Universität Bremen

Better early than never: Formal and Practical Techniques for the Complex System Design Process using Virtual Prototypes

Dissertation
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften
– Dr. Ing. –

von

Pascal Pieper

– Digitalversion –

Vorgelegt im Fachbereich 3
(Mathematik und Informatik)
der Universität Bremen
im April 2023

– This page is intentionally left blank –

Acknowledgements

I would like to thank my supervisor *Prof. Dr. Rolf Drechsler* for creating a great atmosphere in which my scientific endeavours could be supported with an invigorating freedom and without whom this dissertation would not have been possible.

I also would like to thank all my current and former colleagues at AGRA/DFKI; in particular *Dr. Vladimir Herdt* for guiding me into self-sufficiency; as well as *Christina Plump*, *Niklas Bruns*, *Sören Tempel*, *Sallar Ahmadi-Pour*, and *Dr. Fritjof Bornebusch* (in no particular order) for inspiring, scientific, and not-so-scientific talks.

Furthermore, I thank *Prof. Dr. Daniel Große* for inviting me into the AGRA, and *Dr. Ralf Wimmer* for supporting me with our publication and their tool NlView™.

Lastly, an honorable mention goes to the *WMF S2000s* coffee machine in the *Teeküche* that makes neither great nor terrible coffee if mixed with enough milk.

Pascal Pieper

Bremen, April 2023

– This page is intentionally left blank –

Abstract

Modern System-on-Chip (SoC) designs are produced in increasingly faster project cycle times, while their complexity rises together with the need of a continuously decreasing cost. To cope with this high demand and pressure on a manufacturer's ability to maintain a reliable and secure end-product, Virtual Prototype (VP) based design processes are gaining popularity in the industry. A VP creates the possibility to design, evaluate, and verify an executable prototype of the system in an early design stage by modeling the future hardware on a behavioral or structural level. In contrast to more traditional design flows like *hardware-then-software*, this enables both the iterative design evaluation and a parallel development of the (actual) hardware *and* software early in the product conception phase. Additionally, after development of the lower level hardware stages (e. g. on register transfer level, gate level, or physical hardware), VPs can be used as golden reference models with test and verification methods for comparison between the system level behavior and the actual hardware. For this to work, however, the VP and its components need to be verified in the first place.

In this thesis, several techniques are proposed to improve and strengthen the VP-based design process for embedded systems, covering modeling and verification of System-on-Chip scaled hardware models, as well as novel debugging, analysis, and educational tools. The main goal of this thesis is to both improve existing processes and create novel approaches for the early design of complex embedded systems on the architectural and behavioral level.

"To err is human, but to really foul things up you need a computer."
– *Paul R. Ehrlich*

Contents

Chapter 1
Introduction

In computer science, the steadily increasing gap between the technological possibilities and the ability to design such systems is referred to as the design gap [1–3]. With the ever-growing number of transistors in Integrated Circuits (ICs), this trend is expected to increase even further as predicted by "Moore's Law". While this growing trend gave way to an unprecedented life-style with millions of devices working in the background, ranging from simple Internet of Things (IoT) devices, through System-on-Chips (SoCs) in smartphones, Electronical Control Units (ECUs) in cars or airplanes, or up to high-end processors in server farms; managing this complexity has become the main factor in advancing the state-of-the-art or even in meeting tight *time-to-market* and cost requirements.

Most of these SoCs are made up of individual units that would make up a whole desktop computer of the *90s*: A Central Processing Unit (CPU) (or multiple heterogeneous CPUs) with caches and different memories, a Graphics Processing Unit (GPU) for driving 3D graphics, cryptography-, vector-, neural net-, and many more accelerators, wireless functionality, analog domains, and so on.

All of these units are condensed in a package that fits on a thumb with incredible speed and power efficiency, and with features that a human eye can no longer see because normal light can not resolve these; no single individual can manufacture (or even completely understand!) ICs alone without the help of Electronic Design Automation (EDA) tools. Moreover, the actual behavior of these systems also depends on many abstraction layers of software (SW) stacks from drivers to high-level user applications.

One of the many tools at the system designer's disposal is *virtual prototyping* [4, 5], which nowadays is widely used in the industry. A Virtual Prototype (VP) is an executable SW model of some physical hardware (HW) and opens up the opportunity to explore the implications of design choices (called *design space exploration*) as early as possible [6], without the slow and expensive need to implement such HW in full detail first. Furthermore, in case of the model being a processor,

it allows SW to be developed and run on the *virtual* HW early on. Nowadays, for the development of digital systems, SystemC emerged as the go-to option for rapid, HW-centric prototyping due to its flexibility in level of abstraction against simulation speed [7, 8] (see also Section 2.2). As SystemC is a modeling language on top of C++, it allows SW developers to interact, inspect and even create the HW models they use, with full control over the (virtual) HW using SW debugging and analysis tools, which is still a promising field of research.

VPs allow for an increase in development speed, but the combination of SW and HW in embedded systems requires not only a verified *functionality*, but also an increasing level of *security* as the ubiquitousness and mass production by increasingly smaller design teams lift every small error to a huge attack vector and thus impose a multitude of risks.

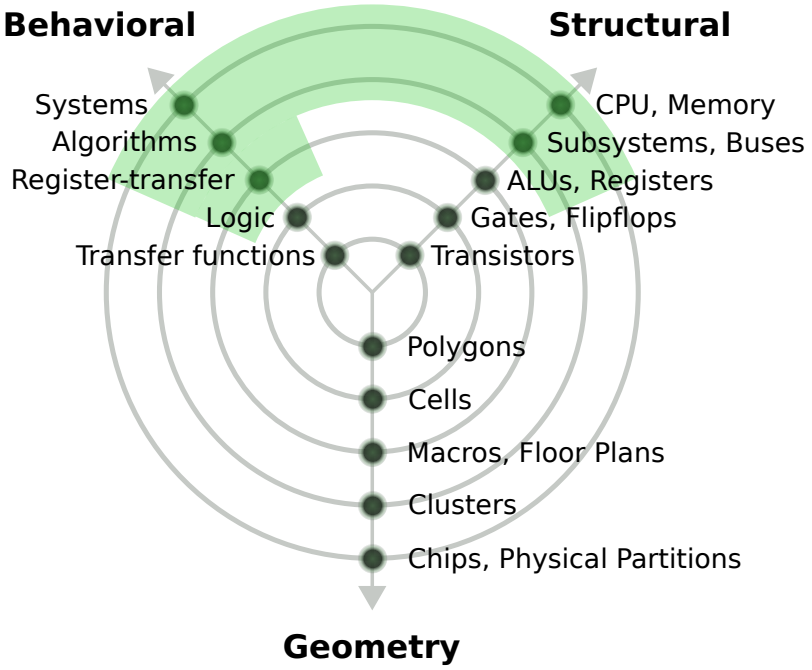


Figure 1.1: Abstraction levels addressed in this dissertation highlighted in green. Gajski-Kuhn Y-Model, redrawn, from [9].

HW is usually modeled in different abstraction levels of the three main factors behavior, structure, and geometry (see Figure 1.1). The behavioral system level is where functions of unit groups behave as if they were already complete, without regard on *how* the function is actually implemented or how the individual units communicate. With the focus on VPs, which can be used early in the behavioral system level abstraction, the abstractions usually range from Transaction Level Modeling (TLM), where communication between units "just happens" (see Chapter 2, Figure 2.2), to the Register Transfer Layer (RTL), where even the task of

passing information between units needs to be synchronized and orchestrated. These abstractions enable the simultaneous development of SW drivers while the actual HW is not even finished yet; benchmarking attributes like run-time, power consumption or memory resources on varying levels of accuracy with the trade-off on simulation speed *continuously* along the growing detail of the model during development.

Without such knowledge, a design team has to estimate these trade-offs by either prior knowledge from other projects or by actually spinning up the whole chip fabrication process. This can only be done at great cost both in terms of money and time, however. While this situation somewhat relaxed by the use of Field-Programmable Gate Arrays (FPGAs), especially in the application specific *embedded systems* area, there is still a considerable workload with time, cost, flexibility, and power demands fixed that needs to be addressed in the design phase. This system design is *complex* on multiple different abstraction layers in the actual word sense of being hard to control and predict; in contrast to being *complicated*, which still may not be easy but ultimately knowable. As an example; one of the common design questions that need to be decided as early as possible in complex system design can be whether a certain functionality should be designed in HW or SW. This is called *Hardware/Software Partitioning* [10] and carries a number of known or unknown trade-offs, as there is not *the one* SW or HW layer (cf. Figures 1.2 and 2.1). Mainly flexibility, re-usability of designs, and a fast idea-to-MVP (Minimum Viable Product) time are additionally shared, secondary goals to being able to design the system in the first place.

Figure 1.2 shows the flexibility of change against the expected execution speed of a certain functionality as one of the considerations to make. The flexibility to change a certain aspect of a designed system is important and heavily dependent on the use-case. This is also related to the level of detail needed in the underlying simulation for a functionality to be properly tested and verified (cf. Figure 1.1). It starts with easy-to-change user scripts that may control high-level behavior. Scripting may be used, e. g., in smart home assistant automation with an interpreted language such as Python or Lua, where the execution speed is comparatively slow, but the code may be changed even in running systems. Scripts can also be used and tested on simple mock-up systems running on a desktop computer. Broadly speaking, this level applies mainly to tasks orchestrating underlying systems, modeling the overall behavior, and modifying high level process parameters. Less flexible, and usually implemented in faster high-level languages, are the Operating System (OS), system-specific tasks, and drivers for the underlying HW. This level also applies to calculation-intensive tasks and libraries that provide a set of primary functions. Functions like OSes and low-level drivers already need more sophisticated simulation systems such as virtual machines or even system-level

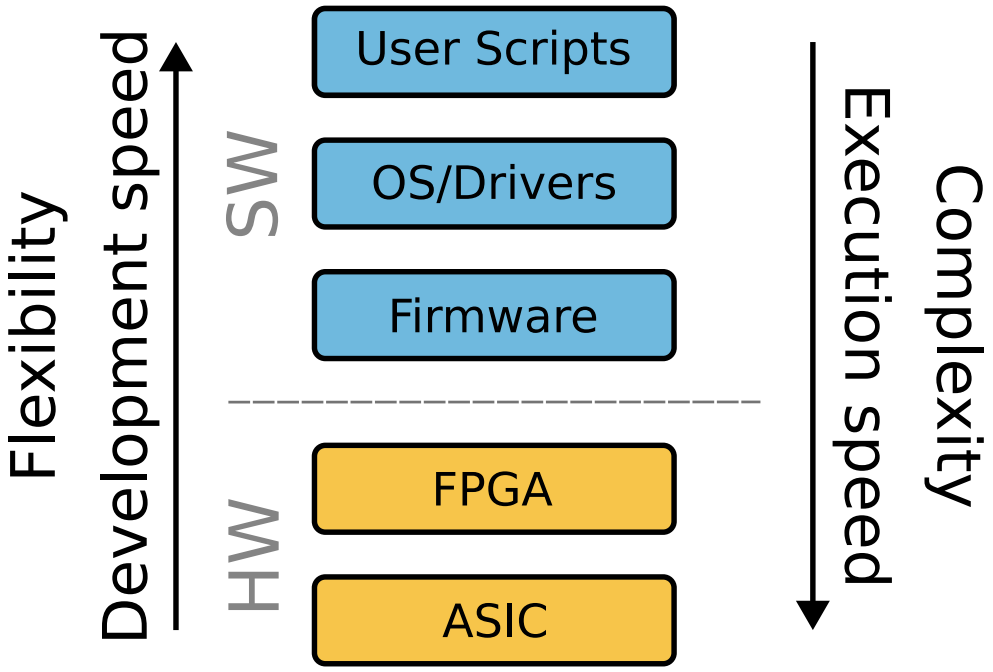


Figure 1.2: Possible SW/HW layers where a given functionality can be implemented, with the trade-off between flexibility and execution speed.

VPs, as they can be just enough detailed to run as fast as possible. The lowest SW level is represented by the machine-native firmware (sometimes called *bare-metal*), which ranges from early system-setup code to instruction sequences / microcode images for controllable on-chip devices. Here, the usable resources are limited, as the firmware usually contains early start-up code where devices such as Dynamic Random Access Memory (DRAM) can not yet be used, as they are being initialized only during this run-time. The firmware is tightly coupled to the underlying HW, which needs to be simulated in more detail than just the common virtualization techniques for test and verification, with more focus on the interaction with the SoC's on- and off-chip peripheral devices. Here, system- or algorithm-level VPs are well-suited for an accurate test and verification simulation (which is the focus of this work), while still being comparatively fast.

Entering the HW domain, FPGAs are a way of spinning up small-volume HW implementations. These can be changed multiple times, but only once the HW model is completely designed. The execution speed depends on the workload, but usually is in the hundreds of megahertz range and used for specific, highly parallel tasks or early HW validation. In contrast, the complete test and verification of such designs requires a lot of computing power for simulation while still being slow. Finally, the highest speed and lowest flexibility offer Application-specific Integrated Circuits (ASICs). The time to production ranges from several months

to years to a usable chip, but once finished, may implement whole SoCs clocking on the gigahertz to terahertz range. Here, simulating or verifying the whole system or even sub-systems remains a huge challenge even for the largest companies. This figure can thus also be read as development speed (left) vs. verification effort / complexity (right).

