FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING
DEGREE PROGRAMME IN WIRELESS COMMUNICATIONS ENGINEERING

MASTER'S THESIS

Pre-validation of SoC via Hardware and Software Co-simulation

Author	Valtteri Karmitsa
Supervisor	Jukka Lahti
Second Examiner	Marko Pakaslahti
(Technical Advisor	Petri Palosaari)

October 2023

Karmitsa V. (2023) Pre-validation of SoC via Hardware and Software Co-simulation. University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 63 p.

ABSTRACT

System-on-chips (SoCs) are complex entities consisting of multiple hardware and software components. This complexity presents challenges in their design, verification, and validation. Traditional verification processes often test hardware models in isolation until late in the development cycle. As a result, cooperation between hardware and software development is also limited, slowing down bug detection and fixing.

This thesis aims to develop, implement, and evaluate a co-simulation-based pre-validation methodology to address these challenges. The approach allows for the early integration of hardware and software, serving as a natural intermediate step between traditional hardware model verification and full system validation. The co-simulation employs a QEMU CPU emulator linked to a register-transfer level (RTL) hardware model. This setup enables the execution of software components, such as device drivers, on the target instruction set architecture (ISA) alongside cycle-accurate RTL hardware models.

The thesis focuses on two primary applications of co-simulation. Firstly, it allows software unit tests to be run in conjunction with hardware models, facilitating early communication between device drivers, low-level software, and hardware components. Secondly, it offers an environment for using software in functional hardware verification.

A significant advantage of this approach is the early detection of integration errors. Software unit tests can be executed at the IP block level with actual hardware models, a task previously only possible with costly system-level prototypes. This enables earlier collaboration between software and hardware development teams and smoothens the transition to traditional system-level validation techniques.

Key words: co-simulation, QEMU, virtual platform.

Karmitsa V. (2023) Järjestelmäpiirin esivalidointi laitteiston ja ohjelmiston yhteissimulaatiolla. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 63 p.

TIIVISTELMÄ

Järjestelmäpiirit (SoC) ovat monimutkaisia kokonaisuuksia, jotka koostuvat useista laitteisto- ja ohjelmistokomponenteista. Tämä monimutkaisuus asettaa haasteita niiden suunnittelulle, varmennukselle ja validoinnille. Perinteiset varmennusprosessit testaavat usein laitteistomalleja eristyksissä kehityssyklin loppuvaiheeseen saakka. Tämän myötä myös yhteistyö laitteisto- ja ohjelmistokehityksen välillä on vähäistä, mikä hidastaa virheiden tunnistamista ja korjausta.

Tämän diplomityön tavoitteena on kehittää, toteuttaa ja arvioida laitteisto-ohjelmisto-yhteissimulointiin perustuva esivalidointimenetelmä näiden haasteiden ratkaisemiseksi. Menetelmä mahdollistaa laitteiston ja ohjelmiston varhaisen integroinnin, toimien luonnollisena välietappina perinteisen laitteistomallin varmennuksen ja koko järjestelmän validoinnin välillä. Yhteissimulointi käyttää QEMU suoritinemulaattoria, joka on yhdistetty rekisterinsiirtotason (RTL) laitteistomalliin. Tämä mahdollistaa ohjelmistokomponenttien, kuten laiteajureiden, suorittamisen kohdejärjestelmän käskysarja-arkkitehtuurilla (ISA) yhdessä kellosyklitarkkojen RTL laitteistomallien kanssa.

Työ keskittyy kahteen yhteissimulaation pääsovellukseen. Ensinnäkin se mahdollistaa ohjelmiston yksikkötestien suorittamisen laitteistomallien kanssa, varmistaen kommunikaation laiteajurien, matalan tason ohjelmiston ja laitteistokomponenttien välillä. Toiseksi se tarjoaa ympäristön ohjelmiston käyttämiseen toiminnallisessa laitteiston varmennuksessa.

Merkittävä etu tästä lähestymistavasta on integraatiovirheiden varhainen havaitseminen. Ohjelmiston yksikkötestejä voidaan suorittaa jo IP-lohkon tasolla oikeilla laitteistomalleilla, mikä on aiemmin ollut mahdollista vain kalliilla järjestelmätason prototyypeillä. Tämä mahdollistaa aikaisemman ohjelmisto- ja laitteistokehitystiimien välisen yhteistyön ja helpottaa siirtymistä perinteisiin järjestelmätason validointimenetelmiin.

Avainsanat: yhteissimulointi, QEMU, virtuaalialusta.

TABLE OF CONTENTS

FOREWORD

This thesis focuses on developing, implementing, and evaluating a co-simulation-based pre-validation methodology for System-on-Chips (SoCs). The primary objective is to enable early hardware and software integration, facilitating more efficient verification and validation processes. This thesis work was conducted at Nokia Solutions and Networks Oy. I am grateful to my line manager, Sami Jylhä, for the opportunity to work at Nokia, for providing me with this fascinating yet challenging topic, and for the overall supportive attitude towards my studies.

Developing and understanding the needed components for the co-simulation architecture was a highly educational process made possible only with the support of various individuals from Nokia. I want to thank Petri Palosaari for serving as my technical advisor, proofreading my thesis, and providing invaluable assistance with the verification environment challenges. I want to express my gratitude towards Joona Rintamäki and Ari Hautala for their invaluable assistance in comprehending the problem from a validation standpoint and for their overall encouraging attitude towards this project. I also sincerely thank Ville Lahtinen for aiding with software-related configurations and unit testing and Hannu Nieminen for his general guidance on software-related topics. Furthermore, I would like to thank Panu Poiksalo and Jari Mäkinen for their technical assistance with the Target SoC Simulator (TSS). Panu's enthusiasm and positive words were invaluable during the entire project. While it is not possible to mention everyone by name, I want to thank numerous other employees at Nokia who have helped me with this project.

I want to thank Jukka Lahti from the University of Oulu for guiding my thesis, giving valuable feedback, and professional teaching throughout my studies. Jukka's excellent digital design courses initially introduced me to the world of digital techniques. Additionally, I would like to thank Marko Pakaslahti for serving as the second examiner of this thesis. Finally, I want to express my gratitude to my parents for their encouragement and support in everything I do. The can-do attitude with which they raised me has been the foundation for all my achievements.

Oulu, October 21, 2023

Valtteri Karmitsa

LIST OF ABBREVIATIONS AND SYMBOLS

AFM	Arm Fast Model
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BFM	Bus Functional Model
CPU	Central Processing Unit
CSR	Control and Status Register
DBT	Dynamic Binary Translation
DMA	Direct Memory Access
DPI	Direct Programming Interface
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
DTB	Device Tree Blob
DTC	Device Tree Compiler
DTS	Device Tree Source
DUT	Device Under Test
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
GDB	GNU Debugger
GIC	Generic Interrupt Controller
HDL	Hardware Description Language
HW	Hardware
IP	Intellectual Property
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group
NoC	Network-on-Chip
OS	Operating System
PSS	Accellera Portable Test and Stimulus
QEMU	Quick Emulator
RAL	Register Abstraction Layer
RAM	Random Access Memory
RO	Read-Only
RobotFW	Robot Framework
ROM	Read Only Memory
RPC	Remote Procedure Call
RTL	Register-Transfer Level
RTOS	Real-Time Operating System
SRAM	Static Random Access Memory
SoC	System-on-Chip

SPI	Shared Peripheral Interrupt
SSH	Secure Shell
SV	SystemVerilog
SW	Software
TCG	Tiny Code Generator
TCP	Transmission Control Protocol
TLM	Transaction-Level Modelling
TSS	Target SoC Simulator
UART	Universal Asynchronous Receiver/Transmitter
UC	Use Case
UT	Unit Test
UVM	Universal Verification Methodology
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VIP	Verification IP
VM	Virtual Machine
VP	Virtual Platform
W1C	Write-One-to-Clear
WO	Write-Only
XML	Extensible Markup Language

1 INTRODUCTION

System-on-chips (SoCs) integrate various hardware (HW) and software (SW) components onto a single chip to provide complete system functionality. As SoCs become increasingly complex, verifying their correct functionality is crucial yet challenging [1], accounting for almost 70% of the total chip design effort and expenses [2].

The design of a complex application-specific integrated circuit (ASIC) SoC involves creating models of the system at different levels of abstraction. As the design progresses, these models are transformed and refined until all the details required for silicon manufacturing are available. This progression through different models representing increasing levels of implementation detail is referred to as the SoC design flow. A critical step in this design flow is partitioning system functionality between hardware and software components [3], as depicted in Figure 1.

Following the architecture partitioning, corresponding teams can proceed to design hardware components and code software elements. Integration is where new software meets the new hardware, and their interoperability can be validated. Early integration benefits from finding integration difficulties earlier in the SoC development process and allows for software usage in hardware configuration during functional hardware verification phases. [4]

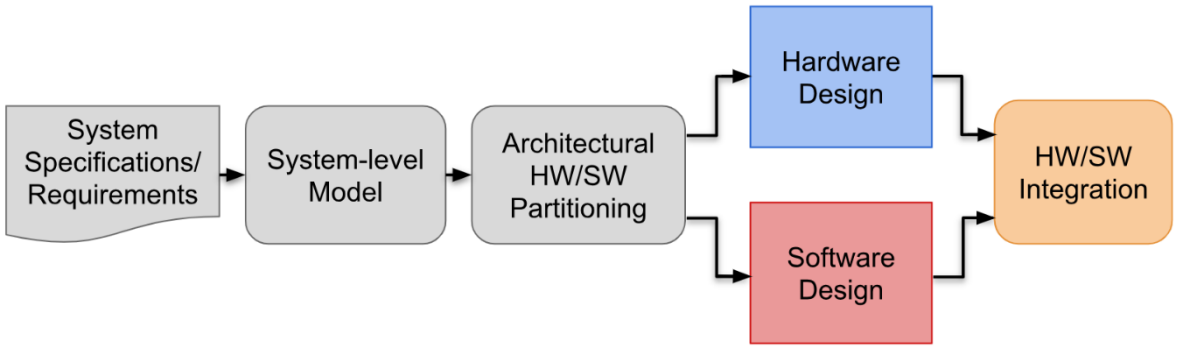

Figure 1. Hardware and software partitioning and integration

The complexity of verifying and validating modern SoCs arises from the complicated interplay between hardware components and layers of software. SoCs integrate third-party and legacy intellectual property (IP) blocks alongside new internal IPs into a coherent architecture. Each IP block undergoes verification in isolation by teams specializing in verification. These blocks must work collectively at the system level, requiring extensive integration testing. Conversely, software development often lags behind hardware creation, resulting in late-stage integration and delayed bug identification. [1]

Traditional verification approaches rely heavily on testing hardware models in isolation until late in the development cycle before integrating hardware and software [5]. Verification teams use their own methods, like Universal Verification Methodology (UVM). This siloed approach delays the discovery of integration issues and necessitates duplicate efforts between teams. Earlier integration of software and hardware models could alleviate this problem.

The primary goal of this thesis is to develop and assess a co-simulation approach for early SoC pre-validation, an intermediate step between standalone hardware verification and full SoC validation. This involves executing software on a virtual Central Processing Unit (CPU) emulator while simulating the rest of the system as register-transfer level (RTL) hardware models. The aim is to integrate and start testing of hardware and software together earlier in the design flow, a concept known as "shift-left."

The traditional waterfall model postpones integration until hardware prototypes or emulators are available due to logic simulation performance limitations [1]. This late integration shrinks the time available for software development. It also leads to extensive validation overhead from discovering bugs arising from untested interactions between hardware and software components. Another drawback concerning traditional prototyping methods is vendor lock-in to expensive commercial virtual platforms. In contrast, the pre-validation approach developed in this thesis utilizes an open-source Quick Emulator (QEMU) [6] for software execution and standard RTL simulators, avoiding vendor lock-in to any specific tools.

Custom extensions to QEMU, developed by Nokia, facilitate communication between software executing on QEMU and external hardware models in RTL simulation, creating a full SoC virtual platform. The aim of the co-simulation architecture is to allow early integration and testing of production software in conjunction with the RTL hardware model, pre-validating their interplay before the traditional validation activities take place.

The objective of this thesis is to enable early-stage software integration and testing with RTL hardware models, aiming for earlier hardware and software integration testing to reduce validation overhead in later stages and pre-emptively identify potential integration issues.

The second chapter focuses on introducing the basic building blocks of a typical system-on-chip (SoC), including the CPU, memories, IP blocks, interconnects, and software stack components like device drivers and operating systems, and shows that by combining the different elements, a complete system is formed.

The 3rd chapter will build upon the 2nd chapter and give an overview of hardware verification, software testing, and system-level validation techniques used to ensure correct SoC functionality. It emphasizes the functional verification of hardware models and the challenges of verifying complex hardware and software interactions before the availability of prototypes.

The 4th chapter delves into the components enabling the co-simulation environment, including the QEMU emulator to run the software, the TSS Server to facilitate transaction-tevel modeling (TLM) communication, and the integration of RTL hardware models. It discusses the potential benefits of this virtual platform approach for early integration, along with an analysis of associated implementation challenges.

The 5th chapter demonstrates the two key use cases for the co-simulation architecture: software unit testing and the method for software-driven functional verification. To obtain perspective on the utilization of co-simulation for pre-validation, the performance and realism of execution are discussed.

The 6th chapter discusses potential future work and offers an analysis of completed work, while the 7th chapter concludes the thesis with a summary.

2 SYSTEM-ON-CHIP ARCHITECTURE

A System-on-Chip (SoC) is an integrated circuit that integrates the main components of a computer into a single chip. Microprocessors, predecessors of SoCs, combined just the Central Processing Unit with its execution and control units. Improvements in VLSI technology now allow placing the CPU, primary memory, and input/output devices on the same chip. [7]

SoCs are used in many embedded and mobile applications, such as smartphones, tablets, IoT devices, and automotive electronics. An SoC combines pre-designed hardware blocks with well-defined functionality known as "intellectual properties" or "IPs," which can be augmented with accompanying software. The SoC design philosophy aims to accelerate the development by configuring the IPs for the target SoC use case (UC). Fast system design is facilitated through standardized communication interfaces between the IPs, thereby enabling rapid design iteration. [8]

2.1 Hardware on a SoC

Figure 2 provides a high-level overview of the main components of a typical System-on-Chip (SoC). The components can be logically divided into four categories: control, communication, computation, and storage [9]. Or equivalently to - CPU, interconnects, IP blocks, and memory. IP blocks can be further grouped into subsystems that contain multiple IP blocks, as seen in the filter subsystem in Figure 2. These subsystems can then be integrated at the top level of the SoC architecture alongside the other major components like CPU and interconnects.

The CPU provides the main computing power and acts as a central controller controlling other components. Memories hold data and instructions close to the CPU. IP blocks provide application-specific functionality, such as a filter subsystem containing digital signal processing (DSP) IP for signal processing. Interconnects like high-speed and slower peripheral buses tie everything together and facilitate communication between the hardware components. [7]

In Figure 2, one of the IP blocks is the interrupt controller, which works closely with the CPU. It forwards interrupt lines from other IP blocks to the CPU. In multicore systems, interrupts can be statically routed to a particular core or dynamically routed based on the availability of a non-interrupted core. IP blocks like Arm’s Generic Interrupt Controller (GIC) can implement this functionality. [7]

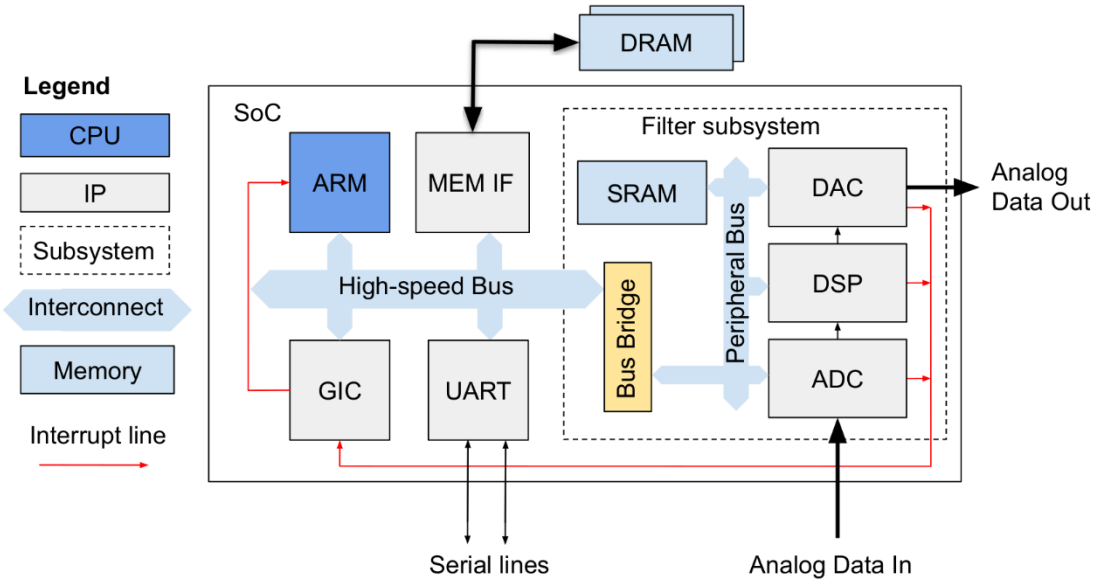

Figure 2. Example SoC Hardware Architecture.

2.1.1 Central Processing Unit

The central processing unit (CPU) runs all the software one instruction at a time and controls the entire system's behavior. The fundamental job of the CPU is to execute instructions from software programs by fetching instructions from memory, decoding them, executing the operations specified, and writing results back to registers or memory. The CPU runs the operating system and applications software on the SoC by scheduling processes and threads. Through device drivers and the OS, the CPU also manages all the other hardware components in the SoC, like peripherals, memory controllers, and accelerators. CPUs provide a consistent instruction set architecture abstraction to software programmers so that code can be written in programming languages like C/C++ and other languages targeting the CPU's Instruction Set Architecture (ISA). [7]

Arm and RISC-V are popular CPU architectures used in SoCs. The CPU defines the instruction set architecture (ISA) and programmer's model, such as registers, execution modes, memory access, and interrupts. An ISA is a set of instructions a specific processor family can execute. SoCs today often utilize a multi-core architecture integrating 2, 4, 8, or more identical CPU cores on one chip. Multi-core CPUs provide greater processing power, throughput, energy efficiency, and flexibility through parallel execution. CPU cores can be packaged with caches, memory management units, debug components, and other logic as reusable IP blocks that system integrators can integrate into SoCs. [7]

2.1.2 Memory

Memory is a critical component of any SoC, often occupying over 50% of the chip area. Different memory technologies serve distinct needs for speed, density, and non-volatility. Static random access memory (SRAM), built from cross-coupled inverters, offers fast access times, making it ideal for caches, register files, and local scratchpad memories. SRAM retains data while powered on but is less dense than other memories. [7]

A SoC utilizes denser dynamic RAM (DRAM) for more extensive primary storage needs. DRAM stores data as charge in capacitors, necessitating periodic refresh cycles. Slower than SRAM, DRAM acts as the main system memory, with sophisticated controllers to maximize throughput. Non-volatile memories like read-only memory (ROM) retain data while the system power is off. [7]

The memory has a massive impact on overall SoC performance and efficiency. Various memory technologies are strategically deployed across the SoC to balance speed, density, leakage power, and reliability requirements. The processor or DSP IPs rely heavily on fast SRAM caches close by and DRAM controllers to access larger main memory. [7]

2.1.3 Special Purpose IP Blocks

In addition to processor and memory components, any SoC has a collection of specialized IP blocks that provide connectivity, acceleration, control, and sensing capabilities. These IP blocks are integrated alongside the processor and memory to create a complete SoC. Interface IP blocks like USB, Ethernet, and PCIe enable the SoC to connect with peripheral devices, networks, and high-speed expansion slots. Display interfaces like HDMI and DVI connect the SoC to external monitors and TVs. Interface IP is selected based on required peripherals and performance. [7]

Various controller IP blocks handle critical SoC management tasks like interrupts, direct memory access (DMA) transfers, DRAM access, and power management. Interrupt controllers route external events to the CPU cores. DMA controllers offload heavy data transfers, relieving the main CPUs from handling these low-level duties. [7]

To accelerate the processing of workloads like signal processing, graphics, and vision, SoCs integrate specialized hardware accelerator IP blocks [7]. Digital signal processors (DSPs) efficiently implement signal processing algorithms like Fast Fourier Transforms (FFT), finite impulse response filters (FIR), and Viterbi decoders for digital communications systems [10]. Hardware accelerators exploit parallelism and custom architectures to massively outperform CPU-based software execution [7]. Analog interface IP blocks like ADCs and DACs are required for any SoC to interact with real-world analogue components, which convert between digital signals and analogue voltages [7].

