer size of 256.

Figure 6.1: Simulation times, measured in clock cycles, for different data sizes and implementations, when dealing with a single accelerator instance.



6.5 Analysis of Results

This section presents an analysis of the results obtained from both experiments.

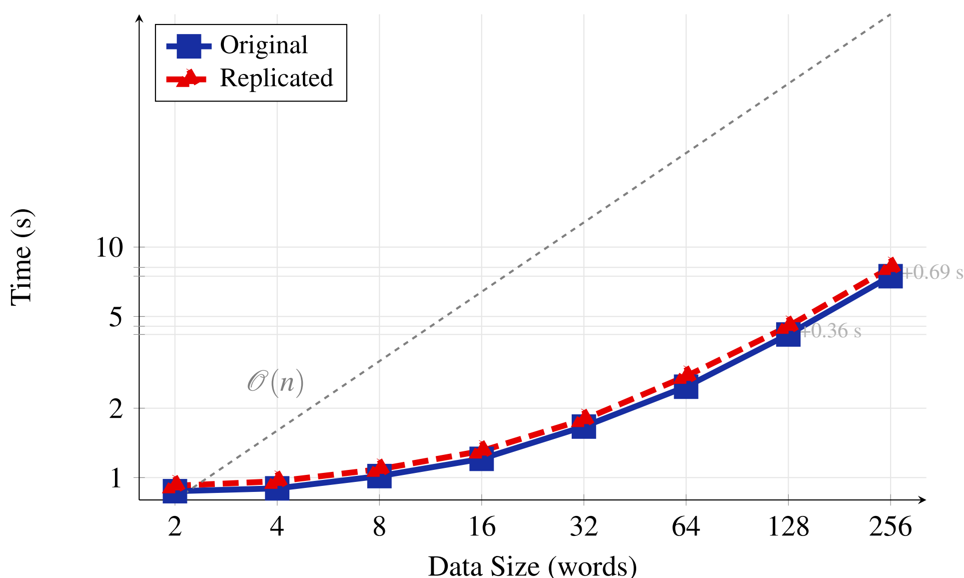
6.5.1 Experiment 1: Results Discussion

Figure 6.1 presents the number of clock cycles required to complete the simulation for various data sizes, comparing the original and replicated implementations. For all data sizes except the smallest one ($N = 2$), both implementations perform identically in terms of cycle count. At $N = 2$, the replicated version requires an additional 100 cycles (2102 vs. 2002). This difference in cycles probably indicates that the accelerator's C++ implementation is *not* mirroring the original's logic with complete precision. However, we believe that results are still quite satisfactory, given that for every other value of N, simulation times were identical.

However, performance discrepancies are accentuated when examining actual wall-clock simulation times in Figure 6.2. Here, we observe that while both implementations exhibit a linear trend with increasing data size, the replicated version consistently takes longer to complete. This is true even when both versions consume the same number of clock cycles, as the discrepancy stems not from algorithmic differences but from additional overhead related to the framework's underlying communication layer.

The performance gap is emphasized at data sizes $N = 128$ and $N = 256$, where the replicated version takes +0.36s and +0.69s longer than the original, respectively. This suggests that performance becomes worse as data sizes increase.

Figure 6.2: Wall-clock simulation times, measured in seconds, for different data sizes and implementations, when dealing with a single accelerator instance.



To better quantify this trend, Figure 6.3 plots the ratio between the wall-clock times of the replicated and the original implementations. The ratio stays between 1.06x and 1.11x, with the smallest overhead (6%) observed for $N = 2$ and the largest (11%) for $N = 64$.

These results are summarized in Table 6.3. ST_R and ST_O indicate the simulation cycles for the replicated and original implementations, respectively. Conversely, WC_R and WC_O measure the wall-clock simulation times, in real-time seconds, of the replicated and original implementations. The last column, r_{WC}, computes the ration between WC_R and WC_O.

6.5.2 Experiment 2: Results Discussion

Figure 6.4 plots the growth of simulation cycles for buffer sizes of 256, using 2, 4, and 8 accelerator instances. Both implementations achieved the exact same number of simulation clock cycles.

Figure 6.5 shows the wall-clock simulation times for both implementations, demonstrating that the replicated approach introduces a small overhead.