Concluding, the HW/SW partitioning question involves a lot of architectural assumptions like the processor type, interface styles, and memory structure to name a few [11]. All of this makes VPs essential for successful designs, as these architecture characteristics can be modeled fast and to the desired accuracy of the advancing process phases.

Stepping into more details of the actual system design, *embedded systems* are mostly heterogeneous systems, tailored to a very special set of tasks. One of the most far-reaching design decisions is the Instruction Set Architecture (ISA) of the processor; as this defines power consumption, available HW modules (so called Interlectual Properties (IPs)), area, available SW (driver, operating systems, libraries), and lastly, licensing costs. Traditionally, the choice was effectively limited to an x86 CISC[1] or ARM RISC processor model [13] [14, 15]. If the design company was too small or the chip was targeted to a lowest-cost scenario, the only resort were some niche MIPS[2] processors. In the last years, however, a new movement arose: RISC-V [16–18], first introduced in 2019, is a freely available and highly modifiable RISC ISA that is being adopted widely, even in big companies [19, 20], for its flexibility and license-free nature (see also Section 2.3). Thus, VPs and other Instruction Set Simulators (ISSs) implementing this ISA can be published and extended with liberal and free licenses.

In the context of this dissertation, RISC-V is mainly used as a case-study to showcase the developed techniques while maintaining a real-world context that benefits an active community shaped by academia and industry alike.

[1] CISC/RISC: Complex/Reduced Instruction Set Computer. See also Section 2.3 and [12].
[2] MIPS: Microprocessor without Interlocked Pipelined Stages.

1.1 Design Flow with Virtual Prototypes

The main benefit of *virtual prototyping* is the parallelism in developing software and HW simultaneously, essentially cutting the development time in half [21]. Following Figure 1.3, there previously were three mainly sequential design phases:

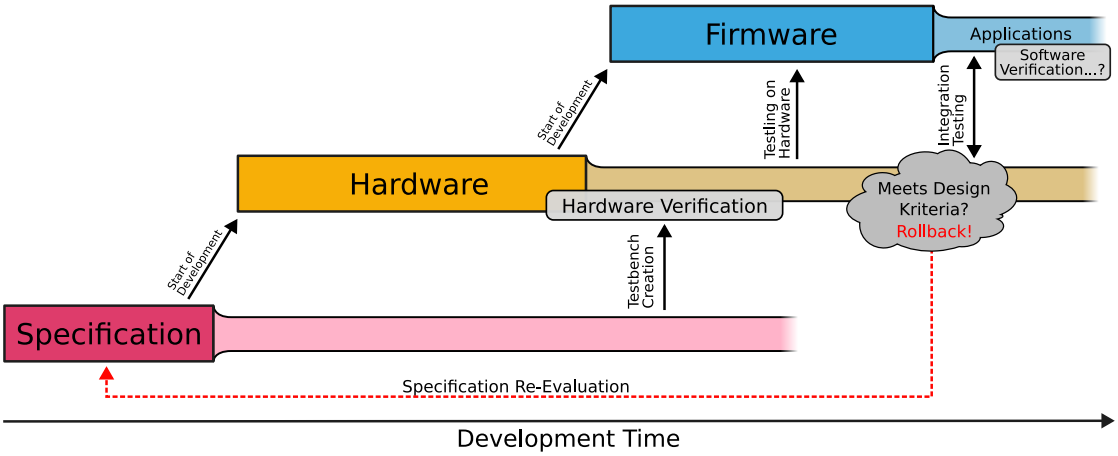


Figure 1.3: Traditional *SW-then-HW* design flow. Notice the possible design rollback due to the missing design space exploration.

The *Specification* phase (in red), the *HW* design phase (in yellow), and finally the *Software* design phase (in blue). So, after a certain specification had been made, the HW had to be modeled to a near complete level to **1)** verify the system for conformity to the specification, and **2)** evaluate this specification whether it meets the design requirements at all (e. g. execution speed, power consumption, accuracy, chip area, overall cost, etc.).

Not only did this cause a huge time overhead in development; possible design specification misjudgments in terms of incorrect assumptions or changed requirements could only be detected or corrected far into the design process, which requires critical rollbacks or re-designs. Furthermore, special care had to be taken in design specification to maintain a high level of security in the IoT, as embedded systems are both very critical *and* very susceptible to security and safety issues, as they usually do not offer risk-mitigating features commonly found in desktop-grade environments such as memory space partitioning, address layout randomization, or memory virtualization techniques in user / supervisor / hypervisor modes [22, 23]. These projects thus heavily depended on either luck no error occurred, or on so-called "rock-star developers". These are individuals that exceed the norm, handle complex problems exceptionally well, and are far less susceptible to errors, enabling projects to overcome the baseline hurdles and pitfalls mentioned earlier. While certainly some star projects succeeded, this dependence makes the

traditional approach unreliable which is thus unsuitable for modern, complex projects, where *anybody* might start developing systems that once were feasible only for the biggest of companies. This *SW-after-HW* approach also allows for verification gaps in the HW/SW interaction layer, which are traditionally only correctable by developing a one-time-use system level model which also needs to be verified against the specification to have any meaningful statement. This adds complexity and misses a lot of re-usable verification chances such as Software Driven Verification (SDV) [8] or Cross-Level Verification (CLV) that re-use the previously one-time-used intermediate model as a *golden reference model*.

While *virtual prototyping* recently grew as an option to the industry, the modeling and verification of such systems is mostly still a manual, and thus error-prone, process. Due to the effort of industry towards maintaining their IPs secret, every new competitor or private individual has to start from the ground up. This slows down the overall innovation rate and reduces the average quality of products. Thus, there is the need for stable, open-source RISC-V VPs that everyone can use, modify, and build upon (as can be observed with recently founded companies like *MachineWare* [24]). Also, for VPs to become a trustworthy source of "correctness", verification *must not* be an afterthought, but rather a constant part of an agile design process. Public incidents, ranging from privacy-related inconveniences [25] to the loss of human lives [26] show the necessity of improving the quality of embedded systems in general and IoT devices specifically. This level of quality is currently especially hard to achieve for smaller companies that do not have a big verification division because of the missing availability and cumbersome application of existing tools.

As such, there is a strong demand of accessible system level design automation tools [27] and agile chip development [28]. In fact, the automation part is what is most important: The best verification tools are meaningless if they are too cumbersome to use or do not map to reality. Also, in the real world without rock-star developers and ever faster product cycles, it is likely that a necessary re-spin of the design process is just not carried out if the errors are found too late, leaving known design or implementation flaws open to save money or time.

To sum up, the following problems in the traditional design flow can be identified:

- Long development time with *HW-then-SW* flow
- Inefficient *design space exploration* encouraging design rollbacks
- Too little re-use of intermediate models
- Missing early system analysis tools that would improve design understanding
- High verification hurdle causing insufficient product quality

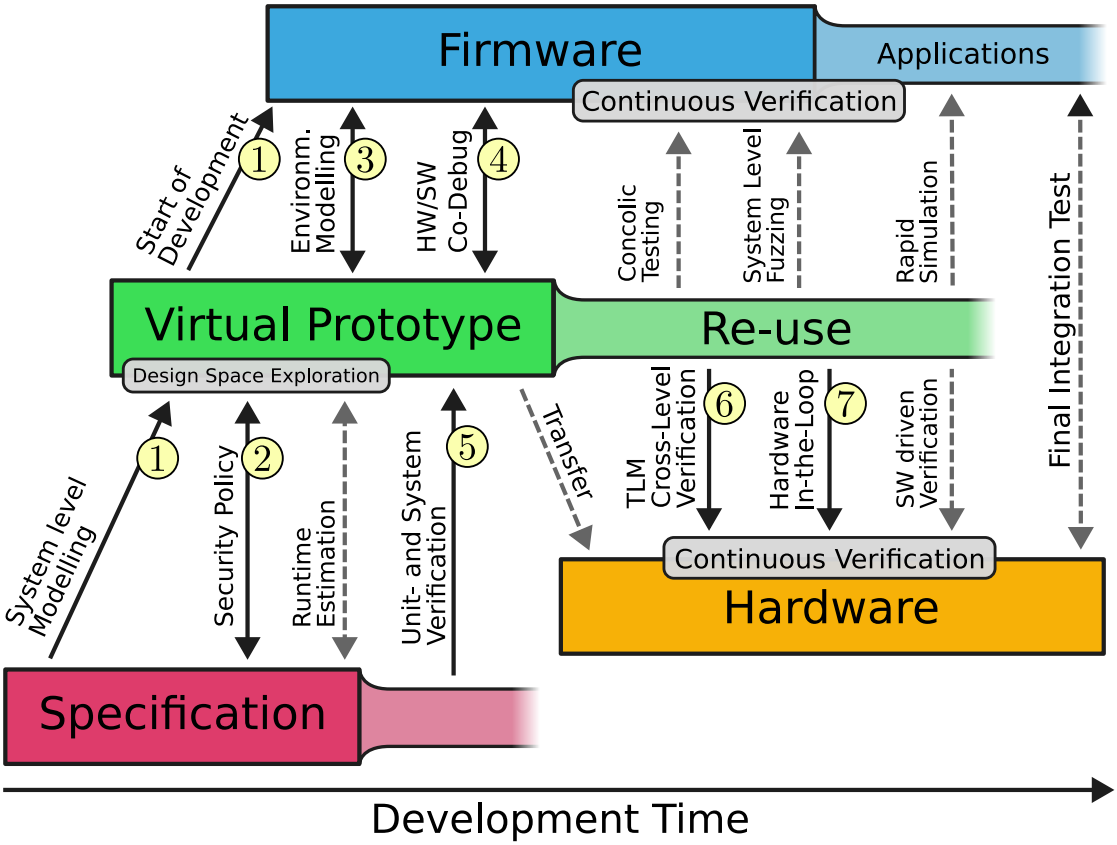


Figure 1.4: Overview of the proposed design flow for fast and agile development of embedded systems.

The modern system design flow proposed in this thesis tackles these problems in an agile way: It is a VP-centric approach that starts with a highly abstracted VP that is developed closely along the specification using advanced modeling approaches. This enables an early design space exploration (*fail early, fail cheap; cf.* [29]) required for complex projects, allowing the SW to be developed simultaneously and thus to be a part of the evaluation process.

Figure 1.4 depicts this advanced design flow. The main work packages are, again, separated into *Specification* (in red), *Software* (in blue), and *Hardware* (in yellow); but with the additional *Virtual Prototype* package in green in between. Processes marked with a circle (ⓝ) are presented in-depth in this thesis, while grey dashed arrows indicate techniques that are important to the proposed design flow but are referenced in related work.

In this proposed advanced design flow, the development of a VP can start alongside the specification phase, synchronous to the abstraction level. This initial system level prototype bring-up is enabled by using the *RISC-V VP* (①, Section 3.1). The RISC-V VP also enables the complete exploration of the design space, as initial SW development can already start for low-level drivers and firmware. Highly secure system designs are additionally improved by using the early security policy evaluation (②, Section 4.3) to analyze and adapt security policies and their feasibility for the given tasks. Along with other criteria that can be estimated with VPs (e. g. runtime, power consumption, complexity), virtual prototyping enables a fast-paced, agile way of exploring requirements in the specification. This, however, is only possible if SW can be brought up fast, too: HW/SW co-design and -debug tools like advanced off-chip environment modeling (④, Section 3.2) and peripheral state visualization (④, Section 3.3) allow the SW-developers to quickly build the firmware from drivers up to the operating system (cf. Figure 1.2).

Decoupled from the development state of the software, during the completion of the specification phase, the conformity of a VP can continuously be verified using SW development tools and techniques (⑤, Section 4.1) and the security policy evaluation approach ②.

After the VP has been verified thoroughly and thus achieved a certain quality, the HW development can start. Here, one of the benefits of SystemC stand out: A subset of SystemC is synthesizable to Hardware Description Languages (HDLs) (see also Section 2.2), so depending on the abstraction level of the SystemC VP, HW-modules may either directly be synthesized or at least interpreted in a way that is native to the HW developer [8]. As the VP is verified thoroughly, it can continuously be used as a (behavioral) golden reference model for cross level verification for the HW models down to the Register Transfer Layer (RTL) layer, securing a high quality and shorter time spent in functional verification. The most important techniques in this phase are early Transaction Level Modeling (TLM)-translated cross level verification (⑥, [30, 31] and Section 4.2) and Hardware-in-the-Loop (HWITL) processes (⑦, Section 3.4), where HW developers can focus on developing the unique selling point devices first, alongside the parallel SW development and the corresponding specialized verification tools (e. g. [32–35]).

