Informatics

System-level verification and testing of a safety-critical SoC using HW/SW Co-Simulation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Jonas Bodingbauer
Matrikelnummer 11802486

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung (TU Wien):	Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Betreuung (CERN):	Dr. Hamza Boukabache
Mitwirkung:	Univ.Ass. Dipl.-Ing. Stefan Tauner

Wien, 18. Mai 2022

Jonas Bodingbauer   Andreas Steininger

TU WIEN Informatics

CERN

System-level verification and testing of a safety-critical SoC using HW/SW Co-Simulation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Jonas Bodingbauer
Registration Number 11802486

to the Faculty of Informatics

at the TU Wien

Advisor (TU Wien):	Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Advisor (CERN):	Dr. Hamza Boukabache
Assistance:	Univ.Ass. Dipl.-Ing. Stefan Tauner

Vienna, 18th May, 2022

Jonas Bodingbauer   Andreas Steininger

Erklärung zur Verfassung der Arbeit

Jonas Bodingbauer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Mai 2022

Jonas Bodingbauer

Acknowledgements

Firstly, I want to thank my great team at CERN for providing support during this project. I especially want to thank Dipl.-Ing. Katharina Ceesay-Seitz who gave valuable feedback and input during development. Questions regarding the process of verification and the technicalities of it were never left unanswered.

I want to thank Univ.Ass. Dipl.-Ing. Stefan Tauner for his prompt replies to my questions. Furthermore, I want to thank Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger and Dr. Hamza Boukabache for general supervision of this thesis.

Gratitude is also deemed to my parents who always supported me during my studies and will hopefully continue to do so as they further progress. I also want to thank Lena, who was available at all times when I needed to tell someone about problems I faced while developing and writing this thesis.

Kurzfassung

In der Europäischen Organisation für Kernforschung (CERN) produzieren unterschiedliche Quellen ionisierende Strahlung. Diese muss konstant überwacht werden, um Arbeiter-Innen und Umwelt vor Effekten bei zu großer Dosis zu schützen. Das CERN Radiation Monitoring Electronics (CROME) Projekt wurde für diese Anwendung entwicklelt und verwendet ein heterogenes System-on-Chip (SoC), einen Zynq-7000, als zentralen Datenverabeitungschip.

Diese Arbeit zeigt einen Ansatz zur Verifikation auf Systemebene für ein solches SoC, welcher auf einer Hardware/Software-Co-Simulation basiert. Der Simulator setzt sich aus einem Verbund eines kommerziellen Hardware-Simulators und QEMU, einem Software/Maschinen Emulator, zusammen. Die Simulation ist schnell genug, um Linux in wenigen Minuten zu booten, während sie realitätsgetreu genug für Verifikation bleibt.

Im Rahmen dieser Arbeit wird weiters eine "constrained random stimulus" Testbench-Architektur, welche auf dem Industrie-Standard UVM basiert, entwickelt. Diese Testbench kann gleichermaßen Software und Hardware überwachen und ansteuern. CROME hat eine ähnliche Auswahl an Schnittstellen wie Geräte im Internet of Things (IoT). Während die Software mittels TCP/IP verbunden ist, interagiert die Harware mit unterschiedlichen Sensoren. Die vorgestellte Topologie ist daher auch für eine Reihe anderer Geräte geeignet.

Abstract

At the European Organization for Nuclear Research (CERN) different sources produce ionizing radiation. This radiation needs to be monitored to protect people and the environment from effects caused by high exposure. The CERN Radiation Monitoring Electronics (CROME) are developed for this application and use a heterogeneous System-on-Chip (SoC), a Zynq-7000, as its main computing device.

An approach for performing system-level verification on such an SoC is shown within this thesis, which is built on a Hardware/Software Co-simulation. The simulator is created by connecting an off-the-shelf hardware simulator and QEMU, a full machine (software) emulator. Its speed is high enough to boot Linux in minutes while keeping the model close enough to real-life to enable verification.

A constrained random stimulus testbench architecture, based on industry-standard Universal Verification Methodology (UVM), for interfacing both software and hardware is developed within this work. CROME features interfaces similar to Internet of Things (IoT) devices: software interacts via TCP/IP and hardware connects various sensors. The presented topology is therefore suitable for a range of other hardware as well.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The European Organization for Nuclear Research (CERN) operates a variety of particle accelerators, which are used for diverse experiments studying fundamental physics of our universe [Bou+17]. In the last run, particles were accelerated up to an energy of $13\,TeV$ in the Large Hadron Collider (LHC)[PS19]. It accelerates particles in opposing directions which are then collided at four experiments installed on the LHC: ATLAS, CMS, Alice and LHCb [PS19]. When particles collide at these energy levels, new particles are formed and scattered into various directions [Bou+17].

Ionizing radiation is produced at these, and other experiments on the CERN site from interaction of energetic particles with stable matter, such as the accelerators and experiments themselves as well as with other particles. Additional radiation comes from radioactive sources used by many experiments. Exposure to ionizing radiation at the working environment needs to be monitored precisely, since, depending on the accumulated dose a person is exposed to, it may have adverse health effects. Furthermore, the environment in which CERN operates has to be monitored as well, to protect people and environment near the experiments. One measure to limit exposure to generated radiation is shielding. In the case of the LHC, it is installed $100\,m$ below surface level. Other experiments use thick walls of concrete and other materials to ensure safety. Therefore, most of the radiation protection monitoring devices are also installed behind these shielding.

Complying with the safety regulations, imposed by France and Switzerland as host states as well as CERN, measuring the radiation doses is one of the responsibilities of the Radiation Protection (RP) Group at CERN. Radiation monitoring devices are installed in different places at CERN. The newest, in-house developed system, CERN Radiation Monitoring Electronics (CROME) is created by the Instrumentation & Logistics Section, which is part of RP Group [Cee19]. More than 150 CROME devices are currently in use, with more installations planned in the following years.

CROME is a safety-critical system, which, according to the IEC 60532 radiation protection instrumentation standard, should meet Safety Integrity Level 2 (SIL 2). Therefore, the hardware as well as the software needs to fulfill certain criteria to reduce the risk of dangerous failures during operation [Cee19]. The hardware fulfills the SIL 2 requirements as shown by the reliability analysis of CROME [Hur18]. The main computing device of this system is a Xilinx Zynq-7000 System-on-Chip (SoC) which embeds a hard-core CPU as well as an FPGA. Verification of the software as well as the hardware (programmable logic) in this SoC is needed, according to the methodology described in "Automated verification of a System-on-Chip for radiation protection fulfilling Safety Integrity Level 2" [Cee19]. A verification engineer already applied block-level tests on the hardware side, which contains all safety-critical functions like dose and dose-rate calculation, temperature compensation and alarm evaluation and triggering [Cee19; Bou+17]. This thesis describes a method for executing system-level tests on a simulated system, exercising the communication between software and hardware.

1.2 Objectives

The main question answered by this thesis is: "How to verify a tightly integrated system using a heterogeneous SoC including a hard-core and FPGA on system-level?"

System-level testing on a heterogeneous SoC or system is not a trivial task since no directly available method for simulation or formal verification of the entire system exists. Adoption of such devices in various safety-critical applications is becoming more widespread [Mic; doo15; Bro12] and therefore a method for verification is needed.

One technique for verification is testing, which can be executed in various ways. One possibility is simulation of the software running on the CPU as well as the logic implemented by the Field Programmable Gate Arrays (FPGA) at the same time - also referred to as Hardware/Software Co-simulation. Ideally, this simulation can be executed on a single general purpose computer for development, debugging and verification. This allows for wider application and usability of the simulation. The simulation speed should be fast enough that an operating system like Linux can be booted in a reasonable amount of time, i.e., within a few minutes. Furthermore, the capability to run automated tests is another important aspect for verification of an embedded device.

Goal of this work is to design and implement a Hardware (HW)/Software (SW) Co-simulation that can be used for debugging and verification. The properties mentioned in the previous paragraph should be fulfilled by this implementation. Additionally, an approach for a testbench which is based on the developed simulator for automated tests on a combined hardware and software design is developed.

1.3 Outline of this thesis

This thesis presents the development of a HW/SW Co-simulation for a device using a Zynq-7000 SoC and is structured in 5 Chapters. Chapter two gives a theoretical background of the devices that are supposed to be simulated, methods for simulation and verification. Special attention is given on verification using testing and methods for evaluating respective verification progress.

The third chapter gives an overview of the architecture of the device this simulation was developed for. The architecture of the simulation is explained and software and tools used to develop the simulator are discussed.

In chapter four, an architecture for a testbench for automated testing, based on the simulation in chapter three, is presented. The Universal Verification Methodology (UVM), an industry-standard for writing modular and flexible testbenches [Sof20], is used to show an approach for applying stimulus and monitoring outputs on both the software and hardware domain.

The last chapter discusses results of the work developed within the scope of this thesis. An outlook on further development of co-simulation methods for heterogeneous SoCs is given.

CHAPTER 2

Background

2.1 System-on-Chip - SoC

An SoC is a

> "single silicon chip which can be used to implement the functionality of an entire system, rather than several physical chips being required." [Cro+14, p. 2]

This term can therefore describe a wide range of devices and integration levels. It can reach from a "simple" digital application-specific integrated circuit (ASIC), to mixed-signal chips including both analog and digital components. In specialized applications, even optical components as well as advanced radio frequency components can be integrated.

As far as the digital components go, it can include memory, processing devices, logic amongst other elements. When not using an SoC for creating a design, those components would usually come on separate chips which need assembly. With an SoC this is avoided, therefore it is usually cheaper at scale, has more reliable and faster interconnections between components and blocks and is physically smaller at the same time. Furthermore, it reduces energy consumption compared to a solution from discrete components [Cro+14, p.2].

2.1.1 Field Programmable Gate Arrays - FPGA

A technology to create programmable hardware devices are FPGAs. These devices allow writing code in a hardware description language like Verilog and Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Chisel and quickly implement it on hardware. Other, usually less complex devices achieving similar functionality are Programmable Logic Devices (PLDs) and Complex Programmable Logic Devices

A	B	C	$(A \wedge B) \oplus C$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 2.1: Example LUT

(CPLDs). Compared to the fabrication of an ASIC, the initial cost of using such devices is much cheaper. This means it is cheaper to use an FPGA for small production runs and prototyping instead of producing an actual silicon chip.

2.1.2 Working Principle of FPGAs

While the details can differ between manufacturers and also between different chips of one manufacturer, the working principle remains similar. Modern implementations are usually more complex to provide even more flexibility and speed.

Lookup table Logic Cells

One of the main building blocks of an FPGA are Lookup tables (LUTs) capable of implementing any truth table with a given amount of inputs and outputs. Instead of synthesizing the truth table by using simple gates and wiring them up in different configurations, the truth table can be imagined to be written into a small memory. For implementing a logic function like $(A \wedge B) \oplus C$, the truth table 2.1 is used to look up a single bit output. An additional benefit of this flexibility is, that while routing signals between LUTs, the signal does not need to be routed to a specific pin since the inputs can easily be switched around by reordering the LUT [KHZ16, p. 13]. A LUT can be implemented using different memory technologies. Static RAM (SRAM) is commonly used, but while giving more flexibility, this also causes the need to configure the FPGA on each startup since SRAM is volatile. For large devices this can take up to a few seconds [KHZ16, p. 14]. Some vendors also provide non-volatile technologies like flash memories for instant startup [Mic17] and less susceptibility to radiation-induced bit-flips.

Logic Blocks

LUTs, Flip-Flops, as well as other blocks like adders are combined into configurable logic blocks for implementing higher level functions. These logic blocks are therefore more capable compared to its individual components. These can again be grouped into even

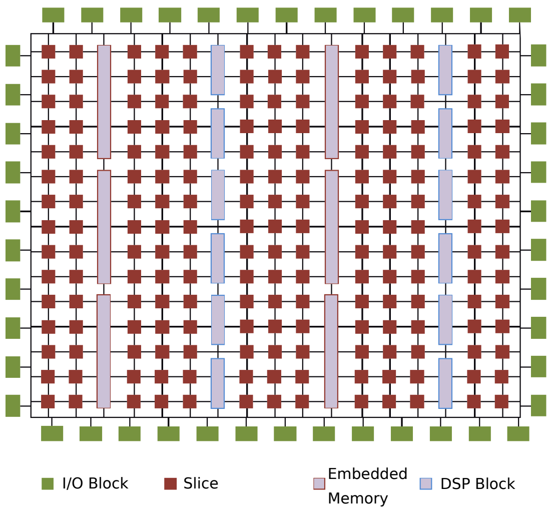

Figure 2.1: Conceptual layout of a FPGA [RF16]

higher level blocks. In general, the combination of lower level devices into one larger entity is a key building block of an FPGA [Cro+14, p. 24].

Additional Block Types

Memory Blocks Since many system designs need memory for storing values in a dense fashion, FPGAs usually include memory blocks for dedicated storage. This can be more efficient than using the flip-flops inside the logic blocks. Special functionalities like dual-port RAMs or FIFO queues which are often available for these blocks, can further simplify development [Xil19].

DSP Blocks Implementing multiplication and addition within a design can be sped up by using dedicated Digital Signal Processing (DSP) blocks. These are usually specialized to process longer vector lengths in dedicated silicon which saves LUT resources [Cro+14].

IO Blocks For interfacing the outside world, input and output driver blocks are also needed. These can be configurable to different voltage levels as well as speed characteristics and modes or even can drive more complex interfaces for instance for high speed data communications. As an example, a simple IO block may be configured for an Inter-Integrated Circuit (I^2C) Bus. This configuration needs to support a given logic level like, e.g., $3.3\,V$ and open-drain outputs, which would not be possible by using the direct signals from inside the FPGA as they usually do not support tri-state logic and typically use different logic leves.

Interconnections

To achieve complex functionality, many blocks are assembled into a grid-like structure with interconnecting wires. For each block, wires can be connected to the grid which enables routing of signals between all blocks. The connections are an important aspect for speed as well as signal integrity as signals need to arrive at their dependent blocks at a specified timing which is influenced by the routing taken. According to Koch roughly $\frac{2}{3}$ of the latency of the critical path within a design is caused by the interconnections. The connection points are conceptually made up by many multiplexers which route the signals between the available wires [KHZ16].

2.1.3 SoC including FPGA and CPU

SoCs which include both FPGA and a dedicated hardware processor are available from different manufacturers. At first glance it could seem redundant to include a hardware processor in silicon, when one can implement a soft-core processor within the FPGA. There are benefits and drawbacks to either choice which will be briefly discussed.

Soft-Core processor A soft-core processor is made up by the FPGAs general purpose logic [SOS08, p.2]. This means that it consists of the slices, DSP slices and other general segments which the FPGA provides. This allows for easy reconfiguration and choosing fitting characteristics of the processor for its purpose.

Hard-Core processor A hard-core processor is implemented in silicon at the transistor-level [SOS08, p.2]. It does not allow for modifications except different configuration of peripherals, but usually has other benefits compared to the soft-cores such as general processing speed and less power consumption.

Processing Speed

A soft-core processor is limited by the speed of the FPGA fabric in terms of clock speed. If processing speed for general purpose applications is needed, a hard-core can achieve higher clock-speeds which, for many applications, results in higher processing speed.

Power Consumption

Hard-core designs usually feature different power saving modes. These can shut down parts of the device, or even stop or slow down the clock for the entire device resulting in much less power consumption compared to it running at full capacity. FPGAs implementing a soft-core are not power efficient by nature. They usually do not use their logic blocks very efficiently, with no mechanism of cutting power, while also using memories for creating the LUTs [MC12, p.12].

Visibility of signals to developers

Soft-Cores allow to view all signals inside the processor since they must be available as a netlist to be implemented in the FPGA [MC12, p. 7]. Hard-Core systems do not provide this option while usually featuring some way of tracing and debugging applications, like the *CoreSight* peripherals of ARM processors [ARM13].

Whether the designer chooses a soft- or hard-core, the SoC requires both software and hardware to run. This results in a tight coupling between the hardware and driving software. This can result in very efficient and fast systems, but also complicates development as both aspects need to be co-designed.

2.2 Verification of an SoC

The objective of verifying a system, according to the IEC 61508 standard for functional safety of electrical/electronic/programmable safety-related systems, is to show that all the outputs of a system meet the specified requirements [Com10, p. 49]. This means the engineers who build the system ask “Are we building the system right?”. Checking whether whatever is built, fulfills the functional specification exactly.

This process is to be differentiated from validation, which asks “Are we building the right system?”. Therefore, checking whether the specification of the system fulfills all the requirements by the client [Cee19].

Verification is especially relevant for devices which are related to safety-critical functions, as a malfunction might cause damage, or worst case fatalities. The above-mentioned standard defines different target failure measures for four different categories. For devices with *low-demand* (functionality used less than once per year) the probability targets are specified as probability of failure per demand, for *high-demand* devices as average frequency of failures per hour. According to the required category SIL 1-4 different approaches of verification have to be chosen [Com10].