To better quantify this overhead, Figure 6.6 presents the ratio between the wall-clock simulation times of the two versions. The overhead remains under 10% regardless of the number of copies.

These results are generally consistent with those from the first experiment, indicating that the framework induces a constant overhead that does not grow disproportionately with the number of accelerator instances.

Table 6.4 provides a summary of the obtained simulation cycles and wall-clock simulation times for 2, 4, and 8 accelerator instances. ST_R and ST_O indicate the number of simulation cycles for the replicated and original versions, respectively. Conversely, WC_R and WC_O represent

Figure 6.3: Ratio between wall-clock simulation times for different data sizes and implementations, when dealing with a single accelerator instance.

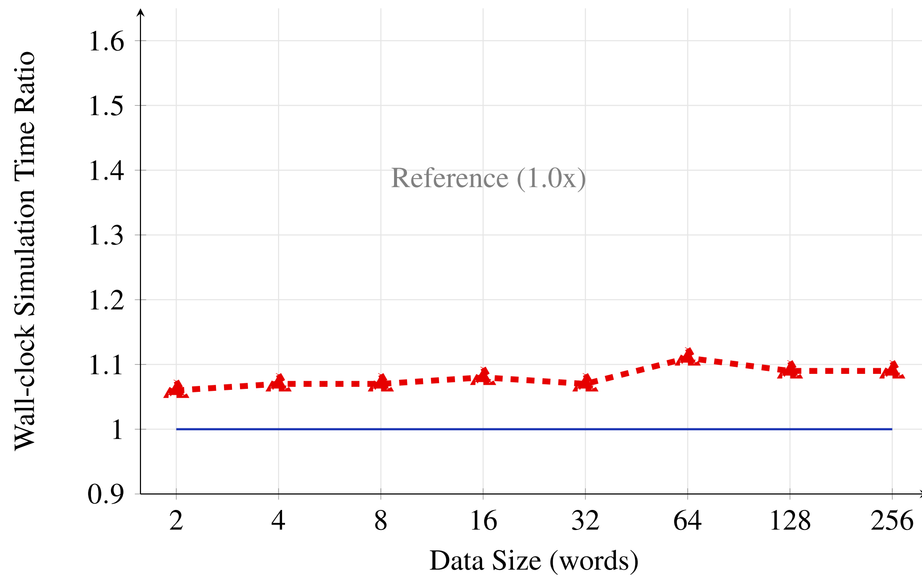
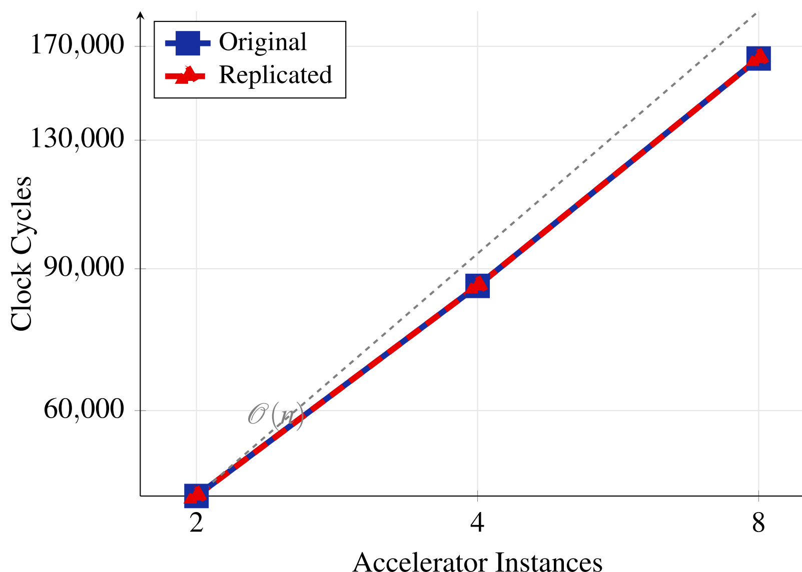


Table 6.3: Experiment's 1 performance summary for varying data sizes and implementations, when dealing with a single accelerator instance.