On the SW work package, this modern, VP-centric approach also allows the SW to be developed while *continuously* being verified on and against the VP using

different testing and verification approaches that specifically require VPs as their basis [32, 36, 37]. In summary, the main benefit of the proposed design flow for fast and agile development of embedded systems are:

- Early and advanced design space exploration and thus less probable project rollbacks (①, ②, ③)
- Increased re-use rate of existing designs (①, ⑥, ⑦)
- Faster product cycles and thus *time-to-market* (①, ③)
- Better design understanding and debugging capabilities (①, ③, ④)
- Improved handling of complex full-stack systems (① to ⑦)
- Trusted systems with multi-stage verification in different abstraction levels (②, ⑤, ⑥)

1.2 Thesis Contribution

The contributions of this thesis can be summarized in two main areas: **Modeling** and **Verification** of embedded systems using Virtual Prototypes.

By extending and implementing an advanced RISC-V VP (①, [38]), capable of running multiple operating systems including Linux, and an environment modeling system (④, [39, 40]) for live interaction to and from simulated off-chip devices, a strong foundation could be laid out for further tools and techniques. Especially the RISC-V VP, as it was released as an open source project, fueled academic interest and was cited, to date of this publication, at least 50 times by peer-reviewed papers. This includes comparison papers where the RISC-V VP was deemed the best modeling tool for SoCs with its good timing vs. performance balance and small modeling effort [41]. Both modeling tools of the approaches are actively used in educational lectures of domestic and international universities, which was captured by a course study [42]. Also, both contributions are recognized well in academia with [38] being regularly presented at workshops and conferences (lastly the *Latch-Up 2023* in Santa Barbara), and with [40] being mentioned in first place of the *Chip Industry's Technical Paper Roundup: October 2022* in the *Semiconductor Engineering* news-group [43] as well as being featured online on the *Journal of Low Power Electronics and Applications* (*JLPEA*) front page. Also based on the combination of the RISC-V VP and the environment modeling system, the visualization and HW/SW co-debugging tool (③, [44]) offers a strong introspective for design understanding and analysis.

In the verification area, two main VP-based applications could be contributed: A Dynamic Information Flow Tracking-powered approach (②, [45]) for early design space exploration *and* verification of system-level security policies of the full HW/SW stack, and the verification of SystemC TLM peripherals using symbolic execution techniques (⑤, [46]). Also, this thesis' topic was presented and discussed in the *DAC22* and is admitted at the *Design, Automation & Test in Europe* (*DATE*) *'23* conference as part of their respective PhD-Fora.

To increase adoption of the modern design process, all tools of the scientific contributions mentioned in this thesis (except for [44] due to licensing reasons) have been made publicly available as open-source projects to stimulate further research and broaden the RISC-V community [47–51].

In summary, this thesis features the following peer-reviewed publications of the author:

1. [45] Pascal Pieper, Vladimir Herdt, Daniel Große, and Rolf Drechsler (2020). Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes. In *2020 57th ACM/IEEE Design Automation Conference* (*DAC*) (*pp. 1-6*).
2. [38] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler (2020). RISC-V based virtual prototype: An extensible and configurable platform for the system-level. In *Journal of Systems Architecture* (*JSA*) (pp. 109).
3. [44] Pascal Pieper, Ralf Wimmer, Gerhard Angst, and Rolf Drechsler (2021). Minimally Invasive HW/SW Co-Debug Live Visualization on Architecture Level. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI* (*GLSVLSI*) (*pp. 321–326*). ACM.
4. [39] Pascal Pieper, Vladimir Herdt, and Rolf Drechsler (2022). Advanced Environment Modeling and Interaction in an Open Source RISC-V Virtual Prototype. In *Proceedings of the Great Lakes Symposium on VLSI* (*GLSVLSI*) *2022* (*pp. 193–197*). ACM.
5. [46] Pascal Pieper, Vladimir Herdt, and Rolf Drechsler (2022). Verifying SystemC TLM Peripherals using Modern C++ Symbolic Execution Tools. In *2022 59th ACM/IEEE Design Automation Conference* (*DAC*) (*pp. 1-6*).
6. [40] Pascal Pieper, Vladimir Herdt, and Rolf Drechsler (2022). Advanced Embedded System Modeling and Simulation in an Open Source RISC-V Virtual Prototype. In *Journal of Low Power Electronics and Applications* (*JLPEA*).
7. [52] Pascal Pieper, Sallar Ahmadi-Pour, and Rolf Drechsler (2023). Virtual-Peripheral-in-the-Loop: A Hardware-in-the-loop Strategy to Bridge the VP/RTL Design-Gap. Under review for the *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis* (*CODES+ISSS*).

1.3 Thesis Organization

The following chapters present the approaches mentioned in the advanced design flow and experimental results: First Chapter 2 is an overview over the most common background knowledge necessary to understand this thesis, such as the structure of embedded devices, the RISC-V standards, and SystemC / TLM. In the following central Chapters 3 and 4, the two main cornerstones of this thesis' contribution are explained in full detail: System modeling and SW/HW co-design (Sections 3.1 to 3.4), and the verification of such systems, with a focus on the SoC's peripherals (Sections 4.1 and 4.2) and security policies conformity using dynamic information flow tracking (Section 4.3). Lastly, the conclusion and possible future research directions are discussed in Chapter 5.

The assignment of peer reviewed papers to the individual sections are as follows:

- Chapter 3: System Modeling with VPs
  - Section 3.1: SoC Modeling - [38]
  - Section 3.2: Off-Chip Modeling - [39, 40]
  - Section 3.3: SW/HW Co-Debugging - [44]
  - Section 3.4: RTL development speedup with HWITL - [52] (*under review*)
- Chapter 4: Verification with VPs
  - Section 4.1: TLM Peripheral Verification - [46]
  - Section 4.2: TLM/RTL Cross-Level Peripheral Verification
  - Section 4.3: Security Policy Verification using Dynamic Information Flow Tracking - [45]

Chapter 2
Preliminaries

This section gives an overview over the common topics the following work builds upon, to keep this thesis self-contained. Individual, per-chapter specific topics (such as security lattices, Section 4.3) will however be introduced in their respective subsections.

As the proposed advanced design flow is focused on embedded devices (ranging from simple micro-controllers to huge SoCs), the concept of embedded devices and how they are usually laid out is explained first. The development of such devices is nowadays aided with VPs, so the *status quo* of virtual prototyping with a focus on modeling with SystemC and TLM in particular is explained, and the trade-off between simulation accuracy and execution speed is introduced. Lastly, a summary and the rationale of the RISC-V ISA and its ecosystem, which is used as a functional case study in this thesis, are outlined.

2.1 Embedded Devices

Embedded devices are usually a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function [53, 54]. They range from the simplest 4-bit devices such as melody-playing greeting-cards to highly sophisticated control systems in airplanes or high-throughput Digital Signal Processors (DSPs) in wireless access points for cell-phone networks. Their key common property is that they are purpose-built for very specific tasks. Due to their nature, they often work at an interplay between HW and SW, creating the term *firm*ware for the programming that drives these systems. This firmware might often be left unchanged for years without end or reboot, managing single or multiple tasks with sometimes mixed real-time requirements.

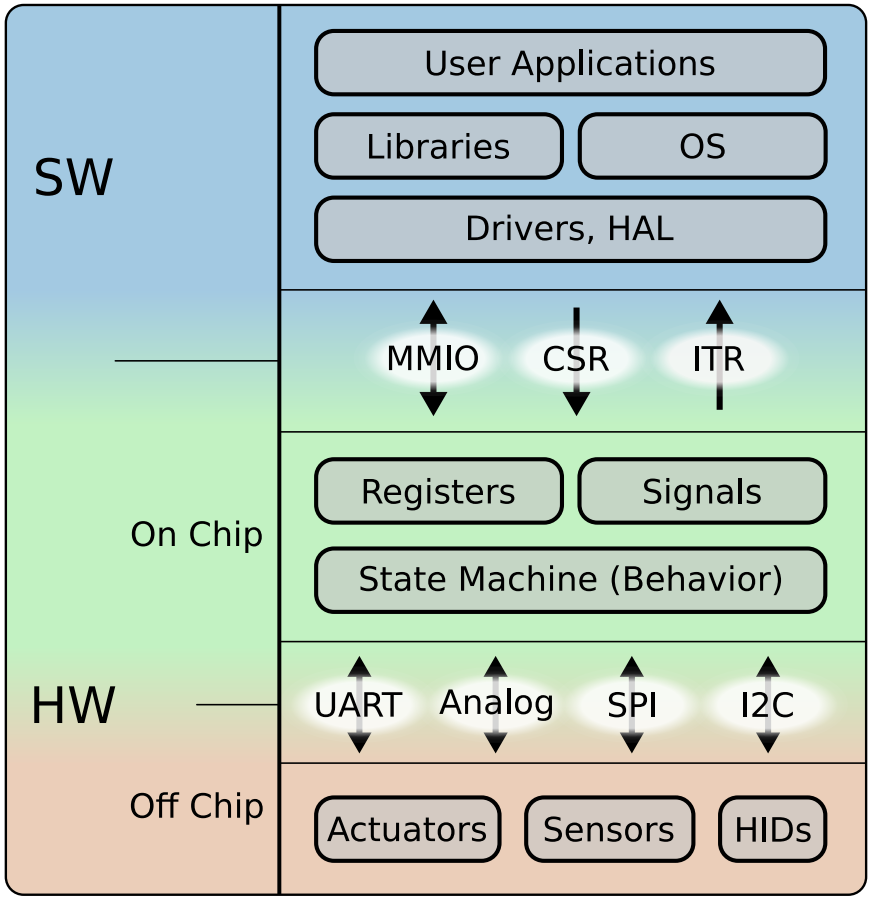


Figure 2.1: Behavior model of an embedded device in the scope of this thesis. It consists of three layers: The software stack (binary running on a chip, in *blue*), the on-chip peripherals (hardware functionality, in *green*), and off-chip devices that are part of the deployed system (*orange*).

A simplified view on a general embedded system is given in Figure 2.1. The boundaries between software (blue), on-chip hardware (green) and off-chip hardware (red) are blurred, as the communication channels and functionalities are not well-defined to reside on one side or the other.

Starting with the SW, the highest abstraction layer is usually the user applications. Here, the term *user* refers to the engineers that develop the actual mission-related tasks, in contrast to the lower layers that focus on housekeeping, managing the device-specific hardware and enabling the execution of these tasks in the first place. *Libraries*, *Drivers*, the (optional) OS, and potential Hardware Abstraction Layers (HALs) usually interface with the HW in multiple ways. Besides the processor specific (or extended) ISA (see Section 2.3) and their defined Control and Status Registers (CSRs), control over the hardware will usually be exercised in *von Neumann* architectures by memory-mapped I/O, while interrupts / traps close the control loop back to the software. Interrupts and traps offer a way for the HW to signal readiness of new data or required attention from the SW in general, while memory-mapped I/O offers means of reading and writing data to and from on-chip hardware peripherals via a bus-like shared connection. This data may be interpreted as commands, payload, or whatever else, thus creating a very flexible and cheap interface method. Please note that, while *isolated* I/O schemes exist, they come with several drawbacks. These drawbacks include the need for dedicated read/write assembler commands and irregular microarchitectures, and are thus not part of modern RISC ISAs like RISC-V. On this level, the ISA can be seen as the "contract" between the SW and HW defining the interface, and the microarchitecture as the actual CPU implementation with its own trade-offs and costs. On SW side, these accesses are simply read or write instructions to certain addresses, as if they were contained in memory. The actual data/instruction memory in this case may just be one of many peripheral bus participants, albeit an important one.

The on-chip peripherals are then usually ranging from simple devices such as the General Purpose Input/Output (GPIO)-system, over Analog-to-Digital Converter (ADC)/Digital-to-Analog Converter (DAC) modules or digital protocol links such as Universal Asynchronous Receiver / Transmitter (UART), to state-machine controlled hardware-accelerators and Direct Memory Access (DMA) controllers. Especially in this domain, the complexity against speed or other trade-offs have a big impact on decisions where a certain workload will be implemented in, or the next bottleneck will reside.

At the other end, embedded systems are usually not just one chip device, but form a network between high- and low-level control loops, sensors, and actuators. In the following, a small outline of the protocols and devices mostly used in

the case-studies is presented. The list is backed by [53, 54] and is by no means exhaustive.

UART Universal Asynchronous Receiver / Transmitter is, as the name implies, a serial asynchronous protocol that transmits simplex and unclocked bytes. Depending on the actual configuration, it most often comes in the configuration *8-bit, single stop bit, no parity* in baud-rates from 9600 to 115 200 symbols per second, with dedicated receive and transmit lines. It is usually used to transmit/receive text-based commands or raw bytes in a custom, higher level protocol like talking to a modem or an In-System Programmer (ISP).

SPI The Serial Peripheral Interface is a high-speed, clocked protocol with dedicated receive and transmit connections (Master-In-Slave-Out (MISO), and Master-Out-Slave-In (MOSI)), along with a Chip Select (CS) line. It is intended to be used in a one-to-many connection from a session-initiating host, and mostly used for either extended program memory (e. g. SPI-Flash) or interface with high-data-rate sensors. Readyness/Attention of clients is usually indicated by a separate interrupt pin.