Abstraction Level Verification can be done on different abstraction levels of the design such as block-level, module-level and system-level. This is possible because hardware designs are usually made up hierarchically by different design blocks. For example a multiplication block, a timing block, a clock generation block and so on. These blocks are then connected together to make up the entire system. Different techniques for verification can and need to be applied at different levels. Each level focuses on different aspects. On the block-level, verification can be more specific and fine-grained, while on system-level the correct orchestration of the blocks must be checked. This means that bus-protocols are fulfilled, timings are met and all the interfaces between the blocks are correct [ST12].

Many modern systems feature many configurations and temporal dependencies which cause issues with the complete verification of the system. In a perfect world, one could show that for each and every state of the system, the output is as expected. This is usually not possible in reasonable time because of the so called *state-space explosion*. Each variable which is modifiable in the system, increases the amount of states exponentially. While a system with five, one byte wide variables already has $2^{8 \cdot 5} \approx 10^{12}$ states, a real life system with thousands of multibyte variables features exponentially more. This calls for verification at different levels, allowing for different objectives and intensities, to still gain confidence in the correctness of the design.

Verification Methods Different approaches have to be taken to overcome this issue and still show, with a given confidence, that the system behaves as it should. Two general approaches to verification can be differentiated: **testing** and **formal** verification [Cee19].

2.2.1 Testing

A quite straightforward approach of verification is testing a system by applying stimuli and checking whether the outputs react as they should. This means that a way to input and observe defined signals from the system is needed. With those one can see if it reacts accordingly. Different approaches at different development stages of the project exist.

ASICs

For ASICs a coarse split can be made by pre-silicon and post-silicon verification.

ASICs are silicon chips which are designed and implemented on hard-wired silicon. The design process of such a device consists of many steps. A key milestone is marked by the "tape-out" which refers to the point where the design is produced in silicon [Tar21]. All verification performed before the first available hardware is considered pre-silicon.

Pre-silicon verification is the process to check if the design behaves as it should, before having the actual hardware available. This means, that either simulation or emulation has to be used to check the functionality of the system. Simulation is usually much slower than real-time while emulation (at least in the semiconductor context) usually refers to specialized hardware which can speed up these tests by emulating the code on a programmable logic device [Mis+17].

Post-silicon testing refers to the process of exercising the actual hardware inside a physical testbench which models the environment the design is supposed to operate in and allows the verification engineers to specify inputs and outputs. This form of verification faces its own challenges since it is much harder to control and observe the Device under test (DUT). Debugging is also harder in this setting, since the internal signals of the system are not available for easy access. Developers can overcome this issue by including on-chip instrumentation which enables observing internal signals. Choosing which signals to observe is another difficulty during design as one cannot modify the choice of signals after tape-out [Mis+17].

FPGA

Since these devices are configurable in nature, the approaches which can be taken can be more varied compared to ASICs, although the techniques as described above stay similar. Simulation like in *pre-silicon*, as well as the physical testbench approach similar to *post-silicon* verification can be chosen. Since an FPGA can be reconfigured, the design can be tested on hardware at earlier stages and can also be debugged more easily. In general, the flexibility of this device can be exploited by developers to test it in different configurations. It is also easier to include instrumentation such as integrated logic analyzers to gain visibility and control of signals within the hardware. For even more visibility of the design as well as avoiding synthesis and routing, simulation can still be a very useful tool. In general, all methods used for ASIC verification can be used with fewer constraints.

SoC including FPGA and Hard-Core

As discussed in Subsection 2.1.3, vendors produce devices which have both an FPGA and Hard-Core which are tightly connected. This kind of system causes additional testing challenges. To simulate a system like this, different approaches can be taken.

Many HDL designs are made up from blocks, which are then connected using busses [Zab12, p. 2]. For testing the individual blocks, fully contained within the HDL code, conventional hardware simulation can still be used.

For system-level testing, designs on these platforms are often tightly coupled, meaning the software which controls the hardware is dependent on the interactions with the hardware as well as the other way around [Zab12]. That means that for system-level testing, new, more capable approaches need to be found which this work tries to contribute to. Chapter 3 and Section 2.3 are going into more details.

2.2.2 Stimulus Generation

For testing a design, different input scenarios have to be generated.

By writing *directed tests*, different scenarios for the DUT are created by the verification engineer. This means that the input sequences are handcrafted and the output values of the design are compared to values anticipated by the tester. This can be repeated in different ways and for different scenarios. All of these scenarios should be generated from the specification, to verify the expected behavior in every relevant case [BGB02]. While this approach can cover interesting cases and is usually fast to get started with, every test case only covers a small set of scenarios depending on the implementation.

Another approach which can be taken for generating input stimulus is *constrained random testing*. This method generates random input stimuli which can be constrained to only generate valid/invalid/interesting/... cases. Completely random input values could generate scenarios that are not possible in real operation and thus are not useful for verification. Constraints may restrict values to fall in specific ranges, or set them to values

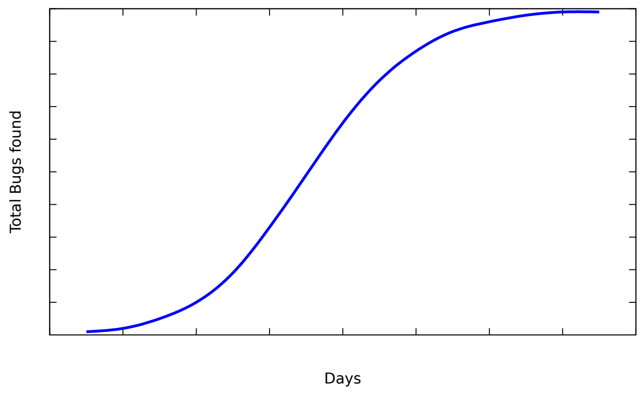

Figure 2.2: Expected bug discovery during verification

depending on other random values. This allows for generating complex scenarios which are still random. A positive effect of this method is, that with increasing simulation time, more combinations of states of the design are exercised. This means that the amount of tested scenarios is not dependent on the time spent developing test cases anymore, but dependent on the simulation time. The time invested into writing the random generation and checking is usually greater than writing directed tests, but once written the amount of tested scenarios can be much higher [Cee19]. Since the DUT is receiving random inputs, a way of checking the outputs is needed. A reference-model, which receives the same inputs as the DUT can be used to achieve a self-checking testbench, which compares the outputs of the design with the model.

2.2.3 Measuring Verification Progress

A difficult question during verification of a device is “Is the design tested enough?”. Edsger W. Dijkstra comes to the following conclusion regarding testing: “Testing shows the presence, not the absence of bugs” [BR70, p. 16]. Therefore, with testing we can never say that a system is bug-free unless it is very small and all combinations of inputs can exhaustively be tested. In real-life and within the engineering process, metrics to shed some light on the progress like coverage and bug-discovery-rate are used to make an educated decision on when the design is sufficiently tested. The bug-discovery-graph is usually expected to follow an S-curve which starts slow, increases and plateaus again as seen in Figure 2.2 [BGB02].

Coverage

There are different types of coverage metrics. All represent some percentage of what has been checked within the design. Optimally, verification achieves full coverage (100%) on all relevant metrics. This might not be possible in some cases, but in general, higher coverage of the design means more confidence in its functionality. Keeping the previously mentioned statement of Dijkstra in mind, that testing cannot show the absence of bugs, even a coverage of 100% cannot guarantee that all bugs of a specific type have been found except in very simple cases. This is the case because every metric abstracts the execution of a program, which can be seen from the selection of metrics introduced in the next paragraphs.

Another difficulty is, that no formal way of comparing metrics is available (yet). So no "optimal" metric exists and one metric might show different areas of the design which are still not sufficiently tested, compared to another one [TK01].

Despite all these issues just described, coverage metrics are still used very commonly since they still can be used for gauging the verification process. So for an actual verification task, these metrics may be arbitrarily chosen and can vary widely in complexity. Tasdiran and Kreutzer provided guidelines on choosing suitable metrics:

- Overhead of measurement should be tolerable
- Improving the coverage metric by generating new stimulus should be possible with reasonable effort
- The available tools should not be needed to be modified a lot to measure the chosen metrics

[TK01, p. 3]

Code coverage This type of coverage stems from software testing, but can also be applied to hardware testing. It is measuring which fractions of the code have been executed. In software, this concept is straightforward, in hardware a simulator needs to measure signals and relate them to their generating code.

Line Coverage measures which lines of code have been executed while running the code. Measuring if a line is executed is comparatively easy to achieve, unfortunately coverage can be at 100% while functions of the design have never been executed. More sophisticated metrics have been developed for this reason.

Branch Coverage measures the fraction of branches that have been taken. Meaning that in an `if else` code fragment, it is required that both the `if` and the `else` branch are taken.

Expression Coverage measures the different outcomes of an expression in the code. Meaning that to achieve full coverage on the expression `a and b` needs to evaluate to true and false once. Meaning two executions with `a=true b=true` and `a=false b=true` would result in full coverage, although not all variables changed. Other metrics, like Modified Condition/Decision Coverage (MC/DC) take this into account.

Path Coverage measures the amount of paths in the control flow graph that are taken. This metric is quite hard to check, and a simple loop can make it impossible to fulfill [TK01].

Circuit-structure-based coverage These class of metrics use the circuit as the measured component. Hardware designs can be split into the data and control section: A metric can measure data flowing from *register-to-register* for instance, where each feasible path must be exercised. For a counter this might mean that it must be initialized, reset and reaching its minimum and maximum values.

Finite-state-machine-based coverage Finite state machines (FSMs) are used to implement control functionality in hardware designs. Different techniques such as *state*, *transition* or *path* coverage can be used to measure the verification progress of an FSM. In real designs, abstract FSMs might be needed to model a more controllable, abstract representation of the design [TK01].

Functional Coverage All the previously mentioned coverage metrics are related to the structure of the design. Functional coverage is different from that, since it is derived from the functionalities a system should provide. It is usually specifically written for one design and represents a set of functions and scenarios a design should be tested in [TK01]. Functions to be exercised need to be identified and derived from the specifications. The specified functions need to be described in terms of the input and output values of a system and their relations. This task is optimally done by an independent verification engineer who does not yet know the assumptions embedded in the design and uses the specification to derive these scenarios [Cee19].

Different methodologies to minimize errors exist for finding all relevant scenarios in the specification, which is a task executed by an engineer and therefore prone to human error (forgetting scenarios and configurations). For defining a functional coverage model, splitting scenarios in different groups can be helpful. An approach would be to define mutually exclusive and collectively exhaustive (MECE) groups. This can help in identifying and distinguishing functionality while also limiting it to one occurrence [Cee19, p. 12]. Other groupings, which divide scenarios semantically, exist as well. An example for dividing cases are semantic categories like: Use cases, modes of operations, states in state machines; Illegal conditions and unexpected use cases; Interesting scenarios. An in depth discussion into the use and pitfalls of such groups is made by Sprott et al. [SMG15].

2.2.4 Formal Verification

Formal verification is a way of showing that a design does what it is supposed to, not by applying stimulus, but by proving that it behaves as it should, according to a formal specification. In other words, the design fulfills given formal properties. These proofs are usually assisted by software, since doing it by hand is both infeasible and error-prone. As in simulation, different techniques exist.

Formal Property Verification

Formal properties are written in temporal logic, a formalism which allows for describing temporal dependencies. For hardware design, a common language in which these properties are written is the Property Specification Language (PSL) which supports Linear Temporal Logic (LTL) operators like *always*, *next* and *until*. A temporal property can look like: `assert always req -> next (ack until grant)`. This property makes sure, that whenever `req` is true in a timestep, `ack` is true in the next timestep and `ack` remains true until `grant` becomes true. Another language which is often used for such properties is SystemVerilog Assertions (SVA).

These properties can be shown for a design by tools applying the method of *(bounded) modelchecking*. While formal property verification can be very useful for verification, it is usually applied on smaller parts of the design like single blocks. The reason is, that the amount of states for a single block is usually much smaller than the states of the entire system which makes it easier for automatic tools to prove given properties [Cee19, p. 24]. From the verification engineer's point of view, commercial or free tools read the HDL design as input and proof the properties specified by the engineer. So although the proofs are mathematically grounded, it is still in the hands of the verification engineer to formally specify the intended behavior which, in itself is not a trivial task.

Formal Equivalence Checking

While property verification checks if a design fulfills defined properties, equivalence checking proofs that two versions (for example for different abstraction levels: Register-Transfer-Level (RTL) and Netlist) of the design are equivalent to each other. In this context, equivalence means that certain, defined, key points of the design behave the same [Cee19, p. 24]. This can for example refer to only the inputs and outputs of the design while the internals are not considered - also known as a "black box".

This type of verification is, as hinted above, often applied on different abstraction levels. So from RTL to Netlist or even from a more abstract level like C to HDL. With High level synthesis (HLS), combined with formal equivalence checking, the entire design can be written and tested in a high level language and then translated to low-level hardware while knowing that it is still equivalent to the high level design in given aspects. This can result in shorter development times compared to a traditional design flow.

Formal Co-verification of HW/SW Designs

As mentioned in chapter 2.1.3 in an SoC hardware and software are usually tightly coupled which needs specific tools for formal verification. Mukherjee et al. present *HW-CBMC* which allows validation of co-designs written in Verilog and C [Muk+17]. Their technique employs *bounded modelchecking* with special purpose primitives in C which are used to model the HW-SW interface. Furthermore, this tool supports a subset of PSL for specifying properties the system must fulfil. The special primitives can both describe "transaction level properties", which describe interaction between hard- and software and vice versa, as well as “component-level” properties. These stay within the level of an individual component of the design which can either be a hardware model, a software model or an interface model [Muk+17]. Other approaches are discussed in scientific literature such as the works from Li et al. [Li+10], Große et al. [GD05] and others. Unfortunately no implementations of tools except HW-CBMC seem to be available for now.

Formalizing Properties

Although formal methods are able to proof that a design fulfills given properties, this does not mean that the design does not exhibit erroneous behavior. This is because the formal specification is still written by a human who can interpret the specification differently from what the design engineer had in mind. Or an error happened while translating a specification to a formal language. To have better control over this issue, different techniques have been developed. Usually a semiformal (graphical) language is used to communicate the design requirements to people from different backgrounds. One such graphical language is the Unified Modelling Language (UML) to describe the design on an abstract level which can then be translated to a formal specification [Cee19].

Another semiformal approach, proposed by Ceesay-Seitz et al. are *natural language properties* which can be understood by persons with different backgrounds and can be easily translated to a formal property language [Cee19]. An example property like “Cycle is a MC and alarm function is not deactivated and integral value is greater than or equal to threshold implies that alarm is on.” [Cee19, p. 79] can be understood without knowing a special formal language. Since it follows a semiformal syntax, it can also be translated directly to SVA (or PSL) by the verification engineer, making sure that the intent of the properties is correct.

2.3 Integrating Hardware and Software Simulation

Hardware/Software Co-Simulation is the process of simulating both a hardware description, as well as software interacting with it. This can be useful at many stages in the development of a project. *Virtual Prototyping*, as another term for HW/SW Co-simulation, can be used when hardware of the system is not available yet to already write software using the system. Depending on the accuracy, these models can be used to estimate performance and help with the overall design of the project. In the special case of SoCs with an included FPGA, co-simulation can furthermore help development because the visibility of signals is increased which helps with debugging and analyzing the actual behavior of the design. If the simulation is accurate enough, it can also be used for functional verification.

2.3.1 Hardware Simulation

There are different methods of hardware simulation. Methods can be split into time-discrete and continuous. Discrete can furthermore be split into time controlled and event-oriented categories. The chosen method has influence on the abstraction of time and the models which can be used. This also impacts the similarity to the real world. In continuous simulation, transistors as well as other components are modelled with differential equations and are then solved with numeric methods. This is relevant for analog simulations, or simulation of the behavior of individual gates, but for Very Large Scale Integration (VLSI) these models are not viable because of their comparatively low speed. In discrete simulation, time progresses in discrete timesteps. This simplifies simulation, but also decreases the accuracy of the model [Mat01].

Discrete simulation methods

Time-controlled simulation A fixed timestep (Δt) is taken, and at each step the system is analyzed. Everything that happens in between those timesteps in the real system is only processed in the next timestep. Therefore, the choice of the length of this step is crucial to the accuracy of the simulation and also heavily depends on the system itself. The smaller Δt is chosen, the closer the simulation is to the real system. The bigger it is, the faster the simulation is running [Mat01]. This means, that a compromise between accuracy and speed has to be found.

Discrete event simulation (DES) DES is an approach where time increments are of variable size and calculations are only done, when something is happening within the system. To achieve this, the system is split into objects with states which are described through a set of state variables. A change of an object state, meaning that one of its state variables changes in value, is represented by an event. Each event has a timestamp (which does not necessarily need to represent the time progression of the real world) which tells when an event will occur. Additionally, to the timestamp, each event also has a routine which is executed when the event is processed. This routine can change the object state as well as add or delete events to the event list. For simulation, all events are saved in an event list. This event list stores all events sorted by their timestamps so that the next event in time can easily be accessed [Mat01].