2.1.4 Interconnections

Interconnects are responsible for routing data between on-chip blocks, ensuring that all system components, such as CPUs, hardware accelerators, memory controllers, and I/O interfaces, cooperate. Interconnections can be divided into two types: shared buses, which connect multiple devices to a single communication pathway, and point-to-point buses, which form dedicated lines of communication between two blocks. [9]

In any bus system, the devices attached play one of two roles. They are either bus masters, which can initiate transactions, or bus slaves, designed to only respond to transactions. The main CPU is a typical example of a bus master. However, peripherals supporting DMA can also serve as bus masters. A dedicated DMA controller facilitates communication, particularly in systems where devices cannot initiate transactions. This controller can perform DMA transactions on behalf of bus slaves, ensuring that data is transmitted as intended. [7]

Bus-based interconnects may become restrictive in complicated SoCs. The architecture of Network-on-chips (NoCs) can be leveraged to address this issue. NoCs are state-of-the-art on-chip communication architectures that have become standard in high-performance SoCs. Instead of the conventional bus methodology, NoCs use packetized communication, sending data between routing switches via dedicated interconnect links. Parts of a SoC may continue to use local traditional interconnects linked to the NoC through bridges. [7]

The AMBA, or the Advanced Microcontroller Bus Architecture, is an example of bus architecture that is widely used. Developed by ARM, this architecture offers multiple variants for varying needs. These include the APB (Advanced Peripheral Bus), AHB (Advanced High-performance Bus), and the AXI (AMBA Advanced Extensible Interface). Each of these variants has slightly different characteristics, with some performing better in bandwidth, others in speed, and others being able to operate in multiple clock domains. Despite being ARM's architecture, AMBA buses are also used in SoCs that do not include an ARM CPU. AXI4-Stream, for instance, is AMBA's version of point-to-point connect, operating equivalently to a standard AXI4 bus in one direction, excluding the address bus. [7]

By using bridges between interconnect buses, hierarchical communication can be facilitated between IP blocks and the main CPU by forwarding transactions from one bus to another. This is made by bridges acting as both a master on the originating bus and a slave on the destination bus. For example, IP blocks that do not require high performance, like universal asynchronous receiver/transmitter (UART), can be placed lower in the bus architecture and utilize buses like AMBA APB instead of AXI. Bridges use addresses for determining the destination slave. Bridges asymmetrically connect buses, often allowing forwarding only in one direction, but they can also serve as masters and slaves, offering greater flexibility. [9]

Figure 3 provides an overview of how the main components, including the main CPU and memory, interface directly with the high-performance NoC. Complementing this direct connection are bridges that are connected to the primary NoC. These bridges forward transactions to subsystems and the IP blocks within them. The illustration also depicts the use of a point-to-point bus. It functions similarly to a shared bus, consisting of a master and a slave, but without an address space. This design choice reduces transaction routing overhead, making the data transfer resemble an infinite data stream. Point-to-point interfaces, such as AXI4-Stream, are essential in parts of an SoC, like the main data path, where high performance is needed, and communication addressing is not required. [7]

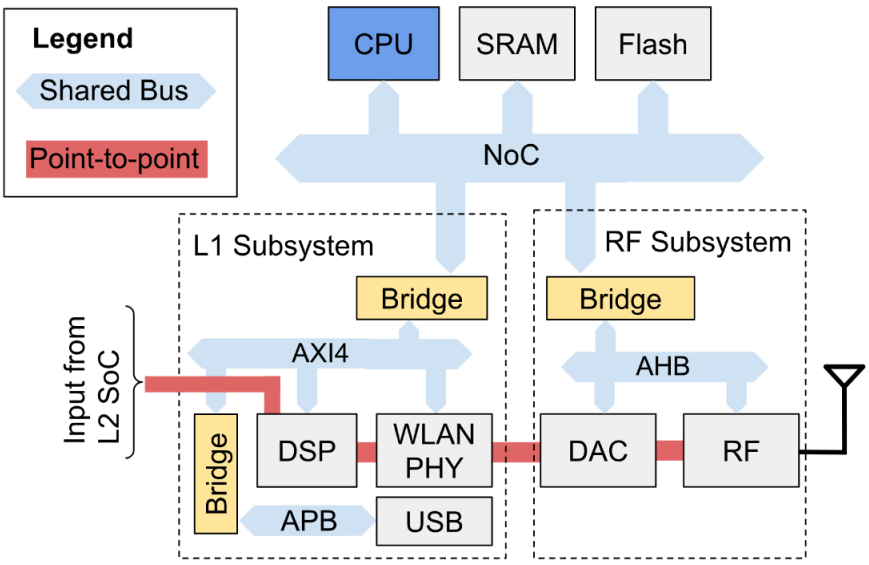

Figure 3. Example SoC Interconnects.

2.2 Software on a SoC

The software components in a SoC deliver the control programs, protocols, and applications that enable the system functionality. The software determines the use cases, features, and ultimate end-user experience the SoC platform provides. The software in an embedded SoC can be logically divided into several layers based on the level of abstraction [9]. Figure 4 depicts a scenario at a high level in which a CPU and IP block are connected to an on-chip bus. An operating system with the IP block-specific device driver and API runs on the CPU. Typically, each IP block has its IP-specific device driver and possible API layers. These software components part of a particular hardware IP block can be considered software IPs [3].

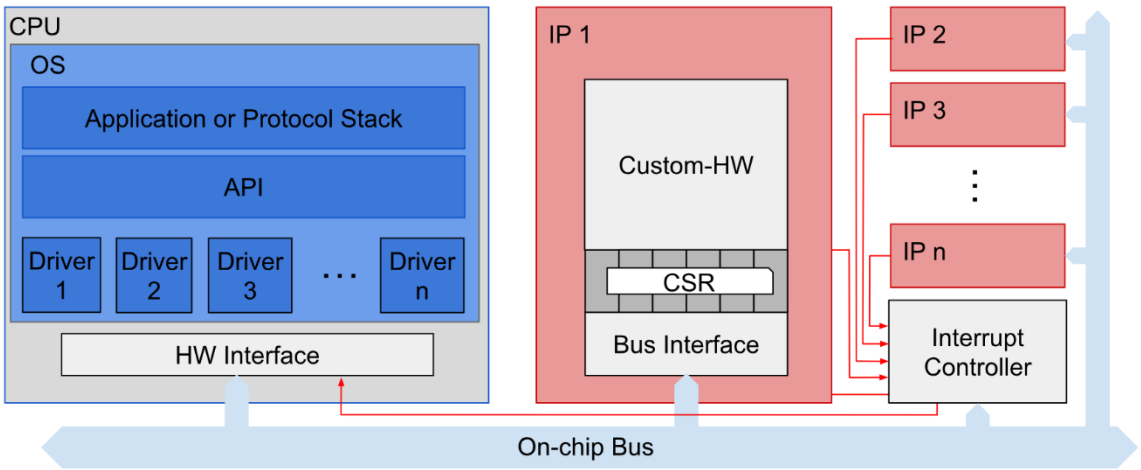

Figure 4. Example SoC Software Components.

Hardware IP blocks communicate with the CPU through reserved memory-mapped addresses. The CPU writes and reads these addresses to transfer data to and from the IP block's Control and Status Registers (CSR). The bus interface connects to the block's internal register

bank, making it appear to the CPU as memory. The IP blocks' bus interface handles communication protocols and data transfer to and from the on-chip bus. [9]

The CPU also has an interface to connect to the on-chip bus. This interface includes hardware and firmware, allowing the software on the CPU to access the bus. A device driver acts as the interface between the software API on the CPU and the CPU's hardware connection. The driver wraps the hardware transactions into software function calls. [9]

IP blocks' software API hides the lower-level details of the device driver [9]. An API may implement a complicated IP block configuration behind a single function call. For example, a DSP IP's API could implement a function that allows the configuration of the IP with a specific sample rate, which would otherwise require hundreds of manual register writes/reads.

2.2.1 Device Driver

Device drivers are one firmware component operating at the lowest level of software on the CPU. Device drivers enable communication between IP block devices and higher-level software layers. When an application or protocol wishes to access a device or when a device wishes to send a message to higher software layers, the device driver is responsible for facilitating the communication. When the device driver wishes to communicate with the device, the device register interface is utilized. Alternatively, when a device has something to say to the device driver, it can raise the interrupt line, which is routed through the interrupt controller to the CPU. [11]

The hardware side exposes addressable control and status registers (CSR) that software can read and write. The ability to access registers by device drivers can be used to configure the device, read status information, or instruct the device to start some functionality by writing to command registers. For some kinds of devices, registers may also act as data buffers. For example, a UART driver may configure baud rate and transfer parameters to control registers and transmits data by writing to a data register. [11]

Hardware also generates interrupt signals to notify software of events like data ready or errors. During the device operation, some event might occur that needs to be solved by the software. For example, an IP block can manage real-world events like sending and receiving data packets in communication devices. UART driver might receive data by reading a register when the hardware interrupts upon data reception. The IP block may have buffers to store packets, but there's a limit to how many they can hold. Packets need to be processed quickly; otherwise, if buffers fill up, it could cause issues. [11]

Interrupts can be divided into vectored and non-vectored interrupts. When a non-vectored interrupt occurs, the CPU always goes to a standard interrupt service routine (ISR). An interrupt handler, or ISR, is a software code the CPU executes in response to an interrupt. This ISR checks each device to identify the interrupt source. If multiple devices cause interrupts, they are addressed sequentially within the same ISR. In the case of a vectored interrupt, the CPU is directly informed of the interrupt and its source. As a result, vectored interrupts are more responsive than their non-vectored. The CPU, interrupt controller, and devices are the hardware components involved in interrupt handling. [11]

When an interrupt occurs, its signal is sent to the interrupt controller(s) either through the system bus or specific interrupt lines, like in Figure 4. In ARM architecture separate interrupt lines connected from each IP to the interrupt controller form a shared peripheral interrupt (SPI) vector [12]. Typically, each device only sends an interrupt request to the controller via one line or the system bus. This means that all interrupts from a particular device converge on a single

line. Such an arrangement simplifies hardware design. To identify the source of the interrupt, the interrupt service routine must check the CSRs - seen in Figure 4.

2.2.2 Real-Time Operating System

The operating system (OS) is an optional software component in an embedded system. It is not required for all embedded devices. The OS can run on any processor architecture it has been ported to. The OS sits on top of the hardware or device drivers in the software stack. [13]

The OS in an embedded SoC provides an abstraction layer between hardware and software and manages hardware and software resources. By running device drivers and hiding hardware implementation details, the OS facilitates the development of software and applications that operate on top of the operating system. [13]

The kernel is the core component of the OS that contains key functionality to manage resources through process, memory, and I/O system management. Process management refers to how the OS handles scheduling, switching between, and tracking multiple processes or tasks running on the SoC. This includes managing interrupts and errors triggered by different sources. Memory management refers to how the OS allocates memory space between processes. This ensures security for critical system regions and efficient utilization. I/O system management refers to how the OS manages sharing I/O devices between processes, including access control and allocation. [13]

Compared to general-purpose operating systems, real-time operating system (RTOS) better meets the needs of an embedded system, where timing and determinism are important [13]. The main distinction is that tasks executed on RTOS are guaranteed to meet their execution deadlines, making them predictable or deterministic [13]. RTOS are typically lighter and faster, making them ideal for resource-limited environments [13]. Standard operating systems used in embedded SoCs include FreeRTOS, Zephyr, and Linux [11].

2.2.3 Application Programming Interface

At the highest application level, user programs invoke OS application programming interfaces (APIs) to utilize underlying drivers and hardware without needing to directly access registers or handle interrupts. An application may invoke transmit/receive API functions for the UART, causing lower-level drivers to, for instance, write data to the UART IP and then instruct it to begin transmission with a certain baud rate determined by another write operation. This enables portable application software that is isolated from the specifics of the SoC hardware implementation. [11]

3 SOC TEST METHODOLOGIES

The design of a complex ASIC SoC involves creating models of the system at different levels of abstraction. As the design progresses, these models are transformed and refined until all the details required for silicon manufacturing are available. This progression through different models representing increasing levels of implementation detail is referred to as the SoC design flow. Effective use of Electronic Design Automation (EDA) tools is crucial to manage the complexity inherent in SoC designs. EDA tools automate the translation of models from one level of abstraction to the next. [7]

At the highest level, the system model captures the desired functionality and specifications. Lower-level models increase implementation detail as the design progresses while preserving consistent functionality. Verification at each stage ensures conformity with the preceding higher-level model. The final physical model contains complete information on how to manufacture the physical silicon chip. This multi-level design flow enables complex SoC implementation through stepwise refinement. [7]

To avoid potentially catastrophic and costly errors in complex SoC, complete and rigorous verification, software testing, and validation are required throughout the SoC design flow, accounting for over half of the total effort and cost of SoC design projects [1]. Finding and correcting bugs late in the product development cycle or after a product has been manufactured can cause massive financial costs [1]. Most of the hardware verification is performed on RTL hardware models written in hardware description languages (HDLs) such as VHDL and SystemVerilog, where hardware behavior is represented by bit and cycle-accurately modeling combinational and sequential logic of digital circuits [7].

Modern SoCs integrate many hardware components and software elements. Verifying these elements individually and collectively as an integrated system is crucial to ensure correct functionality. Various verification methods are employed, including simulation, emulation, FPGA prototyping, and formal verification techniques pre-silicon. Once the actual silicon is produced, further post-silicon testing and validation is carried out. [1]

Verification and validation are critical yet distinct phases that ensure the final product’s quality, functionality, and performance. Verification is a quality assurance process to ensure that each development phase’s outcomes meet the conditions and requirements initially specified for that phase. It focuses on the "building the product right" aspect, ensuring the system or component is designed and developed correctly at each life cycle step. The objective is to confirm that the system or component satisfies standards, practices, and conventions throughout its development, operation, and maintenance. [14]

Validation is evaluating a system or component at different stages or at the end of the development process to determine if it meets the specified requirements. This process aims to confirm that the system "solves the right problem" and fulfills its intended use and user needs. It provides evidence that the design, software, or hardware satisfies all allocated requirements after each life cycle activity. [14]

In System-on-Chip development, verification ensures that each hardware IP and subsystem within the SoC meets the design specifications and functional requirements. Various techniques, such as simulation, formal methods, and hardware emulation, confirm that the individual blocks and their interconnections are correctly implemented. The verification process also includes checking compliance with industry standards and practices at each development phase. Here, the focus is on confirming the design’s correctness, completeness, and consistency before it goes into fabrication. [5]

SoC Validation takes a more comprehensive approach, focusing on whether the SoC meets the end-user's needs and the system-level requirements. This process involves testing the SoC in an environment that closely mimics its final application to validate its functionality, performance, and reliability. Validation ensures that the components of the entire SoC system, with complex hardware/software use-case scenarios, satisfy the intended application and user expectations. [15]

So, while verification of an SoC ensures that each component and its interconnections are correctly implemented according to design specifications, validation ensures that the entire chip meets the system-level requirements and is fit for its intended purpose. Verification occurs throughout the development phases, whereas validation is generally performed towards the end or post-fabrication to confirm that the SoC meets all specified requirements and user needs.

3.1 Platforms

Before manufacturing the physical hardware, the verification platform runs a design description, expressed with HDLs, to ensure it works as expected. These design representations can be run on a variety of platforms. There are four primary techniques for executing a hardware design: logic simulation, simulation acceleration, emulation, and hardware prototyping. Each technique offers unique debugging options with varying advantages and drawbacks. These methods range in speed and debugging capabilities from the slowest and most flexible to the fastest but less flexible. [4]

3.1.1 Logic Simulator

Logic simulation, the main platform to execute and perform RTL verification, uses software simulators running on workstations to simulate the behavior of the hardware design. Logic simulators are event-driven, tracking signal values and queueing future events. They can use interpreted code for flexibility but slower speed or compiled code for faster execution but longer compile times. Simulation generally provides a flexible environment with good debugging capabilities but slower overall execution speed. [4]

3.1.2 Simulation Acceleration

The second type is simulation acceleration or co-emulation [16], which involves mapping the synthesizable portions of a design into custom hardware optimized for parallel execution. This takes advantage of the parallelism in HDLs. The remaining portions of testbench used for verification still run in a software simulator. The overall performance depends on the percentage of the design that can be accelerated. This approach removes events from the software simulator for faster evaluation in hardware. [4]

3.1.3 Emulator

The third type is emulation, which maps the entire design into a hardware platform like an field-programmable gate array (FPGA) for maximum performance [4]. FPGA can be viewed as a

larger silicon chip programmed to function similarly to a smaller silicon chip, with the advantage that the functionalities can be rewritten compared to ASIC. Since there is no constant connection to the workstation, the emulator can run at full speed without waiting for communication. Emulation provides the fastest execution but historically less flexibility for debugging [4]. However, modern hardware emulators have good debugging capabilities and RTL design visibility close to logic simulators [16].

3.1.4 Hardware Prototype

The fourth type is hardware prototyping, which constructs physical custom hardware or reusable prototyping boards to create a realistic hardware version of the final system. This allows engineers to have a working prototype available sooner by making trade-offs in requirements like performance and packaging. A common technique is using FPGAs instead of final ASICs for faster availability. In summary, the options range from pure software simulation to realistic hardware prototyping, with trade-offs in speed, debugging, flexibility, and cost. If a SoC design is too large for any of the available FPGAs and emulators, the only method to perform system-level verification may be to create an ASIC prototype of the SoC. [4]

3.2 Functional Hardware Verification

Functional pre-silicon verification is the main verification activity during and after hardware development and implementation. It is a continuous process that increases in maturity and complexity as the design evolves. [1]

Most industrial SoC designs include a mix of legacy and new IPs created in-house or from third-party providers. An IP verification team (in-house or third-party) verifies the IP in a standalone environment to ensure it functions appropriately on its own. IPs are then integrated into an evolving SoC model, and system-level verification is performed to ensure the IPs work correctly together. [1]

IPs are delivered to the SoC team as hard IPs (physical layouts) or soft IPs (RTL or netlists). More verification is required for hard IPs. Traditionally, IP verification involves testing the IP in isolation to build a robust portfolio of reusable IPs. Recently, there has been a push towards "right-sized verification" to test only target SoC use cases. So, IP verification should focus on IP use cases tied to SoC use cases. [1]

Rather than attempt exhaustive verification, a focus on verifying key intended use cases of the product helps direct verification efforts toward the most critical scenarios. Use cases usually exercise complex interactions between hardware, firmware, and software tailored to meet the target device’s performance, power consumption, and functional requirements.