I²C Inter-Integrated Circuit, or sometimes *twowire*, is a serial protocol with the key feature of requiring only one, duplex, data line and a clock line. It is usually used for sensors and smaller actuators such as power distribution devices on mainboards in a bus-like configuration. The devices share the same connections and are addressed using, usually, 7 bit identification numbers, followed by a variable number of data bits that can be interpreted as an in-device register address or actual payload data. Every transaction is either acknowledged or dismissed by a following bit from the target device.

PWM Pulse Width Modulation is a common technique to emulate analog signals *on average* on digital lines. Especially Metal-Oxide Semiconductor Field-Effect Transistors (MOSFETs), used as output drivers of a chip's GPIO pins, can deliver a comparatively high current only when they are fully turned on. To still drive in-between values for, e. g., motor drivers or Light Emitting Diode (LED) dimming, the output pin can be rapidly turned on and off. The ratio between on- and off-time is called *duty-cycle* and defines the averaged analog output value. Usual frequencies are in the kilohertz range where a human eye can not perceive the individual on- and off-cycles, or the inertia of a physical motor filters the pulses.

LED Light Emitting Diodes are semiconductors that use differently doped silicon to stimulate electrons of a certain energy to emit light in a corresponding wavelength. The energy (and thus, color) depends on the band-gap between the two semiconductors, which nowadays ranges from infra-red to ultra-violet. This process is, compared to incandescent light bulbs, very efficient. LEDs are non-Ohmic conductors and need a separate current limiter when driven with a voltage source. Due to a mostly very narrow operating voltage range, dimming (i. e. limiting output power) is mostly done via PWM which also increases the efficiency compared to actual current controlling circuits.

PCB Printed Circuit Boards are connection boards that host electrical devices like chips, transistors, passive components, and other components categorized as through-hole- and surface-mounted devices. Nearly every electrical device consists of one or multiple PCBs, acting as the foundation of all components. PCBs consist of copper on passive media like FR4, a flame retardant epoxy resin and glass fabric composite. This is layered (usually ranging from 1 to 10 layers), as the copper is systematically etched away to only remain in previously specified locations, building up the connections. After etching, the layers are glued together, drilled, and connected with so-called *vias*. Finally, the components can be soldered on to the PCB.

2.2 SystemC / TLM

SystemC [4] is an industry-proven modeling standard to model digital systems [55], especially VPs [56]. It is not a new language however, it rather is a C++ class library which includes an event-driven simulation kernel [4, 57]. The main benefit of SystemC is the flexible trade-off between timing accuracy and simulation time, operating from abstract TLM down to the RTL. This support for multiple abstraction levels enables developers to refine the design and even re-use the model to verify the final hardware [58, 59].

The structure of a SystemC design is described with ports and modules, whereas the behavior is modeled in processes which are triggered by events. The execution of a process is non-preemptive, i. e. the kernel receives the control back if the process has finished its execution or suspends itself by calling `wait()`. SystemC provides three types of processes with `SC_THREAD` being the most general type, i. e. the other two can be modeled by using `SC_THREAD`. For event-based synchronization, SystemC offers many variants of `wait()` and `notify()` such as `wait(time)`, `wait(event)`, `event.notify(delay)`, `event.notify()`, etc. The most prominent implementation of SystemC, the Accellera™ SystemC library [60], features a user-space scheduler instead of using the Operating System's scheduling to increase the speed of frequent context-switches at the cost of a complex code base. For every execution step, it jumps to the time of the next active event(s). It will then execute all waiting threads or methods that are "sensitive" to these events in an unspecific order. If other events were notified during this time-step, they are also executed in *zero time*, i. e. without advancing the simulation time. If all communications are settled, the simulation jumps to the next time where time-sensitive events are waiting.

Communication between SystemC modules is abstracted using TLM transactions at the cost of timing accuracy, but with significant improvements in simulation speed. Compared to RTL simulations, the execution is usually faster by a factor of 1000 (although retaining cycle-accuracy is possible [61]). A TLM transaction object essentially consists of a command (e. g. read/write) and the data (payload) to be transmitted. Transactions are routed based on their address from an initiator to a target socket which is all defined in the SystemC TLM-2.0 standard and allow very fast interactions between modules. Optionally, a transaction can be associated with a delay (modeled as `sc_time` data structure), which denotes the execution time of the transaction and allows to obtain a more accurate overall simulation time estimation. Furthermore, those TLM transactions can be used with *blocking* and *non-blocking* communication mechanisms (see Figure 2.2), in Loosely Timed (LT) and Approximately Timed (AT) coding styles. In the LT coding style, communication is done by temporal decoupling of accesses between models, i. e.

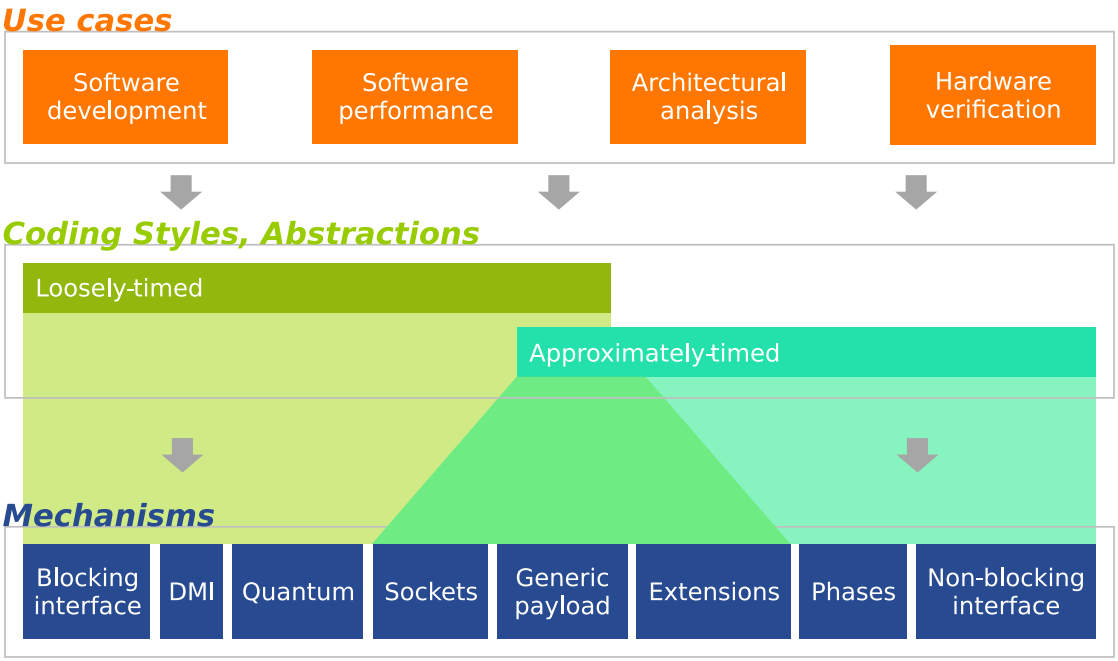


Figure 2.2: Classification of degrees of SystemC timing accuracy [4]. From left to right the models gain timing accuracy with a decreasing execution speed.

possibly finishing the transmission "in the future". These transactions can either *read* or *write* at a specified target socket, carrying a generic payload along with a cumulative delay, and may return success or error status values. This delay is increased by every model passing the transaction and added to a global quantum afterward. The global quantum tracks the time difference a transaction "jumped" in contrast to the actual simulated time. If this difference is bigger than a certain maximum allowed time, the SystemC kernel will initiate a global synchronization. This allows for a fine control over the trade-off between simulation speed and accuracy in LT mode.

The non-blocking interface (in AT models) however consists of a number of (user-defined) phases, in where the progress of the communication through individual transmissions need to be handled by the respective models. This also needs to be synchronized with time and added status information, down to the accuracy of RTL models; which in turn heavily decreases the simulation's performance.

2.3 RISC-V Instruction Set Architecture

RISC-V is an open and free Instruction Set Architecture. It was started in 2014 as a purely academic project and gained traction in the last years; mainly because of its extensibility and lack of licensing fees. With the benefit of no need of backwards-compatibility, a completely new and modern ISA could be developed, reaching or outperforming existing ISAs while being more efficient in general [62]. A particularly interesting choice was to simplify the microarchitecture by moving a part of the complexity to the compilers – a traditional RISC-approach. This idea played out well, as nowadays next to no-one needs to write whole programs in assembly languages, shifting the trend more to high-level (compiled) languages. This factor, together with increasing computational power in general, enables processors both in the highly specific low-power / embedded domain and the high-power server market to have a common base ISA. Obviously, though, the set of features differ enormously between the two poles. This is where the extensibility shines: There is a shared base instruction set, but for everything else there is is an ever-growing set of standardized and intentionally left out (user-) Instruction- and Control and Status Register extensions (see Table 2.1). This drew attention especially from the machine learning industries, as there existed a need to incorporate, e. g., neural-net and cryptography accelerators neatly and efficiently into their SoCs [63]. Nowadays, with the huge ecosystem that has evolved around RISC-V, it also lets companies focus on their main selling-point (such as HW accelerators), without having to reinvent the metaphorical wheel again in the form of instruction set cores and infrastructural and SW components.

The base ISA consists of a mandatory base integer instruction set (denoted `RV32I`, `RV64I` or `RV128I` with corresponding register widths) and various optional extensions denoted as single letters, e. g. `M` (integer multiplication and division), `C` (compressed instructions), etc. Thus, `RV32IMAC` denotes a 32 bit core with `M`, `A` and `C` extension. The base instruction set is very compact, `RV32I` consists of 47 instructions and, for example, the `M` extension adds additional 8 instructions. The second volume of the RISC-V ISA defines privileged architecture description [17]. The RISC-V ISA also defines CSRs, which are registers serving a special purpose. For example the *mtvec* (Machine Trap-Vector Base-Address) CSR stores the address of the trap/interrupt handler.

Furthermore, the RISC-V ISA provides a small set of instructions for interrupt handling (`wfi`, `mret`) and interacting with the system environment (`ecall`). For a comprehensive description of the RISC-V ISA, please refer to the official specifications [16, 17]. All `RV32IMA` instructions have a 32 bit width and use at most two source and one destination register. The `C` extension adds 16 bit encoding

for common operations (saving encoding space by, e. g., grouping source and destination register), which can be expanded into an existing 32 bit `I` instruction.

Abbreviation	Version	Status	Description
RVWMO	2.0	**Ratified**	Weak memory ordering model
RV32I	**2.1**	**Ratified**	Integer base instructions 32-bit
RV64I	**2.1**	**Ratified**	Integer base instructions 64-bit
RV32E	**2.0**	**Ratified**	Reduced base instructions 32-bit
RV64E	**2.0**	**Ratified**	Reduced base instructions 64-bit
RV128I	*1.7*	*Draft*	Integer base instructions 128-bit
M	**2.0**	**Ratified**	Multiplication / division instructions
A	**2.1**	**Ratified**	Atomic operations
F	**2.2**	**Ratified**	Single-precision floating-point instructions
D	**2.2**	**Ratified**	Double-precision floating-point instructions
Q	**2.2**	**Ratified**	Quad-precision floating-point instructions
C	**2.0**	**Ratified**	Compressed instructions
Counters	*2.0*	*Draft*	Performance counter CSRs
L	*0.0*	*Draft*	Decimal floating point instructions
P	*0.2*	*Draft*	Packed Vector instructions
V	*0.7*	*Draft*	Vector instructions
Zicsr	**2.0**	**Ratified**	Control and Status Registers
Zifencei	**2.0**	**Ratified**	Synchronization between instruction fetches
Zihintpause	**2.0**	**Ratified**	Hint for low-power operations
Zam	*0.1*	*Draft*	Allows misaligned atomics in `A`
Zfh	**1.0**	**Ratified**	Half-precision floating-point operations
Zfhmin	**1.0**	**Ratified**	Minimal subset of `Zfh`
Zmmul	**1.0**	**Ratified**	Multiplication-only instructions in `M`
Ztso	**1.0**	**Ratified**	Total Store Ordering for memory model

Table 2.1: Excerpt of current (as of June 2023) ratified or soon-to-be ratified extensions of the RISC-V ISA, extended, from [16]. The top lines are *Base* instructions where at least one instance needs to be implemented, while the lower ISA definitions are optional *Extensions*.