The simulation is executed by retrieving the next event in time, increasing simulation time to its timestamp and then executing its routine. This means, that timesteps are not uniformly incremented, which also means that times when no activity in the design can be observed are skipped. Thus, DES can be faster than other approaches like the time-controlled simulation. A high level algorithm description is shown Listing 2.1.

Simulation of HDL Code

Using the mentioned methods, hardware can be simulated on different levels of abstraction, such as gate-level, RTL as well as higher and lower levels [Mat01]. For verification as

Algorithm 2.1: Basic discrete event simulation algorithm [Mat01]

```
while event_list not empty do
    e ← next event from event_list
    global time ← timestamp of e
    new_events ← execute routine of e
    add new_events to event_list
end
```

well as debugging of FPGA designs, RTL is a suitable and often used model, as many designs are written on this level of abstraction, in VHDL or Verilog.

DES is a useful simulation method at this level, where an event corresponds to a change of value in a signal. All gates or higher level blocks, which are influenced by this change must re-evaluate their outputs and might change those as well, which generates new events.

The method for DES described above, must be extended a bit since changes of the outputs of a gate occur later in time than the change of the inputs, but the timestamps for "regular" observable events should not be influenced by this. Because of this reason, *delta timesteps* are introduced. They describe a different time dimension and are used for all timesteps after the output of a gate has changed. This means that on a higher level - as a reaction to an external event - the output is available after this regular timestamp, but for calculating all the intermediate signal values which are caused through reevaluation of truth tables, *delta timesteps* are used [Mar08].

A demonstration of this process is given in Figure 2.3 and Table 2.2 on the example of an RS Flip-Flop.

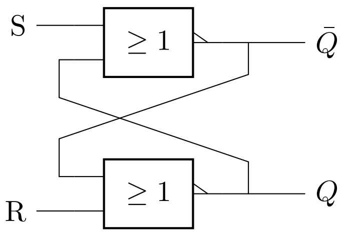

Figure 2.3: RS-Flipflop for demonstration of delta cycles

In the beginning the output Q of the FF is one and at t_s an external event, where the R signal is set to 1, occurs. The lower NOR-Gate processes this information and generates an event one delta cycle later where its output Q is updated to 0. This changes the inputs to the upper gate which causes an event at 2δ to change $\bar{Q}$ to 1. This again propagates to the lower gate, which evaluates to the same output as before. So at 3δ no event is issued anymore and a stable state for this timestep has been reached [Mar08]. For the

	$t_s + 0\delta$	$t_s + 1\delta$	$t_s + 2\delta$	$t_s + 3\delta$
R	1	1	1	1
S	0	0	0	0
Q	1	0	0	0
$\bar{Q}$	0	0	1	1

Table 2.2: Signal values with increasing delta cycles

observer, who does not care about the delta timesteps, the result is already visible in this timestep.

2.3.2 Software Simulation / Machine Emulation

Compiled software is targeted to a specific machine and instruction set. While desktop CPUs usually feature the AMD64 instruction set (with extensions), embedded systems often have different instruction sets such as ARM, AVR and others. Most of them also use different data busses, peripherals and operating systems (if they have one at all). This means, that to run the exact same software that is running on an embedded system on a desktop device, a simulation of the hardware is needed.

In this context, running multiple OSes (for the same instruction set and hardware) on one CPU is called virtualization [Rod+12]. When running a different OS or program written for another instruction set and hardware, it is referred to as emulation. Since SoCs usually have different instruction sets as the computer the software is developed on, emulation is needed for running the software on a usual desktop CPU.

QEMU

QEMU is an open source project initiated by Fabrice Bellard which is a “fast machine emulator using an original portable dynamic translator” [Bel05]. It allows an unmodified *target* operating system to run on another *host* operating system. To achieve this, it is made up by subsystems: the CPU module which emulates the instruction set, emulated devices like the peripherals in a microcontroller, generic devices which connect emulated devices to actual host devices, debugger and UI [Bel05].

Dynamic binary translation in QEMU To emulate the target CPU instruction set, QEMU uses a technique called dynamic translation which converts target instructions to host instructions in real-time. To speed up emulation, the translated code is cached for later use [Bel05]. QEMU solved the issue of easy portability by splitting the *target* CPU instructions in simpler *micro operations*.

In the first version of QEMU those *micro operations* were implemented in a short piece of C code and compiled for each *host* CPU by GCC, which generates an object file for the required *host* instruction set. This object code is then used to generate the dynamic code generator which translates the *guest* binary by combining and patching the *micro* *operations* to execute the correct *host* instructions [Bel05]. Nowadays, the Tiny Code Generator (TCG) is used which also converts *target* instructions into an intermediate representation which are then translated to *host* instructions with optimizations applied [Rav11]. This system still focuses on easy portability between different systems.

More techniques are used by QEMU to solve further emulation issues like register allocation, memory management, self-modifying code and condition code optimizations. Details about these topics can be read in the initial paper describing QEMU [Bel05].

To actually enable full system emulation, devices need to be emulated as well which is covered in more detail in Section 3.3.3.

2.3.3 Connecting parallel simulators

Connecting two, in the scope of this thesis different, simulator processes is needed for hardware/software co-simulation. Different techniques exist solving this issue for n processes. The main issue of distributing the simulation of a physical system is to find a way to synchronize all processes. This means that each process, although only communicating by messages, operates on the same time increments and therefore deterministically simulates the entire system.

Synchronous parallel simulation

Synchronous parallel simulation handles the issue of time by having a central time controller which handles progress of each individual process. This means, at each timestep is handled by the individual processes and then forwarded to the communication backbone. It then forwards this to the receiving processes which can continue evaluation. While this method is comparatively easy to implement, the slowest process always determines the speed of the rest of all the other processes. Depending on the distribution of the load between the processes, this can lead to a severe slowdown [Mat01]

Asynchronous parallel simulation

Another approach for synchronization is to let every Logical Process (LP), which is another name for the process running on the computer, keep its own current simulation time and infer the relationship of its current time by the messages exchanged with other processes.

The first methods for achieving such a distributed network were described by Bryant [Bry77], Chandy and Misra [CM79] and thus are often called CMB-Algorithm [Mat01]. A brief introduction on this method for n processes is made in the following paragraphs.

A LP is a single process of the simulator which tries to simulate a physical system, called PP. Each LP can be connected to any amount of other LPs. Each LP only communicates by messages containing a timestamp and a message. A special kind of message, the `NULL` message can be sent which must be distinguishable from the messages sent for

Algorithm 2.2: Logical Process Pseudo-Algorithm from Chandy and Misra [CM79]

```
Initialize all timestamps to 0
while Simulation is running do
    //Select next processes
    next_in = all connected LP which sent their last message with a timestamp
      of local $t$ (and this time is smaller than $t + T_{ij}(t)$)
    next_out = all connected LP which received their last message from this
      process with a timestamp of local $t$
    //Compute step
    compute the new message for each LP in next_out according to the simulated
      Physical Process (PP)
    //I/O step
    do in parallel
        foreach next_out do
            if output has not changed then
                send NULL with the timestamp $t + L_{ij}(t)$
            else
                send the calculated timestamp and message
            end
        end
        Wait to receive messages from the expected processes next_in
    end
    t = the minimum timestamp a message was received or sent.
end
```

communicating the state of the PP [Bry77]. Every LP i also keeps track of the latest timestamp it received T_{ij} and sent T_{ki} a message for each connection.

Furthermore, each LP must be able to predict the future of its outputs from its current inputs to some extent. This means, that LP i must be able to predict that if it received messages until the time t, it will not create an event for its connected process j, before $t + L_{ij}(t)$ where $L_{ij}(t)$ is an arbitrary function with a value greater than 0. In the case of simulating a logical gate, the value of $L_{ij}(t)$ is simply the delay induced by a gate, so $L_{ij}(t)$ becomes a constant. In other words, a logic gate can only react on an input once it received it, and it only outputs a new value after a specified amount of time. This information must be known by each LP. The same algorithm is running on each process.

The algorithm 2.2 running on each LP consists of three steps. In the first step it selects the appropriate processes of which it is expecting input or needs to send output. The second part actually calculates the behavior of the PP, so the actual simulation happens here. At last, the input and output operations are performed. This step is where the synchronization between processes occurs since each process waits to receive information of the state of the other processes. The use of `NULL` messages is of high importance in this step. By including a newer timestamp, they are used to tell the process that there will be no event in this time period, and it is safe to continue in time. If no `NULL` messages were in this algorithm, a process would stop and wait indefinitely on messages on its inputs (if no event occurs in each process) and therefore deadlock.

This class of algorithms is guaranteed to be deadlock free and also terminates for a given simulation end time [CM79]. A drawback of this solution is, that the amount of exchanged `NULL` messages might be quite high. Remedies for this problem, for example by sending these messages only on demand, have been developed [VH99].

Another way of synchronizing processes is to use optimistic methods. This means, that each process runs at its own speed and when an event arrives that lies in the past, the process must revert its state and all the messages it sent starting from this point. Such a rollback is usually quite an expensive operation. Different approaches for this method exist [Mat01].

2.3.4 State-of-the-art HW/SW Co-simulation approaches

Zabołotny discusses different approaches in "Development of embedded PC and FPGA based systems with virtual hardware" [Zab12]. Further distinctions were made by Naia [Nai15].

Using a physical platform as a testbench An approach very similar to ASIC post-silicon verification where the FPGA is exercised by a physical testbench.

Using a hardware simulator on the whole system If the Central Processing Unit (CPU) core of the system is available as source or netlist, it could be used to run the system on a single hardware simulator. This is too slow for running an entire OS [Zab12]. A visual representation can be seen in Figure 2.4.

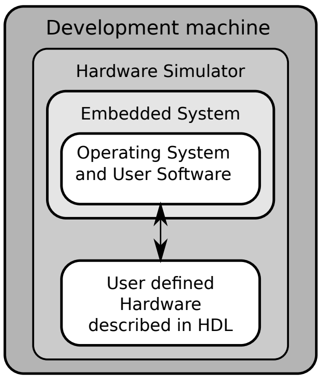
Development machine
Hardware Simulator
Embedded System
Operating System and User Software
User defined Hardware described in HDL

Figure 2.4: Hardware simulator for whole system [Zab12]

Hardware simulator coupled with software on host-machine The software usually running on the embedded core is executed on the host machine. Communication with the hardware must be modifiable, so it can be coupled to a hardware simulation as well as the real hardware. The software also needs to be compilable on the host PC. This method can be very useful for relatively simple software as demonstrated by Zabołotny, but the lack of the actual environment the software is intended to run on can be an issue. This method does not suit emulation of "bare-bones" software, meaning that the software is running directly on the CPU without an OS in between abstracting the hardware. Shown in Figure 2.5.

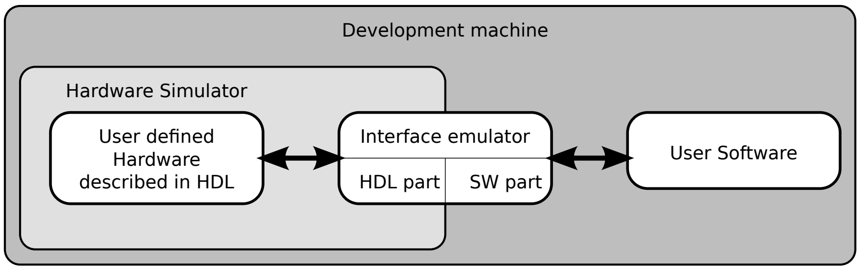

Figure 2.5: Hardware simulator coupled with software running on host[Zab12]

Co-simulation using hardware-level and software-level simulators This method uses two simulators. One must be capable of simulating the target embedded processing system and another one is capable of the hardware simulation. A link which provides synchronization between those programs is needed. The result of this approach would optimally be fast simulation of software and cycle-accurate simulation of the hardware [Zab12]. Can be seen in Figure 2.6.

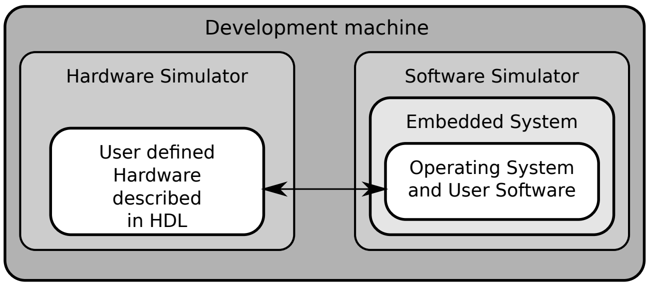
Development machine
Hardware Simulator
Software Simulator
Embedded System
User defined Hardware described in HDL
Operating System and User Software

Figure 2.6: Hardware simulator coupled with software simulator[Zab12]

Full System Software Simulation Hardware is implemented and integrated into software simulation. The hardware can be modelled on different abstraction levels but needs to be accurate enough that software can sufficiently use them. Two representations of the device need to be created which are consistent with each other. This can be quite demanding, but can also help development if used early in the design process [Nai15]. Visualized in Figure 2.7.

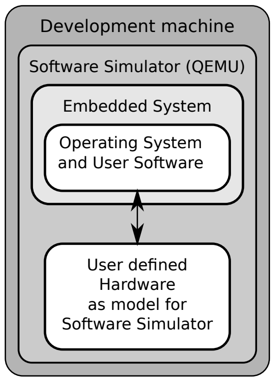
Development machine
Software Simulator (QEMU)
Embedded System
Operating System and User Software
User defined Hardware as model for Software Simulator

Figure 2.7: Software simulator for the whole system[Zab12]

All of these methods can be used to run a simulation of the system which can then be tested in different conditions. They differ in speed, accuracy and usability for different processes within the design of the system, so there is no best solution which works in every single situation.

CHAPTER 3

Hardware/Software Co-simulation

3.1 CROME Architecture

As briefly outlined in the introduction, CERN Radiation Monitoring Electronics (CROME) is an ionizing radiation monitor developed by the Instrumentation & Logistics Section at CERN. It is designed to continuously measure the ambient dose rate, generate alarms at different levels and interlock experiments [Bou+17]. Furthermore, it communicates with a supervision software which logs the measured data and sets functional parameters.

The system actually consists of several units:

- CROME Measuring and Processing Unit (CMPU)

[1]Redrawn with kind permission of the author

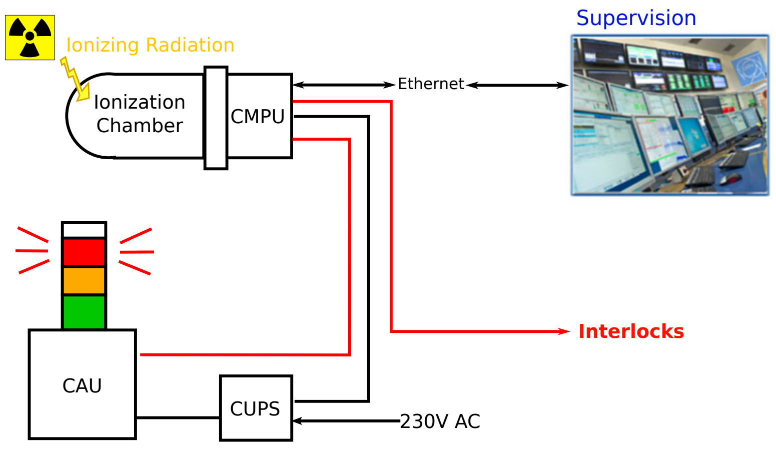

Figure 3.1: Overview of CROME System Architecture [Bou+17] [1]

- CROME Alarm Unit (CAU)
- CROME Uninterruptible Power Supply (CUPS)
- CROME Junction Box (CJB)

The CMPU is available in two variants, the rack- and the bulk-version. The rack version is separate from the ionization chamber and meant for measurement in high radiation zones, as the processing system can reside in a low radiation area. The bulk variant is one compact unit combining the ionization chamber and electronics which can be wall mounted. It is intended for monitoring of low radiation areas. Ionizing radiation also has effects on operating electrical devices which can result in a Single Event Upset (SEU) and other effects. A SEU can temporarily cause a bit-flip which introduces a fault in the system. Many measures are available to mitigate this issue, and while CROME uses global triple modular redundancy and scrubbing using soft error mitigation Intellectual Property (IP), it is best to avoid exposure to radiation anyway.

3.1.1 CMPU

Frontend The frontend of the system is designed to measure radiation dose over a high dynamic range of 9 decades. Ionizing radiation is converted into electrical charges by the use of an ionization chamber which are then measured by this frontend. As different ionization chambers may be connected to the system, the characteristics of the input current may vary. The lowest, with reasonable accuracy, measurable current must lie in the femtoampere range. As there is a temperature dependence of the entire system, each unit is calibrated and can compensate for temperature. Additionally, each unit is able to provide a configurable high voltage to drive the ionization chamber [Bou+17].