SoC integration verification involves system-level simulation and defining use cases. The Purpose of the use case is to describe SoC configuration as if it would be initialized and configured in the actual product. Many use cases require hardware and software co-execution, which is infeasible in simulation. That is why complete use case tests are deferred until emulation and FPGA prototyping when the design matures, and faster execution speed is available. Verifying use cases end-to-end uncovers integration issues that may be missed by verifying components in isolation. [1]

Simulation is the primary methodology used to verify complex system-on-chip (SoC) designs in today's industry. The simulation-based verification process involves test generation, results checking, and coverage collection to verify the functionality thoroughly. [1]

3.2.1 Quality Measurement

One of the most critical aspects of functional verification is assessing its effectiveness or "goodness." This measurement is typically conducted using functional coverage metrics. Functional coverage allows to track what portions of the design's functionality have been exercised by the verification. A quantitative assessment of how well the design intent has been verified can be done by mapping the verification plan to specific cover points and cover groups in SystemVerilog's (SV) functional coverage language. It is essential to remember that there is never absolute certainty that the design is error-free; only a certain level of trust that the behavior is as expected can be achieved, or the design can be shown not to operate following the requirements. [5]

Functional coverage is not the only metric, however. While not a substitute for functional coverage, code coverage provides a complementary view by ensuring that every line of code, condition, and state has been exercised. The two together can be used to assess the verification status. [5]

3.2.2 Typical Test Environment

Driving test stimuli to the design is central to functional verification. The method of stimulus generation can vary depending on the complexity of the design and the verification requirements. Constrained-random test generation has become an industry standard for this purpose. It allows writing constraints that guide the random generation of test stimuli, ensuring that tests are random and meaningful. Constrained-random testing is beneficial for exercising edge cases that directed tests might not cover. [5]

In the context of System-on-Chip designs, test stimuli often need to be driven to multiple IPs and their interconnections. The Universal Verification Methodology (UVM) offers a structured approach to operating test environment [18]. By separating test generation from the simulation environment, UVM enables higher reusability and flexibility in driving test stimuli to complex SoC designs. This separation is crucial when IPs come from various sources or when some are legacy components, as it enables more straightforward integration and less duplicate work. When an IP block is embedded in a subsystem, existing UVM tests can be reused at the subsystem level with minimal changes. [5]

The UVM is a standardized approach for creating reusable verification components and environments. It defines a base class library and set of APIs in SystemVerilog that enable modular, scalable verification environments. The main goals of UVM are to improve verification IP (VIP) interoperability across tools and teams, reduce rewrite costs for new projects, and enable easier reuse of verification components. It provides a methodology to create modular verification environments using packages, components, sequences, and other testbench elements. Environments are structured into modules like agents and scoreboards. [17]

In the UVM framework, VIPs are reusable and modular components that facilitate efficient and effective stimulus generation for the Device Under Test (DUT) [18]. For instance, the AXI VIP is invaluable for verifying complex AXI protocols in SoC designs [19]. The architecture

of such VIPs includes several essential UVM components like the sequencer, driver, and monitor, all encapsulated within an agent [17].

In functional verification, the testbench and Device Under Test (DUT) are two main entities that interact through multiple VIPs, as illustrated in the example Figure 5. The testbench employs three different AXI agents for specific roles in this example. The first AXI Stream agent feeds the test input stimulus into an AXI VIP interface connected to one end of DUT's point-to-point interconnect. The second AXI Master Agent is responsible for the runtime configuration of the hardware by writing to CSRs and configuring the DUT and the internal IP blocks to suit specific use cases. The third AXI Stream Agent receives input from the opposite end of the DUT's point-to-point interconnect. This agent also compares the DUT output to reference data to determine a pass/fail status and whether the DUT behaves as expected.

The AXI VIP interface simplifies the process of stimulus generation by managing low-level operations and facilitating communication with the DUT via its AXI bus. The VIPs play a dual role in a self-checking testbench setup. First, they drive a sequence of transactions based on the functional specifications of the DUT, which may include various use cases, boundary conditions, and stress scenarios. Second, they monitor the DUT outputs, comparing them to expected or "golden" reference data.

A VIP serves an essential function by translating high-level test stimuli into pin- and cycle-accurate signals through bus function models (BFMs). Similarly, BFMs convert the DUT's output back into transaction-level data for analysis. If a mismatch is detected, debugging can proceed at the signal or pin level, enhancing the efficiency of the verification.

The primary advantage of a self-checking testbench is its ability to autonomously identify mismatches between the expected behavior and the DUT's output. When a mismatch is detected, the testbench provides immediate feedback, often specifying the input conditions and the nature of the discrepancy. This feature significantly accelerates debugging, making it more reliable than manual checks. [3]

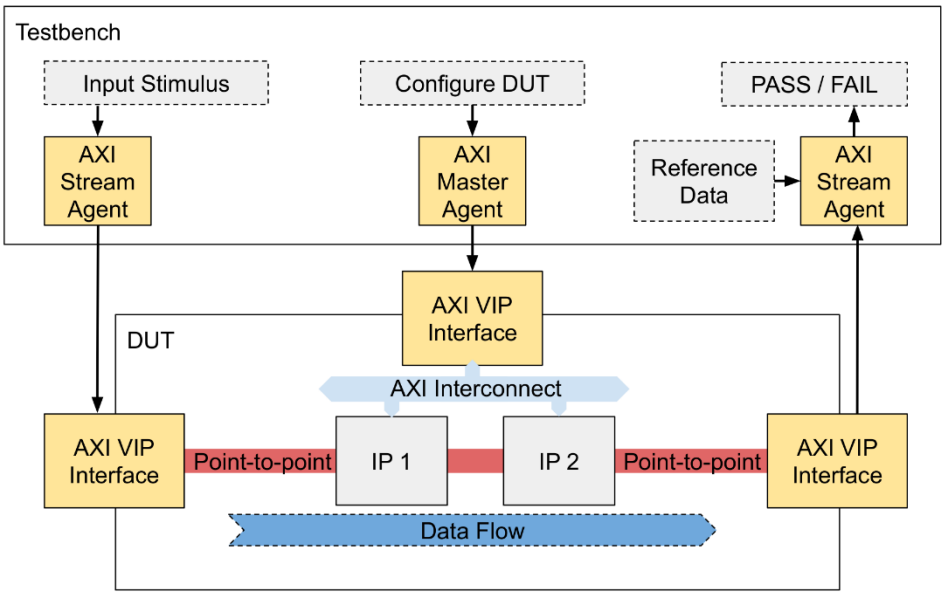

Figure 5. Example VIP Usage in a Testbench.

3.2.3 DUT Configuration with UVM

The UVM defines a standardized testbench structure with pre-determined phases coordinating the testbench components. One key phase is the configure phase, which programs the device under test (DUT) for the specific test case. For example, the test case may target one of the SoC's use cases. During the configure phase, the testbench uses agents like an AXI master in the middle of Figure 5 to write register and memory values that set up the DUT for the desired functionality [17], which would, in a complete SoC be done by the software running on CPU as discussed in section 2.2.

The following main phase then stimulates the DUT with transaction sequences tailored for the test case, simulating real-world stimuli generated by components such as an AXI stream agent. Monitor components like another AXI stream agent on the read interface concurrently capture outputs. In this way, the UVM methodology structures the testbench first to configure the DUT to the test mode, then apply stimuli and monitor outputs accordingly. The phased approach coordinates components and enables configuration, stimulus, and checking capabilities for comprehensive verification. [17]

The UVM Register Abstraction Layer (RAL) provides a standardized model for representing and accessing registers and memories in a DUT. The RAL creates register and memory objects to manipulate DUT's register values, including methods for reading and writing. The RAL model manages all the details of driving the necessary cycles on the actual interfaces.

A key capability of the RAL is enabling backdoor accesses that bypass the physical interfaces and access the DUT's registers directly. For example, the peek() and poke() methods instantly read and write register values without consuming simulation cycles. This backdoor access is handy for fast configuration during the configure phase of a UVM testbench. Testcases can call backdoor methods to set the DUT registers and memories in zero time before the stimulus for functional verification begins. [17]

Backdoor accesses also enable monitoring by peeking at register values anytime without interfering with regular operation. Testcases can peek at status or result registers to monitor the DUT state during long stimulus sequences. In summary, the backdoor capabilities of the UVM RAL enable high-speed configuration and monitoring of the DUT state. [17]

3.2.4 SystemVerilog DPI-C

The SystemVerilog Direct Programming Interface (DPI-C) enables interlanguage communication between C and SystemVerilog. With DPI-C, C function calls can be done directly from the SystemVerilog environment and vice versa. When a C-program, including DPI-C header file, is compiled into a shared library, it can be loaded into an RTL-simulator. [20]

Functions implemented in C can be declared in SystemVerilog code using import "DPI-C" declarations, which are then referred to as imported tasks and functions. Conversely, tasks and functions implemented in SystemVerilog can be declared using export "DPI-C" and are then callable from C. [20]

In SystemVerilog, the imported or exported functions can be declared anywhere where typical SystemVerilog functions can be, such as in a package. Both imported and exported functions have somewhat limited argument types, including types like void, byte, shortint, int, longint, real, and shortreal. The SystemVerilog functions to be called from C are declared as "extern" in C. In addition, the DPI-C header file must be included in the compilation. There is

a separation between pure and context functions. If a C function is imported to call exported SV functions or access SV data objects, it should be imported as a context function. [20]

The extensibility of testbenches using SystemVerilog's DPI-C capabilities is limited only by the user's imagination. It can be used, for instance, to extend the UVM framework's ability to communicate with SystemC [21] or to connect a design's reference model directly to the UVM environment [22], [23].

3.2.5 Challenges in Verification

The complex interactions between different IPs and the system-level behavior that results from these interactions make SoC verification a complicated process. More critically, low-level software is becoming increasingly crucial in hardware IPs' runtime configuration and functioning [24]. Because of this operational dependency, software availability also affects SoC integration testing. In the historical development waterfall, the software is tested only after the hardware is complete [1]; hence, actual software cannot be used for runtime configuration of hardware during functional verification.

Hardware-software co-verification is often necessary to capture the complete system behavior. This co-verification is generally not feasible in pure logic simulation due to performance limitations and is usually deferred to later stages of validation involving emulation or prototyping [5]. This necessitates the development of an environment that permits software development and execution on early hardware models, preferably already at the IP level.

3.3 Software Testing

Ensuring thorough and accurate testing of embedded software is crucial. Given the complexity of the system in which embedded software operates, flaws may expose to significant risks or financial losses in the event of any delays [25]. SoCs usually use CPUs with unique instruction set architectures (ISAs), like ARM [7], which differ from those in personal computers where software development commonly occurs. This makes testing embedded software more challenging compared to testing general-purpose software. Completing software testing as an independent task is essential before integrating new hardware and software for a seamless combination. Although there are various test methodologies and testing levels in software testing[26], like there is in hardware verification, this thesis will narrow its focus to unit testing, the most basic level of software testing.

A software unit test (UT) is a code designed to validate a specific behavior or functionality within the software under development. Typically developed parallel with the production code, unit tests are not included in the final software product. These tests create an isolated environment for executing a certain action on a code unit and verifying the result. The objective is to keep these tests short, simple, and independent of each other to avoid test coupling. [26]

Unit testing forms the foundation of a multi-level verification process, serving as an early stage for identifying issues. Detecting bugs at the unit testing stage is particularly cost-effective and sets the stage for subsequent testing levels, ensuring system reliability. [26]

A framework provides a structured environment and tools for writing, running, and reporting unit tests. It offers a foundation upon which individual tests can be built and executed, streamlining the process of verifying code functionality and quality. One prominent example

of a unit testing framework is Google Test [27], designed for C++ programming and can be applied, for example, to software API layer testing.

Thorough software testing necessitates having sufficiently mature hardware models and prototypes to run on. The sooner the device driver can be tested alongside the actual hardware model it is intended to control, the better. Hardware/software co-verification is vital since the complete functionality depends on the intricate interaction between hardware and software elements [11]. Therefore, comprehensive unit testing of software cannot be performed before actual hardware or hardware models are available.

3.4 System-Level Validation

Validation activities in SoC development can be divided into pre-silicon and post-silicon stages. In the pre-silicon phase, designers have the advantage of extensive monitoring and control of the design. In contrast, the post-silicon step provides limited observability, complicating the process of error identification. As the project transitions from the pre-silicon to the post-silicon stage, the complexity and realism of potential errors increase, leading to higher costs for bug detection in the later stages. Therefore, early identification and resolution of problems are essential. These validation activities aim to ensure successful software and hardware integration, confirming that the SoC is functional and aligns with its intended purpose. [1]

3.4.1 Pre-silicon Validation

Emulation and FPGA prototyping are technically part of pre-silicon verification, but they form an essential bridge to post-silicon verification/validation. The RTL model is mapped to a reconfigurable architecture like an FPGA or emulator that runs hundreds to thousands of times faster than RTL simulation. This enables executing hardware/software use cases like booting an OS in hours. However, this speed comes at the cost of reduced controllability and observability compared to logic simulation. [1]

In simulation, any internal signal can be observed at any time. In FPGA prototyping, observability is limited to a few thousand internal signals that must be selected before generating the FPGA bitstream. Reconfiguring observability requires time-consuming bitstream recompilation. Thus, emulation and FPGA prototyping are used when the design is relatively mature, with stable functionality and debug observability. [1]

Recent pre-silicon validation platforms address observability and speed limitations [16], [28]. The primary distinction between emulation and FPGA prototyping is that emulators may have debugging capabilities comparable to simulators, but FPGAs have severely limited debugging capabilities [16]. On both platforms, a significant improvement in execution speed relative to simulation enables complete validation of complex hardware/software use cases [1].

3.4.2 Post-silicon Validation

Post-silicon validation involves testing an actual silicon chip, physical hardware, instead of a hardware model, like the RTL description. Before mass production, samples of silicon chips are debugged in the lab to validate the chip, focusing on functionality, timing, power,

performance, electrical characteristics, and physical stress effects. It is the final validation before starting mass production of the chip. [1]

Running tests at the target clock speed allows executing long use cases like booting an OS in seconds and exercising power management and security features. However, controlling and observing silicon execution is far more complicated than RTL simulation or FPGA/emulation models. Also, changing observability in silicon is impossible. [1]

Overall, post-silicon validation enables running complex software use cases to validate the design entirely. However, controllability and observability are severely limited compared to RTL verification. It is a critical yet challenging phase before releasing to high-volume manufacturing. [1]

3.5 Pre-validation

The purpose of pre-validation is to serve as an intermediate step between traditional validation methods such as hardware emulation and FPGA prototyping and the actual hardware-software (HW/SW) integration. Traditional validation activities are crucial for ensuring the reliability and functionality of a SoC design before it enters mass production. However, relying solely on conventional validation can introduce significant time and resource overheads. This is because, as most of the verification is completed and the first prototypes are created to run software on, the hardware is relatively mature at this point, making future changes to the hardware improbable. Challenges arise as the software meets the real hardware for the first time, necessitating extensive debugging to identify and correct hidden bugs during standalone hardware or software testing.

Three scenarios illustrate the influence of various validation strategies on project schedules. Figure 6 outlines a project timeline without an emulator or FPGA prototyping. In this case, HW/SW integration begins only after the physical chip is manufactured, allowing for only post-silicon validation. This method comes with considerable risks, including the complexity of debugging due to limited visibility into the physical chip. In worst-case scenarios, this approach could necessitate the re-spin of the project due to a functional bug in hardware [5].

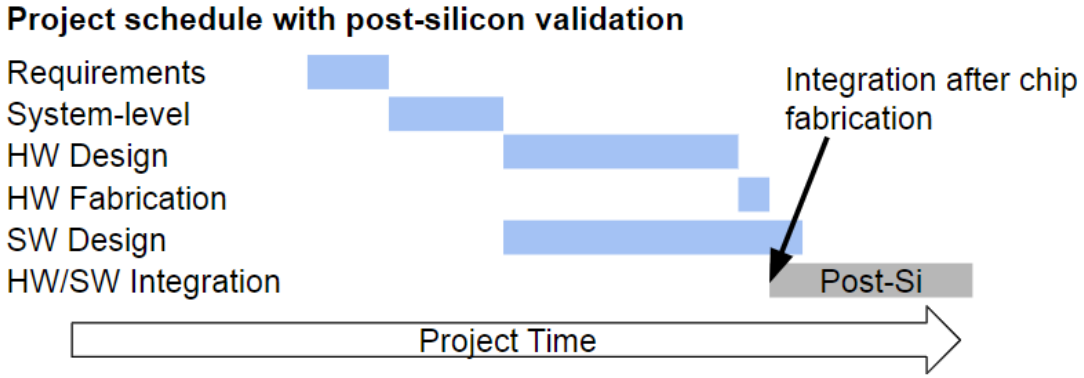

Figure 6. Project schedule with post-silicon validation.

The second approach, shown in Figure 7, incorporates traditional pre-silicon validation methods, such as emulation and FPGA prototyping. This enables a left-shift in the project schedule, accelerating time-to-market. However, employing these pre-silicon validation methods requires that the top-level architecture of the SoC be finalized and that the HW and SW are relatively mature. This restricts the flexibility to make design changes and compresses

the development time of both HW and SW components. A major issue with traditional pre-silicon validation, such as emulators, is that software and hardware are created independently and still meet at an unfortunate late stage of SoC development. A preferable way would be to integrate real software with early hardware models, involving collaboration among software developers, hardware designers, and verification engineers, improving system awareness and communication.

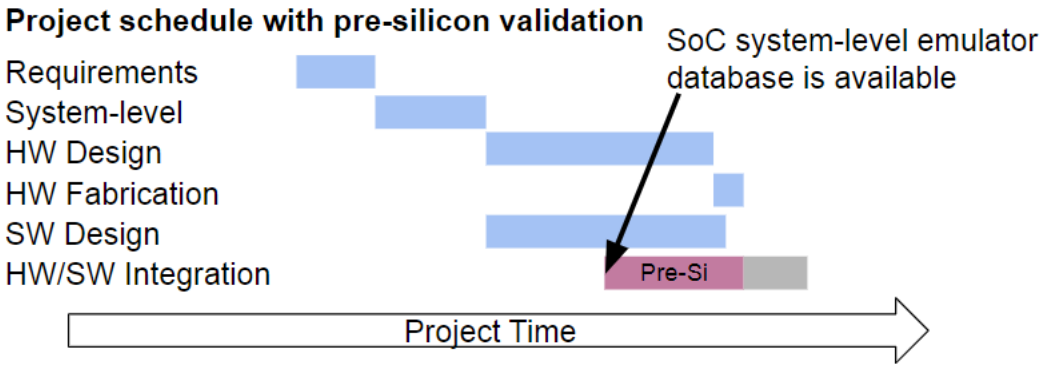

Figure 7. Project schedule with pre-silicon validation.

In contrast, Figure 8 presents a project schedule that leverages pre-validation. This allows for HW/SW integration to begin already at the IP block level, utilizing early HW and SW models. This results in a further left-shift in the project timeline, enabling an even quicker time-to-market. This allows for earlier detection of bugs and faster maturation of HW and SW, reducing the time spent on their development, allowing earlier traditional validation methods to occur, and thus allowing for earlier fabrication of the physical chip. The time spent on conventional pre- and post-silicon validation is also reduced, as most common bugs are already identified during the pre-validation phase. This allows engineers to focus on more complex validation scenarios with a mature HW and SW base.

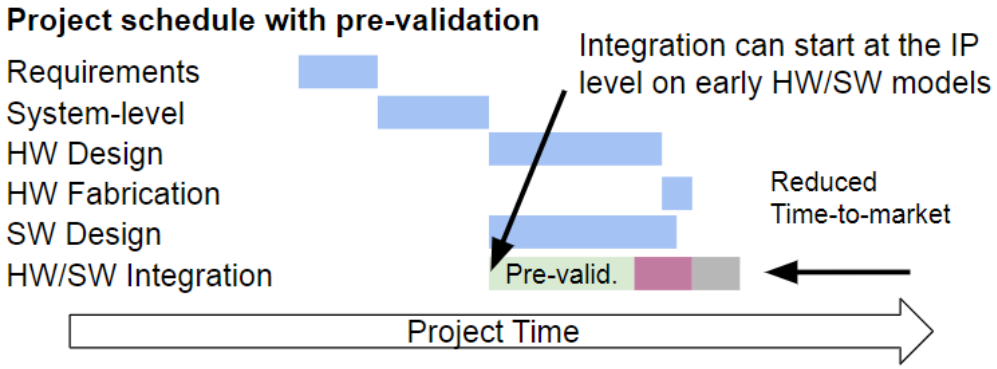

Figure 8. Project schedule with pre-validation.

Pre-validating an SoC by co-simulating HW and SW facilitates the early identification of potential issues and makes subsequent validation efforts more efficient. The practice aligns well with the "shift-left" approach, where testing and validation activities occur earlier in the development lifecycle. This proactive strategy results in quicker issue identification and resolution, reducing time-to-market, and lowering overall costs.

Performing pre-validation at the IP or subsystem level offers several advantages. One of the key benefits is the reduced compile time, especially when compared to hardware emulators.

While emulators can accommodate large designs, their slow compile times often negate their ability to process billions of cycles quickly, particularly for smaller designs [5]. Simulation-based pre-validation can be more productive in such cases, allowing quicker design iterations, including compilation, execution, and debugging.