Chapter 3

Hardware and Environment Modeling

The ubiquity of IoT devices opens a new world of possibilities for both personal and industrial applications. The key idea of the IoT is to have a multitude of tiny and low power sensors and actuators connected in big, distributed wireless networks. All of these devices are heavily specialized and, for this idea to be feasible, need all of their on- and off-chip components to be as cheap as possible. Hence, as an open and free instruction set architecture, RISC-V is gaining huge popularity for IoT. As stated in Section 2.3, RISC-V is a modern ISA that, with its open nature and a combination of a clean and modular design, has enormous potential to become a game changer in the IoT era. It is to no surprise that a large ecosystem is already available around RISC-V, including various RTL implementations at one end and high-speed ISSs at the other end. These implementations can, as always, be categorized in the execution speed against accuracy and development time trade-offs. Among the fastest implementations of RISC-V interpreters, besides actual FPGA/ASIC implementations, are Instruction Set Simulators like [64] (fast execution speed) or [65] (fast development time of the ISS, early adoption). These focus only on the execution of RISC-V assembler instructions, with either none or only highly abstracted peripheral devices for the sake of execution speed and simplicity. Because of this, ISSs are usually used either for ISA design space exploration or testing of high-level SW (e. g. operating systems). The ISSs, when executed on powerful host machines, sometimes achieve 1:1 performance (simulation time vs. wall-clock time) or better. However, being predominantly designed for speed, they can hardly be extended to support further Electronic System Level (ESL) use cases such as power/timing/performance validation, analysis of complex HW/SW interactions, or analysis and simulation of off-chip devices [66]. On the lowest level, HDLs models on the RTL like the Berkeley out-of-order machine [67], can be simulated in the magnitude of ten to a hundred thousand seconds per second simulation time at the benefit of cycle-time accuracy. This simulation

necessitates, of course, a fully developed system with a detailed microarchitecture and peripherals, which is *too late* in the design process of complex systems.

Virtual Prototypes are designed to fill this niche in-between: At execution speeds next to real-time and hundreds of seconds per second of simulation time, they enable the design team to continuously adapt the executable model to the current project scope from a high-level behavioral system level down to the RTL.

This chapter presents a number of hardware modeling approaches; starting from the developed RISC-V VP, over an Environment Model GUI simulating off-chip devices and protocols virtually connected to the RISC-V VP, debugging and visualization techniques for on-chip peripherals with RISCVIEW, to finally a hardware-in-the-loop system for virtually connecting RTL peripherals on FPGAs to the RISC-V VP.

Firstly, in Section 3.1, a RISC-V based VP is proposed and implemented with the goal of filling this gap between early system design and fully finished system, presented in an extended version of the original publication [38]. It provides a 32- and 64-bit RISC-V core supporting the `IMACFD` instruction sets with different privilege levels, the RISC-V Core-Local Interruptor (CLINT) and Platform Level Interrupt Controller (PLIC) interrupt controllers and an essential set of peripherals. The simulation of (mixed 32 and 64 bit) multi-core platforms is supported, and SW debug and coverage measurement capabilities are provided, along with support for the FreeRTOS [68], Zephyr [69], RIOT [70], and Linux operating systems. The VP is designed as extensible and configurable platform with a generic bus system and implemented in standard-compliant SystemC and TLM-2.0. The latter point is very important, since it allows leveraging cutting-edge SystemC-based modeling techniques needed for the mentioned use cases. The RISC-V VP allows a significantly faster simulation compared to RTL, while being more accurate than existing ISSs. Finally, the RISC-V VP is fully open source (MIT license) to help expanding the RISC-V ecosystem and stimulating further research and development.

In the following Section 3.2, an Environment Model GUI is presented to broaden the application domain for virtual prototyping in the RISC-V context in form of an extended version of the original publications [39, 40]. It builds upon the previously presented RISC-V VP and adds a standalone Graphical User Interface (GUI) for visualization purposes of the environment using the Qt C++ library which communicates to the VP through a Transmission Control Protocol (TCP) connection. Additionally, appropriate libraries were designed to offer hardware communication interfaces such as GPIO and SPI from the VP to an interactive environment model. The approach is generally designed to be integrated with VPs that leverage a TLM communication system to prefer a speed optimized simulation. To show the practicability of an environment model, a set of building blocks

such as buttons, LEDs and Organic Light Emitting Diode (OLED) displays are provided, including configurations for two demonstration environments. Moreover, for rapid prototyping purposes, another modeling layer is provided that leverages the dynamic Lua scripting language to design components and integrate them faster and more easily with the VP-based simulation. Lastly, an evaluation with two different case-studies is given that demonstrates the applicability of the approach in building virtual environments effectively and correctly in matching the real physical systems. To advance the RISC-V community and stimulate further research, the extended VP platform is provided along with the environment configurations and visualization toolbox and the case studies on GitHub [48].

Next up, Section 3.3 extends the previously presented RISC-V VP with an easily configurable graphical debugging tool called RISCvIEW, as published in [44]. It features a GUI that allows developers to debug hard- and software and their interaction in an early design stage, aided by the standard software debugger. The GUI visualizes the internal state of the hardware, rendered automatically using the industrial-strength drawing engine Nlview™ [71]. The schematics are annotated with live simulation data, i. e. the current values of signals, busses, and registers while executing software instructions. These annotations can include internal values that are not accessible from the hardware's interface via software instructions, but are still necessary for understanding and debugging the system's state. At the same time, the software debugger monitors and allows to manipulate the state of the software. This co-visualization supports design understanding and live debugging of the HW/SW interaction. The usefulness is demonstrated with a case-study where an OLED display driver is debugged while running on the RISC-V VP.

Finally, Section 3.4 presents a method to close the TLM/RTL gap by bridging memory-mapped peripherals between a SoC-like VP and RTL models on FPGAs. While virtual prototyping lessens the design-gap and improves the verification abilities of the SoC design process, there exists another gap, as the step from an architectural level VP implementation on the TLM to the RTL implementation is considerably big.

Especially when a company wants to focus on their Unique Selling-Point (USP), the HW Design Space Exploration (DSE) and acceptance tests should start as early as possible. Traditionally, this can only start once the (minimum) rest of the SoC is also implemented in the RTL. As SoCs consist of many common subsystems like processors, memories, and peripherals, this may impact the time-to-market considerably. This is avoidable, however: This section proposes a Hardware-in-the-Loop strategy called Virtual Peripheral in-the-Loop (VPIL) that allows to bridge the gap between VP and RTL designs that empowers engineers to focus on their USP while leveraging an existing suite of TLM IPs for the common base-system

components. It is shown how VPs and partial RTL implementations of a SoC can be combined in a HWITL simulation environment utilizing FPGAs. The proposed approach allows early DSE, validation, and verification of SoC sub-components, which bridges the TLM/RTL gap. The approach is evaluated with a lightweight implementation of the proposed protocol, and three case-studies with real-world peripherals and accelerators on HW. Furthermore, the capabilities of the approach are assessed and practical considerations for engineers are offered, to utilizing this HWITL approach for SoC design. Finally, further extensions are proposed that can boost the approach for specialized applications like high-performance accelerators and computation.

3.1 RISC-V based Virtual Prototype: An Extensible and Configurable Platform for the System-level

This section includes and extends published material from a previous conference paper [72] and the following journal extension [38]. After a motivation in Subsection 3.1.1 and a discussion of related work in Subsection 3.1.2, Subsection 3.1.3 reviews essential background information on RISC-V instructions and architecture, as well as a more specific dive into the SystemC TLM bus architecture. Next, an overview of the VP architecture is given in Subsection 3.1.4. Then, in Subsection 3.1.5, the VP interaction with the SW and environment, by means of interrupts and syscalls (system calls), is presented in more detail. Subsection 3.1.6 shows and explains performance optimizations, and Subsection 3.1.7 describes the extension to simulate multi-core platforms. In Subsection 3.1.8, additional examples are provided that further demonstrate the configurability and extensibility of the RISC-V VP. The quality, applicability and performance of the RISC-V VP is evaluated in Subsection 3.1.9. Finally, Subsection 3.1.10 provides a discussion of future work and Subsection 3.1.11 concludes this section.

3.1.1 Introduction

Enormous innovations are enabled by the IoT-market since every device is connected to the Internet. Forecasts see additional economic impact resulting from industrial IoT. In the last years, the complexity of IoT devices has been increasing steadily with various conflicting requirements. On the one hand, IoT devices need to provide smart functions with a high performance including real-time computing capabilities, connectivity and remote access as well as safety, security and high reliability. At the same time they have to be cheap, work efficiently with an extremely small amount of memory and limited resources and should further consume only a minimal amount of power to ensure a very long runtime.

To meet the requirements of a specific IoT system, a crucial component is the processor. Stimulated from the enormous momentum of open source software, a counterpart on the hardware side recently emerged: *RISC-V* [16, 17]. As stated in Section 2.3, RISC-V is an open-source ISA which is license-free and royalty-free. The ISA standard is maintained by the non-profit RISC-V foundation and is appropriate for all levels of computing systems, i. e. from microcontrollers to supercomputers. The RISC-V ecosystem is rapidly growing, ranging from HW, e. g. various HW implementations (free as well as commercial) to high-speed ISSs. These ISSs facilitate functional verification of RTL implementations as well as early SW development to some extent. However, being designed predominantly for

speed, they can hardly be extended to support further system-level use cases such as design space exploration, power/timing/performance validation or analysis of complex HW/SW interactions.

A major industry-proven approach to deal with these use cases in earlier phases of the design flow is to employ VPs [73] at the abstraction of ESL [66]. In industrial practice, the standardized C++-based modeling language SystemC and TLM techniques [4, 74] are being heavily used together to create VPs. Depending on the specific use case, advanced state-of-the-art SystemC-based techniques beyond functional modeling (see e. g. [75–79]) are to be applied on top of the basic VPs. The much earlier availability as well as the significantly faster simulation speed in comparison to RTL are among the main benefits of SystemC-based VPs.

In this section, a RISC-V based VP is proposed and implemented to further expand and bring the benefits of VPs to the RISC-V ecosystem. With the goal of filling the mentioned gap in supporting further system-level use cases, SystemC is necessarily the language of choice. The VP is therefore implemented in standard-compliant SystemC and TLM-2.0 and designed as extensible and configurable platform with a generic bus system. The RISC-V VP provides a 32 and 64 bit RISC-V core supporting the `IMACFD` instruction set with different privilege levels, the RISC-V CLINT and PLIC interrupt controllers and an essential set of peripherals. Furthermore, it supports the simulation of (mixed 32 and 64 bit) multi-core platforms, provides SW debug and coverage measurement capabilities and supports a number of operating systems (e. g. FreeRTOS, Zephyr, RIOT, and Linux) operating systems. This section further demonstrates the extensibility and configurability of the RISC-V VP by three examples: addition of a sensor peripheral, describing the integration of the GNU Project debugger (GDB) SW debug extension, and configuration to match the RISC-V HiFive1 board from SiFive. The evaluation section demonstrates the quality and applicability of the RISC-V VP to real-world embedded applications and shows the high simulation performance of the RISC-V VP. As always, the RISC-V VP is made fully open source on GitHub [49] (MIT license) to stimulate further research and development.

3.1.2 Related Work

As mentioned earlier, the RISC-V ecosystem already has various high-speed ISSs such as the reference simulator SPIKE [65], RISCV-QEMU [64], RV8 [80] or DBT-RISE [81]. They are mainly designed to simulate as fast as possible and predominantly employ dynamic binary translation (to `x86_64`) techniques. This is however a trade-off as accurately modeling power or timing information for instructions becomes much more challenging.

The full-system simulator gem5 [82], also has initial support for RISC-V. *gem5*

provides more detailed models of processors and memories and can in principle also be extended for accurate modeling of extra-functional properties. Renode [83] is another full-system simulator with RISC-V support. Renode puts a particular focus on modeling and debugging multi-node networks of embedded systems. However, both gem5 and Renode employ a different modeling style and thus hinder the integration of advanced SystemC-based techniques.

FORVIS [84] and GRIFT [85] are Haskell-based implementations that aim to provide an executable formalization of the RISC-V ISA to be used as foundation for several (formal) analysis techniques. SAIL-RISCV [86] aims to be another RISC-V formalization that is implemented in Sail, which is a special language for describing ISAs with support for generation of simulator backends (in C and OCaml) as well as theorem-prover definitions.

The project SoCRocket [87] that develops an open-source SystemC-based VP for the SPARC V8 architecture, can be considered comparable to this proposed VP.

Finally, commercial VP tools such as Synopsys Virtualizer or Mentor Vista might also support RISC-V , but their implementation is proprietary.

The RISC-V VP is implemented in standard compliant SystemC TLM-2.0, which is an industry-proven modeling standard (IEEE-1666 [4]), and published as open-source to extend the RISC-V ecosystem and lay the foundation for advanced SystemC-based system-level use cases for RISC-V.

3.1.3 Preliminaries

3.1.3.1 RISC-V: Atomic Instruction Set Extension

The RISC-V Atomic instruction set extension enables implementation of synchronization primitives between multiple RISC-V cores. It provides two types of atomic instructions:

1) A set of `amo` instructions. For example, `amoadd.w` loads a word `X` from memory, performs an addition to `X` and stores the result back into the same memory location. Furthermore, `X` is also stored in the destination register `RD` of the `amo` instruction. The load and store operation of the `amo` instructions are executed atomically, i. e. they are not interrupted by other memory access operations from other cores.

2) The `lr` (Load Reservation) / `sc` (Store Conditional) instructions. `lr` loads a word from memory and places a reservation on the load address. The reservation size must at least include the size of the memory access. `sc` stores a result to memory. `sc` succeeds if a reservation exists on the store address and no other operation has invalidated the reservation, e. g. a store on the reserved address range. Otherwise, `sc` fails with an exception, which in turn triggers a

trap. A sequence of corresponding `lr` / `sc` instruction should satisfy the RISC-V *"forward-progress"* property, i. e. essentially, contain no more than 16 simple integer instructions (i. e. no loads, stores, jumps or system calls) between the `lr` and `sc` pair.