Size	ST_R	ST_O	WC_R **(s)**	WC_O **(s)**	r_{WC}
2	2.1K	2.0K	0.922	0.873	1.06
4	2.2K	2.2K	0.964	0.898	1.07
8	2.6K	2.6K	1.089	1.015	1.07
16	3.3K	3.3K	1.305	1.206	1.08
32	5.0K	5.0K	1.787	1.665	1.07
64	8.1K	8.1K	2.752	2.479	1.11
128	14.4K	14.4K	4.535	4.177	1.09
256	27.0K	27.0K	8.166	7.479	1.09

Figure 6.4: Simulation times, measured in clock cycles, for different numbers of accelerator instances.



the wall-clock simulation times, measured in real-time seconds, for the replicated and original versions. The last column, r_{WC}, is the ratio between WC_R and WC_O.

These results are summarized in Table 6.4.

6.6 Overall Evaluation and Takeaways

The replicated implementation exhibited the expected overheads compared to the original, which is consistent with the nature of the framework's approach. Importantly, the scaling of simulation times with increasing data sizes and number of accelerator instances closely follows that of the original implementation.

In terms of simulation cycles, the only observed deviation occurred at $N = 2$, during the first experiment, where a difference of 100 cycles was recorded. While this technically disqualifies the implementation from being cycle-accurate across all cases, we believe this discrepancy stems from minor timing differences that the `C++` version did not manage to accurately capture. Given that all other tested configurations produced identical cycle counts, we are confident that the implementation remains a very close approximation of the intended behavior.

Table 6.4: Experiment's 2 performance summary when using buffer sizes of 256 words for different numbers of accelerator instances.

Instances	ST_R	ST_O	$WC_R(s)$	$WC_O(s)$	r_{WC}
2	47002	47002	16.436	15.706	1.05
4	85702	85702	29.767	28.233	1.05
8	164202	164202	61.004	56.331	1.08

Figure 6.5: Wall-clock simulation times, measured in seconds, for different numbers of accelerator instances.

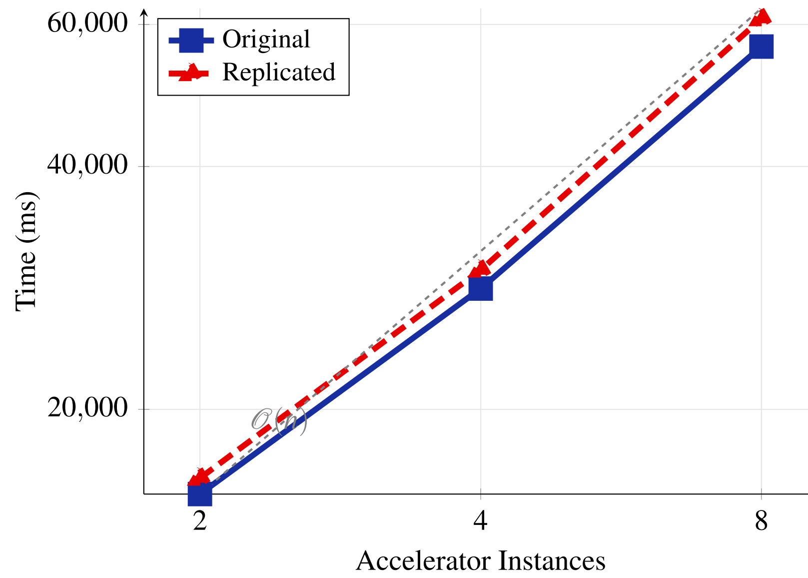
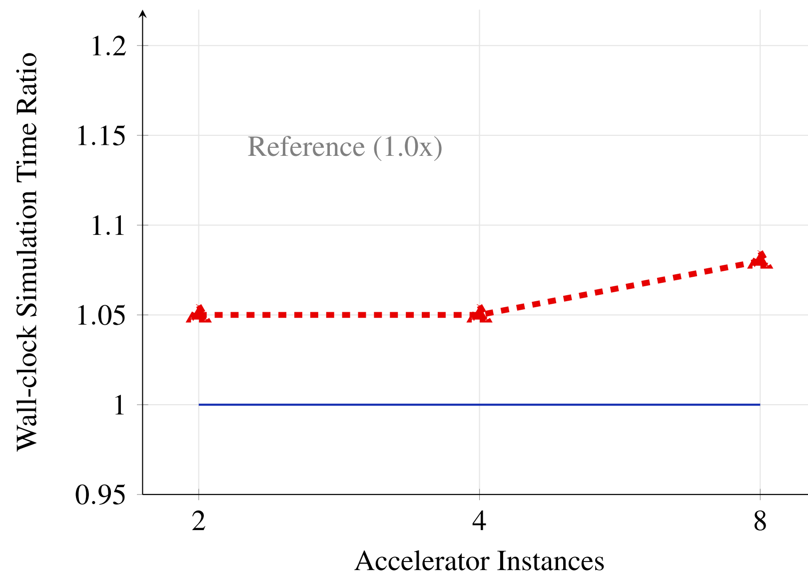