3.6 Existing Solutions for Pre-silicon HW/SW Integration

Co-simulation is when two or more separate programs work together to create a complete simulation result. Each part of co-simulation specializes in a particular aspect of the system being studied, and by working together, they offer a more comprehensive understanding of the entire system. Co-simulation allows earlier HW/SW integration pre-validation, also known as co-verification. HW/SW co-verification, typically not performed until system-level validation, verifies the correct execution of embedded system software on the hardware design. [4]

Co-verification for pre-silicon validation can be achieved through multiple approaches, differentiated mainly by how the hardware and microprocessor are modeled and what execution engine is used. A comprehensive co-verification environment should offer the ability to control software and hardware and monitor the system. The concept extends to "virtual prototyping" and "physical prototyping" if the hardware is not the final fabricated chip. Virtual prototyping refers to simulating the hardware design as a software program, allowing to test how the software interacts with the hardware even before the hardware is physically built. Essentially, any method that verifies the software on a representation of the hardware that is not the final product qualifies as co-verification [4], meaning that pre-silicon validation involving both hardware and software can also be considered a form of co-verification.

Compared to real hardware, co-simulation gives access to all internal signals and registers of the simulated hardware block. Even with the physical ASIC in hand, co-simulation can replicate the problems encountered with the actual chip, making it easier to identify the root cause of failure [11]. However, simulating hardware has the inherent limitation of becoming increasingly slow as the number of events and state changes to track grows exponentially with the design size [8].

Several approaches for early hardware/software integration and co-simulation have been proposed. These approaches aim to facilitate early integration, reduce verification overhead, and uncover pre-silicon bugs. Co-simulation of a system’s hardware and software components creates a virtual platform (VP), which can be used even before implementing the hardware’s RTL description if a behavioral system model is used to model the hardware [5].

Most approaches presented in the literature utilize emulation or virtualization to speed up co-simulation, which allows the use of the host machine’s resources to run the guest system. The most frequent method for building a virtual platform and achieving high speed is to use the open-source CPU emulator QEMU to simulate software execution behavior [29]. SystemC is typically used for hardware modeling because it can describe both TLM and RTL models [30]. The transaction level is the most useful abstraction level for communication between software and hardware since it is precise enough to check functioning while not considering all the specifics that a gate level would, therefore achieving better performance [31].

A core challenge is the timing accuracy difference between software and hardware models. Software simulators focus on functional accuracy, while hardware models require cycle or RTL precision. This difference requires synchronization mechanisms between simulators. Another significant limitation is the long simulation run times caused by the number of triggered events, which grows exponentially with design size. [32]

3.6.1 QEMU

Quick Emulator (QEMU) is a versatile CPU emulator that allows the execution of applications and binaries compiled for different instruction sets. One of the technologies behind QEMU's functionality is dynamic binary translation (DBT), which is why it performs well. In DBT, the tiny code generator (TCG) translates target instructions into a machine-independent intermediate language called TCG ops. These are then compiled for the host architecture at runtime. This method of dynamic translation enables QEMU to support multiple Instruction Set Architectures (ISAs), such as ARM and RISC-V. [6]

QEMU can emulate ARM CPUs and their tightly coupled Generic Interrupt Controllers (GIC). This capability is particularly beneficial for simulating complex hardware systems that rely on ARM architecture. QEMU supports debugging with GNU debugger (GDB), allowing software to be debugged in the same manner as with a low-level debug facility such as joint test action group (JTAG) on actual hardware. In addition to CPU and GIC, QEMU can emulate various types of physical host hardware, such as disk drives, network cards, and audio interfaces. For network connectivity, QEMU can emulate network cards that share the host's network through network address translation [6]

3.6.2 Virtual Platforms

AMD, previously Xilinx, provides a customized version of QEMU that models the ARM-based processing system in their products to execute ARM instructions without needing real hardware. AMD's Vitis emulation platforms utilize QEMU to emulate ARM cores and co-simulate them with RTL or SystemC models of other on-chip components. [33]

Previous work demonstrated usage of Xilinx QEMU to emulate the ARM cores of the Zynq SoC and SystemC models for programmable logic [30]. Communication between QEMU and SystemC uses a custom library with remote-port protocol over transmission control protocol (TCP) sockets [30]. An open-source SystemC-TLM bridge [34] is used between the simulators and enables communication over AXI interfaces. This methodology boots Linux on the simulated SoC in minutes while maintaining accuracy for verification [30]. A UVM testbench architecture also applies constrained random stimulus to hardware and software interfaces, including TCP/IP configuration [30]. The approach enables pre-silicon verification and debugging of tightly coupled hardware/software SoC design of Zynq FPGA SoC [30].

VPSim is one of the virtual platforms developed for early hardware/software integration of embedded systems, which integrates QEMU into SystemC, providing CPU and peripheral models as TLM modules while leveraging QEMU's dynamic binary translation for speed [35].

Another virtual platform, QBox or "QEMU in box," integrates the QEMU into the SystemC simulation environment as a standard SystemC module to facilitate transaction-level interfacing to external hardware components [29]. QBox allows QEMU's efficient simulation of complex IPs like CPUs in the SystemC virtual platform simulation context. Comparing QBox to an alternative approach called QEMU-SC [36], where QEMU runs the simulation and SystemC models are incorporated as peripherals, QBox is better suited for heterogeneous multicore simulation [29]. QBox is utilized, for example, in Nvidia's NVDLA to simulate its hardware model in SystemC while running software on a QEMU ARM processor model [37].

It is demonstrated that by connecting QEMU to SystemC hardware models and enabling simulating software like drivers and applications on QEMU's virtual ARM machine alongside SystemC or RTL hardware models, hardware architecture and software of a GPU SoC can be rapidly explored and verified [38]. Alternatively, by connecting QEMU running server software with a logic simulator and linking the virtual machine's (VM) virtual PCIe/NIC devices to simulation bridges with identical interfaces as the hardware platform, enables full-system simulation of actual server software and bit-accurate hardware without modifications, providing rapid iteration, full visibility, and the ability to simulate complex systems [39].

Techniques to integrate heterogeneous VPs built with different languages/simulators are proposed to enable full system validation of independently developed software. Inter-process communication via shared memory exchanges data between QEMU and commercial ARM Fast Model (AFM) accelerated VPs. The shared memory-based techniques facilitated the creation of full system VP by combining subsystem VPs for software validation. [40]

An alternative to QEMU is to use an instruction set simulator (ISS) to model the architecture of the target CPU. Integrating an ISS to logic simulation using the SystemVerilog DPI is proposed. ISS is a program that interprets machine code, such as ARM, on a host machine's x86 processor. Combining the software debugger provided a virtual prototype for early software testing, easier hardware bug reproduction, and rapid iteration. The DPI integration enabled effective co-verification by linking logic simulation to a C/C++ ISS and existing software tools. [41]

A virtual prototyping platform based on open virtual platforms (OVP) ISS processor model and SystemC with TLM showcased benefits in concurrent hardware and software co-design and co-verification of Multiprocessor SoCs. The processor model enabled early software development before completing RTL. The platform allowed to explore architectures, facilitating hardware/software co-design, early software testing, architecture optimization, and functional verification. [42]

In addition to open-source virtual platforms, commercial solutions may also leverage QEMU for co-simulation. For example, Aldec provides a QEMU bridge that connects their Riviera-PRO RTL simulator to QEMU. This facilitates the co-simulation of SoC designs by running software on QEMU while the programmable logic is simulated in Riviera-PRO. [43]

Veloce Hycon from Siemens EDA is another commercial virtual platform that enables early software verification and validation alongside hardware models. It integrates configurable virtual platforms built using abstraction techniques like ARM Fast Models (AFMs), QEMU, and SystemC TLM with RTL simulation and emulation engines. Hycon has the advantage of mapping hardware models to emulator while employing higher-level models to represent the CPU, considerably improving speed compared to simulation-based or complete hardware emulation execution. [28]

In contrast, Synopsys also offers a hybrid prototyping solution that combines Virtualizer virtual prototyping with HAPS FPGA-based prototyping. One of the key features is the high-performance HAPS Universal Multi-Resource Bus, which transfers data between the virtual and FPGA-based environments. The hybrid approach leverages strengths from both virtual and FPGA-based prototyping, offering efficient debug capabilities through the Virtualizer environment and high-performance, cycle-accurate execution through FPGA-based prototyping. [44]

Cadence also offers a Hybrid solution that combines high-performance transaction-level models running on the Cadence Virtual System Platform (VSP) with register-transfer level (RTL) models running on the Palladium platform, promising software execution speeds that outperform standalone emulation and, in some cases, FPGA prototypes. [45]

3.6.3 Host Execution Techniques

Another approach for early software/hardware integration is host execution techniques, in which the software targeted for another ISA is run on the host machine's ISA. They cannot execute software with the target instruction set, resulting in a lack of accuracy and real-world system feel compared to target ISA execution, for example, in QEMU-based approaches. One method presented is based on Accellera Portable Test and Stimulus Standard (PSS) and Zephyr RTOS to co-develop IP hardware and associated software drivers. The approach is demonstrated on a DMA IP example, developing a modular driver package and IP-level tests. Reuse at the SoC level avoids rewriting drivers and simplifies system test creation. [24]

Another paper discussing host execution methods enables hardware/software co-verification by integrating software execution with UVM components. A custom DPI library maps software reads/writes to UVM transactions that drive VIP sequence items. A host execution library intercepts software memory accesses and redirects them through DPI to the UVM environment, which facilitates running unmodified software while leveraging UVM features like constrained random stimulus and coverage metrics. The technique avoid expensive instruction set simulators (ISS) or virtual platforms. The methodology successfully uncovered hardware bugs from result comparison and hardware/software interaction issues. [46]

Another work specifically interested in hardware verification and developing alternative methods to UVM enables hardware/software co-verification of designs with a Very High-Speed Integrated Circuit Hardware Description Language (VHDL) RTL model and software component on a host with open-source tools to avoid licensing restrictions. The VHDL "virtual device" model in GHDL simulator communicates with software over Unix pipes. This facilitates complete system testing before hardware availability. A fault injection platform case study achieved high code coverage, demonstrating the virtual device concept provides effective co-verification with free tools. [47]

3.6.4 Summary

In summary, various effective approaches for early hardware and software integration and co-verification have been proposed in the literature and provided by EDA vendors. Key benefits include enabling early software development and collaboration among hardware/software teams, finding hardware/software interaction bugs pre-silicon, exploring architectures, and reusing tests across simulation and prototyping. Interest in using software to drive the DUT instead of traditional UVM tests is also shown.

4 CO-SIMULATION ARCHITECTURE

This chapter introduces the components of the co-simulation architecture developed for pre-validation. The entire SoC hardware model, including the CPU, could theoretically be simulated in a logic simulator, but this would be extremely slow and impractical because the CPU is often the most complex block in a SoC, dominating simulation performance. To accelerate the simulation, the developed methodology replaces the CPU, with a faster model to run the software stack on. This method of replacing the main processor with a quicker representation is consistent with the numerous approaches discussed in section 3.6.

Nokia's Target SoC Simulator (TSS) provides a patched version of QEMU with back-end modifications that allow adding a custom TSS-bridge virtual device to QEMU, enabling transaction-level communication between the QEMU and external hardware models with an easy C-language interface. The TSS-bridge device interacts with the hardware model through a TSS server acting as the interface between QEMU and the hardware model(s). The co-simulation architecture involves three key entities shown in Figure 9: QEMU to execute the SoC software, TSS Server to function as a link, and a logic simulator for the RTL-description hardware models. The Table 1 below provides an overview of the co-simulation components shown in Figure 9.

Table 1. Co-simulation components

Component	**Description**
QEMU	Emulator for ARM Cortex-A55 CPU and related components. Patched by Nokia's TSS to include a TSS-bridge for transaction-level communication.
TSS-bridge	Virtual device in QEMU enabling transaction-level communication with external hardware models via the TSS server.
TSS-server	Interface between QEMU and external hardware models. Routes transaction objects to destination endpoints.
RTL-simulator	Logic simulator for all hardware of SoC except the CPU, which is emulated by QEMU.
TSS-endpoint	C program connecting RTL simulator to TSS server's shared memory channel. Enables transaction-based communication.
TSS-transaction	Objects created by endpoints to specify parameters like target address and operation type. Routed by TSS-server to matching endpoints.

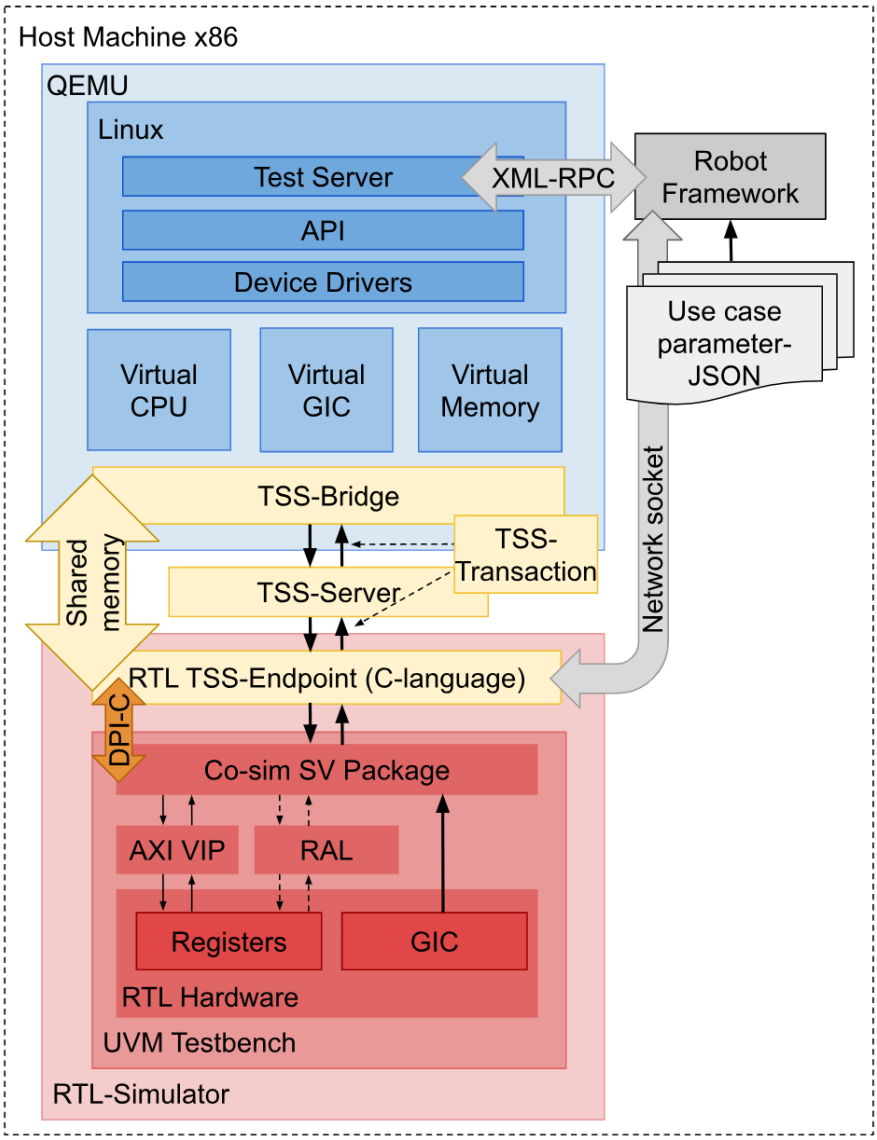

Figure 9. Co-simulation Architecture.

4.1 QEMU

QEMU aims to emulate the ARM Cortex-A55 CPU and other closely related components like memory and interrupt controller to run the target software stack fast in the co-simulation environment. Compared to the basic upstream QEMU, the key difference is a patch provided by Nokia's Target SoC Simulator (TSS), which includes a TSS-bridge device to enable TLM communication with external components in the simulation environment through TSS-server.

In co-simulation, the QEMU is configured to boot up with the unmodified binaries of the target software, including the Linux OS, device drivers, API layers, and software test servers to facilitate communication between API and external applications. A startup script configures the QEMU environment appropriately by loading the correct kernel, root filesystem, CPU parameters, memory size, and TSS-bridge devices. The script also enables port forwarding from the guest machine (QEMU) to the host machine, allowing the host access to QEMU through secure shell (SSH). This allows communication with the software from the host machine.

This architecture allows the running of unmodified software on QEMU as if running on the actual silicon while integrating external hardware models through the tss-bridge device. The software running inside QEMU is fully unaware of the simulation environment and is not affected by the communication mechanisms implemented in TSS. Because of QEMU's dynamic binary translation, software execution achieves excellent performance, booting a full Linux OS in a couple of minutes.

The advantage of the TSS-based server architecture, which is not utilized in this thesis, is that for SoC architectures with multiple processors, multiple QEMU instances can be launched to emulate each CPU. In the co-simulation architecture of this thesis, only one instance of QEMU is launched for the Cortex-A55 CPU. However, in theory, for a SoC with an ARM A55 as the primary application processor and an ARM M3 utilized internally in an RTL subsystem, two QEMU instances would be defined for the A55 and M3, each of which would execute the relevant software. The remaining SoC hardware outside these processors would run in the RTL co-simulation.

This approach would provide an efficient simulation environment for complex SoCs containing multiple CPU architectures tailored to separate roles. Each processor's software could run on its corresponding instance of QEMU, which would synchronize with the RTL simulator via TSS to co-simulate the heterogeneous multicore SoC in its entirety.

4.1.1 TSS-bridge

The TSS-bridge device enables routing transactions from the QEMU to external hardware models through the TSS server. QEMU handles some address ranges internally, like those used for booting Linux and loading programs. For an address space to be visible to TSS, a TSS-bridge device must be declared when launching QEMU. The bridge devices are mapped into the simulated system bus, so accesses to those ranges are forwarded externally. The bridge device also provides interrupt generation and synchronization support with hardware models.

TSS-bridge transparently converts QEMU software register accesses into TSS transactions and forwards them to the TSS Server and further to the hardware model at the endpoint. The tss-bridge is initialized with a specific address range during QEMU startup. Any software register access to addresses within this range is captured by the bridge device and forwarded as TSS transactions to the server.

Multiple non-contiguous bridge devices can be declared in simulated system bus, allowing flexibility in routing different address regions. A single large bridge device can cover the entire SoC I/O space for a full SoC simulation, making all IPs reachable through transactions to addresses within that range. The memory regions bridge devices claim must match address ranges declared in TSS endpoint models. In the co-simulation architecture of this thesis, a single bridge device covers the entire address space of the SoC's RTL hardware models.

4.1.2 Co-simulation Device Tree

QEMU relies on a device tree blob (DTB) file that describes the hardware configuration to boot with proper configuration [6]. A device tree is a systematic approach to representing hardware [48]. It consists of device tree source (DTS), and device tree include (DTSI) files [48]. When these source files are compiled, they produce a device tree blob (DTB) [48]. The DTB describes

the hardware configuration and can be loaded by software such as QEMU and Linux to recognize the system's devices without hardcoding.

The device tree enables automatic device discovery and configuration during system boot. Nodes in the device tree source files are defined for hardware IPs and other devices in the system, like CPU cores, memory, and interrupt controller. These nodes contain compatible IDs, register addresses, interrupts, and other properties. The kernel matches nodes to device drivers based on compatible strings during boot. When a match is found, the kernel calls the driver's probe callback, passing resources like register addresses from the matching node. The driver then initializes the device based on this information. In this manner, the device tree allows the configuration of devices without hardcoding addresses in drivers. [49]

To enable co-simulation, DTS files were created to describe configurations with different combinations of IP blocks and subsystems included. As shown in Figure 10, the process starts by decompiling the SoC DTB to obtain the DTS file. This DTS file is broken down into smaller DTSI files, representing various hardware IPs and subsystem nodes. These DTSI files can be included in different DTS files to suit the specific requirements of co-simulation scenarios, be it IP-block level testing, subsystem testing, or full SoC testing to register the drivers similarly to the physical chip. This device tree-based approach allows booting the same software stack on physical hardware and in a virtual simulation environment. Only minor changes are needed in the device tree sources to include simulation IP block nodes.

QEMU provides a feature to extract the initial DTB using the -dumpdtb switch. This switch allows obtaining a compiled version of the device tree containing all QEMU's internal virtual devices [6], the starting point for further customization. Once the initial DTB is obtained, it can be decompiled back into its textual DTS presentation using a device tree compiler (DTC) [50]. This decompilation allows for the editing and inclusion of additional hardware nodes presented in DTSI files.