System on Chip The Xilinx Zynq SoC inside the CMPU handles multiple functionalities such as controlling the frontend, readout of all connected sensors and processing of this data. It triggers interlocks and alarms based on the configuration received from the supervision which is saved on the device. Alarms and Alerts are triggered based on the average or the integral of dose over a given time period. A physicist decides, based on the monitoring requirements derived by a risk assessment, which value - integral or average - is used in which area. Moreover, all data is logged and can be read out by supervision. The system is designed to fulfill the time-critical needs of the readout as well as communication via Transmission Control Protocol (TCP) with a higher level, custom protocol. All safety-critical functionalities are running with Triple Modular Redundancy (TMR) within the FPGA. A block diagram of the most important data handled by the Programmable Logic (PL) (the FPGA within the Xilinx Zynq device) can be seen in Figure 3.2.

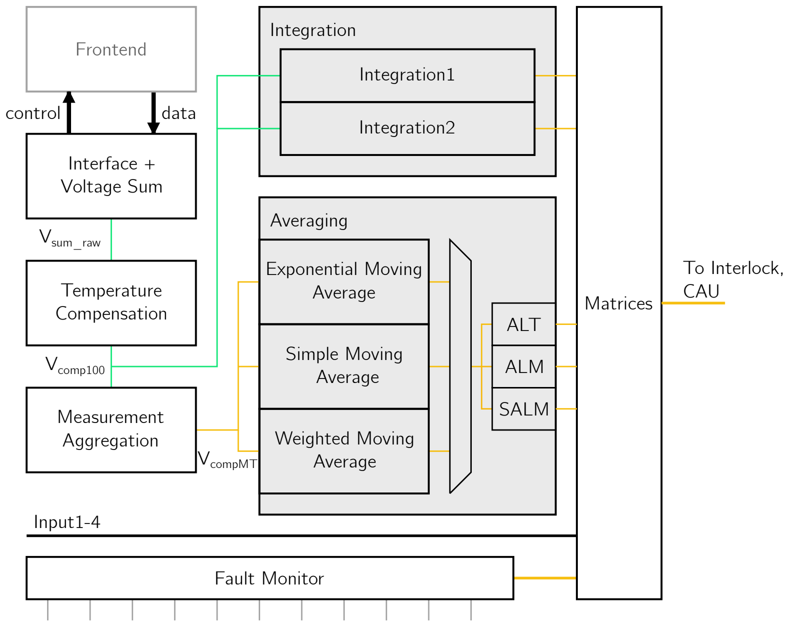

Figure 3.2: Block diagram of the data flowing within the CMPU Programmable Logic (PL) [Laf21] [2]

3.1.2 CAU, CUPS and CJB

The CAU receives alarms and alerts from the CMPU and can indicate those with different colored lights and sounds [Bou+17]. The "Alert" condition is less severe than "Alarm". Power, in the form of 24 V DC for the system is delivered by the CUPS. It is an uninterruptible power supply, meaning that in case of main power failure, the system can keep running autonomously. While each of the other components also needs to fulfill safety criteria, the main focus of this thesis lies on the CMPU as it contains comparatively complex software.

3.1.3 Zynq-7000

The Zynq-7000 SoC was developed by the company Xilinx Inc. and is a very flexible device, as it integrates a dual-core ARM Cortex-A9 processor with an FPGA. Those are interconnected by industry-standard Advanced eXtensible Interface (AXI) interfaces, with high bandwidth and low latency. The processor is usually referred to as Processing System (PS), the FPGA fabric as PL. The tight integration on a single chip offer lots of possibilities for the systems design and allow it to be used for demanding applications [Cro+14].

Nine different busses are available for communication between the two domains inside the Zynq. Four general-purpose AXI interfaces (split into two initiator controllers and two

[2] Reproduced with kind permission of the author

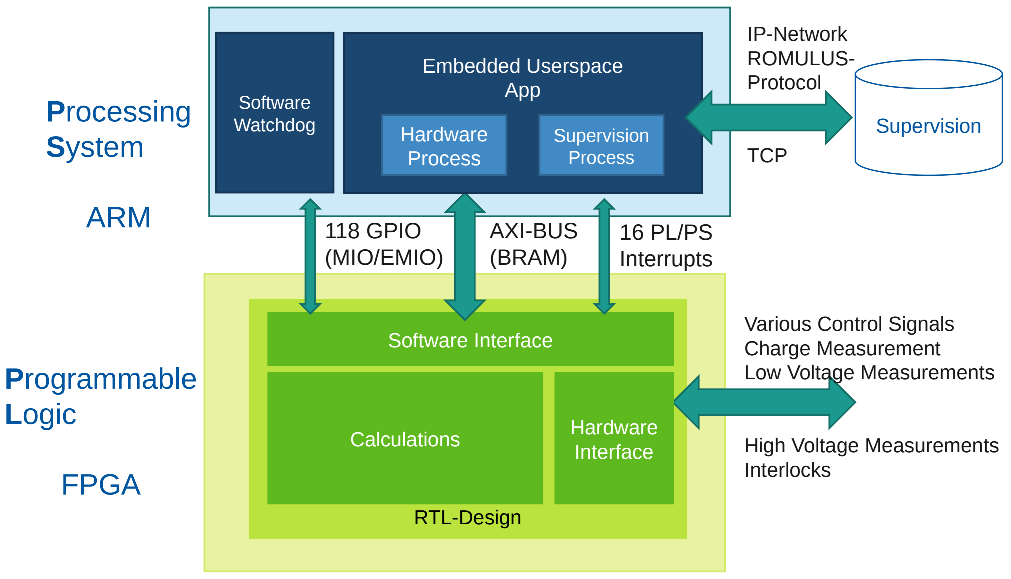

Figure 3.3: Block diagram of the partitioning of CMPU functions

followers[3]), one Accelerator Coherency Port (ACP) interface and four high performance AXI interfaces [Cro+14]. Additionally, a total of 64 Extended Multiplexed I/Os (EMIO) connections as well as further individual interrupt and watchdog lines are routed between PL and PS.

3.1.4 CMPU Architecture

The main control unit of the CMPU is a Zynq-7000 All-Programmable SoC as presented above. The safety-critical functionality of the CMPU is contained within the PL which can be seen in the lower (yellow/green) area in Figure 3.3. The communication with the supervision and logging of data is handled by the PS. It uses only one of the nine possible AXI interfaces for communication, namely the `M_AXI_GP0` Bus (Initiator, General Purpose). EMIO and interrupt lines are used as additional signals between PS and PL.

Processing System

An embedded distribution of the Linux Operating System (OS) is running on the CPU of the CMPU. It is developed with PetaLinux, an embedded Linux Software Development Kit (SDK) from Xilinx [Xil20a]. PetaLinux provides many tools for configuring the OS and its components. Furthermore, it can integrate the build process of software running on the device.

[3]Terminology is different in the AXI specification but changed within this document on purpose

As shown in Figure 3.3, an embedded application developed in the programming language C is running on the PS. It contains two parallel processes, one for handling the communication with the hardware and one which handles the network requests.

Hardware Process Communication with the hardware, referring to the PL, from the PS consists of transferring the configuration from the PS to the PL, as well as current measurement data the other way around. This process follows a protocol which was created for securely transferring parameters between the two domains. A detailed description of it can be found in the master thesis of Nicola Gerber [Ger17]. In the Linux environment, the parameters can just be read and written by accessing memory addresses belonging to two Userspace I/O (UIO) devices. Two interrupts can be read out via the UIO devices and are used to inform the software whenever new data from the PL is available or when the data transfer was erroneous. This means, the hardware process needs to wait for interrupts, read and write updated data and transfer this data to the supervision process.

Supervision Process The supervision software, used by the CERN operators, communicates with the CMPU over TCP and a custom protocol called Romulus. Current values, alarm conditions and other variables are displayed via that interface. It also allows them to reset alarms or change configuration of the device. The supervision process inside the embedded application handles all of these network requests and can also stream the most current values to the supervision software. It waits on requests, receives new data from the hardware process and handles higher level functions of the system. For example, it spawns new threads for managing data which is stored in the file system and answering historical requests (retrieving old log values).

Software Watchdog The Software Watchdog is an additional process which monitors if the embedded application is running and if it does not, tries to restart it.

Programmable Logic

As mentioned several times, the PL contains all safety-critical functionality. It is developed in the hardware description language VHDL. To make this part more secure against SEUs, triple modular redundancy and SEM-Intellectual Property (IP) is used. This means, that parts of the system are replicated three times and the outputs of them are then presented to a voter. The voter chooses the output according to the majority of its inputs. In the case of three systems, the majority are two equal signals.

Communication with the PS The software writes configurations to a Block RAM (BRAM) within the PL via the AXI interface. A Xilinx IP converts the bus transactions to actual memory controls. A checksum is used to verify correct transfer of the data which is then further transferred into an Error-Correcting Code (ECC) BRAM for safe storage [Ger17].

Connected Sensors The CMPU takes measurements of the environment using different sensors connected via I^{2}C and Serial Peripheral Interface (SPI). In total three ADCs are used, one for the radiation measurement and two measuring the high and low voltages on the board, detecting faults. Additionally, three temperatures and the relative humidity are measured using three sensors for temperature compensation and fault monitoring. Time between those measurements can be freely configured in multiples of $100\,ms$.

3.2 Hardware/Software Co-simulation Architecture

The approach “Co-simulation using hardware-level and software-level simulators” as described in Subsection 2.3.4 is chosen because of the requirements to the simulation. It must be accurate for verification which also means that as little as possible should be modified from the source code to enable testing. Furthermore, it must be able to boot the embedded Linux in reasonable time which excludes complete hardware simulation. Additionally, open-source projects and resources like a blog entry [Wer20] also exist which help to develop this approach.

The proposed architecture of the HW/SW co-simulation for the Zynq-7000 is suitable for basically any system using this device. It is capable to simulate all interfaces between PL and PS presented in Subsection 3.1.3, although speed might vary greatly depending on the amount of communication between PL and PS.

It relies on the use of open source projects as well as a mixed-language RTL simulator.

- (Xilinx) QEMU [Xilb] for simulation of the ARM Cortex A9 and the software running on it.
- LibSystemCTLM-SoC [Xil22] for communication between QEMU and the RTL-simulation.
- QEMU-devicetrees [Xila] is needed for instructing QEMU which devices are communicating with the RTL simulator.

The RTL simulator needs to be able to handle multi-language simulation because the design to be verified and simulated is written in VHDL but the library which handles communication between the two simulators is written in SystemC. Furthermore, the testbench introduced in Chapter 4 is developed using SystemVerilog. At the time of writing, only commercial simulators like Questa, Incisive or VCS to give some names, support all of these.

3.2.1 Overview

An overview of the proposed architecture can be seen in Figure 3.4.

The main program used to the PS side is QEMU, which handles execution of the Linux OS. Within, the applications mentioned in Subsection 3.1.4 are running. It is somewhat

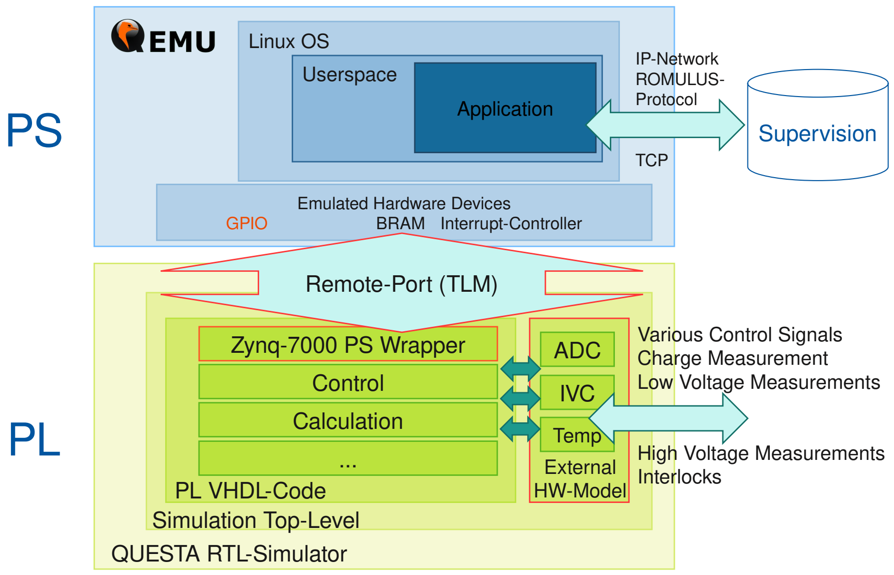

Figure 3.4: Overview of the co-simulation architecture (Red sections are developed and configured within this work)

fast and achieves emulation speed comparable to a real device. QEMU also handles the emulation of devices connected to the ARM processor such as memory, General Purpose I/O (GPIO) and interrupt controller. It was chosen because it is open-source, Xilinx already offers support for PetaLinux and also provides the remote-port protocol which helps in creating co-simulation [Xil20b].

For RTL simulation Questa, from the company Siemens, has been chosen because a license was easily available to the author. It is responsible for simulation of the entire hardware design as well as the connected external devices. Furthermore, it is able to run the library for communication LibSystemCTLM-SoC [Xil22]. In Figure 3.4 a few layers of wrappers are visible which transform the signals into easily accessible and higher level representations.

In the real world, the PL handles communication with peripherals like ADCs and temperature sensors. To enable full system simulation, these devices also need to be modelled. Fortunately, these devices do not need to be synthesizeable, which means that more abstract language constructs may be used for writing them. The development effort is therefore reduced compared to a synthesizeable version.

The big arrow in Figure 3.4 indicates the remote-port protocol used by the library LibSystemCTLM-SoC. It is used for communicating between both simulators and also provides synchronization. Each interface mentioned in subsection 3.1.3 can be modelled via this library. It uses Transaction Level Modeling (TLM) which cares about transactions instead of lower-level concepts like toggling individual bus-signals. One TLM transaction can represent a complete write process on a bus. This means, one transfer usually also only uses one or two exchanged messages per transaction.

3.3 Simulating the embedded software on QEMU

The software which runs on the embedded system is written in C and written to run on a Linux OS. Interaction with the real hardware is therefore abstracted by various interfaces and communication with the peripherals is handled by device drivers. That means, that the individual registers which configure and control peripherals like e.g. GPIO are usually accessed indirectly via pseudo-files.

QEMU is versatile and can emulate entire machines without modification. To achieve this the Central Processing Unit (CPU) needs to be emulated which, in the context of embedded devices, often has a different instruction set and architecture compared to the host. Additionally, peripherals connected to the CPU need to be modelled to allow for smooth emulation of entire system. More information about the instruction set emulation of QEMU can be found in section 2.3.2.

Running a simple image from PetaLinux - which uses the Xilinx fork of QEMU can be as simple as issuing the command [Xil20a]

```
# petalinux-boot --qemu --kernel
```

This command conveniently creates everything required, like the boot image and starts the respective Linux OS on the emulated machine. At the start of emulation, QEMU can use a devicetree (for embedded system emulation) which is usually used by embedded Linux to determine addresses and configurations of peripherals to create the emulated peripherals. This means no additional configuration file is needed as the devicetree, which is discussed in more detail in 3.3.2, provides all relevant information.

3.3.1 Memory-mapped peripherals on Zynq-7000

The ARM Cortex A9 within the Zynq SoC does not stand alone within the PS but also is connected to in-silicon peripherals like a DMA controller, SPI controller or GPIO controller which are also part of the PS. These peripherals are memory-mapped, which means that the registers which control those peripherals are accessible within the normal addressable memory range. Probably the simplest example of a peripheral is the GPIO peripheral of the Zynq which will be used as an example throughout the explanation of the simulation. The next paragraphs give a brief introduction to its functionality.

GPIO Peripheral

A single GPIO pin can have multiple configurations and statuses. In the Zynq-7000 two kinds of outputs Multiplexed I/O (MIO) and EMIO exist. They differ in the connection locations but feature the exact same control registers and controller.

Each pin can be configured as *input or output* and if the *output* is *enabled.* If it is disabled, the pin is set to a high-impedance state, meaning that it does not drive '0' or '1' and looks like an open line. Additionally, it can be configured as a source of an *interrupt* with falling, rising, both edge or level triggering. The *input state* can also be read or the *output value* set.

All of these settings can be written to, and read from a unified memory address space. When using an OS like Linux, compared to a bare-metal program, reading and writing to specific memory addresses is not as straightforward. Normally, a Memory Management Unit (MMU) performs address translation between virtual and actual addresses and protects from writing and reading to not related areas. To provide a uniform interface for different device types, a device driver abstracts interaction with those registers and takes care about the correct memory mapping as well as setting the correct bits to the correct values. But to emulate and understand the device, the actual registers and their semantics need to be known.

Register Name	Address	Width	Type	Reset Value	Description
`DIRM_1`	0xE000A244	22	rw	0x00000000	Direction mode (GPIO Bank1, MIO)
`OEN_1`	0xE000A248	22	rw	0x00000000	Output enable (GPIO Bank1, MIO)

Table 3.1: Example GPIO registers, Zynq-7000 Technical Reference Manual [Xil21]

Table 3.1 describes two exemplary registers from the GPIO peripheral, one of which controls the *direction* (so input/output) and the other one controls the *output enable* for each pin. They are located at the physical address 0xE000A244 and 0xE000A248 which are found as an offset to the base address of the peripheral in the data sheet. The GPIO peripheral has four banks, where each bank holds either 22 (Bank 1) or 32 pins. This split into four banks means, that when pin 35 should be set as an output, the bit at index three at address 0xE000A244 needs to be set to one because $35 - 32 = 3$ because `bank0` has 32 pins.