Figure 6.6: Ratio between wall-clock simulation times for different numbers of accelerator instances.



Wall-clock times for the replicated versions were consistently higher, but the overhead remained modest, reaching a maximum of around 11%. More importantly, this overhead is stable and does not increase disproportionately with the workload. Despite the exponential increase in data value, the overhead curve remains flat, indicating that the replication framework imposes a roughly constant cost rather than a scaling penalty. This containment of architectural complexity is a positive outcome that prevents disproportionate performance degradation.

Another important advantage is the reduction in turnaround time when modifying the simulator. Unlike traditional flows that require costly regeneration and recompilation of the entire SoC simulator (for example, with Verilator), in our framework, changes to the replicated peripheral model can be tested rapidly without recompilation of the whole system. This aligns well with the framework's goal of providing an agile prototyping and testing environment for peripherals integrated in complex systems.

It is worth noting that the accelerator used in these experiments is intentionally simple, with minimal input requirements and trivial domain logic. While this setup serves as a reasonable initial test, it may represent a worst-case scenario for co-simulation approaches where the workload is dominated by data transfer rather than computation. In such cases, higher-level implementations (for instance, in `C++` or Java) may even outperform cycle-accurate RTL simulation. Therefore, further testing with more complex and computation-heavy accelerators is necessary to fully evaluate scalability and robustness.

Nonetheless, these initial results provide strong proof of concept. Performance was never the primary goal of our framework. From the beginning, we prioritized developer experience, aiming to create an accessible environment that enables rapid iteration and testing of simulator implementations.

Overall, the framework's architectural overhead is manageable and consistent with the trade-offs made during design, while offering practical benefits in simulation workflow efficiency.

Chapter 7

Conclusion and Future Work

This chapter outlines the current framework's limitations, discusses possible enhancements, and provides a summary of the conducted research.

7.1 Summary of Research

This thesis presented the design, implementation, and evaluation of a co-simulation framework aimed at simplifying the integration of high-level simulators into existing SoC platforms. The motivation from this work stemmed from the challenges traditionally associated with modeling custom hardware components using low-level RTL and incorporating them into complex SoC environments, a process often hindered by lengthy build cycles and steep learning curves.

To address these challenges, the proposed framework enables developers to declaratively express their simulation and communication needs, automatically integrating their simulators into the X-HEEP platform.

The framework was evaluated through two main experiments, which analyzed the framework's functional correctness and performance with regard to simulation time, versus Verilator-only simulation of examples from X-HEEP's repository. The results demonstrated that while the replicated simulation environments introduced moderate overhead compared to native RTL implementations, they remained within acceptable performance bounds and offered significant gains in development agility. Additionally, the framework was shown to support scenarios involving multiple concurrent accelerators.

With our current knowledge, we can answer the initially proposed research questions:

RQ1 *Can the developed co-simulation framework enable bidirectional communication between a Verilated RTL design and a hardware component modeled as a higher-level software process?*

> **A1** Yes, it can. Our experiments demonstrated almost identical simulation times when replicating an example from X-HEEP's official suite.

RQ2 *Is a messaging protocol effective for managing inter-process communication between heterogeneous components?*

A2 Yes, it is. Our results showed a maximum overhead of 11% in wall-clock simulation times when executing a workload that is dominated by data transfer. This suggests that the communication overhead induced by networking libraries stays within reasonable bounds.