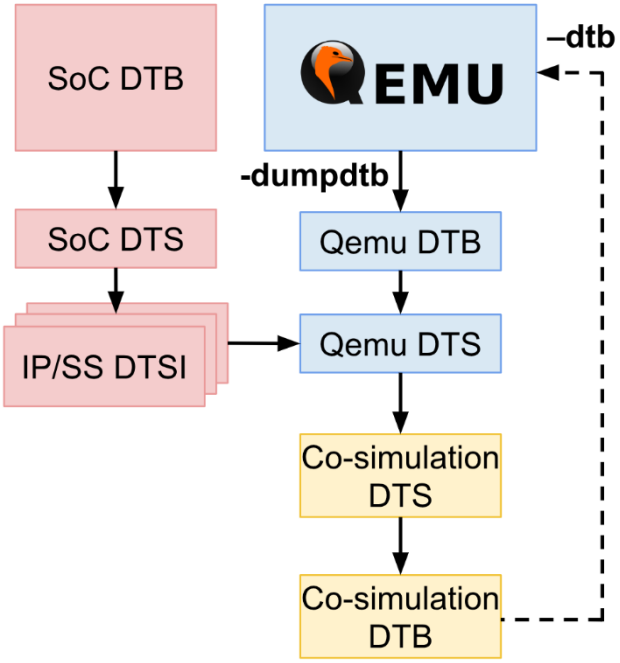

Figure 10. Device Tree Setup Flow.

With a DTS presentation of the QEMU virtual hardware, DTSI files corresponding to individual simulated RTL IP blocks are integrated into the decompiled DTS file. These DTSI

files contain nodes that describe the IP blocks and their configurations, like register addresses and interrupts. Including these DTSI files enables the creation of custom hardware configurations depending on the simulation or testing requirements.

Figure 11 presents an example DTS for co-simulation that includes a subsystem defined in a separate DTSI file. The “cosimulation.dts”-file presents the structure of what a dumped QEMU device tree might look like after including a DTSI file, serving as the root description for the hardware system. The root node of the device tree is denoted by “/”, within which multiple child nodes and properties are defined [48]. The model property is set to "linux,dummy-virt," indicating the system’s model used by QEMU [6]. It also shows “cpus” and “memory” nodes, which must be present in every device tree’s root node [48]. For clarity, other QEMU’s virtual devices are omitted from the example root node.

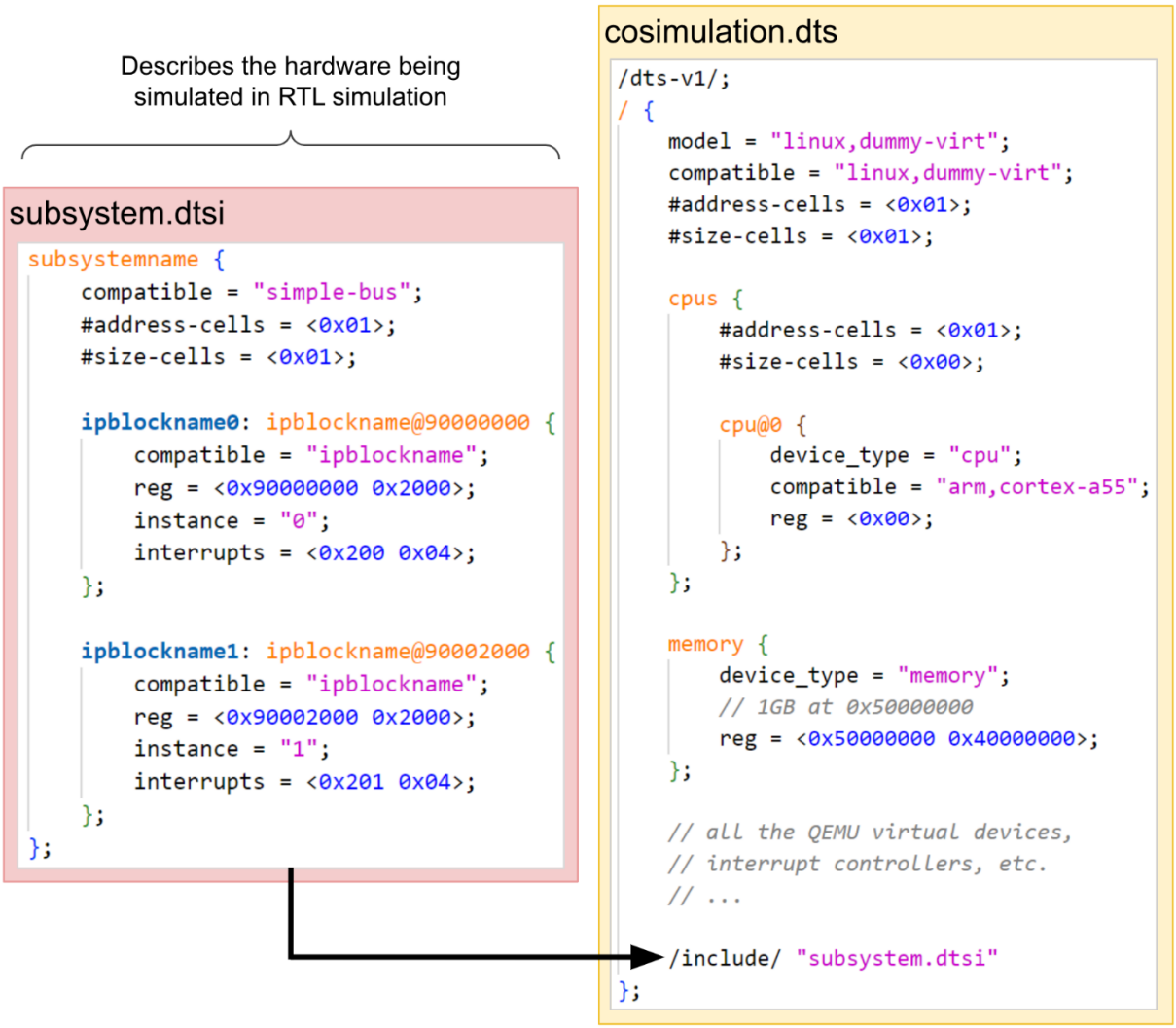

Figure 11. Co-simulation Device Tree Example.

Addressing in the device tree is managed through #address-cells and #size-cells properties, which can be used in a device node that has child nodes. Detailed descriptions [48] for properties can be found in Table 2. In this example, both are set to one 32-bit cell, which means that each address and size specified in the reg property of a child node will consist of one 32-bit value. The cpus node defines the CPU configuration of the system, specifying that the CPU is of type ARM Cortex-A55. The reg property within the cpu@0 node indicates that this is the

first CPU instance. The memory node specifies that the system has 1GB (0x40000000 in hexadecimal) of memory starting at the address 0x50000000. [48]

The subsystem DTSI, on the other hand, first defined the node name for the subsystem, “subsystemname”. The compatible field now contains the value “simple-bus”, which indicates that this subsystem acts as an internal I/O bus, and child nodes on the bus can be directly accessed [48]. The subsystem node has two child instances of an example IP block, ipblockname0 and ipblockname1. The compatible property for both is set to "ipblockname," indicating the type of IP block, which the drivers use to probe devices accordingly [48]. Each instance has a unique base address and size defined by the reg property. An additional instance property differentiates between the two instances with values "0" and "1," respectively. The interrupts property specifies each IP block’s interrupt numbers and types [48].

Table 2. Example device tree properties

Property	Description
/dts-v1/;	Specifies the DTS version.
model	Defines the model of the SoC or system.
compatible	Indicates the compatible hardware or system for the given node.
#address-cells	Specifies the number of 32-bit cells used to represent the base address in the reg property.
#size-cells	Specifies the number of 32-bit cells used to represent the size in the reg property.
device_type	Specifies the type of device (e.g., "cpu" or "memory").
reg	Specifies the base address and size of the memory region or device.
cpus	Node that contains CPU definitions.
cpu@0	Specifies the first CPU instance and its properties. (e.g., ARM Cortex-A55).
memory	Specifies the memory available on the system.
/include/	Includes another DTSI file.
ipblockname0	Node specifies the first instance of an IP block.
ipblockname1	Node specifies the second instance of an IP block.
instance	Additional property to differentiate between multiple instances of the same IP block.
interrupts	Specifies the interrupt numbers and types associated with the device. (e.g., <0x200 0x04> defines the interrupt number as 512 and active high level-sensitivity).

After incorporating the necessary DTSI files, the modified co-simulation DTS file is recompiled back into a DTB file with DTC, as seen in Figure 10. This new DTB file now contains the updated hardware configuration for co-simulation, including the added IP blocks that will be run on RTL simulation. At QEMU startup, the DTB file is loaded using the -dtb switch. This provides QEMU with the device tree describing the virtual hardware, including any TSS endpoints to be co-simulated. QEMU and Linux then initialize based on this hardware description without hard-coded dependencies [49].

4.2 TSS-server

The TSS-server acts as the interface facilitating communication between the QEMU and external hardware models in the developed co-simulation architecture. It leverages a transaction-based approach using TSS transactions to enable interaction between different simulation endpoints by routing transaction objects to their destination endpoints.

In a TSS architecture, there is a possibility to use multiple endpoints representing different hardware models and the QEMU running the software stack. Each endpoint runs as a separate Linux process and declares TSS objects called endpoints to announce its ability to respond to transactions targeting a given address range.

To initiate communication, endpoints create TSS transaction objects specifying parameters like the target address, data, and read/write operation. The endpoint then posts the transaction using the TssPostTransaction() API call. The TSS Server scans all declared endpoints to find a matching address range and queues the transaction to that endpoint's priority queue.

The receiving endpoint is notified of the incoming transaction and handles it accordingly, performing reads or writes. It then indicates transaction completion, so the initiator can resume. This transaction workflow facilitates inter-process communication between different simulation endpoints.

The TSS Server manages transactions between QEMU and hardware models, enabling hardware-software co-simulation in which software runs normally as in actual silicon while models achieve full observability. The transaction-based communication model makes adding new simulation endpoints straightforward and is the optimal solution for early integration testing [32].

The co-simulation environment enables full system simulation comparable to previous work [40]. QEMU, TSS Server, and RTL simulation communication occur via shared memory using standard C-language APIs. This architecture allows each component, like QEMU, TSS, and the RTL simulator, to execute concurrently on separate threads. QEMU efficiently emulates the target CPU through dynamic binary translation. Meanwhile, the RTL simulator provides cycle-accurate hardware modeling.

The shared memory communication path and multi-threaded execution produce a high-performance environment for the co-simulation of multiple simulation entities. The modular, server-based shared memory architecture also simplifies integrating new simulation endpoints and provides better performance than approaches involving inter-process communication through TCP/IP sockets [51]. In the co-simulation architecture of this thesis, only one endpoint is defined for the simulated RTL model, and one QEMU instance is used for the Cortex-A55 CPU.

4.3 RTL-Simulation

The logic simulator simulates the RTL models of all hardware components except the main CPU, which the TSS QEMU instance replaces in SoC-level simulation. Common commercial logic simulators were used to simulate the RTL models. Integrating the compiled C endpoint program with these simulators is straightforward through standard SystemVerilog DPI-C capabilities.

Figure 12 represents a side-by-side comparison of two distinct scenarios. The left-hand side illustrates a SoC RTL simulation setup where the main CPU is modeled in RTL and simulated. The right-hand side represents the co-simulation setup where the main CPU RTL model is

replaced by an AXI VIP master interface connected to the QEMU virtual platform, thereby enabling TSS transactions originating from software execution to propagate into the RTL simulation as if they were coming from the actual CPU model. At IP level co-simulation, if the IP-block's connectivity is implemented with the AXI standard, the AXI VIP interface would be directly connected to the IP's bus, bypassing any higher-level bus architectures and bus bridges. If it were a different bus standard, the AXI VIP could be simply replaced with another VIP available in the UVM environment.

In addition to the CPU, QEMU also emulates tightly coupled components like the GIC and memories. The RTL simulation monitors the GIC's shared peripheral interrupt signals and propagates changes to the virtual GIC in QEMU for accurate interrupt handling. In this manner, QEMU realistically emulates key components to run the software while the remaining hardware models are in RTL simulation.

The RTL simulation environment was mostly unchanged compared to a typical UVM verification setup. The key differences were the addition of DPI-C code to enable the C endpoint and the integration of this endpoint with UVM components. The C endpoint is an intermediary between the TSS server and SystemVerilog testbench to facilitate co-simulation. Mainstream commercial simulators were utilized successfully by integrating the C endpoint program, demonstrating the approach is portable across common logic simulation tools.

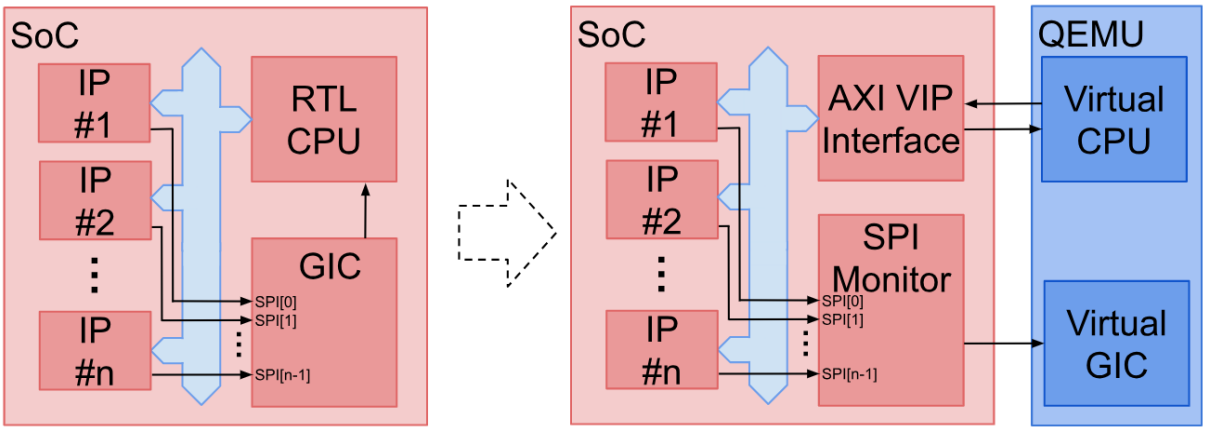

Figure 12. SoC level RTL DUT vs. SoC level co-simulation DUT.

4.3.1 TSS-endpoint

The C endpoint program developed in this thesis work serves three key functions to enable hardware-software co-simulation. First, it initializes the TSS endpoint and connects it to the shared memory communication channel provided by the TSS server. Establishing this endpoint allows the RTL simulation to participate in the transaction-based communication of the TSS framework, receiving transactions originating from software execution on QEMU.

Second, the C program propagates these incoming TSS transactions into the UVM testbench. The program contains logic to map TSS read and write transactions to appropriate SystemVerilog DPI calls. These DPI calls trigger AXI bus transactions or UVM register access, depending on the desired behavior. In this manner, the C code acts as an intermediary between TSS transactions and the UVM environment.

Third, the C endpoint program incorporates networking code to communicate with Robot Framework (RobotFW) over a TCP socket. This interface allows RobotFW to send control commands like "break" to control the RTL-simulator part of the co-simulation. When the C

program receives these string-based commands, it executes suitable actions, such as ending the co-simulation and allowing the UVM test to continue freely.

While the implemented socket-based networking approach enables proof-of-concept RobotFW integration, it has limitations. The required use of predetermined ports may cause conflicts in multi-user environments. A more robust containerized architecture could avoid these issues by isolating the environment. However, the current solution demonstrates the feasibility of integrating external test frameworks like RobotFW via the C endpoint to coordinate hardware-software co-simulation scenarios.

In summary, this thesis's custom C endpoint program bridges TSS transactions originating from QEMU software execution, UVM testbench components, and the external RobotFW test framework. It is mainly responsible for integrating RTL models through the SystemVerilog testbench into TSS QEMU for hardware-software co-simulation.

4.3.2 Co-simulation SystemVerilog Package

To allow communication between the UVM testbench and C code, the implemented C endpoint program requires SystemVerilog functions, which are defined in a package and included in the UVM testbench compilation. This package contains the necessary functionality to create the TSS endpoint from the SystemVerilog environment and to propagate TSS transactions from the C endpoint into the SystemVerilog environment and finally to the RTL DUT. For instance, functions to perform read and write operations from the C program are defined as exported DPI-C functions callable from C. Meanwhile, functions to initialize the TSS endpoint are imported from C to be called from the SystemVerilog.

A key responsibility of the package is implementing the tss_tick_loop() task shown in Figure 13. This loop facilitates continuous communication between the RTL model and the TSS server. The tick loop starts two concurrent threads: one that checks for incoming TSS transactions by calling the C function tss_dpi_tick() and another that monitors RTL interrupt signals and forwards any changes to QEMU. In between tss_dpi_tick() calls, the loop can break out of the loop or advance RTL simulation time, allowing design changes to be propagated.

SystemVerilog Package

```
task tss_tick_loop();
    fork
      forever begin // Register access fork
          tss_dpi_tick();
          if (breakLoop) break;
          if (freeRunning) #delay;
      end
      forever begin // interrupt monitor fork
          tss_check_irqs();
      end
    join_any
endtask
```

Figure 13. SystemVerilog co-simulation loop.

4.4 Co-simulation Startup

A series of steps unfold to start the co-simulation environment, as shown in Figure 14.

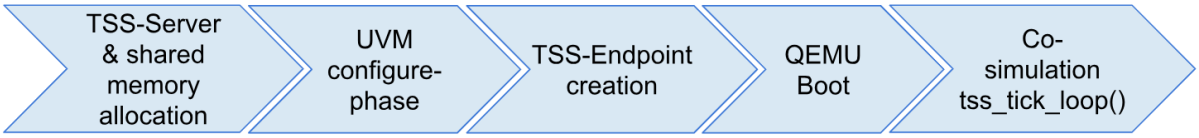

Figure 14. Co-simulation startup.

First, the TSS server is launched, setting up shared memory for communication among the server, endpoint, and bridge. Once the server is up and running, the hardware's RTL simulation and a UVM test are initiated. The UVM test goes on until it reaches the point where the UVM environment and RTL model are ready enough to receive incoming transactions, typically in the configuration phase. At this point, a C-function tss_dpi_init() is called from the UVM test through DPI-C to establish a TSS endpoint for the RTL-modeled hardware and to connect it to the shared memory.

Figure 15 shows the imported function declaration in the SV package and the function defined on the C-program side below, which gives a name for the RTL model and then creates an endpoint corresponding to the address and size defined in the QEMU TSS-bridge device definition. The base address and memory size are given as command line arguments to the UVM test, enabling the user to define endpoints with different size and address easily. In the case of subsystem co-simulation in Figure 11, for instance, the base address and size would be 0x90000000 and 0x4000, respectively, to cover only the subsystem address range. By limiting the endpoint only to cover the active logic, transactions to other areas not covered by the newly generated endpoint would be routed to a dummy memory model to avoid access to non-simulated destinations. Handling register accesses to non-simulated address ranges is an issue that has also been covered in the literature [52].

SystemVerilog Package

```
import "DPI-C" function int tss_dpi_init(
  int unsigned baseAddress,
  int unsigned memSize
  );
```

TSS-Endpoint C-language

```
int tss_dpi_init (unsigned baseAddress, unsigned memSize)
{
  TssBridgeOpen("RTL_MODEL_NAME");
  e = TssCreateEndpoint(EP_AP, baseAddress, memSize, EP_FLAGS);
  return 0;
}
```

Figure 15. TSS-Endpoint initialization function.

After initializing the C endpoint, it can receive transactions from the TSS Server through shared memory. As a final startup step, QEMU boots up, loaded with the software binaries and a TSS-bridge device mapped to the same shared memory region. QEMU is launched with a

separate script that defines key parameters like the target kernel, filesystem images, and devicetree relative to the RTL hardware models, as presented in Section 4.1.2.

At this stage, all the simulation components are connected to the shared memory facilitated by the TSS Server. To complete the co-simulation startup, the tss_tick_loop() task is called from the UVM test. This is when the RTL model is ready to handle incoming register access transactions and propagate interrupts to the QEMU virtual machine.