3.3.2 Devicetree

A devicetree is a data structure that is intended to be used by boot or client programs to figure out which peripherals are present and how they can be accessed.

> "A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics

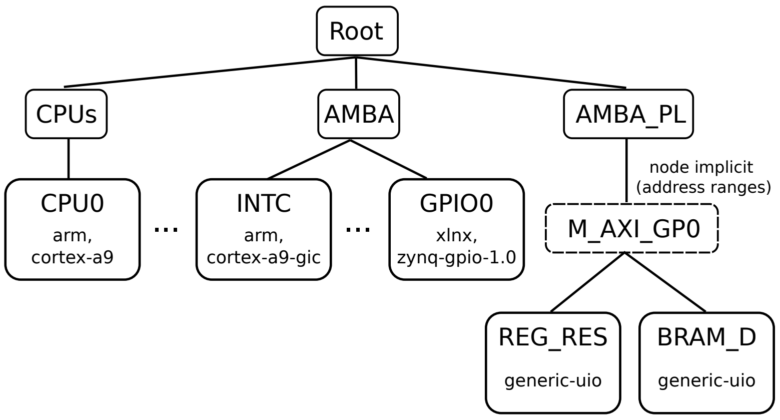

Figure 3.5: Selected parts of a Devicetree from a Zynq-7000 device. Small text indicates the compatibility string.

> of the device being represented. Each node has exactly one parent except for the root node, which has no parent." [dev20]

It is intended for embedded systems which differ from general purpose computers in terms of hardware customizability, because the embedded peripherals are usually fixed within silicon or on the Printed Circuit Board (PCB). A general purpose computer needs to probe the connected devices to know how to access them, for an SoC the devices are static and can therefore be described by this configuration.

Selected parts of an exemplary device tree can be seen at Figure 3.5. It is normally represented by one or more text files which describe the available hardware of a device. Each node can contain multiple properties which provide information about the respective peripheral. An important property is the `compatible` property which defines the specific programming model of the device [dev20]. This means, that a driver can be selected using that identifier corresponding to the device type. Other properties like `reg` and `#address-cells` can be used to describe address ranges a bus has or a device uses.

Listing 1 shows the GPIO node of the devicetree inside the Zynq-7000 described in Subsection 3.3.1. The address 0xE000A000 can be found in the `reg` property as well as the length 0x1000 (4096 decimal) of the respective memory area. Other information like interrupt connections and handling are also encoded.

3.3.3 QEMU Devices

As discussed in Subsection 2.3.2 QEMU not only needs to emulate the instruction set of the CPU, but also the devices connected to it. On a general purpose computers those are usually connected via busses like Peripheral Component Interconnect (PCI) and Universal Serial Bus (USB) which are meant to make the computer customizable. For this kind of system, command line arguments can be used to tell QEMU which peripherals are

```
gpio0: gpio@e000a000 {
    compatible = "xlnx,zynq-gpio-1.0";
    #gpio-cells = <2>;
    clocks = <&clkc 42>;
    gpio-controller;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupt-parent = <&intc>;
    interrupts = <0 20 4>;
    reg = <0xe000a000 0x1000>;
};
```

Listing 1: GPIO peripheral of the Zynq-7000 devicetree

```c
#include "qdev.h"
#define TYPE_MY_DEVICE "my-device"

// No new virtual functions: we can reuse the typedef for the superclass.
typedef DeviceClass MyDeviceClass;
typedef struct MyDevice{
    DeviceState parent;
    int reg0, reg1, reg2;
} MyDevice;

void my_device_class_init(ObjectClass *klass, void *class_data){
    MyDeviceClass *dc = DEVICE_CLASS(klass);
    //Do something to init class (static)
}

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
    .class_init = my_device_class_init,
};

static void my_device_register_types(void){
    type_register_static(&my_device_info);
}
type_init(my_device_register_types)
```

Listing 2: Minimal Type of the QOM with initialization [Doc21] (slightly modified)

connected. For embedded devices, QEMU can also use a devicetree to configure which components are in the emulated system. This means the same file which is used to get the configuration of the system can be used to create the system for emulation.

Historically, devices were developed for QEMU using different Application Programming Interfaces (APIs). The most recent API for this purpose is the QEMU Object Model (QOM), which provides an object-oriented system for dynamically registering types and allowing single-inheritance for types and multiple inheritance of stateless interfaces [Hab16].

A minimal device written in the QOM can be seen in Listing 2. It registers a new type which holds three variables `reg0` to `reg2`. Furthermore, it defines a name which is used for referencing it. It inherits from `TYPE_DEVICE` which is a basic device.

Such an emulated device may have functionality like any real device. It can be plugged into a bus, receive and create interrupts and claim a specific memory area for memory mapped I/O among other things.

The life-cycle of such a device is started by its initialization and then another step, realization which can be used to set up and construct additional things. It ends when it is finalized and all references to it are deleted [Fär13; Bon14].

3.3.4 QEMU Hardware Devicetree for co-simulation

The Xilinx QEMU fork [Xil20b] has the option to define two (different) devicetrees for one system. This is needed for the communication with the remote simulator. One devicetree is passed to the *guest* system. Another devicetree is used by QEMU to add, change or remove devices. This can be used to exchange "normal" devices with ones that support communication to the remote simulator. An example is seen in Figure 3.6 where the top tree is the representation the software sees and the bottom the one for QEMU. The dark red colored devices are special devices which implement the communication with the remote simulator.

3.3.5 QEMU devices for co-simulation

To make the presented QEMU devices communicate with another simulator, Xilinx QEMU uses the configuration of devices shown in 3.6. One virtual device called "COSIM" is created which does not represent an actual physical device, but handles communication the remote simulator. Upon creation, it opens a socket which is used for exchanging messages. It receives all incoming messages and forwards them to the concerned device (indicated by a number). This means, that if the remote simulation sends a packet to update the value of a pin on the GPIO device, it sends a packet to QEMU. The "COSIM" device receives this packet and forwards it to the GPIO device which can then further process it and set the registers to match that situation.

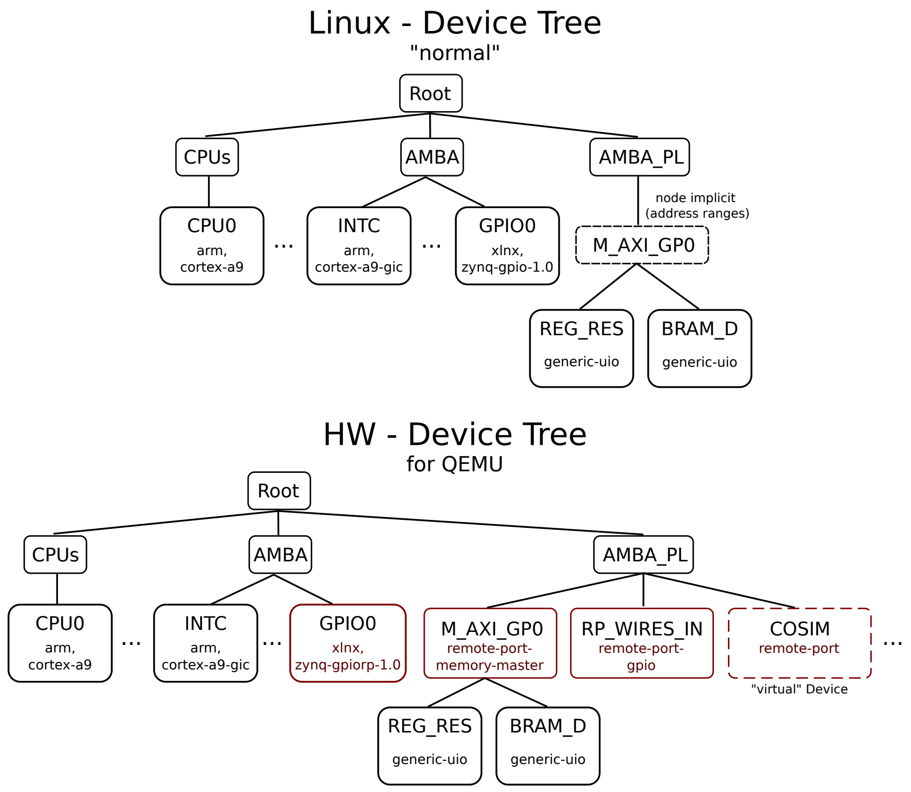

Figure 3.6: Different device trees for Linux and QEMU. The dark red devices implements the remote communication protocol

All setup and configuration of these connections to the remote devices is made in the hardware devicetree which highlights the importance of splitting the tree that the OS and QEMU sees. This split results in full transparency of the communication to Linux.

3.4 Connection between QEMU and Hardware Simulation

QEMU and the hardware simulation need to communicate to enable interaction between the hardware design and the software which controls it. Xilinx has developed the library LibSystemCTLM-SoC [Xil22] which largely helps with this issue. Not only does it provide the remote-port protocol which is used for the messages that get exchanged, but also provides an implementation for QEMU and SystemC. Furthermore, it provides bridges for SystemC which can translate the remote-port packets, which use the TLM approach, back to RTL wire signaling. One example of such a bridge is an `axi2tlm` bridge, which translates packets to AXI bus transactions on pin level.

SystemC is a C++ class library for system and hardware design. It provides functionalities to simulate concurrent processes which are written in C++ and the constructs

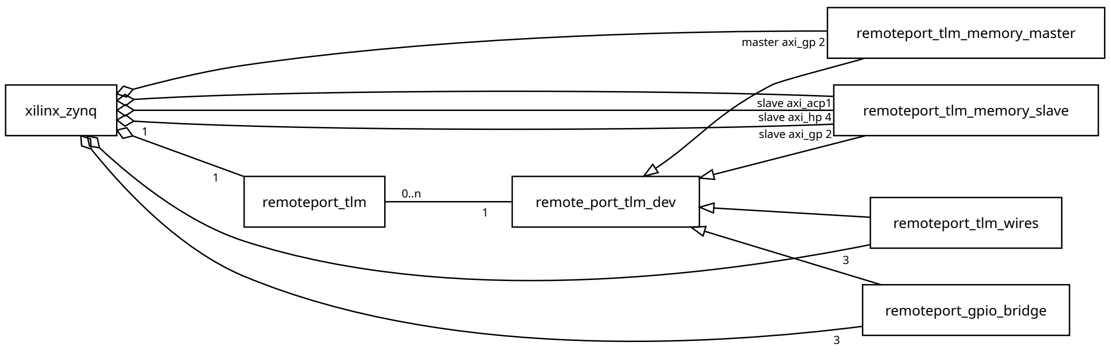

Figure 3.7: Class diagram of the Zynq-7000 SystemC SoC model

provided by SystemC which help modelling hardware devices. A target use case for this library is system-level modelling which can be somewhat abstract and is therefore intended to have a faster simulation speed compared to RTL simulation. Many mixed-language simulators have included this language in their set of supported languages which means that simulations exchanging signals and values between VHDL entities and SystemC modules are possible (with some limitations).

Communication between simulators is based on reliable point-to-point connections [Xil22]. These are enabled by using socket-based communication with TCP/IP-sockets or Unix domain sockets. The messages follow the remote-port protocol which may have various types. Remote-port's message types include `HELLO`, `WRITE`, `READ` and `INTERRUPT` messages, as well as `SYNC` packets. For initializing communication, the `HELLO` packet is exchanged which presents the capabilities of each simulator and initializes the connection. The other types are used for transmitting of read and write requests (for bus transfers), interrupts and synchronization. Each packet has a timestamp which is also used for synchronization.

All setup of the connection and forwarding to the relevant devices is handled by a device implemented in QEMU (Figure 3.6 "COSIM") and a class in SystemC (Figure 3.7 `remoteport_tlm`). They arbitrate packages and send and receive them over the sockets. These interfacing devices are then used by implementations on either side of the simulation. A QEMU device is implemented with the usual QOM and instantiated via the hardware devicetree. For SystemC the connection is created via an observer-pattern where each `remote_port_tlm_dev` observes the single instance of `remoteport_tlm`.

3.4.1 Zynq-7000 PS model for SystemC simulation

Figure 3.7 shows a class diagram of the Zynq-7000 SoC model partially provided by LibSystemCTLM-SoC. Functionality was added within this thesis to support all GPIO pins. Its purpose is to create access to all interfaces which are possibly available for communication between PL and PS on the software simulation side. Each bus or set of signal lines is modelled by a `remote_port_tlm_dev` which has virtual callback functions like `virtual void cmd_write(...)` implemented by respective devices. All of those devices are registered with a device number at the `remoteport_tlm` device which receives all messages over the communication socket and informs the relevant device when needed. The numbers are used as indication in both QEMU and the SystemC part, for routing each TLM packet to the correct device.

The `xilinx_zynq` class has multiple devices which handle and process these requests. It also provides an interface to the rest of the SystemC simulation with the various sockets as well as `sc_signals`. The sockets are still at the abstraction level of TLM, which means that a bridge from the LibSystemCTLM-SoC library needs to be used to convert it to signal level again.

TLM to Wire bridges LibSystemCTLM-SoC provides bridges which translate transactions from TLM generic payloads to wire signaling and vice versa [Xil22]. This simplifies development greatly as the hardware simulation is based on wire signaling and everything else, such as QEMU and the remote-port protocol is based on transactions.

Timing is implemented with TLM loosely timed semantics [Xil22]. In a loosely timed model, a single transaction is used for a complete read or write across a bus. This is in contrast to the approximately-timed model, which breaks down transactions into a number of phases, which correspond more closely to the actual hardware protocol [Ben08]. Furthermore, temporal decoupling [Ben08] is used as the synchronizing strategy between the simulators. The synchronization quantum needs to be equal on both the SystemC and the QEMU side. On QEMU this is handled by the `-sync-quantum` command line option, on the SystemC side this is set within the code.

3.4.2 Example Zynq-GPIO device

The GPIO device on the SystemC side needs to do what all the other bridges do - translate TLM transactions back to RTL signals. A difficulty for this device is handling the tri-state functionality of the MIO pins. Both simulators can potentially write to and read from the same signal which may cause collisions that need to be resolved.

This issue of multiple drivers with tri-state outputs is handled by HDLs with multistate logic. In VHDL the `std_logic` type can have 9 different states. Verilog and SystemC use four states. ‘0’ and ‘1’ represent the normal logic states, ‘Z’ stands for high impedance and ‘X’ for unknown. ‘X’ is used for example, if a signal is written to by two drivers: one driving ‘0’ and the other one ‘1’. This would result in an unknown state in real life and is therefore an ‘X’. Table 3.2 shows how two signals are resolved for SystemC’s `sc_logic` type, when competing writes happen on a resolved signal.

Handling collisions of multiple writers to MIO The communication between simulators is shown in the sequence diagrams shown in Figures 3.8 and 3.9. Some details

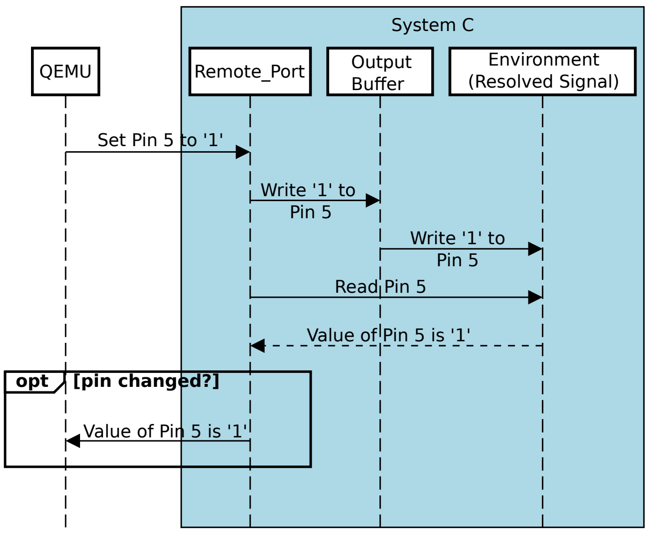

Figure 3.8: Sequence diagram of the communication between QEMU and libsystemctlm-SoC where the change event comes from software

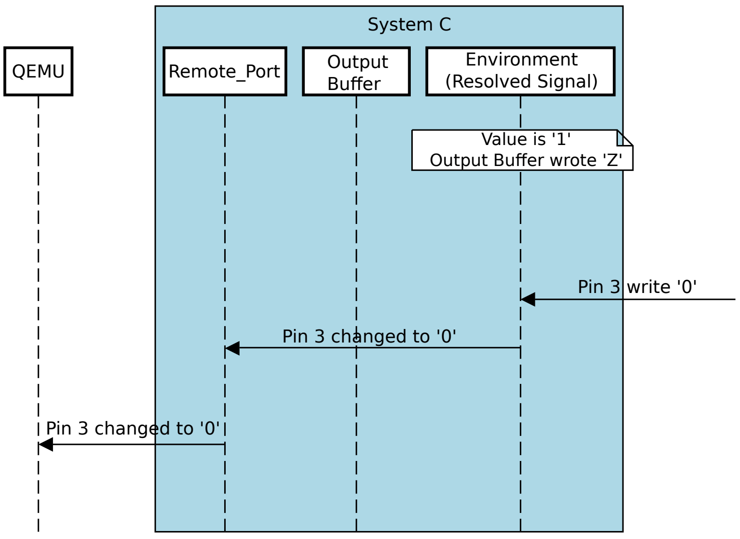

Figure 3.9: Sequence diagram of the communication between QEMU and libsystemctlm-SoC where the change event comes from hardware

D1 \ D2	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	Z	X
X	X	X	X	X

Table 3.2: SystemC Resolve table for two drivers D1 and D2

of the communication mechanism are not shown in the diagram. In the implementation messages are received, processed and forwarded to the concerning device by a remote-port handler on each side - the previously mentioned `remoteport_tlm`. As can be seen in the figures, only changes of signals are transmitted to limit messages between the simulators.