RQ3 *Can high-level simulators, when integrated through this framework, independently issue memory transactions such as reads and writes to the SoC?*

A3 Yes, they can. The high-level APIs offer a memory controller module that simulators can use to issue memory read and write requests independently, on demand, and includes capability for burst access to memory.

Ultimately, we hope that this research provides meaningful value to simulator developers by offering a foundation for rapid prototyping.

7.2 Future Work

This section addresses the current known limitations of the framework and outlines possible areas for future improvements.

7.2.1 Limitations

The most prominent limitation lies in the framework's tight coupling to the X-HEEP platform. Many of the core abstractions and high-level components, such as the memory controller from the high-level API, are specifically designed for X-HEEP. As a result, these components cannot be readily reused with other platforms that rely on different bus protocols or have alternative memory hierarchies.

Furthermore, the framework's code generation pipeline was explicitly designed to align with the integration requirements of the X-HEEP infrastructure. Platform-specific glue code is generated under the assumption that the simulator will be embedded into the X-HEEP system. Porting this integration process to a different SoC would therefore require a substantial reworking of the generation logic.

Despite these limitations, the underlying communication architecture of the framework remains broadly applicable.

7.2.2 Performance Enhancements

The current communication setup between the SoC and the high-level simulator introduces unnecessary overhead that impacts simulation speed.

At each clock cycle, the framework performs both `send` and `recv` operations, regardless of whether new data needs to be transmitted. While this ensures correctness and simplifies control flow, it results in constant interaction with the messaging backend, even during idle periods. This frequent, redundant messaging increases computational load and contributes to slower wall-clock performance, especially as simulation complexity grows.

Another limitation is the lack of interrupt support for the simulator's corresponding peripheral. Instead of receiving a trigger for when the simulator has completed its task, the processor must continuously poll the peripheral for a status update. A polling mechanism is inefficient and adds further overhead since the processor wastes cycles checking for changes rather than performing useful work.

Improving performance in future versions could involve two key changes. First, replacing the constant polling mechanism with event-driven communication or conditional messaging would reduce the number of unnecessary messages. Second, implementing interrupt-like behavior would allow the processor to wait passively and react only when needed.

References

[1] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research. *J. Syst. Archit.*, 133(C):12, 2022.

[2] Mossaad Ben Ayed, Ayman Massaoudi, Shaya A. Alshaya, and Mohamed Abid. System-level co-simulation for embedded systems. *AIP Advances*, 10(3):035113, 03 2020.

[3] Andrew Boutros and Vaughn Betz. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[4] Y. Braatz, D. S. Rieber, T. Soliman, and O. Bringmann. Simpyler: A compiler-based simulation framework for machine learning accelerators. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, volume 2023-July, pages 213–220. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 2; Conference name: 34th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2023; Conference date: 19 July 2023 through 21 July 2023; Conference code: 193186; CODEN: PIAAF.

[5] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416.

[6] G. Busnot, T. Sassolas, N. Ventroux, and M. Moy. Standard-compliant parallel systemc simulation of loosely-timed transaction level models. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, volume 2020-January, pages 363–368. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 4; Conference name: 25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020; Conference date: 13 January 2020 through 16 January 2020; Conference code: 158875.

[7] Christopher Celio, Pi-Feng Chiu, Krste Asanović, Borivoje Nikolić, and David Patterson. Broom: An open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro*, 39(2):52–60, 2019.

[8] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yuanyuan Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : Industrial product. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64, 2020.

[9] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009.

[10] Enfang Cui, Tianzheng Li, and Qian Wei. Risc-v instruction set architecture extensions: A survey. *IEEE Access*, 11:24696–24711, 2023.

[11] Tarek Darwish and Magdy Bayoumi. 5 - trends in low-power vlsi design. In WAI-KAI CHEN, editor, *The Electrical Engineering Handbook*, pages 263–280. Academic Press, Burlington, 2005.

[12] Fares Elsabbagh, Blaise Tine, Priyadarshini Roshan, Ethan Lyons, Euna Kim, Da Eun Shim, Lingjun Zhu, Sung Lim, and Hyesoon kim. Vortex: Opencl compatible risc-v gpgpu. 02 2020.

[13] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), page 33–40, New York, NY, USA, 1960. Association for Computing Machinery.