3.1.4 RISC-V based VP Architecture

The VP is implemented in SystemC and designed as extensible and configurable platform around a RISC-V `RV32/64IMACFD` (multi-)core with a generic bus system employing TLM-2.0 communication and support for the GNU toolchain with the SW coverage measurement tool Gnu Coverage Tool (GCOV) and debug capabilities via GDB. Overall, the VP consists of around 12 000 lines of C++ code with all extensions. Figure 3.1 shows an overview of the VP architecture. This is described in the following subsections in more detail.

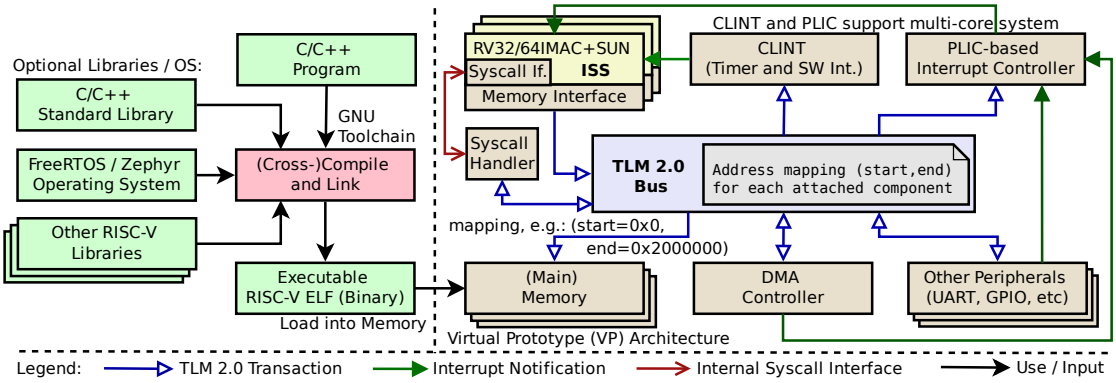


Figure 3.1: RISC-V VP architecture overview.

3.1.4.1 RV32/64 (Multi-)Core

The CPU core loads, decodes and executes one instruction after another. RISC-V compressed instructions are expanded on the fly into regular instructions in a pre-processing step before being passed to the normal execution unit. A 32-bit and 64-bit core is provided, supporting the RISC-V `RV32IMACFD` and `RV64IMACFD` instruction sets, respectively. Besides the mandatory Machine mode, each core implements the RISC-V Supervisor and User mode privilege levels and provides support for user mode interrupt and trap handling (`N` extension). This includes the CSRs for the corresponding privilege levels (as specified in the RISC-V privileged architecture specification [17]) as well as instructions for interrupt handling (`wfi`, `m/s/uret`) and environment interaction (`ecall`, `ebreak`). More details on the

implementation of interrupt handling and system calls (environment interaction) follow in the coming subsections.

Multiple RISC-V cores can be integrated to build a multi-core platform (see upper middle of Figure 3.1). It is also possible to mix 32 and 64 bit cores. The Atomic extension provides instructions to enable synchronization of the cores. Each core is attached to the bus through a memory interface. Essentially, the memory interface translates load/store requests into TLM transactions and ensures that the atomic instructions are handled correctly. More details on simulation of multi-core platforms will be provided in Subsection 3.1.7.

3.1.4.2 TLM-2.0 Bus

The TLM bus is responsible on routing transactions from an initiator, i. e. a bus master, to a target. Therefore, all target components are attached to the TLM bus at specific non-overlapping address ranges (cf. right side of Figure 3.1). The bus mapping will match the transaction address with the address ranges and dispatch the transaction accordingly to the matching target. Please note that in this process the bus performs a *global-to-local* address translation in the transaction. For example, assume that a sensor component is mapped to the address range (`start=0x50000000`, `end=0x50001000`) and the transaction address is `0x50000010`. This instructs the bus to 1) route the transaction to the sensor, and 2) translate the transaction address to `0x00000010` before passing it on to the sensor. Thus, the sensor works on its local address range without the need for knowing its position in the global address space.

Also, the TLM bus supports multiple masters initiating transactions. Currently, the CPU core as well as the DMA controller are configured as bus masters. Please note that a single component can be both master and target, as for example the DMA controller receives transactions initiated by the CPU core to configure source and destination address ranges, and is also able to initiate transactions by itself to perform memory access operations without the CPU core.

3.1.4.3 Traps and Interrupts

Both traps and interrupts result in the CPU core performing a context switch to the trap/interrupt handler (based on a SW configurable address stored in the *mtvec* CSR). Traps are raised to perform a system call or when an execution exception, e. g. invalid memory access, is encountered.

Two sources of interrupts are available: 1) local and 2) external. Essentially, there are two sources of local interrupts: SW as well as timer interrupts generated by the RISC-V specific CLINT (Core Local INTerruptor). The timer is part of

the CLINT and the interrupt frequency can be configured for each core through memory-mapped I/O. CLINT also provides a memory-mapped register for each core to trigger a SW interrupt for the corresponding core. External interrupts are all remaining interrupts triggered by the various components in the system. To handle external interrupts, it is neccessar provide the RISC-V specific PLIC (Platform Level Interrupt Controller). PLIC will collect and prioritize all external interrupts and then route them to each CPU core one by one. The core that claims the interrupt first will process it. CLINT and PLIC are shown on the upper right part of Figure 3.1. According to the RISC-V specification, external interrupts are processed with higher priority than local interrupts, and SW interrupts are higher prioritized than timer interrupts. The interrupt handling process will be described in more detail in Subsection 3.1.5.

3.1.4.4 System Calls

The C/C++ library defines a set of system calls as abstraction from the actual execution environment. For example, the `printf` function performs the formatting in platform independent C code and finally invokes the `write` system call with a fixed char array. Typically, an embedded system provides a trap handler that redirects the `write` system call to a UART/terminal component.

Also, a *SyscallHandler* component is provided to emulate system calls of the C/C++ library by re-directing them to the simulation host system. The emulation layer for example allows opening and reading/writing from/to files of the host system. This functionality is used for example to support SW coverage measurement with GCOV.

The syscall handler can be called in one of two ways: 1) Through a trap handler that re-directs the system call to the syscall handler from SW using memory-mapped I/O (this approach enables a flexible re-direction of selected system calls), or 2) directly intercept the system call (i. e. the RISC-V `ECALL` instruction) in the CPU core, instead of jumping to the trap handler. The behavior is configurable per core. The system call handling process is presented in more details in Subsection 3.1.5.

3.1.4.5 VP Initialization

The main function in the VP is responsible to instantiate, initialize and connect all components, i. e. to setup the architecture. An Executable and Linking Format (ELF) loader is provided to parse and load an executable RISC-V ELF file into the memory and setup the program counter in the CPU core accordingly. Finally, the SystemC simulation is started. The ELF file is produced by the GNU toolchain [88]

by (cross-)compiling the application program and optionally linking it with the C/C++ standard library or other RISC-V libraries (see left side of Figure 3.1). Also, and specially, the approach supports a bare-metal execution environment without any additional libraries and is tested to work with the FreeRTOS, Zephyr and RIOT operating systems.

3.1.4.6 Timing Model

The RISC-V VP provides a simple and configurable instruction-based timing model in the core and, by following the TLM-2.0 communication standard, transactions can be annotated with optional timing information to obtain a more accurate timing model of the executed software. It is also extended by a plugin interface that allows integrating external timing information into the core, with the option to integrating more accurate timing models like [35].

3.1.5 VP Interaction with SW and Environment

In this section, more details on the HW/SW interaction are provided; in particular on interrupt handling, and environment interaction via system calls in the RISC-V VP.

3.1.5.1 Interrupt Handling and HW/SW Interaction

In the following, an example application is shown that periodically accesses a sensor to demonstrate the interaction between hardware (VP-side) and software with a particular focus on interrupt handling. Firstly, the software application running on the VP is described, continuing over a minimal assembler bootstrap code to initialize interrupt handling, closing with a description on how interrupts are processed in more detail. Later in Subsection 3.1.8.1, the corresponding SystemC-based sensor implementation in the RISC-V VP is presented.

Software Side Listing 3.1 shows an example application that reads data from a sensor and copies data to a terminal component. The sensor and terminal are accessed through memory-mapped I/O. Their addresses are defined at the top of the program. They need to match with the configuration in the VP. The sensor periodically triggers an interrupt, denoting that new data is available. The main function starts by registering an interrupt handler for the sensor interrupt (Line 27). Again, the interrupt number specified in SW has to match the configuration in the VP. Next, the sensor is configured in Lines 29 to 30 using memory-mapped I/O. The `SCALER` denotes how fast sensor data is generated and the filter setting what kind

```
1  #include "stdint.h"
2  #include "irq.h"
3
4  static volatile char* const TERMINAL_ADDR = (char* const)0x20000000;
5  static volatile char* const SENSOR_INPUT_ADDR = (char* const)0x50000000;
6  static volatile uint32_t* const SENSOR_SCALER_REG_ADDR = (uint32_t*
   ↪ const)0x50000080;
7  static volatile uint32_t* const SENSOR_FILTER_REG_ADDR = (uint32_t*
   ↪ const)0x50000084;
8
9  bool has_sensor_data = 0;
10
11 void sensor_irq_handler() {
12   has_sensor_data = 1;
13 }
14
15 void dump_sensor_data() {
16   while (!has_sensor_data) {
17     asm volatile ("wfi");
18   }
19   has_sensor_data = 0;
20
21   for (int i=0; i<64; ++i) {
22     *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
23   }
24 }
25
26 int main() {
27   register_interrupt_handler(2, sensor_irq_handler);
28
29   *SENSOR_SCALER_REG_ADDR = 5;
30   *SENSOR_FILTER_REG_ADDR = 2;
31
32   for (int i=0; i<3; ++i)
33     dump_sensor_data();
34
35   return 0;
36 }
```

Listing 3.1: Example application running on the VP to demonstrate the HW/SW interaction.

of post-processing is performed on the data. Finally, the copy process is iterated for three times (Lines 32 to 33) before the program terminates. Each iteration starts by waiting for sensor data (Lines 16 to 18). The global boolean flag `has_sensor_data` is used for synchronization. It is set in the interrupt handler (Line 12) and unset again immediately after the waiting loop (Line 19). The *wfi* instruction is a hint to the CPU core to wait until the next interrupt occurs.

Bootstrap Code and Interrupt Handling Listing 3.2 shows the essential parts of a bare-metal bootstrap code, which is written in assembler and linked with the application code, to handle interrupts. Support for integration with the C/C++ library is also available, e. g. by executing the instructions at the beginning of the main function or integrating them directly into the *crt0.S* file, which is the entry

```
1  .globl _start
2  .globl main
3  .globl level_1_interrupt_handler
4
5  _start:
6  la t0, level_0_interrupt_handler
7  csrw mtvec, t0       # register interrupt/trap handler
8  csrsi mstatus, 0x8   # enable interrupts in general
9  li t1, 0x888
10 csrw mie, t1         # enable external/timer/\gls{sw} interrupts
11 jal main
12
13 # stop simulation with the *exit* system call
14 li a7, 93            # syscall exit has number 93
15 li a0, 0             # argument to exit
16 ecall                # RISC-V system call
17
18 level_0_interrupt_handler:
19 # ... store registers on the stack if necessary ...
20 csrr a0, mcause
21 jal level_1_interrupt_handler
22 # ... re-store registers in case they have been saved ...
23 mret                 # return from interrupt/trap handler
```

Listing 3.2: Bare-metal bootstrap code demonstrating interrupt handling

point of the C library and similarly to the bare-metal code also calls the main function after performing some basic initialization tasks. The `_start` label is the entry point of the whole program. The registers *mtvec*, *mstatus*, *mie* and *mcause* are CSRs that essentially store the interrupt handler address, core status information, enabled interrupts and interrupt source, respectively. The instructions `CSRR` and `CSRW` read and write a CSR into and from a normal CPU register, respectively. The instruction `CSRSI` sets the bits in the CSR based on the provided immediate value. Before the main function is called (Line 11), the interrupt handler base address (level-0) is stored in the *mtvec* register (Lines 6 to 7) and all interrupts are enabled (Lines 8 to 10). After the main function returns, the exit system call is invoked to terminate the VP simulation (Lines 14 to 16). More details on system calls can be found in the next subsection.