Instance in SystemC model An instance of the briefly presented class `remoteport_gpio_bridge`, representing the entire GPIO-chip, is instantiated within the `xilinx_zynq` class as seen in Figure 3.7. It is registered with the `remoteport_tlm` device to receive packets and provides all the signal lines (so 54 MIO and 64 EMIO) that are present in the physical hardware. The MIO lines are implemented as resolved signals while the EMIO is not resolved because they are split into separate input and output lines within hardware.

3.5 Hardware simulation with off-the-shelf simulator

A large section of the CMPUs functionality is written in VHDL. The hardware described by this code is simulated on a simulator like GHDL, Verilator, Incisive, ICS or Questa. In general, the process of running a simulation of any HDL code with such a simulator can be as simple as clicking a button or launching a script. Although more complex designs also need more complex compilation and more involved scripts.

Code for simulation can usually be split into two parts, the testbench and the DUT. The testbench generates inputs to the design and can also be self-checking, meaning that it also observes and inspects the outputs.

For simple use cases, the designer writes the code of the device, a testbench which generates input signals called "stimulus" and then observes the results. An integral part of this process is the waveform viewer which can be integrated in the Graphical User Interface (GUI) of the simulator. This tool shows the traces of the signals like a logic analyzer or oscilloscope would show when probing the real design. By analyzing these traces, the engineer can decide if everything works like it should. Moreover, this is also an important workflow for debugging.

To make use of an available simulator with the presented simulation architecture, the SystemC parts need to be integrated into simulation.

3.5.1 Choice of Simulator

An interesting open-source project is Verilator which translates SystemVerilog files into SystemC for simulation [Ver22]. This project is unfortunately not viable for simulation of the CMPU because at the time of writing, it only supports a synthesizeable subset of SystemVerilog and the design is written in VHDL.

Another open-source project which would be a candidate is GHDL which simulates VHDL by translating it to machine code [Gin17]. The issue with this program is that it does not support SystemC nor SystemVerilog out of the box which is a convenient language for writing a testbench.

For these reasons the commercial, multi-language simulator, Questa, was chosen. It supports VHDL, SystemC and SystemVerilog (which is a superset of Verilog).

3.5.2 Including the LibSystemCTLM-SoC model into hardware simulation

Designs for the Zynq-7000 are synthesized and placed with Vivado which also includes IP from Xilinx and other manufacturers for creation of the design. One available IP block, which needs to be used when interfacing the PL with the PS, represents the Processing System which allows to configure it and also exposes all connection to the HDL domain. When looking at the file structure of the IP generated by Vivado, a wrapper file called `system_processing_system7_0_0` provides a module/entity which exposes all signals available from the PS. The Xilinx-Verification IP (VIP) which is usually wrapped by the former file provides a functional simulation model which also performs protocol checking for the AXI busses [Xil17]. Since these functions need to be replaced with a custom solution, this file is exchanged with a reference to a custom file. To achieve this, a minor modification in the compilation script generated by Vivado is necessary.

In Figure 3.10 the red line indicates the replaced IP with the custom solution. The wrappers are needed for easily converting SystemC signals into the VHDL domain. The way this is done is likely simulator-specific and in the case of Questa is as simple, as instancing the SystemC component in a VHDL file. As long as the signals use supported types, they are translated automatically by the simulator. The blue border represents the SystemC domain where the model of the PS, described in Section 3.4, sits.

3.5.3 Modelling Peripherals

In the overview Figure 3.4 on page 31 a block which represents the peripherals is shown. These peripherals represent devices like a connected Analog to Digital Converter (ADC) or temperature sensor which are present as physical hardware. In total nine peripherals are connected via different interfaces: SPI, I^{2}C and a custom serial protocol from the CAU.

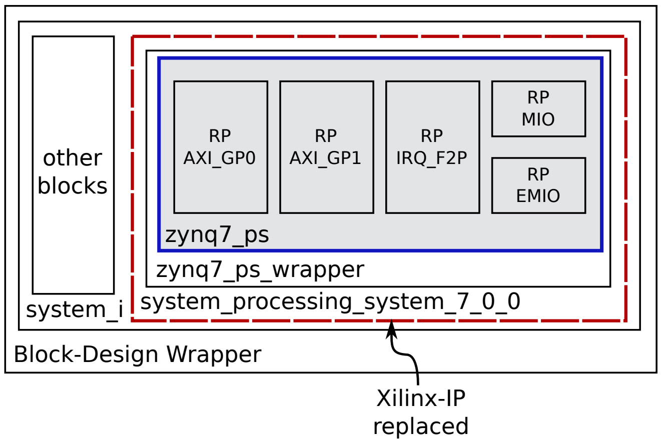

Figure 3.10: Replacing the PS wrapper with custom code. The red line represents the replaced file, the blue line represents the SystemC domain

Some devices are also purely analog like a voltage multiplexer or the IVC102 Switched Integrator [Cor96]. Standards which support modelling analog devices like VHDL-AMS exist but require further licenses and simulators. Therefore, the implementation was chosen to be written in standard VHDL.

Example Device: IVC102

The IVC102 is an integrator for low input currents which can be switched to hold its integration level and reset it. Its functional circuit can be seen in Figure 3.11 An ideal device transforms the input current according to following integral:

$$U_O = -\frac{1}{C_{INT}} \int_0^{t_{int}} I_{IN}(t) \cdot dt$$

The integral can be reset to zero via one control signal which closes switch S2 and t_{int} can be controlled via another signal controlling S1. C_{INT} can be set and connected externally and is in the pico- to nanofarad range.

To model this device in VHDL, this integral has been approximated by a sum.

$$U_O = -\frac{1}{C_{INT}} \sum_{i=0}^{i_{int}} I_{IN}(i) \cdot \Delta t$$

When S1 is open, no current can flow into the integrator. Therefore, the voltage stays constant when it is open. The input of this integrator is connected to the ionization chamber which also needs to be modelled.

It is assumed that during the period in which S1 is open, all generated charges within the ionization chamber are stored in its capacity, which is assumed to have no effect on

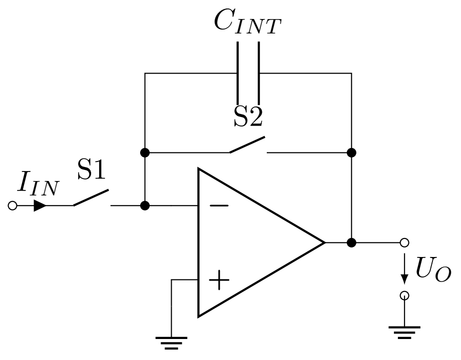

Figure 3.11: Functional circuit of IVC102

charge collection and generation. While this is not valid generally, the time period in which S1 is open is very small compared to the time it is closed, which means that the amount of charge accumulated in the capacity of the chamber is small. This means the electric field between the plates stays close to value set by the external voltage, which justifies this assumption.

The model can be implemented with a rather simple incremental process as seen in Algorithm 3.1. The output of this model can then be connected to another device like an ADC which also needs to be available as a VHDL model. This ADC then handles sampling and quantization of the voltage as well as communication via a given interface like SPI.

Algorithm 3.1: IVC102 increment calculation

```
if S2 closed then
    $outputVoltage = 0V$
    if S1 closed then
        $storedCharge = 0C$
    end
else
    if S1 closed then
        $outputVoltage = outputVoltage - \frac{generatedChargePerSecond \cdot \Delta t + storedCharge}{C_{INT}}$
        $storedCharge = 0C$
    else
        $storedCharge = storedCharge + generatedChargePerSecond \cdot \Delta t$
    end
end
```

Location of peripherals It was chosen to include the peripherals in the topmost layer of the simulation with descriptive names. This helps while debugging as well as writing the testbench because the level of abstraction on the signals can be one layer higher.

This means that a testbench designer does not need to care about individual interfaces, but can rely on e.g. setting a temperature variable or the input current.

Since these devices are only required for the simulation, the entire set of VHDL syntax - especially **`wait`** statements and data types like `real` can be used for developing those. Therefore, writing non-synthesizable models for simulation is simpler and development time faster compared to synthesizable models.

Outlook: Hardware in the Loop

The proposed way of modelling the peripherals in this subsection can also be extended to a Hardware in the Loop (HIL) approach which is fully contained within the SoC. To achieve this, all devices need to be written in a synthesizable way. They get connected to the DUT similarly as in co-simulation. This achieves that the entire Design under verification (DUV) as well as its peripherals are contained within the SoC without the need of extra hardware for verification.

Moreover, this method is interesting for verification of the CMPU as it runs in realtime. Also, because of the special design of the CMPU a speedup of roughly 10 times could be achieved.

An important issue is providing input data to the emulated devices and reading data from them. A various suite of AXI connections is available within the SoC which can be used to solve this issue using existing IP. Synchronization as well as visibility of internal signals are challenges when choosing this approach, but it is appealing because no external devices are needed for development of this method. Furthermore, the actual design and device is exercised during verification which avoids the issues that can arise from the simulation-to-reality gap.

A proof-of-concept was developed but to have a complete and reliable functional verification environment, much more development is needed. However, as seen in Table 3.3 there are several reasons why co-simulation is chosen as means of system-level verification. A big benefit is the flexibility resulting from the simulation compared to the HIL.

Method / Criterion	HW/SW Co-simulation	Hardware in the Loop
Visibilty of signals	Every possible signal within the FPGA as well as variables in the software are visible during simulation.	Special signals must be selected at synthesis (and/or) placement time. Only limited timespans can be captured because of limited memory on the device.
Speed	Considerably slower than real-time, depending on design.	Real-time by nature, depending on the design, faster than real time can be achieved by increasing clock speeds.
Control	Simulation is stoppable at any point in time, any signal can be used for synchronization.	Halting the system clock is harder, signals need to be routed to the outside or made accessible via available busses.
Debugging	Because of the reasons above and support by the simulator to halt execution at breakpoints, debugging is simple.	Harder because less control and visibility.
Hardware	General-purpose computer	SoC
Verification	Model	Real design (although placement within the FPGA may change)

Table 3.3: Comparison between HW/SW co-simulation and SoC contained HIL

CHAPTER 4

Testbench and Functional Verification

4.1 Testbench architecture using UVM

A testbench is the component which exercises the Design under verification (DUV). This means it generates stimuli, applies these stimuli and, if it is self-checking, checks whether the outputs behave correctly, i.e., they behave according to specification.

An ad-hoc approach of writing a testbench is suitable for small designs where the author of the test cases writes a few directed tests which are directly applied to the design. For bigger designs this quickly becomes unmanageable and a structured approach has been proven to be useful.

The object-oriented language SystemVerilog was created as a hardware and verification language which has higher-level concepts and data structures like associative arrays built in [ST12]. The Universal Verification Methodology (UVM) builds on top of SystemVerilog and has many concepts for creating modular and reusable testbenches and testbench components. A brief introduction to the main components of UVM is given and then the application of these components in simulation is shown.

4.1.1 A basic UVM Testbench

A UVM testbench usually consists of the test which instantiates all main components of the test environment and configures them. It creates one instance of the Design under verification (DUV) for simulation and connects it to a defined *interface* which can then be passed to other components in the environment. Once the testbench is compiled, different tests can be selected via command line options. A basic testbench using important UVM components can be seen in Figure 4.1 The testbench is made up

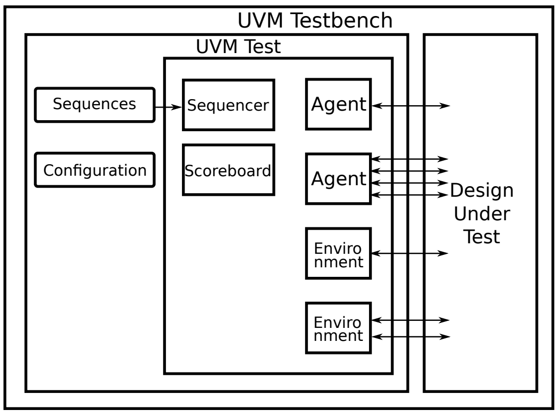

Figure 4.1: Exemplary UVM architecture [Acc15]

from different components which are hierarchically connected. Messages between all of those components are exchanged with TLM ports. This allows for easy addition and removal of components as they are usually only coupled via these messages.

UVM Environment This class is used to group components which are related to each other and needed for driving and comparing stimulus like Agents and Scoreboards.

UVM Scoreboard The Scoreboard is intended to receive transactions of both the input and the output of the DUV. From that, it can check whether it behaved correctly - it can also be used to measure the functional coverage.

UVM Agent An Agent is intended to group components which belong to one interface of the DUV. This usually includes a monitor, a driver and a sequencer.

UVM Driver Actually applying signals to the pins of the DUV is handled by the Driver component. It converts the transactions to pin-level changes and “drives” stimulus into the design.

UVM Monitor This component provides the counterpart of the Driver by observing the pins of the DUV and converting the encoded messages to transactions.

UVM Sequencer The sequencer controls the flow of transactions which are generated by a UVM Sequence to the driver.

UVM Sequence The sequence generates transactions which are sent to the rest of the testbench. Usually, these transactions are generated as constrained random stimuli. Sequences can be quite modular and even contain other sequences for generating higher level test cases.

More information about these classes can be read in the UVM User's Guide [Acc15] and documentation.

4.2 Testbench for the CROME Measuring and Processing Unit (CMPU)

The CMPU has various interfaces, but the most significant split can be made into the software interface and the hardware interface. They are somewhat different in behavior as can be seen in the comparison made in Table 4.1. A testbench for system-level verification of the CMPU must be able to handle both interfaces.

Interface / Criterion	Software	Hardware
Timing	Uncertain timing since processing packets is delayed depending on various factors	Hard timing requirements
Communication	Packets	Pin changes
Complexity	Variable packet width, complex parsing and packaging	Fixed length bitvectors and signals
Interface	TCP/IP socket, configuration files	Values of pins and vectors
Direct SystemVerilog support	No	Yes, built for this purpose

Table 4.1: Comparison between the software and hardware interface

Despite the differences, UVM still proves useful for connecting to both, hardware and software, within the testbench. As already mentioned, UVM uses transactions for communication between components. These transactions can either be mapped to packets for software as well as pin transactions for hardware.

An overview of the hierarchical structure as well as the interfacing between the testbench and the software can be seen in Figure 4.2. Each major interface has an agent which consists of a driver and (usually) two monitors.