[14] J. Haris, P. Gibson, J. Cano, N. B. Agostini, and D. Kaeli. Secda: Efficient hardware/software co-design of fpga-based dnn accelerators for edge inference. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 33–43.

[15] C. Heinz, J. A. Hofmann, L. Sommer, and A. Koch. Improving job launch rates in the tapasco fpga middleware by hardware/software-co-design. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 22–30.

[16] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. The tapasco open-source toolflow: for the automated composition of task-based parallel reconfigurable computing systems. *J. Signal Process. Syst.*, 93(5):545–563, May 2021.

[17] T. Hotfilter, J. Hoefer, F. Kreß, F. Kempf, and J. Becker. Flecsim-soc: A flexible end-to-end co-design simulation framework for system on chips. In *2021 IEEE 34th International System-on-Chip Conference (SOCC)*, pages 83–88.

[18] J. Kwon, S. Oh, and D. Park. Metamorphic edge processor simulation framework using flexible runtime partial replacement of software-embedded verilog rtl models. In *Proceedings - IEEE International Symposium on Circuits and Systems*, volume 2021-May. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 1; Correspondence Address: D. Park; School of Electronic and Electrical Engineering, Kyungpook National University, Daegu, 41566, South Korea; email: boltanut@knu.ac.kr; Conference name: 53rd IEEE International Symposium on Circuits and Systems, ISCAS 2021; Conference date: 22 May 2021 through 28 May 2021; Conference code: 169837; CODEN: PICSD.

[19] Y. C. Lee, T. S. Hsu, C. T. Chen, J. J. Liou, and J. M. Lu. Nnsim: A fast and accurate systemc/tlm simulator for deep convolutional neural network accelerators. In *2019 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2019*. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 7; Conference name: 2019 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2019; Conference date: 22 April 2019 through 25 April 2019; Conference code: 148922.

[20] X. Ling, T. Notsu, and J. Anderson. An open-source framework for the generation of risc-v processor + cgra accelerator systems. In F. Leporati, S. Vitabile, and A. Skavhaug, editors, *Proceedings - 2021 24th Euromicro Conference on Digital System Design, DSD 2021*, pages 35–42. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 13; Conference name: 24th Euromicro Conference on Digital System Design, DSD 2021; Conference date: 1 September 2021 through 3 September 2021; Conference code: 176822.

[21] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), October 2019.

[22] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators, 2024.

[23] G. Michell and Rajesh Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85:349 – 365, 04 1997.

[24] S. Nema, S. K. Chunduru, C. Kodigal, G. Voskuilen, A. F. Rodrigues, S. Hemmert, B. Feinberg, H. Lee, A. Awad, and C. Hughes. Eras: A flexible and scalable framework for seamless integration of rtl models with structural simulation toolkit. In *Proceedings - 2023 IEEE International Symposium on Workload Characterization, IISWC 2023*, pages 196–200. Institute of Electrical and Electronics Engineers Inc. Export Date: 30 October 2024; Cited By: 0; Conference name: 26th IEEE International Symposium on Workload Characterization, IISWC 2023; Conference date: 1 October 2023 through 3 October 2023; Conference code: 193984.

[25] Dima Nikiforov, Shengjun Chris Dong, Chengyi Lux Zhang, Seah Kim, Borivoje Nikolic, and Yakun Sophia Shao. RosÉ: A hardware-software co-simulation infrastructure enabling pre-silicon full-stack robotics soc evaluation, 2023.

[26] Andy D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design Test*, 34(1):77–90, 2017.

[27] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

[28] Sriram Rajagopal. Edge q 5g with an edge. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–13, 2021.

[29] Wilson Snyder, Paul Wasson, and Duane Galbi et al. Verilator. GitHub repository: `https://github.com/verilator/verilator`, 2023. The Verilator package converts Verilog and SystemVerilog hardware description language (HDL) designs into a fast C++ or SystemC model that, after compiling, can be executed. Verilator is not a traditional simulator but a compiler.

[30] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017. Editors.

[31] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.

[32] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-core risc-v chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41(2):36–42, 2021.

[33] Fulai Zhu, Peiyu Xu, and Jiahao Zong. Moore's law: The potential, limits, and breakthroughs. *Applied and Computational Engineering*, 10:307–315, 09 2023.