In summary, the tss_dpi_init() function establishes the TSS endpoint for the RTL model, while the tss_tick_loop() enables continuous communication between the endpoint and the TSS Server. Once the QEMU and RTL simulation endpoint are connected to shared memory, and the tick loop begins, co-simulation can begin with QEMU software execution interacting with RTL hardware models.

4.5 Communication

To enable accurate co-simulation, the communication between the software execution on QEMU and the RTL hardware models must be sufficiently realistic compared to how it would be between components on a real physical chip. This communication between the simulators involves register accesses, interrupts, and synchronization.

First, there must be synchronization mechanisms and options for simulation time advancement. Second, register accesses from software executing on the virtual CPU in QEMU must be communicated to the RTL model and back to QEMU for the hardware to interface with QEMU appropriately. Thirdly, interrupts enable the RTL model to notify software of events and mimic real-world asynchronous interactions.

4.5.1 Synchronization

The QEMU emulator and RTL simulator utilize shared memory allocated by the TSS server for inter-process communication (IPC). Synchronization between the software execution in QEMU and hardware simulation is handled through two primary modes in this thesis: lockstep and free running. Due to the high-level CPU model in QEMU, neither of these techniques is clock accurate. Early integration and software testing use cases do not require cycle-level precision since register-accurate TLM communication is ideal for early integration. The lockstep and free-running modes provide adequate functional synchronization without exact cycle matching.

In lock-step mode, the RTL simulator blocks and waits for the next pending transaction from QEMU before advancing simulation time. It processes one transaction at a time in order without overlap. This mode captures each hardware-software interaction in the bus waveform as a sequence of back-to-back transactions with no idle cycles.

Conversely, free-running mode allows the RTL simulator to continuously advance simulation time independently. It periodically polls the shared memory for pending transactions from QEMU using the same tss_dpi_tick() function. If no transactions are queued, the simulator simulates a predefined number of clock cycles before polling again. As shown in Figure 13, the free-running mode can be easily implemented on top of the lock-step mode by simply adding some delay between calls to the tss_dpi_tick() function that polls for transactions. The delay enables the RTL simulator to advance time in between the polling calls.

In addition to lockstep and free-running, a third synchronization option would be exporting a SystemVerilog function that the C endpoint calls periodically to advance the RTL simulator,

which would allow more precise control over the RTL time progression. Advancing the RTL model time in fixed increments based on QEMU's instruction count could improve consistency but still lack cycle accuracy. QEMU's instruction counting also brings negative side effects, like slowing down the QEMU and limiting multi-core QEMU execution to one host thread [6] and is therefore not utilized in this thesis. This method would also require single-threaded QEMU CPU execution, reducing performance. While better than completely independent models, strict cycle-level synchronization is still not feasible using current QEMU instruction counting features.

4.5.2 Register and Memory Access Transactions

When software running on the virtual CPU in QEMU needs to access a register in the RTL DUT, the request is forwarded through the TSS-bridge to the TSS-server. The TSS-server then propagates the transaction to the TSS endpoint connected to the RTL simulation environment. In the RTL simulation, a continuous tick loop implemented in SystemVerilog polls the TSS-endpoint for incoming transactions using the imported C function tss_dpi_tick(). Figure 16 below shows the procedure when the imported C-function tss_dpi_tick() is called in the tss_tick_loop() shown in Figure 13.

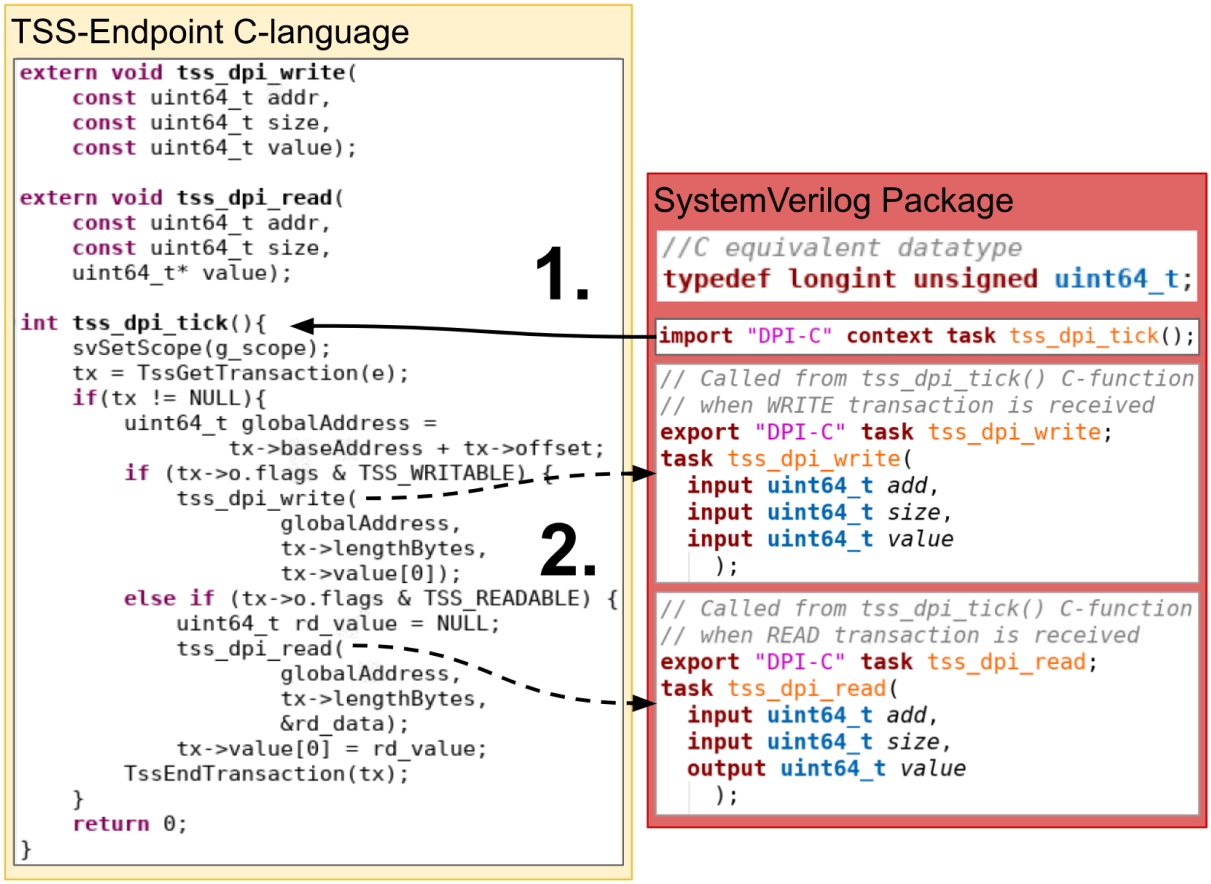

Figure 16. TSS register access procedure.

The tss_dpi_tick() first fetches the transaction details using the TssGetTransaction() API and inspects the flags to determine whether it is a read or write operation. For write transactions, tss_dpi_tick() calls the SystemVerilog function tss_dpi_write(), which is exported to be callable

from C code. The key parameters, including the target address, data value, and access size are passed as an argument to tss_dpi_write() function. This function handles the write using either AXI bus transactions to accurately model cycle timing or direct UVM backdoor access for fast writes without advancing simulation time. On the SystemVerilog package, all inputs and outputs of these functions are defined as C-equivalent uint64_t datatypes for convenience.

The AXI4 VIP used for register accesses supports write strobe for partial writes. The strobe signal has one bit for each byte in the transfer [53]. The tss_dpi_write() function specifies WSTRB based on the size of register access to only modify certain bytes of a register during the write. For example, writing the 2nd byte of a 32-bit register would use a WSTRB mask of 0b0010. In this manner, the access size and correct write strobe value can be used to perform partial register writes through the AXI bus.

For read transactions, tss_dpi_tick() similarly calls the exported SystemVerilog function tss_dpi_read(), passing the target address and access size. The tss_dpi_read() performs the read operation using AXI transactions or fast backdoor peeks and returns the read data value. The tss_dpi_tick() receives this return value on the C side and updates the value field of the original TSS transaction object accordingly.

When using UVM RAL for fast register accesses, the target register is first located using the get_reg_by_offset() method, which returns a handle to the register at that address [17]. Read and write transactions can then be done with the peek() and poke() UVM register methods. If no register match is found, the access may target an RAL memory model instead. In that case, get_mem_by_offset() finds the memory at that address range [17]. Since memories occupy multiple locations, the correct index into the memory must be calculated based on the transaction target address, memory base address, and memory element size. With the memory handle and index, peek() and poke() perform the access to the memory index corresponding to the target address.

After handling the transaction, tss_dpi_tick() concludes it using the TssTargetEndTransaction() API. This makes the updated transaction, now containing any read data, available to return to the TSS-server and the software execution in QEMU. The software is unaware of the underlying propagation steps. The sequence diagram in Figure 17 illustrates the access interactions between the TSS-server and the entities inside the RTL simulator.

A key benefit of the presented register access methodology is that it facilitates integrating different types of transaction-level verification IP (VIP) models into the SystemVerilog package, depending on needs. Since communication with the QEMU software execution occurs using abstract TLM transactions, any interface with a compatible VIP can be easily incorporated. For instance, in addition to the AXI VIP demonstrated, an AMBA AHB VIP could be seamlessly integrated by simply swapping the used VIP sequences in the SystemVerilog package code.

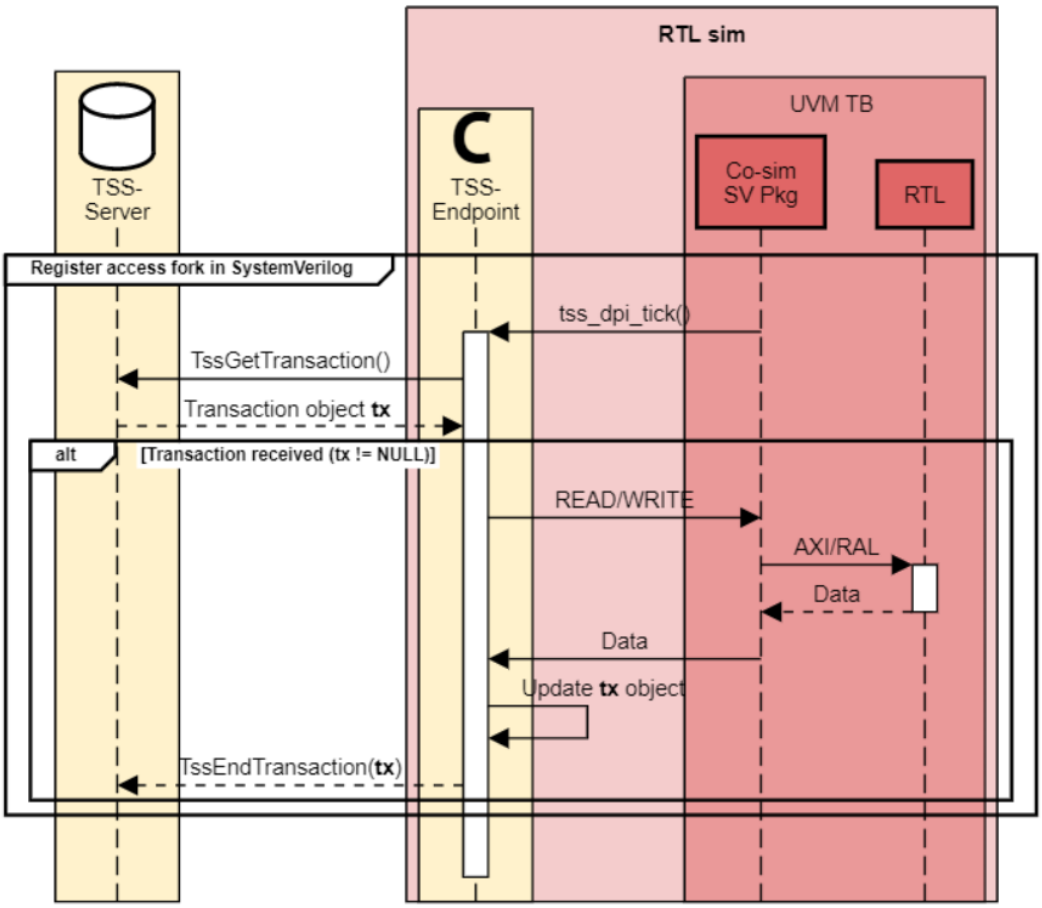

Figure 17. Register Access fork sequence diagram.

4.5.3 Interrupts

The interrupt monitoring is handled by the tss_check_irqs() function shown in Figure 18, which is continuously called in the tss_tick_loop(). It checks the status of each interrupt bit in the Generic Interrupt Controller's Shared Peripheral Interrupt lines from the RTL model. If any interrupts have risen, the interrupt number is passed to the C endpoint via tss_post_irq(irq) shown in Figure 19 to notify the QEMU model.

SystemVerilog Package

```
import "DPI-C" function int tss_post_irq(int unsigned irq);

task tss_check_irqs();
    for (int irq = 0; irq < GIC_SPI_LENGTH; irq++) begin
      if (irq_rose(irq))
        tss_post_irq(irq);
    end
endtask
```

Figure 18. Interrupts monitoring SystemVerilog task.

TSS-Endpoint C-language

```
int tss_post_irq(unsigned irq){
    static TssTransaction *irq_tx = NULL;
    if (irq_tx == NULL){
        irq_tx = TssCreateTransaction();
        irq_tx->addressSpace = IRQ_AS;
        irq_tx->baseAddress = 0x0;
        irq_tx->lengthBytes = 4;
        irq_tx->initialTime = tssTime_getTime();
        irq_tx->o.flags = TSS_PRESENT | TSS_OPEN | TSS_WRITABLE;
        irq_tx->value[0] = 1;
    }
    if (irq >= TSS_IRQ_START) {
        irq_tx->offset = irq - TSS_IRQ_START;
        TssPostTransaction(irq_tx, TSS_NONBLOCKING);
    }
    return 0;
}
```

Figure 19. Interrupt post C-function.

Handling interrupts in a co-simulation environment involving a virtual CPU is crucial for mimicking real-world scenarios, especially during software unit testing. Although interrupts are not essential during use case configuration, where the goal is simply to forward software register accesses to RTL registers and memories, they are required for accurately simulating how a system responds to various interrupt conditions to perform software unit testing. For instance, if an interrupt line from an IP block rises due to an error condition, this change is communicated to a specialized TSS interrupt address space via the C-language program. The handling of interrupts involves multiple layers, from the RTL simulation to the virtual CPU and GIC running in QEMU.

The key aspects of the tss_post_irq() function in Figure 19 are the address space, value, and offset used in the TSS transaction to convey the interrupt. The address space is a dedicated range in TSS allocated for interrupts. The value "1" indicates an active level interrupt has risen. The offset specifies the interrupt number, using TSS_IRQ_START as an offset due to how interrupts are generated in QEMU. This offset transformation maps the interrupt line number from the RTL model to the corresponding offset expected by QEMU's GIC model.

The procedure is illustrated with a sequence diagram in Figure 20. The sequence starts with the activation of the interrupt monitor, which is started simultaneously with the register access loop shown in Figure 13. IP blocks in RTL simulation generate SPIs, which the GIC normally passes to one of the CPU cores. The monitor continuously checks the state of these interrupt lines through a loop that fetches the current state of the GIC SPI vector from the RTL shown in Figure 18. If there's a change in any of the interrupt lines detected by the irq_rose() function, the raised interrupt number is passed to the C-language endpoint with tss_post_irq() function. A TSS transaction targeted at the interrupt address space is initiated by the tss_post_irq() and sent to the TSS server. Finally, this triggers the qemu_irq_raise(irq) function within QEMU, prompting the device driver to react to the interrupt with its ISR.

This flow tries to ensure that interrupts are handled in a manner that mimics the behavior of a real system. The C-language program, SystemVerilog DPI-C package, and UVM Testbench work to monitor, detect, and propagate changes in interrupt lines, thereby providing a robust mechanism for handling interrupts in a co-simulation environment.

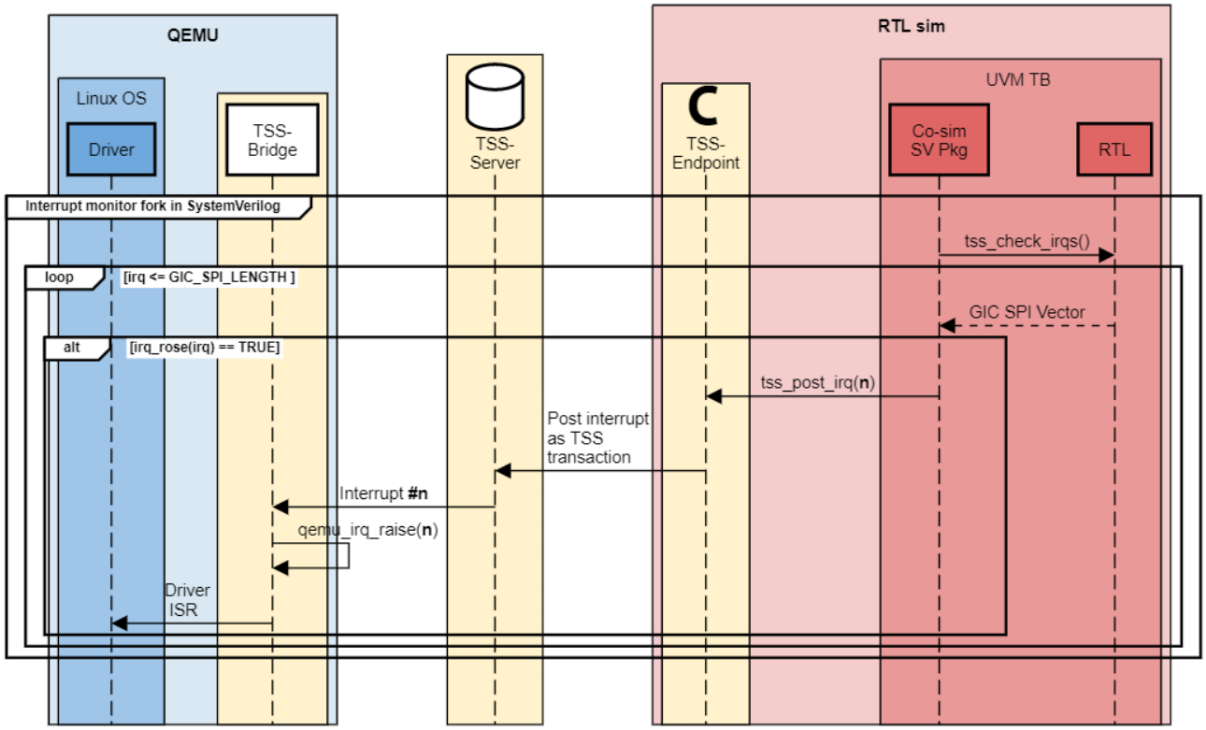

Figure 20. Interrupt monitor fork sequence diagram.

4.6 Robot Framework

Robot Framework (RobotFW) is an existing test automation tool [54] utilized extensively in later validation stages on hardware emulators and sample chips at Nokia. Although RobotFW is not a contribution of this thesis, incorporating it into the co-simulation was, as it allows communication with the RTL simulation via the same TSS-endpoint C-program and controls the software stack running on CPU. QEMU provides port forwarding that map guest ports to the host machine. This allows RobotFW on the host to connect to exposed ports and communicate with the software stack inside QEMU using extensible markup language remote procedure calls (XML-RPC). XML-RPC is a remote procedure call technique that allows function calls to be made across a network [55].

RobotFW integration enables continuity in the validation process by reusing the same system-level test cases across simulation, emulation, and final prototypes. RobotFW test cases developed for traditional pre-silicon and post-silicon testing environments can be executed in the co-simulation environment with only minor changes to communicate with TSS QEMU instead of traditional environments.

RobotFW with software API communication abstracts low-level details of hardware configuration into high-level reusable test cases described by system parameters, like the number of carriers and bandwidth for a telecommunication IP instead of low-level register values and addresses. This facilitates the configuration of realistic use case scenarios with easily machine-readable data format provided by JavaScript object notation (JSON) [56]. The software APIs subsequently interact with device drivers to initiate the necessary register accesses to place the hardware into the desired state, which could be used to replace traditional UVM-based register configuration.