In general, an interrupt can occur at any time during execution of the application SW. All interrupts propagate to the PLIC (i. e. the RISC-V interrupt controller) first and are prioritized there. The CPU core only receives a notification that some interrupt is pending and needs to be processed. The core will prepare the interrupt by storing the program counter into the *mepc* CSR, setting the *mcause* CSR appropriately (to denote an interrupt in this case). The core then reads the base address from the *mtvec* CSR and sets the program counter to that address, i. e. effectively directly jumping to the level-0 interrupt handler (first instruction at Line 20). The interrupt handler (level-0), first in Line 20, reads the reason (i. e. local or external interrupt) for the interrupt into the *a0* CPU register, which according to

the RISC-V calling convention [18] stores the first argument of a function call. Then, in Line 21 an interrupt handler implemented in C is called (level-1, not shown in this example). Essentially, this level-1 handler deals with a local timer interrupt by resetting the timer with an external interrupt by asking the IC for the actual interrupt number with the currently highest priority (through a memory-mapped register access) and then calls the application provided interrupt handler function (Lines 11 to 13 in Listing 3.1, this step is ignored if none has been registered for the interrupt number). Finally, the *mret* instruction restores the previous program counter from the *mepc* CSR. Please note, that the level-0 handler typically stores and re-stores the register values by pushing and popping them to/from the stack before/after calling the level-1 handler, respectively.

```
1  #define SYS_write 64
2
3  ssize_t write(int fd, const void *buf, size_t count) {
4    return syscall(SYS_write, fd, (long)buf, count, 0);
5  }
6
7  long syscall(long n, long _a0, long _a1, long _a2, long _a3) {
8    // store arguments in CPU register and trigger ecall
9    register long a0 asm("a0") = _a0;
10   register long a1 asm("a1") = _a1;
11   register long a2 asm("a2") = _a2;
12   register long a3 asm("a3") = _a3;
13   register long a7 asm("a7") = n;
14
15   // special RISC-V instruction denoting a system call
16   asm volatile ("ecall" : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3), "r"(a7));
17
18   // store potential error code and return result
19   if (a0 < 0) {
20     errno = -a0;
21     return -1;
22   } else {
23     return a0;
24   }
25 }
```

Listing 3.3: System call handling stub linked with the C library (guest side, executed on the VP host system). This example listing is based on the RISC-V *newlib* port available at https://github.com/riscv/riscv-newlib.

```
#define SYS_write 64

// execute syscall on the host system (SyscallHandler)
ssize_t sys_write(int fd, const void *buf, size_t count) {
    const void *p = (const void *)guest_to_host_pointer(buf);
    return write(fd, p, count);
}

long execute_syscall(long n, long _a0, long _a1, long _a2, long _a3) {
    switch (n) {
        case SYS_write:
            return sys_write(_a0, (const void *)_a1, _a2);
        //...
    }
}

// function inside the CPU core
void execute_step() {
  auto instr = mem_if->load_instr(program_counter);
  auto op = decode(instr);

  switch (op) {
    case Opcode::ECALL: {
      if (intercept_syscalls_option) {
        // intercept and redirect syscall to host system
        regs[a0] = execute_syscall(regs[a7], regs[a0], regs[a1], regs[a2],
         ↪ regs[a3]);
      } else {
        // jump to SW trap handler (let SW decide how to handle syscall)
        // SW will either direct the syscall to a peripheral or the
        // SyscallHandler component (which is what the core does
        // directly when intercepting syscalls)
        raise_trap(EXC_ECALL);
      }
    } break;
    //...
  }
}
```

Listing 3.4: Concept on system call execution on the RISC-V VP, either redirect to the host system or take trap.

3.1.5.2 Environment Interaction: Syscall Emulation and C/C++ Library

Syscalls are a way of user-code to access certain functions of an underlying OS. While bare-metal applications do not have such systems, a user might want to use the host's functions for convenience.

Thus, the simplest configuration of the RISC-V VP is able to provide an emulation layer for executing system calls by redirecting them to the host system running the VP simulation. This requires passing arguments from the guest application into the host system and integrate the return values back into the guest application (i. e. memory of the VP). Implementing syscalls enables an easy support for the C/C++ standard library for testing purposes. Furthermore, one can directly use GCOV to track the coverage of the applications simulated on the RISC-V VP, as the GCOV instrumentation requires syscall support to open and write to files.

For example, consider the `printf` function provided by the C standard library: Most of its functionality is implemented as portable C code independently of the execution environment. Essentially, the `printf` function will apply all formatting rules and create a simple char buffer, which is then passed to the `write` system call. At this point, interaction with the execution environment is required. Listing 3.3 shows the relevant part of a stub that is provided in the RISC-V port of the C library. Essentially, the arguments of the system call are stored in the CPU registers *a0* to *a3* and the syscall number in *a7*. Then, the `ecall` instruction is executed. The VP simulator will detect the `ecall` instruction and directly executes the syscall on the host system as shown in Listing 3.4, if configured to do so. Note that it is also possible to execute this by using a trap handler, similar to the interrupt handler described in the previous section. Essentially, it would jump to the level-0 interrupt handler with the *mcause* CSR being set to the syscall identifier, and then redirect the write call to e. g. a UART/terminal component.

In case of the `write` syscall, a pointer argument `buf` is passed. This is a pointer value from the guest system, i. e. an index in the RISC-V VP's memory array `mem`, and has to be translated to a host memory pointer in order to execute the `write` syscall on the host system. Therefore, the `guest_to_host_pointer` function (Line 5) adds the base address of the VP byte memory array, i. e. `mem + buf`. The result of the syscall is stored in the *a0* register and passed back to the C library.

In general, the guest and host system might have different architectures with different word sizes, e. g. in one case the guest system (which is simulated in the VP) can be a 32 bit and the host system (which runs the VP) a 64 bit system. Therefore, one has to be careful when data is passed between the guest and the host. Primitive types, e. g. `int` and `bool`, can be passed directly from the guest to the host, because the host system running the VP uses data types with equal or

larger sizes, thus no information is lost when passing the arguments. When passing values back from the host a check can be performed, if necessary, to ensure that no relevant information is truncated, e. g. due to casting a 64 bit value into a 32 bit one. Pointer arguments need to be translated to host addresses, as described above, before accessing them on the host system, and also to be aligned to the host's word size. A write-access can be thus directly propagated back to the guest application. Structs can be accessed and copied recursively, considering the rules for accessing primitive and pointer types.

Other syscalls are implemented similarly to the `write` syscall, enabling a basic support for non-bare metal applications to run and access the host's ressources.

3.1.6 VP Performance Optimizations

In this section, two performance optimizations for the RISC-V VP are discussed that result in significant simulation speed-ups. The first optimization is a direct memory interface to fetch instructions and perform load/store operations from/to the (main) memory more efficiently. The second is a temporal decoupling technique with local time quanta to reduce the number of costly context switches, especially, in the CPU core simulation. Both techniques are described in the following.

3.1.6.1 Direct Memory Interface (DMI)

The CPU core translates every load and store operation into a transaction which is routed through the bus to the target. Most of the time the main memory is the target of the access, and always accessing the memory through a bus transaction is very costly. Even more so, because fetching the next instruction requires to load it from the memory too. Thus, at least one bus transaction is executed for every instruction. To optimize the access of the main memory, and in particular instruction fetching, using proxy classes with a Direct Memory Interface (DMI) is beneficial. The DMI stores the address offset where the memory is mapped in the overall address space as well as the size and pointer to the start of the memory. The RISC-V VP offers one proxy class for fetching instructions and one to access the memory in general, i. e. to perform load/store byte/half/word instructions. With the proxy classes enabled, the CPU core will first query the proxy class. It will match in case the main memory is accessed (for the *instruction* proxy class, only fetching instructions from main memory is allowed) and otherwise convert the access into a transaction and normally route it through the bus. Using this, of course, requires a consideration to make: Using the DMIs will significantly

improve the simulation performance (cf. Subsection 3.1.9.2) but also decrease timing accuracy.

3.1.6.2 Local Time Quanta

A SystemC-based simulation is orchestrated by the SystemC simulation kernel that switches execution between the various threads (cf. Section 2.2) While this is not a performance problem for most components, since they become runnable on very specific events, context switching can become a major bottleneck in simulating the CPU core. The reason is that a direct implementation will perform a context switch after executing every instruction, because simulation time has passed and the SystemC kernel needs to check for other runnable threads to perform synchronization. However, most of the time no other thread is runnable and the CPU thread is resumed again. Even if some other thread would become runnable it is still fine to keep running the CPU thread for some time (ahead of the global simulation time of the system). For example, even if the sensor thread would be runnable and trigger an interrupt once executed, delaying the sensor thread execution for a small amount of time and keeping the CPU thread running usually does not have an influence on the functional behavior of the system. Generally, the software does not have knowledge of the exact timing behavior and thus is written in such a way, e. g. by employing locks and flags, to always wait for certain conditions.

3.1.7 Simulation of Multi-Core Platforms

The RISC-V VP also provides support for simulation of RISC-V multi-core platforms. It supports instantiating and mixing multiple 32 and 64 bit cores. Each core is attached to the bus using a local TLM memory interface. Furthermore, each core is assigned a unique identifier, starting with zero, during instantiation. This core identifier is denoted as *hart-id*, according to the specification [17]. Access to the hart-id is provided through the read-only, SW accessible, *mhartid* CSR. Based on the hart-id, the SW can decide the behavior for each core. Furthermore, the SW can use atomic instructions for synchronization.

In the following, an example RISC-V multi-core SW is given for illustration and more details on the implementation of the RISC-V atomic ISA extension are given.

3.1.7.1 Example Bare-Metal Multi-Core SW

Listing 3.5 shows an example bare-metal SW that demonstrates the multi-core simulation concept from the SW side. For illustration purposes, it is defined here that the platform consists of two cores. Both cores start at the same time and use

the same entry-point (Line 5, i. e. the `_start` label). By reading the *mhartid* CSR, the SW obtains the id, either `0` or `1`, of the executing core (Line 6). The core id is stored in register *a0*. Based on the id the SW control flow is manipulated. Each core is initializing its stack pointer (*sp* register) address to a separate area (Line 13 and Line 16). Finally, an external main function is called (Line 20) with the core id passed as first argument (in conformation to the RISC-V calling convention that defines the first argument to be provided in register *a0*). The code after the main function ensures that only the second core can proceed to the exit system call (Lines 32 to 34) and stop the simulation for demonstration purposes. The first core that returns from the main function keeps spinning in the loop at Line 29. This synchronization mechanism is achieved by using an `AMO` instruction to read and increment a shared counter.

```
1  .globl _start
2  .globl core_main
3
4  # NOTE: each core will start here with execution
5  _start:
6  csrr a0, mhartid   # return a core specific number 0 or 1
7  li t0, 0
8  beq a0, t0, core0
9  li t0, 1
10 beq a0, t0, core1
11 # initialize stack for core 0 and core 1
12 core0:
13 la sp, stack0_end  # code executed only by core 0
14 j end
15 core1:
16 la sp, stack1_end  # code executed only by core 1
17 end:
18
19 # function argument stored in register a0 (according to RISC-V calling convention)
20 jal core_main
21
22 # wait until all two cores have finished
23 la t0, exit_counter
24 li t1, 1
25 li t2, 1
26 amoadd.w a0, t1, 0(t0) # get current counter value and increase existing value
27 # the first core reaching this point will spin
28 1:
29 blt a0, t2, 1b # jump one label backwards in case a0 < t2
30
31 # stop whole simulation with the *exit* system call
32 li a7, 93          # syscall exit has number 93
33 li a0, 0           # argument to exit
34 ecall              # RISC-V system call
35
36 stack0_begin:
37 .zero 32768 # allocate 32768 zero-initialized bytes in memory
38 stack0_end:
39 stack1_begin:
40 .zero 32768
41 stack1_end:
42 exit_counter:
43 .word 0 # allocate 4 zero-initialized bytes in memory
```

Listing 3.5: Bare-metal bootstrap code for a multi-core simulation with two cores.

3.1.7.2 Implementation of the Atomic ISA Extension

Listing 3.6 shows the relevant part of the memory interface that demonstrates the implementation for atomic instructions. Please note that each core has its own separate memory interface. As already discussed in the preliminaries (Subsection 3.1.3.1), the `A` instruction set extension provides two types of instructions: 1) `amo`, and 2) `lr` / `sc`. In the following, more details are given on how to implement these instructions, while referring, for illustration purposes, to Listing 3.6.

AMO Instructions To execute an `amo` instruction, the core has to perform a load (Line 1) and store (Line 6) operation atomically without intervening memory access operations of other cores. A simple way to ensure the atomic execution property is to lock the bus access during `amo` instructions. Therefore, a shared lock is acquired by the core's memory interface before a load operation (Line 2) and released again after the store (Line 52) operation. In case the bus is already locked, the lock operation will wait until the lock is released. Before performing a memory access operation, each core waits until it has obtained access rights (Line 35), i. e. the bus is not locked by other cores or by the active core itself. This locking scheme also supports DMI operations (Lines 39 to 45).

Peripherals that have write access to the bus (e. g. the DMA controller) are attached through a 1-to-1 TLM interconnect to the bus in order to ensure that they respect the bus locking. The peripheral interconnect transparently forwards all peripheral write transactions, but waits in case the bus is locked by any core. Once the bus lock is released, all waiting (SystemC) processes are notified using a (SystemC) event.

LR / SC Instructions To execute an `lr` instruction, the core (memory interface) tracks a reservation on the load address and acquires the shared bus lock (Lines 12 to 13). The lock is kept acquired while *"forward-progress"* (see preliminaries Subsection 3.1.3.1) is maintained by the core. Essentially, the lock is released in case:

- More than 16 instructions are executed (number chosen arbitrarily).
- A trap (exception) or interrupt is taken.
- A store is performed by the core holding the lock.