Two monitors are used to observe the inputs to the peripheral, as well as the signal actually received by the calculation blocks. This is useful as the accuracy and correctness of the interfacing blocks can be checked with individual scoreboards (not shown in the graphic). For checking the calculations of the system, the exact values received by these blocks must be known. These are received by the output monitors of each interface. As

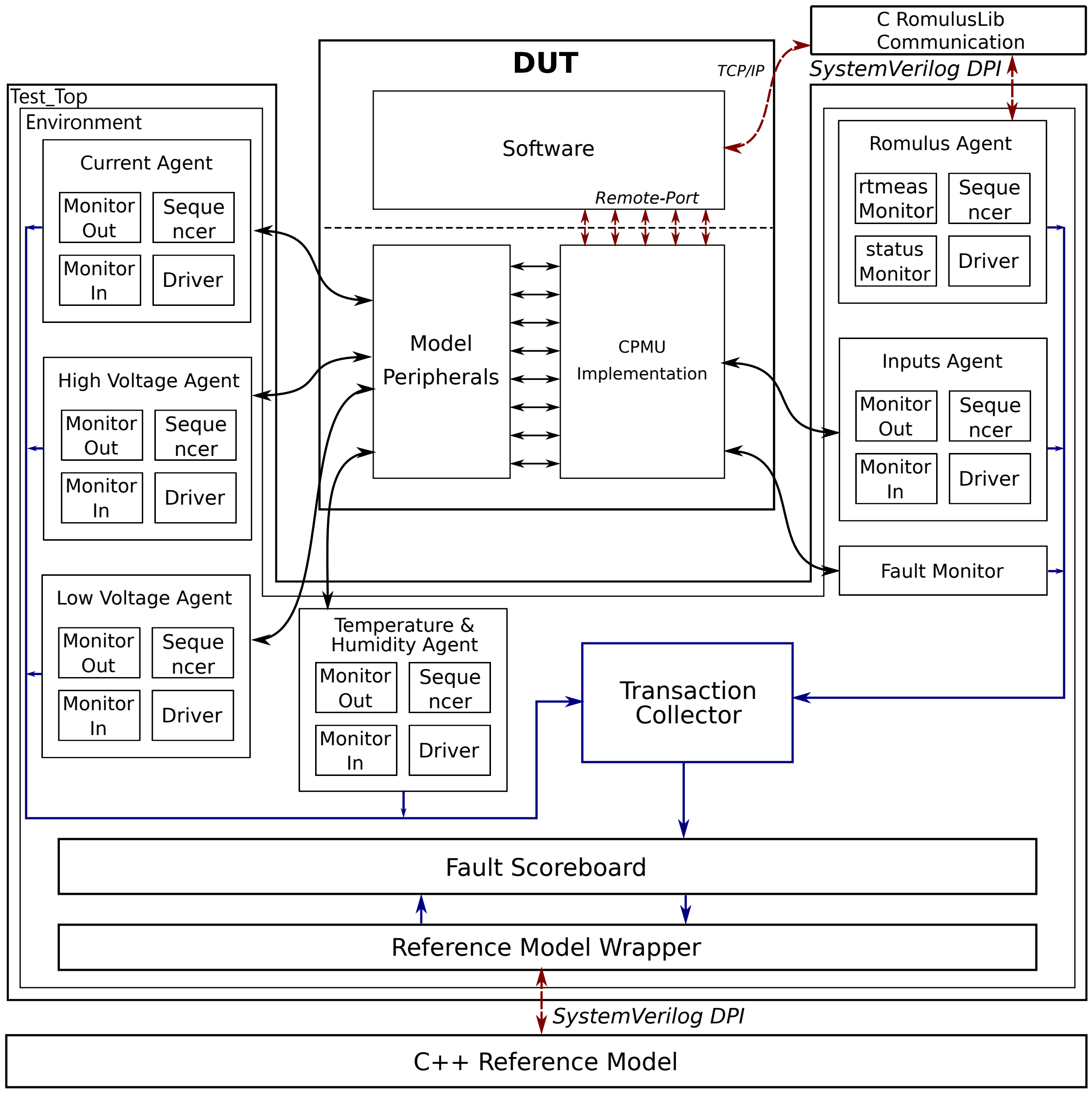

Figure 4.2: Overview of the Testbench for CMPU

long as both, the correctness of the in- to output and the systems outputs corresponding to the exact input values from the output monitors are checked, the entire system is verified.

To elaborate: an input block within the design usually handles a communication protocol like SPI and translates it to a bitvector and some control signals. This bitvector might sometimes not correspond to the exact input value as there might be uncertainty in the input rounding or interpretation of analog signals. To avoid the inaccuracy of the value actually used for calculation, the exact value output by this input block is used for verifying all calculation functionality. To make sure that the input block also conforms to the specification and stays within an error range, a separate scoreboard is used which can check this part individually.

An additional benefit of this two-monitor method is that in case of hardware changes, only the monitors need to be replaced while the rest of the testbench needs no modification.

Red arrows within Figure 4.2 indicate that communication crosses the boundaries of the hardware-simulation domain. The protocols/interfaces *remote-port*, *SystemVerilog Direct Programming Interface (DPI)* and *ROMULUSLib* based on *TCP/IP* are used for this purpose. The blue arrows indicate connections to the transaction collector which handles the synchronization of all transactions within the testbench. This is especially important for the software transactions, as these might come with some delay or interruptions. Therefore, special care is taken to synchronize all messages before passing them to the scoreboard.

Sending TCP/IP packages from the UVM testbench is achieved by developing a library in C, based on the existing ROMULUSLib, which provides high-level functions for sending and receiving those. These functions can be called from SystemVerilog code with the DPI.

SystemVerilog DPI The DPI is a simple and useful interface for calling C/C++ from SystemVerilog code and the other way around. To call a C function from SystemVerilog, it needs to be introduced via the `import` statement. Parameters can be passed as normal arguments and simple types like an integer are directly mapped to the corresponding datatype [ST12]. Arrays and structures can be passed with a little more effort as it is important where the memory is reserved to avoid memory corruptions. To enable the simulator to call the relevant functions, the C code must be compiled as a shared object and passed to the SystemVerilog simulator.

It is even possible to call simulation time-consuming tasks from a C function when they are declared to run within the task context.

4.2.1 Example test case for fault evaluation

In the scope of this thesis, emphasis was set on creating tests for the fault evaluation of the CMPU because it receives input from all interfaces, depends on parameters configured

```
export ''DPI-C'' task spend_simtime;
import ''DPI-C'' context task send_romulus_parameters(input
 ↪  parameterstruct_t array[], output int returnCode);
task spend_simtime(int timeInMs);
    #(timeInMs*1ms);
endtask
```

Listing 3: Imports and exports of the DPI

by the PS and is an important part of the systems self-checking functionality.

The smallest time between two evaluations of parameters of the CMPU is 100 ms. This is also the time which is used to clock the testbench with. All monitored transactions (input and output) are collected and synchronized by the transaction collector block seen in Figure 4.2. It packs all the incoming transactions into one big transaction which is then received by the scoreboard (and potential other components) for further processing.

Scoreboard

With the information received from the transaction collector, the scoreboard can determine functional coverage, and check the results with the connected reference model. The correctness (under the assumption that the reference model is correct) of the design can be checked by comparing all faults in the reference model and in the simulation.

Functional Coverage Another function that is implemented within the Scoreboard is the collection of functional coverage. SystemVerilog has constructs which help in defining this functional coverage as well as collecting it. A `covergroup` can contain `coverpoints`, options, formal arguments and triggers [ST12, p. 333]. A `coverpoint` in a `covergroup` may refer to a value and define different `bins` it might fall into. The value can be sampled at several points during simulation which is then evaluated at these points. If a value belonging to each `bin` was observed, the coverage for this `coverpoint` is 100%. SystemVerilog also supports more elaborate concepts such as transition coverage, conditional coverage and cross coverage [ST12]. With this wide range of options, different functional scenarios can be described.

Coverage of an alarm with hysteresis Alarms within the CMPU usually get triggered by a comparator with hysteresis. This means, that a value needs to pass some threshold to trigger the alarm. But the threshold which needs to be passed so the alarm ceases again, is lower compared to the threshold from before. Meaning that there exists an interval of values in which the alarm may be either on or off, depending on the history of the value. This helps when dealing with noisy signals because a signal which is near the threshold might cause rapid on and off toggling of the alarm which is not desired.

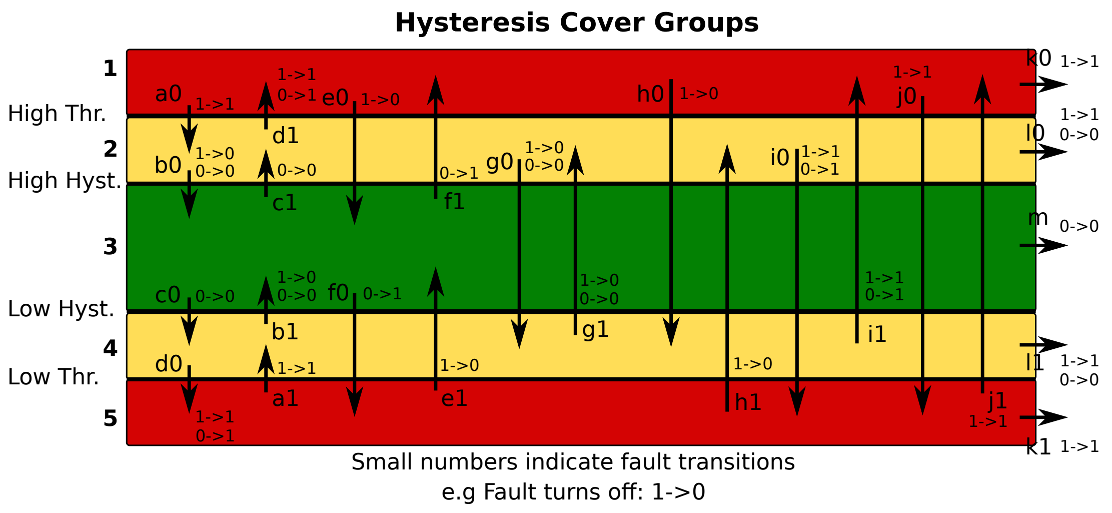

Figure 4.3: Graphical representation of hysteresis covergroups

```
covergroup cg_low_voltage_faults;
    range: coverpoint mapToGroup(lowThr,lowThrHyst,
                                 highThrHyst,highThr,value) {
        bins a0 = (1 => 2);
        ...
    }
    outp: coverpoint fault {
        bins o_o = (1=>1); //One to one
        ...
    }
    fault: cross range,outp{
        bins a0  = binsof(range.a0) && binsof(outp.o_o);
        ...
    }
}
```

Listing 4: SystemVerilog covergroup for hysteresis signal

To make sure randomly generated input covers every interesting transition, a set of coverpoints needs to be found. An illustration of the potential scenarios of the input value can be seen in Figure 4.3. The green area indicates the valid range, yellow can either be in the state where an alarm is active or not. In the red zone, an alarm must be active. All arrows represent transitions that the input signal may make, and the small numbers next to them represent the possible transitions of the alarms.

To capture these scenarios, transition coverage and cross coverage are used which can be seen in Listing 4. The function `mapToGroup` is used to map the input signals to

```
//Function declaration in C++ class
class refModel{
void initModel(uint64_t cycleNumberIn100ms);
}
```

```
//C Wrapper for DPI with pointer to object
extern "C"{
    void cInitModel(void* model,uint64_t cycleNumberIn100ms){
        ProgrammableLogic* pl = static_cast<ProgrammableLogic*>(model);
        pl->initModel(cycleNumberIn100ms);
    }
}
```

```
//Wrapper in SystemVerilog
import "DPI-C" function void cInitModel(chandle model, longint unsigned
↪  cycleNumberIn100ms);
class reference_model;
    chandle refModel;
    function void initModel(longint unsigned cycleNumberIn100ms);
        cInitModel(refModel,cycleNumberIn100ms);
    endfunction
endclass
```

Listing 5: Example of function wrappers for simple use in SystemVerilog

categories (1 to 5) which simplifies reuse of the covergroup for other signals. Labelled cross coverage bins are used to easily describe all interesting combinations of alarm and input value combinations.

4.2.2 Reference Model

A functional reference model for checking whether the DUT is behaving according to specification, is written in C++ and wrapped by C calls for the DPI. In SystemVerilog this process is reversed, meaning a reference model class, which holds a reference to the C++ object and translates the calls is available. An example of these wrappers for communication to a C++ object via DPI can be seen in Listing 5.

At the time of writing, the reference model only implements a subset of the functionality of the CMPU. The main control cycle which determines the evaluation of signals is implemented which receives all input parameters from the testbench. This will make including other blocks in the future as simple as instantiating them within the main `Model` class and linking their function calls. The outputs of the reference model are read by the scoreboard and compared to real model to check if it behaved correctly.

4.2.3 Interfacing the Software running within QEMU

As seen in Figure 4.2, a C program is called using the DPI. The C library "ROMULUSLib" was created by the CROME development team for general communication with the CMPU which is also used by this C Program. QEMU can be configured to forward the relevant ports from the guest to the host machine which can then be accessed from any program running on the host and behaves the same as a regular device (albeit slower). When sending messages to the software from the testbench, special care has to be taken since a deadlock situation may occur.

A simple C Program to set a value on the CMPU sends a request via TCP and waits for the answer. When the answer is received, the program returns the results to the caller. This is a problem when the function is called from a SystemVerilog context because simulation time does not advance while the C function is waiting for the answer. The reason for this is, that SystemVerilog waits for the function to terminate so that it can process the result. This waiting also halts the QEMU execution. Since QEMU is stopped the embedded application running within it never answers to the request issued by the C function and the entire system waits infinitely.

This can be solved by putting the DPI call into the task context as shown in Listing 3 on Page 52 and calling a time-consuming task from the C program while waiting on the result. The time is then passed in RTL simulation and therefore also in QEMU.

4.2.4 Test case stimulus and results

Once all blocks are connected, the system is booted and the embedded application is started, all monitors need to be synchronized. Then the system is configured via the Romulus interface. After configuration is done, the actual test starts which generates random voltages values, evenly distributed between the five presented value categories. The scoreboard measures coverage and correctness of hundreds of stimuli which are applied to the system. The result of this specific test case for the low-voltage fault generation is 100 % which gives good confidence in this part of the design.

CHAPTER 5

Results and Outlook

5.1 Conclusion and Results

A method to verify an SoC using testing and simulation was developed within this thesis. The emulation of the entire chip Zynq-7000 is possible and was created using state-of-the-art tools and methods. Following the methodology introduced by "Automated verification of a System-on-Chip for radiation protection fulfilling Safety Integrity Level 2" [Cee19], a UVM testbench which makes use of constrained random stimulus was built to exercise the design. A reference model was developed and the low-voltage fault generation of the system was successfully checked with a functional coverage of 100%.

Further goals set in Chapter 1 of this thesis are:

1. Simulation runs on a single, general purpose machine.
2. Simulator can be used for debugging and development of new features.
3. The model of the simulation must be accurate enough for verification.
4. Simulation should be fast enough to boot an embedded Linux within a few minutes.
5. Automated tests must be executable with the simulator.

The first three items are fulfilled by design of the hardware/software co-simulation. Regarding point number two, CMPU developers successfully used the simulation for development of new functionalities and debugging. Every signal within the RTL design is visible to the developer with the integrated waveform viewer of the hardware simulator. Software can be monitored and debugged like on a physical device. Even kernel-debugging is possible because of QEMU which supports a GNU Debugger (GDB) connection for this purpose.

The accuracy of the model is limited by the synchronization quantum which determines the time between two synchronization points. The quantum was empirically set to be $50\,\mu s$ which is quite small compared to the $100\,ms$ processing window in which the hardware performs evaluation and the software receives results. This results in a reasonable performance as well as accuracy. The operating system does not have a real-time kernel, this means that the uncertainty of execution time, introduced by the scheduler is overshadowing the effect of the synchronization quantum.

The fourth goal of a reasonable boot time is also fulfilled as shown by tests on two development machines. On a machine with an *Intel i7 3770 @* $3.6\,GHz$ and *8 GB DDR3 RAM* the entire Linux OS boots on average in one and a half minutes. Simulation speed @$2MHz$ clock speed of the hardware is roughly 50 times slower than real-time. This fulfills the goal for the usability for developers as well as verification. This speed allows running various scenarios with durations of a few minutes within a few hours. Different scenarios can also be tested at the same time and checked by the reference model as well, minimizing simulation time and increasing the potential for errors by colliding transactions.

An architecture for a modular, automated testbench which can observe and drive various signals of the device was presented which fulfills the fifth design goal. The presented testbench observes and drives a set of in- and outputs which are common to Internet of Things (IoT) devices, meaning TCP/IP configuration and reporting and low-level sensor communication. This means that the presented testbench architecture is applicable for other IoT devices and can therefore be used for a wider class of devices. Furthermore, the testbench is built using industry-standard UVM and is therefore easily usable by other verification engineers who are acquainted with this methodology.

5.2 Outlook

The simulation and its tools have been developed and optimized for the CROME project. An implementation which is easily adjustable to other designs would be optimal as within CERN, all-programmable SoCs, more specifically Zynq-7000 and Zynq UltraScale+ are used for a non-neglible amount of designs.

More scenarios for the testbench to further the verification process as well as additional development of the reference model are needed. Furthermore, regression tests and integration with a Continuous Integration/Continuous Delivery (CI/CD) workflow is interesting for the ongoing development as well.

Since lots of development for open-source tools is currently happening, an entirely open-source based simulation would be interesting. If Verilator supported VHDL, where efforts are already made, a combination of it and SystemC-UVM [Acc22] or cocotb [coc22], a python based testing framework, could result in a fully open simulation.