One of the integration tasks during the thesis was enabling RobotFW communication with the RTL simulation environment. A custom API was developed to allow RobotFW to send commands over a TCP socket to the same DPI-C bridge, which handles the TSS-endpoint functionality in the logic simulator. RobotFW communicates directly with the RTL simulation environment when it is blocked in the SystemVerilog tss_tick_loop(), waiting to complete register accesses. A TCP socket connection allows RobotFW to send control commands like "break" to terminate the tick loop. This enables RobotFW to first use the software stack to configure the RTL hardware mode, then break out of the tick loop to proceed with UVM's stimulus generation and results checking. The implementation also utilizes a RobotFW listener that forwards log messages over a socket to the C endpoint. These logs are printed in the simulation environment for visibility into RobotFW execution.

4.7 Challenges and Advantages of Co-simulation

While co-simulation presents promising benefits for the early integration of hardware and software, it brings its own technical challenges that must be considered. Simulation speed is a known limitation, as RTL simulation becomes extremely slow as the design size and number of events increase. Co-simulation performance depends on the scope of simulated components. Focusing on IP blocks rather than full SoCs can improve simulation speed.

Precise synchronization between software execution and RTL models remains difficult. The current lockstep and free-running synchronization modes are functionally adequate but do not provide the full cycle accuracy required for final system validation. This difference in cycle accuracy must be accepted when utilizing co-simulation for pre-validation rather than final system validation.

Debugging the co-simulation architecture may require understanding across the software, UVM, RTL, and C environments. Interactions between several simulation elements may require more debugging than pure RTL simulation or commercial platform because all entities must be carefully configured. In contrast to an actual sample chip, co-simulation provides greater visibility and debugging possibilities and a faster compilation time than an RTL emulator. Hardware engineers can fully utilize logic simulator features like examining internal signals and waveforms. In QEMU, software engineers can use GDB to monitor the target kernel and drivers and step through code with breakpoints. This enables both teams to debug using domain-specific tools while interacting across the hardware-software boundary.

The co-simulation architecture handles all register access sizes from one byte to 64 bits without difficulties. However, challenges arise for 128-bit accesses despite AXI4 supporting 128-bit transactions natively. This is because QEMU emulates the ARM processor's external bus as 64-bits wide. So, 128-bit load/store instructions are done in 64-bit chunks – first the lower half, then the upper half—consequently, a single 128-bit software access results in two 64-bit TSS transactions. For simple memory locations, this does not cause issues. But for registers that can change value between reads, it introduces errors. For example, reading the lower 64 bits first could trigger a value change, causing the upper 64-bit read to retrieve an incorrect value. A workaround could be to monitor all transactions coming to known 128-bit wide registers and combine the separate transactions in tss_dpi_tick() before sending them to RTL. But this approach assumes all accesses in a specific address range are 128-bit, otherwise causing errors.

However, despite these challenges, co-simulation delivers valuable benefits that make it highly worthwhile for early pre-silicon hardware and software integration. Co-simulation

enables early software testing and driver validation at the IP and subsystem levels, which is not feasible without SoC-level prototypes. This allows bugs to be caught and fixed earlier.

Hardware and software issues can be uncovered pre-silicon before traditional methods for system prototyping. Finding bugs earlier reduces validation time and effort in later stages. Unified configuration between simulation, emulation, and prototypes is possible using a central software framework like Robot Framework. This way, stimulus generation can be consistent across verification and testing platforms.

Using RobotFW tests, actual low-level software may be used instead of testbench sequences, enhancing realism in DUT configuration. Proven-to-work UVM environments enable test and VIP reuse from IP to SoC level in co-simulation. This avoids duplication and takes advantage of the standardized UVM framework. Also, integrating co-simulation components into existing UVM environments is relatively straightforward using standard interfaces like DPI-C, facilitating easier adoption.

In summary, co-simulation presents challenges and opportunities in the early integration of hardware and software in SoC design. While it poses challenges such as reduced simulation speed, synchronization, and debugging complexities, the methodology offers unquestionable advantages. These include earlier bug detection and the possibility of unified configuration across hardware execution platforms.

5 PRE-VALIDATION METHODOLOGY EVALUATION

This chapter will evaluate the pre-validation methodology utilizing the co-simulation environment. It covers the startup process to initialize the co-simulation components and make the hardware models visible to the software. User interaction techniques for manual testing and executing software unit tests are explained. Finally, the methodology for leveraging production software instead of UVM to configure the simulated hardware DUT is discussed.

5.1 Environment Startup

As discussed in 4.4, the co-simulation environment is launched by initializing the TSS Server, RTL simulator, and QEMU virtual platform components. Once QEMU finishes booting the Linux kernel, device drivers begin probing the hardware nodes defined in the devicetree, as explained in 4.1.2. This automated configuration process allows the software stack on QEMU to interact with the simulated RTL hardware models via TSS transactions already during system initialization. Because of QEMU's speed, the operating system and software are up and running within a few minutes of executing the TSS startup script.

For example, a device driver may read the device ID during probing to confirm the presence of hardware. Such transactions will be forwarded to the RTL simulation through TSS. In this manner, the Linux software stack can communicate with the underlying RTL models as the OS boots up and discovers devices enabled by the devicetree-driven probing process.

With Linux and drivers initialized, the user can interact with the software stack and hardware models through the command line. Software tests targeting specific devices can then be executed on QEMU to stimulate the RTL hardware models further. The initialized drivers allow user-space applications to communicate with the simulated hardware components.

5.2 User Interaction and Test Execution

In the pre-validation workflow, the user has several options for initiating tests and interacting with the simulation environment. For instance, the user can access registers manually by employing the devmem utility, which allows to read or write physical memory locations from the Linux command line [57]. This allows immediate, direct register reads and writes, providing a straightforward way to verify that a specific hardware register or memory location behaves as expected.

During initial co-simulation development, the devmem tool was utilized to validate register communication before advancing to more complex software testing. Devmem enables interactively reading and writing registers from the Linux command line to test register accesses with different widths. Partial accesses are done by leveraging AXI write strobe signals to update bytes within a register selectively.

For example, Figure 21 shows a scenario with a 4-byte register starting at address 0x90000000. The user first writes the register to all zeros using devmem, then updates only the 2nd byte to 0xFF using an 8-bit write. This is followed by 16-bit writes to bytes 3-4. Finally, reading the full 32-bit register in Linux prints the expected written values.

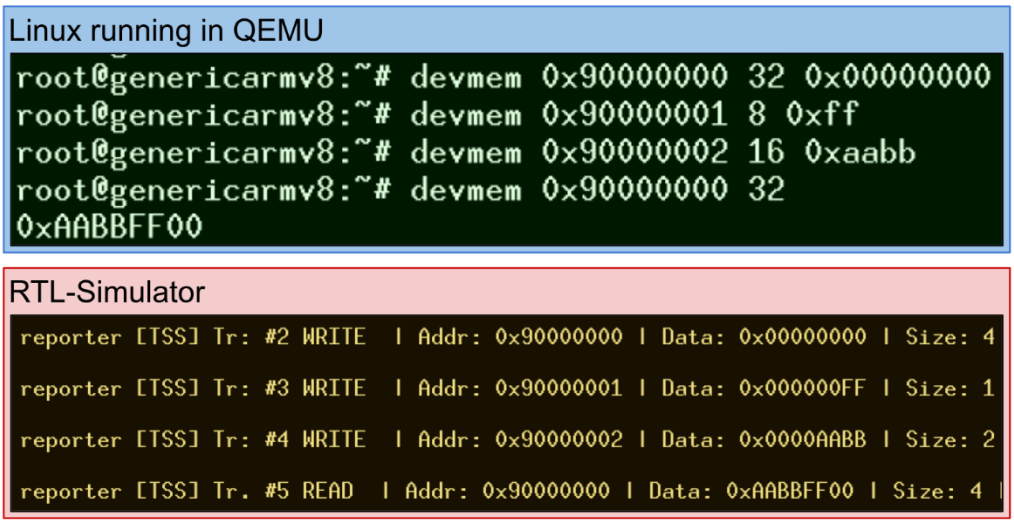

Figure 21. Register access example with devmem.

Figure 22 depicts the AXI bus waveforms in the RTL simulator during this workflow. The WSTRB signals indicate the partial accesses, first targeting just byte 2, then bytes 3-4. This simple workflow was used to verify the register accesses. The devmem provides an interactive way to test register communication between the RTL model and the OS running in QEMU before integrating drivers and higher software layers. Before integrating drivers and higher software layers, it was utilized during initial co-simulation development to check the right behavior of the RTL model and the OS running in QEMU.

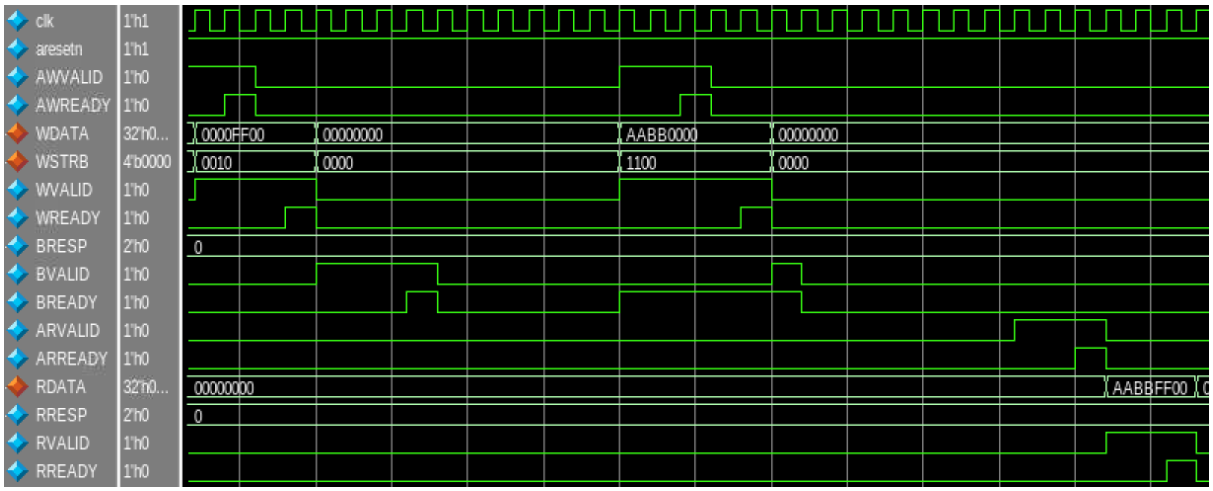

Figure 22. Register access to RTL simulation waveforms.

After verifying connectivity with devmem register accesses, more advanced software testing can commence on the co-simulation platform. One option is executing software unit tests that target IP block-specific software and hardware components. These tests stimulate the RTL models through register and memory operations initiated from the software stack running in QEMU. The unit tests are launched from the Linux command line inside QEMU.

Another possibility is utilizing RobotFW tests launched from the host machine to configure the hardware models into specific use cases defined in JSON files. RobotFW orchestrates register writes through the software stack based on the high-level configuration parameters provided in JSON files.

In both cases, the transactions originating from software ultimately propagate to the RTL simulator via the TSS Server, similarly to the devmem accesses.

5.3 Software Unit Testing

The same software unit tests that are previously designed to run on an emulator-based commercial virtual platform, and physical prototypes can be executed on a virtual CPU connected to a SoC- or IP-level UVM testbench in the co-simulation environment. Unit tests are important in verifying the functionality of the lower API layers of the SoC software stack and device drivers. By executing unit tests at the IP level early on and validating that the drivers are properly designed in connection with the RTL hardware models, the validation of the entire SoC may be completed more efficiently.

The unit tests exercise basic functionality like register reads and writes between the CPU and IP blocks. For example, a test may read an IP's ID register or enable interrupts to validate the hardware communication. They also provide higher-level software testing, allowing for configuring the IP using the software API with specific parameters and then testing whether the RTL design has achieved its intended state by reading CSRs.

A key milestone achieved was successfully running unit tests that trigger interrupts from an IP block, which are then handled correctly in the software stack. Such realistic tests validating interrupt propagation have not been previously possible for software teams without a commercial full SoC prototype or physical sample chips available. The ability to validate register addressing and interrupt behavior provides confidence in the hardware/software integration before the availability of traditional validation environments.

Additional delays between register accesses were sometimes needed when running unit tests to allow the RTL simulation time to propagate and trigger interrupts. Alternatively, free-running synchronization mode continuously advances the RTL simulator, ensuring interrupts will eventually propagate without inserting explicit delays.

Executing software unit tests at the IP level revealed an acceptable speed of approximately 30 register accesses per second when using AXI bus transactions as depicted in Table 3. The trends highlight the advantages of testing at the IP level compared to the SoC level and utilizing UVM backdoor accesses when possible. The IP-level AXI VIP transactions were approximately 30 times faster than the SoC-level AXI VIP in experiments with a single, quite large IP block. However, this performance may vary based on the size of the IP block being tested. Smaller IPs would likely enable even faster transaction speeds. Meanwhile, the UVM backdoor accesses provided a significant 3000x speedup at the SoC level by bypassing transaction delays.

Table 3. Register access performance

Design size and access method	**IP - AXI4 VIP**	**SoC - AXI4 VIP**	**SoC - UVM Backdoor**
transactions/s (approx.)	30	1	3000
cycles/transaction	7	239	0

At the SoC level, QEMU replaces the actual CPU RTL model. To reach a destination IP block, register access transactions done by software must traverse the entire SoC architecture through all the bus bridges and interconnects. In contrast, at the IP block level, QEMU can

shortcut much of the SoC infrastructure by connecting directly to the peripheral bus attached to the IP. Transactions reach the destination IP significantly faster because they bypass all higher-level interconnects and bus bridges, requiring significantly fewer simulation cycles.

The performance difference is partly due to simulation complexity when going from IP to SoC-level, but also significantly impacted by where QEMU is integrated into the co-simulation architecture. Bypassing upper SoC infrastructure provides major performance increase if the goal is IP-level software testing. This reinforces the benefits of modular co-simulation focused on individual IPs or subsystems versus full SoC when utilizing transaction-level communication. These results demonstrate the benefits of testing at the IP rather than SoC level when using more accurate bus transactions. They also highlight the potential utility of selective UVM backdoor accesses to accelerate specific operations during co-simulation and pre-validation.

To maximize performance when running software unit tests, debugging features in the RTL simulator were disabled to provide the fastest simulation runtimes. Logging and waveform visibility were limited to the minimum. With debug options enabled for RTL simulation, performance would be slower. However, some visibility is sacrificed for speed. A trade-off between debuggability and simulation performance must be balanced based on the verification objectives.

Another performance evaluation was performed by first executing unit tests using conventional pre-silicon testing techniques with mocked device drivers in native QEMU and with actual drivers in a commercial emulator accelerated SoC-level prototype, as detailed in Table 4. The same unit tests were executed with actual device drivers in co-simulation with AXI VIP, co-simulation with a TSS RAM as a dummy HW model, and co-simulation with the UVM backdoor scenarios listed in Table 5. Both co-simulation unit tests are run at the SoC-level testbench. This explains the significant difference in execution performance when the backdoor was used instead of AXI VIP, which can perform approximately one register access per second.

Table 4. Traditional pre-silicon software testing environments

Testing Environment	**Driver Type**	**HW Model Type**	**Execution Time**
Native QEMU	Mocked drivers	N/A	18ms
Commercial prototype	Actual drivers	RTL emulated	102ms

Table 5. TSS-based software testing environments

Testing Environment	**Driver Type**	**HW Model Type**	**Execution Time**
TSS QEMU	Actual drivers	Dummy memory	139ms
Co-simulation: AXI VIP	Actual drivers	RTL simulated	399646ms
Co-simulation: UVM Backdoor	Actual drivers	RTL simulated	158ms

In the case of mocked drivers, the dd command was used to create a device like /dev/ipblockname0, acting as a simplified representation of an actual device driver and serving as a Linux file that can be read from or written to, allowing higher-level software to interact with this file as if it were an actual device. However, it's worth noting that this is a very simplified form of a device driver mock, and it won't capture the complexities or specific behaviors of an actual device driver when interacting with actual hardware. However, it is

included in the table to illustrate the execution speed of specific unit tests on the native QEMU for comparison.

The table shows that the commercial SoC-level platform with hardware emulation performs orders of magnitude better, over 5000 times faster than the co-simulation technique at the SoC level. TSS supports the definition of RAM memory for a certain address range, which the TSS QEMU option utilizes to use actual device drivers but to read or write a dummy memory model instead of an RTL endpoint. This metric is provided in the table to compare the performance of TSS QEMU to native QEMU with mocked driver file accesses. It shows that the additional TSS layer and driver software execution degrade performance slightly when compared to native QEMU but enable the use of actual device drivers, which is an advantage over native mocked driver QEMU execution.

The UVM backdoor approach provides speed by not advancing simulation time between operations, leading the RTL model to not respond and propagate state changes between accesses if no extra delays between transactions are applied. Still, backdoor writes provide a more realistic testing environment for early unit tests than mocked drivers, which just manipulate memory structures rather than accessing actual hardware and utilizing the device driver code. For instance, the RTL model can model different behaviors for several types of registers, such as read-only (RO), write-only (WO), and write-one-to-clear (W1C), which are aspects not typically captured when conducting native Linux unit testing with file accesses. The performance decrease is minor when moving from dummy hardware to co-simulation with backdoor accesses, while the benefit is that the actual RTL model is being exercised.

It was demonstrated the unit tests could run at a satisfactory speed, enabling early and iterative driver and low-level software testing in conjunction with real hardware models. Real device driver code may be tested even when utilizing a dummy HW model through TSS, an improvement over earlier native execution approach. This highlights the advantages of performing unit tests with TSS or in the developed co-simulation environment, which provides a closer approximation to real-world hardware behaviors without having access to an SoC-level prototyping platform.

5.4 DUT Configuration with Software

In functional verification, SoC configurations have traditionally relied solely on UVM configuration sequences based on reference values obtained from modeling or engineers' interpretation of specifications. In the real-world operating environment of a SoC, this configuration is done by the corresponding software that comes with IP blocks. This approach delays the use of software-based configuration until SoC-level validation is done. Utilizing software during the functional verification phase provides an alternative to the typical UVM-based configuration, which might be useful for both software and hardware teams since integration issues could be detected and fixed more quickly. This methodology aligns with the "failing fast" principle, facilitating quicker debugging cycles for both software and hardware components.

Compared to hardware emulation, the shorter compilation times for simulators make it faster to spot errors, such as incompatibilities between software and hardware versions in register addressing. This shorter feedback loop might be utilized before the compilation of the emulation database would be ready, checking for common errors, inspecting, and correcting them with the high debuggability of RTL simulation, and then proceeding to the hardware emulator with fixed hardware and software. The objective is not to replace proven UVM techniques required

for early hardware verification but to supplement them by providing an alternative approach that enables earlier software-based configuration through high-level parameters, thereby creating a unified approach to stimulus generation across simulation, emulation, and sample testing.

The co-simulation environment demonstrated in this thesis work enables forwarding all software configuration register accesses to the underlying RTL hardware models. This enables the use of the software stack to configure the DUT for certain use cases. While the equivalence of this method compared to traditional UVM-based configuration was not an objective of this thesis, the environment shows the feasibility of leveraging software for DUT configuration.

In addition to the DUT, the numerous UVM agents utilized in the RTL verification environment must be configured based on the unique use case, which is not possible via production software calls. Future work should study strategies to synchronize UVM component setup with software-driven DUT configuration to enable comprehensive co-simulation scenarios. This process should be straightforward if only the DUT configuration is replaced in each test class with the software method while preserving the previous UVM configuration for all testbench entities. Overall, the environment provides a basis for exploring software-based SoC configuration at a deeper level.

5.4.1 Robot Framework Test Execution

Having the co-simulation architecture up and running, the sequence diagram in Figure 23 illustrates the process of executing a Robot Framework test at a high level, omitting the TSS entities for clarity.