List of Figures

Acronyms

ACP Accelerator Coherency Port. 28
ADC Analog to Digital Converter. 30, 31, 42, 44
API Application Programming Interface. 36
ASIC application-specific integrated circuit. 5, 6, 10, 11
AXI Advanced eXtensible Interface. 27–29, 37, 42, 45

BRAM Block RAM. 29

CAU CROME Alarm Unit. 26, 27, 42
CERN European Organization for Nuclear Research. ix, xi, 1, 25, 29, 58
CI/CD Continuous Integration/Continuous Delivery. 58
CJB CROME Junction Box. 26
CMPU CROME Measuring and Processing Unit. xiii, 25–30, 41, 42, 45, 49, 51–55, 57, 59
CPLD Complex Programmable Logic Device. 5
CPU Central Processing Unit. 2, 22, 23, 32, 34
CROME CERN Radiation Monitoring Electronics. ix, xi, 1, 2, 26, 55, 58
CUPS CROME Uninterruptible Power Supply. 26, 27

DES Discrete event simulation. 17, 18
DMA Direct Memory Access. 32
DPI Direct Programming Interface. 51, 52, 54, 55
DSP Digital Signal Processing. 7, 8
DUT Device under test. 10–12, 41, 45, 54
DUV Design under verification. 45, 47, 48

ECC Error-Correcting Code. 29
EMIO Extended Multiplexed I/Os. 28, 33, 41

FPGA Field Programmable Gate Arrays. 2, 5–9, 11, 18, 22, 26, 27, 46
FSM Finite state machine. 14

GDB GNU Debugger. 57
GPIO General Purpose I/O. 31–36, 38, 39, 41
GUI Graphical User Interface. 41

HDL Hardware Description Language. 39, 42
HIL Hardware in the Loop. 45, 46
HLS High level synthesis. 15
HW Hardware. 2, 3, 15, 16, 30

IoT Internet of Things. ix, xi, 58
IP Intellectual Property. 26, 29, 42, 45
I^{2}C Inter-Integrated Circuit. 7, 30, 42

LHC Large Hadron Collider. 1
LP Logical Process. 20, 21
LTL Linear Temporal Logic. 15
LUT Lookup table. 6, 8

MC/DC Modified Condition/Decision Coverage. 13
MIO Multiplexed I/O. 33, 39
MMU Memory Management Unit. 33

OS Operating System. 23, 28, 30, 32, 33, 37

PCB Printed Circuit Board. 34
PCI Peripheral Component Interconnect. 34
PL Programmable Logic. 26–31, 38, 42
PLD Programmable Logic Device. 5
PP Physical Process. 20, 21
PS Processing System. 27–30, 32, 38, 42, 43, 52, 59
PSL Property Specification Language. 15, 16

QOM QEMU Object Model. 36, 38

RAM Random Access Memory. 7
RP Radiation Protection. 1
RTL Register-Transfer-Level. 15, 17, 18, 30, 31, 38, 39, 55

SDK Software Development Kit. 28
SEU Single Event Upset. 26, 29
SIL 2 Safety Integrity Level 2. 2
SoC System-on-Chip. ix, xi, 2, 3, 5, 8, 15, 16, 19, 26–28, 31, 32, 34, 38–40, 45, 46, 57–59
SPI Serial Peripheral Interface. 30, 32, 42, 44, 51
SRAM Static RAM. 6
SVA SystemVerilog Assertions. 15, 16
SW Software. 2, 3, 15, 16, 30

TCG Tiny Code Generator. 20
TCP Transmission Control Protocol. ix, xi, 26, 29, 38, 51, 55, 58
TLM Transaction Level Modeling. 31, 32, 38, 39, 48
TMR Triple Modular Redundancy. 26

UIO Userspace I/O. 29
UML Unified Modelling Language. 16
USB Universal Serial Bus. 34
UVM Universal Verification Methodology. ix, xi, 3, 47–49, 51, 57, 58

VHDL Very High Speed Integrated Circuit Hardware Description Language. 5, 18, 29, 30, 38, 39, 41–45, 58
VIP Verification IP. 42
VLSI Very Large Scale Integration. 17

Bibliography

[Acc15] Accellera. *Universal Verification Methodology (UVM) 1.2 User's Guide.* 2015. URL: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf (visited on 01/24/2022).

[Acc22] Accellera. *SystemC Verification.* 2022. URL: https://www.accellera.org/activities/working-groups/systemc-verification (visited on 01/27/2022).

[ARM13] ARM. *CoreSight Technical Introduction.* Aug. 2013. URL: https://developer.arm.com/documentation/epm039795/latest.

[Bel05] Fabrice Bellard. „QEMU, a Fast and Portable Dynamic Translator“. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference.* ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41. DOI: 10.5555/1247360.1247401.

[Ben08] Jeremy Bennett. *Building a Loosely Timed SoC Model with OSCI TLM 2.0 A Case Study Using an Open Source ISS and Linux 2.6 Kernel.* 2008. URL: https://www.embecosm.com/appnotes/ean1/ean1-tlm2-or1ksim-2.0.pdf (visited on 02/01/2022).

[BGB02] M.G. Bartley, D. Galpin, and T. Blackmore. „A Comparison of Three Verification Techniques: Directed Testing, Pseudo-Random Testing and Property Checking“. In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324).* Proceedings of 39th Design Automation Conference. New Orleans, LA, USA: IEEE, 2002, pp. 819–823. ISBN: 978-1-58113-461-2. DOI: 10.1109/DAC.2002.1012735.

[Bon14] Paolo Bonzini. „QOM Exegesis and Apocalypse“. 2014. URL: https://www.linux-kvm.org/images/9/90/Kvmforum14-qom.pdf (visited on 01/19/2022).

[Bou+17] Hamza Boukabache et al. „Towards a Novel Modular Architecture for CERN Radiation Monitoring“. In: *Radiation Protection Dosimetry* 173.1-3 (Apr. 1, 2017), pp. 240–244. ISSN: 0144-8420, 1742-3406. DOI: 10.1093/rpd/ncw308.

[BR70] J. N. Buxton and B Randell. *SOFTWARE ENGINEERING TECHNIQUES*. Brussles: NATO SCIENCE COMMITTEE, Apr. 1970. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[Bro12] General Dynamics Broadband. *General Dynamics Broadband Brings Virtualization to Xilinx Zynq-7000 All Programmable SoC*. Oct. 31, 2012. URL: https://www.aerocontact.com//en/aerospace-aviation-news/35811-general-dynamics-broadband-brings-virtualization-to-xilinx-zynq-7000-all-programmable-soc (visited on 01/17/2022).

[Bry77] Randall Everitt Bryant. *Simulation of Packet Communication Architecture Computer Systems*. Cambridge: Massachusetts Institute of Technology, Nov. 1977. URL: https://www.cs.cmu.edu/~bryant/pubdir/MIT-LCS-TR-188.pdf.

[Cee19] Katharina Ceesay-Seitz. „Automated Verification of a System-on-Chip for Radiation Protection Fulfilling Safety Integrity Level 2“. Vienna, Tech. U., Apr. 18, 2019. URL: https://cds.cern.ch/record/2672187 (visited on 10/18/2021).

[CM79] K.M. Chandy and J. Misra. „Distributed Simulation: A Case Study in Design and Verification of Distributed Programs“. In: *IEEE Transactions on Software Engineering* SE-5.5 (Sept. 1979), pp. 440–452. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.230182.

[coc22] cocotb. *Cocotb, a Coroutine Based Cosimulation Library for Writing VHDL and Verilog Testbenches in Python*. cocotb, Jan. 25, 2022. URL: https://github.com/cocotb/cocotb (visited on 01/27/2022).

[Com10] International Electrotechnical Commission. *IEC 61508-1 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. IEC, Apr. 2010, p. 132.

[Cor96] Burr-Brown Corporation. *Precision Switched Integrator Transimpedance Amplifier*. 1996. URL: https://www.ti.com/lit/ds/symlink/ivc102.pdf?ts=1642984406562 (visited on 01/24/2022).

[Cro+14] Louise H. Crockett et al. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. 1. ed. Glasgow: Strathclyde Academic Media, 2014. 460 pp. ISBN: 978-0-9929787-0-9.

[dev20] devicetree.org. *Devicetree Specification 0.3*. Feb. 2020. URL: https://github.com/devicetree-org/devicetree-specification/releases (visited on 01/18/2022).

[Doc21] QEMU Documentation. *The QEMU Object Model (QOM)*. 2021. URL: https://qemu.readthedocs.io/en/latest/devel/qom.html (visited on 01/19/2022).

[doo15] Xylon d.o.o. *logiADAK Zynq-7000 All Programmable SoC Automotive Driver Assistance Kit.* Feb. 2015. URL: http://itersnews.com/wp-content/uploads/experts/2015/03/95928logiADAK_hds.pdf (visited on 01/17/2022).

[Fär13] Andreas Färber. „Modern QEMU Devices“. 2013. URL: https://www.linux-kvm.org/images/0/0b/Kvm-forum-2013-Modern-QEMU-devices.pdf (visited on 01/19/2022).

[GD05] D. Große and R. Drechsler. „CheckSyC: An Efficient Property Checker for RTL SystemC Designs“. In: *2005 IEEE International Symposium on Circuits and Systems.* 2005 IEEE International Symposium on Circuits and Systems. Kobe, Japan: IEEE, 2005, pp. 4167–4170. ISBN: 978-0-7803-8834-5. DOI: 10.1109/ISCAS.2005.1465549.

[Ger17] Nicola Joel Gerber. „Contributions to the SIL 2 Radiation Monitoring System CROME (CERN RadiatiOn Monitoring Electronics) - CERN Document Server“. EPFL, Mar. 2017. URL: https://cds.cern.ch/record/2699738?ln=de (visited on 12/15/2021).

[Gin17] Tristan Gingold. *GHDL Main/Home Page.* 2017. URL: http://ghdl.free.fr/ (visited on 01/21/2022).

[Hab16] Eduardo Habkost. *An Incomplete List of QEMU APIs.* Nov. 29, 2016. URL: https://habkost.net/posts/2016/11/incomplete-list-of-qemu-apis.html (visited on 01/19/2022).

[Hur18] Saskia Kristina Hurst. „Reliability Analysis of the CERN Radiation Monitoring Electronic System CROME“. In: CERN-THESIS-2016-363 (Feb. 6, 2018). URL: https://cds.cern.ch/record/2303168 (visited on 01/11/2022).

[KHZ16] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for Software Programmers.* Cham: Springer International Publishing, 2016. ISBN: 978-3-319-26406-6. DOI: 10.1007/978-3-319-26408-0.

[Laf21] Clyde Laforge. „Datapath in CROME“. In: CERN Internal, 2021.

[Li+10] Juncao Li et al. „An Automata-Theoretic Approach to Hardware/Software Co-verification“. In: *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering.* FASE'10 (2010), pp. 248–262. DOI: 10.1007/978-3-642-12029-9_18.

[Mar08] Peter Marwedel. „Discrete Event Modeling: VHDL“ (Technische Universität Dortmund). 2008. URL: https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/courses/ss10/ies/downloads/es-marw-2.7-discrete-event.pdf.

[Mat01] Martin Matschnig. „Parallele VHDL Simulation Mit Einem Standard Hardwaresimulator“. Wien: TU Wien, Mar. 2001. URL: https://publik.tuwien.ac.at/files/pub-et_4998.pdf (visited on 01/31/2022).

[MC12] Antonio Mondragón-Torres and Jeanne Christman. „Hard Core vs. Soft Core: A Debate“. In: 2012 ASEE Annual Conference & Exposition. San Antonio, Texas, June 10, 2012. DOI: 10.18260/1-2--21446.

[Mic] Microsemi. *SmartFusion2 SoC*. SmartFusion2 SoC. URL: https://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2 (visited on 01/17/2022).

[Mic17] Microsemi. *PolarFire™ Non-Volatile FPGA Family Delivers*. Oct. 2017. URL: https://www.microsemi.com/document-portal/doc_view/1243174-polarfire-fpga-white-paper (visited on 03/28/2022).

[Mis+17] Prabhat Mishra et al. „Post-Silicon Validation in the SoC Era: A Tutorial Introduction“. In: *IEEE Design & Test* 34.3 (June 2017), pp. 68–92. ISSN: 2168-2356, 2168-2364. DOI: 10.1109/MDAT.2017.2691348.

[Muk+17] Rajdeep Mukherjee et al. „Formal Techniques for Effective Co-verification of Hardware/Software Co-designs“. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC ’17: The 54th Annual Design Automation Conference 2017. Austin TX USA: ACM, June 18, 2017. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062253.

[Nai15] Nelson Naia. „Real-Time Linux and Hardware Accelerated Systems on QEMU“. University of Minoh, Oct. 2015. 244 pp. URL: http://repositorium.sdum.uminho.pt/bitstream/1822/51319/1/Nelson%20Pinheiro%20Duarte%20Naia.pdf (visited on 01/31/2022).

[PS19] B Aa Petersen and C Schwick. „EXPERIMENT REQUESTS AND CONSTRAINTS FOR RUN 3“. In: *9th LHC Operations Evian Workshop* (2019), pp. 267–271. URL: https://cds.cern.ch/record/2750301 (visited on 01/31/2022).

[Rav11] Renjith P Ravindran. *Qemu Detailed Study*. Apr. 2011. URL: https://lists.gnu.org/archive/html/qemu-devel/2011-04/pdfhC5rVdz7U8.pdf (visited on 01/18/2022).

[RF16] Bajaj Ronak and Suhaib A. Fahmy. „Mapping for Maximum Performance on FPGA DSP Blocks“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.4 (Apr. 2016), pp. 573–585. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2015.2474363.

[Rod+12] Fernando Rodríguez-Haro et al. „A Summary of Virtualization Techniques“. In: *Procedia Technology*. The 2012 Iberoamerican Conference on Electronics Engineering and Computer Science 3 (Jan. 1, 2012), pp. 267–272. ISSN: 2212-0173. DOI: 10.1016/j.protcy.2012.03.029.

[SMG15] Jason Sprott, Paul Marriott, and Matt Graham. „Navigating The Functional Coverage Black Hole: Be More Effective At Functional Coverage Modeling“. In: DVCon. 2015, p. 12. URL: https://www.verilab.com/files/dvcon_functional_coverage_black_hole_final.pdf (visited on 01/31/2022).

[Sof20] Siemens Digital Industries Software. „2020 Wilson Research Group Functional Verification Study“. In: (2020), p. 13. URL: https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study-ic-asic-fucntional-verification-trend-report (visited on 01/17/2022).

[SOS08] Ahmed Karim Ben Salem, Slim Ben Othman, and Slim Ben Saoud. „Hard and Soft-Core Implementation of Embedded Control Application Using RTOS“. In: *2008 IEEE International Symposium on Industrial Electronics.* 2008 IEEE International Symposium on Industrial Electronics. June 2008, pp. 1896–1901. DOI: 10.1109/ISIE.2008.4677261.

[ST12] Chris Spear and Greg Tumbush. *SystemVerilog for Verification A Guide to Leaning the Testbench Language Features.* Third Edition. Springer New York, 2012. ISBN: 978-1-4614-0714-0.

[Tar21] Vaibbhav Taraate. *ASIC Design and Synthesis: RTL Design Using Verilog.* Singapore: Springer Singapore, 2021. ISBN: 978-981-334-641-3 978-981-334-642-0. DOI: 10.1007/978-981-33-4642-0.

[TK01] S. Tasiran and K. Keutzer. „Coverage Metrics for Functional Validation of Hardware Designs“. In: *IEEE Design & Test of Computers* 18.4 (July 2001), pp. 36–45. ISSN: 07407475. DOI: 10.1109/54.936247.

[Ver22] Veripool.org. *Verilator.* 2022. URL: https://www.veripool.org/verilator/ (visited on 01/21/2022).

[VH99] Voon Yee Vee and Wen Jing Hsu. *Parallel Discrete Event Simulation: A Survey.* 1999. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.7706&rep=rep1&type=pdf (visited on 01/31/2022).

[Wer20] Rick Wertenbroek. *Zynq-7000 HW-SW Co-Simulation QEMU-QuestaSim – REDS Blog.* 2020. URL: https://blog.reds.ch/?p=1180 (visited on 01/28/2022).

[Xila] Xilinx. *QEMU-devicetrees.* URL: https://github.com/Xilinx/qemu-devicetrees (visited on 01/14/2022).

[Xilb] Xilinx. *Xilinx's Fork of Quick EMUlator (QEMU) with Improved Support and Modelling for the Xilinx Platforms.* URL: https://github.com/Xilinx/qemu (visited on 01/14/2022).

[Xil17] Xilinx. *Zynq-7000 All Programmable SoC Verification IP v1.0.* Apr. 28, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/processing_system7_vip/v1_0/ds940-zynq-vip.pdf (visited on 01/21/2022).

[Xil19] Xilinx. *7 Series FPGAs Memory Resources User Guide.* 2019. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[Xil20a] Xilinx. *PetaLinux Tools Documentation: Reference Guide*. 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1144-petalinux-tools-reference-guide.pdf (visited on 01/13/2022).

[Xil20b] Xilinx. *Xilinx Quick Emulator User Guide*. Nov. 2020. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/821395464/QEMU+User+Documentation (visited on 01/17/2022).

[Xil21] Xilinx. *Zynq-7000 SoC Technical Reference Manual*. 2021. URL: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (visited on 01/13/2022).

[Xil22] Xilinx. *LibSystemCTLM-SoC*. Jan. 12, 2022. URL: https://github.com/Xilinx/libsystemctlm-soc (visited on 01/14/2022).

[Zab12] Wojciech M. Zabołotny. „Development of Embedded PC and FPGA Based Systems with Virtual Hardware“. In: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012. Ed. by Ryszard S. Romaniuk. Wilga, Poland, Oct. 15, 2012, 84540S. DOI: 10.1117/12.981877.