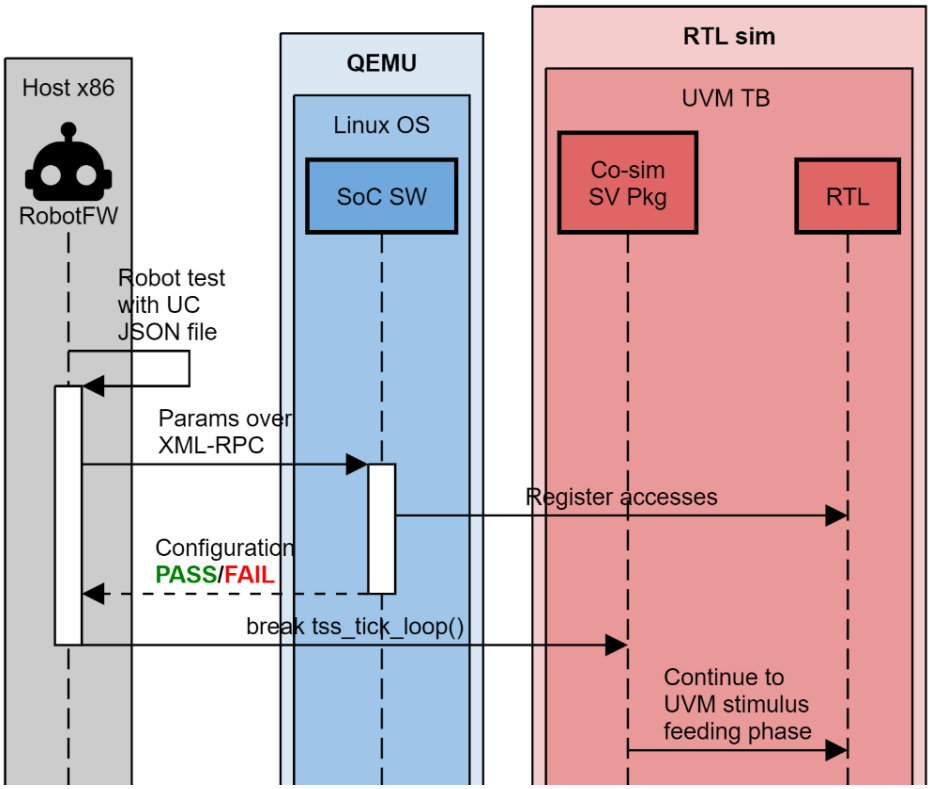

Figure 23. Software-driven DUT configuration.

The JSON files given to Robot tests describe essential parameters for an SoC use case, such as the number of carrier signals and their bandwidth. Previous work at Nokia relied on direct register addresses and corresponding values for configuration [58]. While the prior work demonstrated that it is possible to use machine-readable data formats to represent configuration parameters as address-value pairs and configure them to the DUT using the existing UVM framework, this thesis advances that approach by leveraging real high-level parameters described in JSON files already used in validation testing to enable a more accurate and unified approach with emulation and sample testing. The RobotFW thus becomes a tool for unified stimulus generation across different stages of testing—RTL simulation, hardware emulation, and sample testing.

5.4.2 Dynamic Register Access Configuration

Controlled by UVM test arguments in unit testing, RobotFW could provide a dynamic technique to control the register access mechanism during a simulation. By exploiting the UVM backdoor mechanism, it is feasible to conduct register reads or writes quicker than bus transactions. Current Robot Framework integration employs sockets for basic communication with the C endpoint program, including tick-loop termination. This API could be expanded to enable more advanced capabilities like dynamically switching between regular bus transactions and UVM backdoor accesses. For instance, if the only objective is to write specific values to a large memory area without interest in the RTL model's detailed behavior, the AXI VIP transactions could be safely replaced with UVM backdoor operations, resulting in significantly faster register accesses, as demonstrated in Table 3.

Given the RTL simulation's speed limitation, this may be beneficial in certain scenarios. Robot tests contain many keywords for which the use case parameter JSON file is given as a parameter. For instance, a certain RobotFW keyword may invoke a software API function, which results in hundreds or thousands of transactions to a large memory array within an IP block. Enabling backdoor accesses before calling this keyword in the robot test and turning it off after doing the multiple memory writes before continuing to subsequent keywords, which may need a more realistic bus-based approach, would boost the configuration phase performance, allowing rapidly writing the memory array without each access consuming simulation cycles.

Incorporating RobotFW into conventionally relatively static UVM tests would provide dynamic control of the test flow outside of the RTL simulation by enabling backdoor mode before keywords that access large register or memory areas and reverting to bus-based accesses for subsequent API calls requiring more accuracy. This strategy would selectively utilize quick backdoor accesses to prevent needless bus delays during bulk configurations, balancing between the modes based on the precision demands of each keyword and API call.

6 DISCUSSION

The primary objective of this thesis was to develop and evaluate a co-simulation environment that would allow earlier integration of hardware and software in System-on-Chips. While the basic concept and idea behind the architecture of this thesis are not novel in comparison to the several QEMU-based virtual platforms presented in 3.6.2, the developed environment aligns well with these approaches, providing a vendor-independent environment for Nokia to start hardware and software integration significantly earlier via an architecture based on the open-source QEMU emulator supplemented with Nokia's proprietary patch and logic simulator. In accomplishing this, the thesis significantly exceeds the objectives by enabling not just early integration and register access communication but also the correct handling of interrupts in early software unit testing, something that has previously been unachievable at Nokia until SoC-level prototypes like commercial prototype or physical sample chip.

Nokia's TSS communication architecture, like previous work [40], provides an efficient shared memory C language interface between software execution on virtual CPUs and hardware models. This avoids the additional abstraction layer required when using higher-level modeling languages like SystemC, used primarily in other approaches in the literature. A standard SystemVerilog DPI-C interface facilitates the integration of the C endpoint into current UVM environments. Register access transactions for read and write operations are implemented using approaches similar to those presented in the literature [46]. The generalized TLM interface enables seamless integration of verification IP models depending on the target IP's bus interface.

While earlier efforts at Nokia relied on direct register addresses for configuration [58], this thesis takes a leap forward. It leverages high-level parameters described in JSON files, which are already used in validation testing, enabling a more unified and accurate approach that aligns with emulation and sample testing. This advancement enhances the "shift-left" approach, allowing issues to be identified and addressed when changes are less expensive, thereby boosting overall efficiency.

The most notable outcome of this thesis is the ability to run software unit tests and validate correct interrupt and register behavior in the early stages of the design cycle at IP block level. This feature is especially significant because it provides a level of confidence in early hardware and software integration that was previously unattainable at Nokia due to the need to wait until SoC level RTL architecture was completed. Because the developed environment is vendor-independent, IP teams can comfortably launch SW unit tests in co-simulation without worrying about licensing restrictions associated with commercial platforms.

While the main use case demonstrated in this thesis focused on software unit testing, the environment also shows promise for leveraging production software in functional RTL hardware verification. By passing software register accesses to RTL models, an approach was proposed that allows DUTs to be configured for particular use cases. This area has the potential for future development of integrating software-driven configurations into UVM testbenches.

While co-simulation presents its own set of challenges, such as the potential for the RTL simulator to slow down as design complexity increases, the benefits of early integration largely outweigh these limitations. Future work should explore methods for accelerating co-simulation speed, automating test execution, and further leveraging SystemVerilog functional coverage to assess software-driven tests versus traditional UVM stimuli. Co-simulation speed and test iteration performance would be improved by replacing the RTL simulator with an emulator by using a synthesizable AXI bridge [59] and implementing a co-simulation checkpointing [60] mechanism.

In summary, this thesis explored a co-simulation approach using QEMU and Nokia's TSS along with RTL models to enable early hardware and software integration. The environment has successfully demonstrated a co-simulation approach that enables early hardware and software integration, facilitating the successful execution of software unit tests and low-level software validation alongside RTL hardware models, thereby allowing stronger collaboration between hardware, software, and verification teams earlier in the design cycle.

7 SUMMARY

The key objective of this thesis was to develop and evaluate a co-simulation methodology used for the pre-validation of SoCs. Pre-validation facilitates earlier integration of hardware and software models compared to traditional validation methods. Shifting left on integration attempts to reduce validation effort by identifying problems earlier when changes are less expensive and easier to do.

The co-simulation architecture presented enables unmodified software execution with target ISA on QEMU alongside RTL hardware models. Custom extensions to QEMU provided by Nokia's TSS facilitate transaction-level communication with external hardware models like the RTL simulator. The TSS Server manages these transactions using shared memory IPC, avoiding networking overheads. The QEMU-based method is consistent with several similar methodologies in literature and commercial products.

A custom C endpoint program was developed to bridge TSS transactions between the UVM testbench and the TSS Server. This program integrates with commercial RTL simulators using standard interfaces such as SystemVerilog DPI-C. The C endpoint also included a socket-based API to allow the Robot Framework to control the co-simulation environment.

The significant outcome of this thesis was that the developed co-simulation approach demonstrated the successful execution of software unit tests against RTL models, which do not require the availability of conventional validation prototypes. Most notably, also interrupt handling in addition to other basic functionalities tested in unit tests were validated between software and hardware, providing confidence in the integration, which had previously been postponed until SoC-level prototypes. Furthermore, the environment demonstrated the viability of forwarding software register accesses to RTL DUTs, laying the foundation for future software-driven use case configuration during functional hardware verification.

While the method has known limitations, such as logic simulation performance and the complexity of debugging hardware and software systems, it facilitates shifting left on integration to identify the issues earlier. This framework can potentially increase collaboration between software, hardware, and verification teams earlier in the SoC development cycle.

In conclusion, this thesis provides a QEMU- and RTL-simulator-based co-simulation architecture that enables early software and RTL hardware model integration. The co-simulation was found to have potential future utility, allowing for earlier integration, and reducing subsequent traditional validation integration challenges.

8 REFERENCES

[1] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. C. Wang, "Challenges and Trends in Modern SoC Design Verification," *IEEE Des Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017, doi: 10.1109/MDAT.2017.2735383.

[2] U. Farooq and H. Mehrez, "Pre-Silicon Verification Using Multi-FPGA Platforms: A Review," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 37, no. 1, pp. 7–24, Feb. 2021, doi: 10.1007/S10836-021-05929-1/FIGURES/11.

[3] P. Rashinkar, P. Paterson, and L. Singh, "System-On-A-Chip verification : methodology and techniques," p. 372, 2002.

[4] J. R. Andrews, "Co-verification of hardware and software for ARM SoC design," p. 260, 2005.

[5] A. B. Mehta, "ASIC/SoC Functional Design Verification," *ASIC/SoC Functional Design Verification*, 2018, doi: 10.1007/978-3-319-59418-7.

[6] "QEMU." Accessed: Sep. 20, 2023. [Online]. Available: https://www.qemu.org/

[7] D. J. Greaves, "Modern System-on-Chip Design on Arm – Arm®." Accessed: Aug. 09, 2023. [Online]. Available: https://www.arm.com/resources/ebook/modern-soc

[8] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-Silicon Validation in the SoC Era: A Tutorial Introduction," *IEEE Des Test*, vol. 34, no. 3, pp. 68–92, Jun. 2017, doi: 10.1109/MDAT.2017.2691348.

[9] P. R. Schaumont, "A Practical Introduction to Hardware/Software Codesign," *A Practical Introduction to Hardware/Software Codesign*, 2013, doi: 10.1007/978-1-4614-3737-6.

[10] "IP cores integration in DSP System-on-chip designs | IEEE Conference Publication | IEEE Xplore." Accessed: Sep. 04, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/7071983

[11] Gary. Stringham, "Hardware/firmware interface design : best practices for improving embedded systems development," p. 360, 2010.

[12] "Learn the architecture - Arm Generic Interrupt Controller v3 and v4." Accessed: Oct. 06, 2023. [Online]. Available: https://developer.arm.com/documentation/198123/0302/Arm-GIC-fundamentals

[13] J. J. Labrosse, "Embedded software," p. 770, 2008.

[14] "1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation," 2017.

[15] P. Mishra and F. Farahmandi, "Post-silicon validation and debug," *Post-Silicon Validation and Debug*, pp. 1–394, Sep. 2018, doi: 10.1007/978-3-319-98116-1/COVER.

[16] "Is it Emulation or FPGA-Based Prototyping that I want? – Breaking The Three Laws." Accessed: Oct. 15, 2023. [Online]. Available: https://blogs.synopsys.com/breakingthethreelaws/2012/10/is-it-emulation-or-fpga-based-prototyping-that-i-want/

[17] "1800.2-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual," 2020.

[18] R. Saleh *et al.*, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050–1068, 2006, doi: 10.1109/JPROC.2006.873611.

[19] V. Melikyan, S. Harutyunyan, A. Kirakosyan, and T. Kaplanyan, "UVM Verification IP for AXI," *2021 IEEE East-West Design and Test Symposium, EWDTS 2021 - Proceedings*, Sep. 2021, doi: 10.1109/EWDTS52692.2021.9580997.

[20] “1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,” 2018.
[21] E. Massoud, M. Abdelsalam, M. Safar, and M. Watheq El-Kharashi, “A Reusable UVM-SystemC Verification Environment for Simulation, Hardware Emulation, and FPGA Prototyping: Case Studies,” *2022 International Conference on Microelectronics, ICM 2022*, pp. 38–41, 2022, doi: 10.1109/ICM56065.2022.10005445.
[22] S. Konale and N. B. Rao, “C-based predictor for scoreboard in Universal Verification Methodology,” *2014 International Conference on Advances in Engineering and Technology Research, ICAETR 2014*, 2014, doi: 10.1109/ICAETR.2014.7012913.
[23] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono, “Co-simulation methodology for improved design and verification of hardware neural networks,” *IECON Proceedings (Industrial Electronics Conference)*, pp. 2226–2231, 2013, doi: 10.1109/IECON.2013.6699477.
[24] M. Ballance, “Co-Developing Firmware and IP with PSS,” 2022.
[25] C. Ebert and C. Jones, “Embedded software: Facts, figures, and future,” *Computer (Long Beach Calif)*, vol. 42, no. 4, pp. 42–52, 2009, doi: 10.1109/MC.2009.118.
[26] R. S. Pressman and B. R. Maxim, *Software engineering: A Practioner’s Approach*. New York: McGraw-Hill Education, 2020.
[27] H. Cheddadi, S. Motahhir, and A. El Ghzizal, “Google Test/Google Mock to Verify Critical Embedded Software,” Aug. 2022, Accessed: Sep. 19, 2023. [Online]. Available: https://arxiv.org/abs/2208.05317v1
[28] J. Chen, “Veloce HYCON Software Enabled SoC Verification and Validation on Day 1”.
[29] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, C. Jego, and F. Konrad GreenSocs, “QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0,” Jan. 2016, Accessed: Sep. 08, 2023. [Online]. Available: https://hal.science/hal-01292317
[30] J. Bodingbauer, S. Tauner Wien, and J. Bodingbauer Andreas Steininger, “System-level verification and testing of a safety-critical SoC using HW/SW Co-Simulation.” 2022. Accessed: Jul. 05, 2023. [Online]. Available: http://cds.cern.ch/record/2815552
[31] M. C. Chiang, T. C. Yeh, and G. F. Tseng, “A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 593–606, Apr. 2011, doi: 10.1109/TCAD.2010.2095631.
[32] E. Díaz, R. Mateos, E. J. Bueno, and R. Nieto, “Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms,” *Electronics 2021, Vol. 10, Page 759*, vol. 10, no. 6, p. 759, Mar. 2021, doi: 10.3390/ELECTRONICS10060759.
[33] “QEMU User Documentation - Xilinx Wiki - Confluence.” Accessed: Oct. 10, 2023. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/821395464/QEMU+User+Documentation
[34] “Xilinx/libsystemctlm-soc: SystemC/TLM-2.0 Co-simulation framework.” Accessed: Sep. 20, 2023. [Online]. Available: https://github.com/Xilinx/libsystemctlm-soc
[35] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, “Fast virtual prototyping for embedded computing systems design and exploration,” *ACM International Conference Proceeding Series*, vol. Part F148382, 2019, doi: 10.1145/3300189.3300192.

[36] M. Montón, A. Portero, M. Moreno, B. Martínez, and J. Carrabina, "Mixed SW/systemC SoC emulation framework," *IEEE International Symposium on Industrial Electronics*, pp. 2338–2341, 2007, doi: 10.1109/ISIE.2007.4374971.
[37] "Virtual Platform — NVDLA Documentation." Accessed: Oct. 10, 2023. [Online]. Available: http://nvdla.org/vp.html
[38] S. T. Shen, S. Y. Lee, and C. H. Chen, "Full system simulation with QEMU: An approach to multi-view 3D GPU design," *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pp. 3877–3880, 2010, doi: 10.1109/ISCAS.2010.5537690.
[39] S. Cho, M. Patel, H. Chen, M. Ferdman, and P. Milder, "A Full-System VM-HDL Co-Simulation Framework for Servers with PCIe-Connected FPGAs," 2018, doi: 10.1145/3174243.3174269.
[40] D. Lee, K. Kang, J. Park, B. Lee, J. Kim, and J. Im, "Toward Heterogeneous Virtual Platforms For Early SW Development," *Proceedings - International SoC Design Conference 2022, ISOCC 2022*, pp. 384–385, 2022, doi: 10.1109/ISOCC56007.2022.10031327.
[41] A. Freitas, "Hardware/Software Co-Verification Using the SystemVerilog DPI," 2007.
[42] A. Wicaksana and C. M. Tang, "Virtual prototyping platform for multiprocessor system-on-chip hardware/software co-design and co-verification," *Studies in Computational Intelligence*, vol. 719, pp. 93–108, 2018, doi: 10.1007/978-3-319-60170-0_7.
[43] "HW/SW Co-Simulation with QEMU - Functional Verification - Solutions - Aldec." Accessed: Oct. 07, 2023. [Online]. Available: https://www.aldec.com/en/solutions/functional_verification/qemu_co_sim
[44] "Synopsys Launches Industry's First Integrated Hybrid Prototyping Solution - Jun 4, 2012." Accessed: Oct. 18, 2023. [Online]. Available: https://news.synopsys.com/index.php?item=123390
[45] "Palladium Hybrid | Cadence." Accessed: Oct. 18, 2023. [Online]. Available: https://www.cadence.com/zh_TW/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-hybrid.html
[46] F. Cerisier, "UVM based Hardware/Software Co-Verification of a HW Coprocessor using Host Execution Techniques," 2019.
[47] M. Muñoz-Quijada, L. Sanz, and H. Guzman-Miranda, "SW-VHDL Co-Verification Environment Using Open Source Tools," *Electronics 2020, Vol. 9, Page 2104*, vol. 9, no. 12, p. 2104, Dec. 2020, doi: 10.3390/ELECTRONICS9122104.
[48] "Release v0.4 · devicetree-org/devicetree-specification · GitHub." Accessed: Sep. 28, 2023. [Online]. Available: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.4
[49] F. Vasquez, C. Simmonds, and an O. M. Company. Safari, "Mastering Embedded Linux Programming - Third Edition," p. 758.
[50] "GitHub - dgibson/dtc: Device Tree Compiler." Accessed: Oct. 12, 2023. [Online]. Available: https://github.com/dgibson/dtc
[51] A. Venkataraman and K. Kumar Jagadeesha, "Evaluation of Inter-Process Communication Mechanisms".
[52] Horace Chan and Byron Watt, "How to test the whole firmware/software when the RTL can't fit the emulator," 2019.
[53] "AMBA AXI and ACE Protocol Specification Version E." Accessed: Oct. 06, 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/

[54] “Robot Framework User Guide.” Accessed: Oct. 03, 2023. [Online]. Available: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html
[55] “What is XML-RPC?” Accessed: Oct. 13, 2023. [Online]. Available: http://xmlrpc.com/
[56] “JSON.” Accessed: Oct. 10, 2023. [Online]. Available: https://www.json.org/json-en.html
[57] “busybox/miscutils/devmem.c at master · brgl/busybox · GitHub.” Accessed: Oct. 07, 2023. [Online]. Available: https://github.com/brgl/busybox/blob/master/miscutils/devmem.c
[58] J. Kituniemi, “Reusable hardware configuration methodology for simulation, emulation, and validation.” 2022.
[59] E. Kokkonen, “AXI-Stream VIP Optimization for Simulation Acceleration : a Case Study,” Apr. 2021, Accessed: Sep. 17, 2023. [Online]. Available: https://trepo.tuni.fi/handle/10024/125270
[60] M. Macián, “Checkpointing for virtual platforms and systemC-TLM-2.0,” 2010.