The `sc` instruction succeeds in case the lock is still acquired (Line 19) and a reservation on the store address exists (Line 20). In any case, the bus lock is released by the memory interface after executing the `sc` instruction (Line 24 or Line 52).

```
int32_t atomic_load_word(uint64_t addr) {
    bus_lock->lock(get_hart_id());
    return load_word(addr);
}

void atomic_store_word(uint64_t addr, uint32_t value) {
    assert (bus_lock->is_locked(get_hart_id()));
    store_word(addr, value);
}

int32_t atomic_load_reserved_word(uint64_t addr) {
    bus_lock->lock(get_hart_id());
    lr_addr = addr; // reservation for the load address on the whole memory
    return load_word(addr);
}

bool atomic_store_conditional_word(uint64_t addr, uint32_t value) {
     /* The lock is established by the LR instruction and the lock is kept while
      ↪ "forward-progress" is maintained. */
    if (bus_lock->is_locked(get_hart_id())) {
        if (addr == lr_addr) {
            store_word(addr, value);
            return true; // SC succeeded
        }
        bus_lock->unlock();
    }
    return false; // SC failed
}
void store_word(uint64_t addr, uint32_t value) {
    store_data(addr, value);
}

template <typename T>
inline void store_data(uint64_t addr, T value) {
    // only proceed if the bus is not locked at all or is locked by this core
    bus_lock->wait_for_access_rights(get_hart_id());

    // check if this access falls within any DMI range
    bool done = false;
    for (auto &e : dmi_ranges) {
        if (e.contains(addr)) {
            quantum_keeper.inc(e.dmi_access_delay);
            *(e.get_mem_ptr_to_global_addr<T>(addr)) = value;
            done = true;
        }
    }

  // otherwise (no DMI), perform a normal transaction routed through the bus
    if (!done)
        do_transaction(tlm::TLM_WRITE_COMMAND, addr, (uint8_t *)&value, sizeof(T));

    // do nothing in case the bus is not locked by this hart
    bus_lock->unlock(get_hart_id());
}

// ... other load/store functions and load_data similar ...
```

Listing 3.6: Core memory interface with atomic operation support.

Similar to the `amo` instructions, the shared bus lock ensures that other cores (and peripheral bus masters) do not interfere with the `lr` / `sc` instruction sequence execution and hence the `lr` / `sc` sequence eventually succeeds when maintaining *"forward-progress"*.

```
1  struct SimpleSensor : public sc_core::sc_module {
2    tlm_utils::simple_target_socket<SimpleSensor> tsock;
3
4    interrupt_controller *ic = 0;
5    uint32_t irq_number = 0;
6    sc_core::sc_event run_event;
7
8    // memory-mapped data frame
9    std::array<uint8_t, 64> data_frame;
10
11   // memory-mapped configuration registers
12   uint32_t scaler = 25;
13   uint32_t filter = 0;
14   std::unordered_map<uint64_t, uint32_t *> addr_to_reg;
15
16   enum {
17     SCALER_REG_ADDR = 0x80,
18     FILTER_REG_ADDR = 0x84,
19   };
20
21   SC_HAS_PROCESS(SimpleSensor);
22
23   SimpleSensor(sc_core::sc_module_name, uint32_t irq_number)
24       : irq_number(irq_number) {
25     tsock.register_b_transport(this, &SimpleSensor::transport);
26     SC_THREAD(run);
27
28     addr_to_reg = {
29       {SCALER_REG_ADDR, &scaler},
30       {FILTER_REG_ADDR, &filter},
31     };
32   }
33
34   void transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &delay) {
35     auto addr = trans.get_address();
36     auto cmd  = trans.get_command();
37     auto len  = trans.get_data_length();
38     auto ptr  = trans.get_data_ptr();
39
40     if (addr >= 0 && addr <= 63) {
41       // access data frame
42       assert (cmd == tlm::TLM_READ_COMMAND);
43       assert ((addr + len) <= data_frame.size());
44
45       // return last generated random data at requested address
46       memcpy(ptr, &data_frame[addr], len);
47     } else {
48       assert (len == 4);  // NOTE: only allow to read/write whole register
49
50       auto it = addr_to_reg.find(addr);
51       assert (it != addr_to_reg.end()); // access to non-mapped address
52
53       // trigger pre read/write actions
54       if ((cmd == tlm::TLM_WRITE_COMMAND) && (addr == SCALER_REG_ADDR)) {
55         uint32_t value = *((uint32_t *)ptr);
56         if (value < 1 || value > 100)
57           return; // ignore invalid values
58       }
```

```
59
60        // actual read/write
61        if (cmd == tlm::TLM_READ_COMMAND) {
62          *((uint32_t *)ptr) = *it->second;
63        } else if (cmd == tlm::TLM_WRITE_COMMAND) {
64          *it->second = *((uint32_t *)ptr);
65        } else {
66          assert (false && "unsupported tlm command for sensor access");
67        }
68
69        // trigger post read/write actions
70        if ((cmd == tlm::TLM_WRITE_COMMAND) && (addr == SCALER_REG_ADDR)) {
71          run_event.cancel();
72          run_event.notify(sc_core::sc_time(scaler, sc_core::SC_MS));
73        }
74      }
75    }
76    void run() {
77      while (true) {
78        run_event.notify(sc_core::sc_time(scaler, sc_core::SC_MS));
79        sc_core::wait(run_event);  // 40 times per second by default
80
81        // fill with random data
82        for (auto &n : data_frame) {
83          if (filter == 1) {
84            n = rand() % 10 + 48;
85          } else if (filter == 2) {
86            n = rand() % 26 + 65;
87          } else {
88            // fallback for all other filter values: random printable
89            n = rand() % 92 + 32;
90          }
91        }
92
93        ic->trigger_interrupt(irq_number);
94      }
95    }
96  };
```

Listing 3.7: SystemC-based configurable sensor model that is periodically filled with random data - demonstrates the basic principles on modeling peripherals.

3.1.8 VP Extension and Configuration

The RISC-V VP is designed as a configurable and in particular extensible platform. It is very easy to add additional components (i. e. peripherals/controllers including bus masters) and attach them to the bus system at a new address range, or change the address mapping of the existing components. To provide an initial demonstration, different 32 and 64 bit cores are provided which can also be combined into a multi-core platform. This allows for an easy (re-)configuration of the VP. By following the TLM-2.0 communication standard, transactions can be annotated with optional timing information to obtain a more accurate timing model of the executed software. Support for additional RISC-V ISA or even custom extensions (beyond `IMACFD`) can also be added inside the CPU core by extending the decode-

and execute functions accordingly. The RISC-V VP already provides a large feature set with large potential for additional extensions, which made the RISC-V VP suitable as foundation for different application areas, as can be demonstrated in [32, 34, 35, 39, 40, 89, 90] and many more.

In the following, the extensibility and configurability of the RISC-V VP is demonstrated by three concrete examples: addition of a sensor peripheral, extension of a GDB-based SW debug functionality and configuration matching the RISC-V HiFive1 board from SiFive.

3.1.8.1 Extending the VP with a Sensor Peripheral

This subsection presents the SystemC-based implementation of the VP sensor peripheral, which is used by the SW example presented in Subsection 3.1.5.1. It shows the principles on modeling peripherals and extending the RISC-V VP as well as demonstrates the TLM communication and basic SystemC-based modeling and synchronization. The sensor is instantiated in the main function of the VP alongside other components and is attached to the TLM bus.

The sensor implementation is shown in Listing 3.7. The sensor model has a data frame of 64 bytes that is periodically updated (overwritten with new data, Lines 82 to 91) and two 32 bit configuration registers `scaler` and `filter`. The update happens in the run thread (the run function is registered as SystemC thread inside the constructor in Line 26). Based on the `scaler` register value, this thread is periodically unblocked (Line 78) by calling the `notify()` function on the internal SystemC synchronization event. Thus, `scaler` defines the speed at which new sensor data is generated. The `filter` register allows to select some kind of post-processing on the data. After every update an interrupt is triggered, which will propagate through the interrupt controller to the CPU core up to the interrupt handler in the application SW. Therefore, the sensor has a reference to the interrupt controller (`ic`, Line 4) and an interrupt number provided during initialization (Line 23 and Line 24).

Access to the data frame and configuration registers is provided through TLM transactions. These transactions are routed by the bus to the `transport` function (Line 34). The routing happens as follows: 1) The sensor has a TLM target socket field, which is bound in the main function (i. e. VP simulation entry point) to an initiator socket of the TLM bus. 2) The `transport` function is bound as destination for the target socket in the constructor (Line 25).

Based on the address and operation mode, as stored in the generic payload (Lines 35 to 36), the action is selected. It will either read (part of) the data frame (Line 46) or read/write one of the configuration registers (Lines 61 to 67). In case of a register access, a pre-read/write validation and post-read/write action can be

defined as necessary. In this example, the sensor will ignore invalid `scaler` values (Lines 54 to 58) and reset the data generation thread on a scaler update (Lines 70 to 73). Please note, that the transaction object (generic payload) is passed by reference and provides a pointer to the data, thus a write access is propagated back to the initiator of the transaction as is defined in the TLM standard (cf. Section 2.2). Optionally, an additional delay can be added to the `sc_time` delay parameter (also passed by reference) for a more accurate timing model.

3.1.8.2 SW Debugging Support Extension

In this subsection, the RISC-V VP extension to provide SW debug capabilities in combination with (for example) the Eclipse Integrated Development Environment (IDE) by implementing the GDB Remote Serial Protocol (RSP) interface is described. Debugging for example enables to step through the SW line by line, set (conditional) breakpoints, obtain and even modify variable values and also display the RISC-V disassembly (with the ability to step through the disassembly). Debugging can also be extremely helpful on the VP level to investigate errors, due to the deterministic and reproducible SW execution on the VP.

Using the GDB RSP interface, the RISC-V VP acts as server and the GDB[1] as client. They communicate through a TCP connection and send text based messages. A message is either a packet or a notification (a simple single char `+`) that a packet has been successfully processed. Each packet starts with a `$` char and ends with a `#` char followed by a two digit hex checksum (the sum over the content chars modulo 256). For example the packet `$m111c4,4#f7` has the content `m111c4,4` and checksum `f7`. The `m` command denotes a memory read, in this case "read `0x4` bytes starting from address `0x111c4`". The server might then, for example, return `+$05000000#85`, i. e. acknowledge the packet (`+`) and return the value `5` (two chars per byte with little endian byte order). To handle the packet processing and TCP communication, a gdb-stub component is provoded for the RISC-V VP. The whole debugging extension is only about 500 additional lines of C++ code, with most of them to implement the gdb-stub. On the VP side, only the CPU core has been modified to lift the SystemC thread into the gdb-stub, to allow the CPU to interrupt and exit the execution loop in case of a breakpoint and thus effectively transfer execution control to the gdb-stub.

Debugging works as follows: Start the RISC-V VP in *debug-mode* (command line argument), this will transfer control to the gdb-stub implementing the RSP interface, waiting for a connection from the GDB debugger. In another terminal, start the GDB debugger. Load the same executable ELF file into the GDB (command

[1] In particular, the freely available RISC-V port of GDB, which knows about the available RISC-V register set, the CSRs, and can provide a disassembly of the RISC-V instruction set.

`file main-elf`) as in the RISC-V VP. Connect to the TCP server of the VP (by typing `target remote :5005` to connect to localhost using port 5005). Now the GDB debugger can be used as usual to set breakpoints, continue and step through the execution. It is also possible to directly use a visual debugging interface, e. g. *ddd* or *gdb-dashboard* or even the *Eclipse* IDE.

Please note, the ELF file contains information about the addresses and sizes of the various variables in memory. Thus, a `print(x)` command with an integer variable `x` is already translated into a memory read command (e. g. `m11080,4`). Therefore, on the server side, i. e. the RISC-V VP, an extensive parsing of ELF files is not necessary to add comprehensive debugging support. In total, only 24 different GDB commands needed to be implemented, of which 9 can simply return an empty packet and a few more to return some predefined answer. Relevant packets are, especially: read one register (`p`), read all registers (`g`), read a memory range (`m`), set/remove breakpoints (`Z0`/`z0`), step (`s`) and continue (`c`) through execution.

3.1.8.3 HiFive1 Board Configuration

As a demonstration, in the following will be a configuration presented that matches the HiFive1 board from SiFive [91]. Besides the basic configuration which is described in Subsection 3.1.8.3, the demonstration introduces also a virtual environment for the RISC-V VP. This will be introduced in this section, but is explained in full detail in Section 3.2. This virtual environment GUI allows the SW developer to create and test SW as if they would have a real physical HiFive1 board, including buttons, LEDs etc. since the environment is also simulated. The core idea here is to build a server-based environment interface (detailed in Subsection 3.1.8.3) which provides the GPIOs section to the outside world, and where a client, in this case a Qt-application, can connect. This Qt-application mimics the SoC's environment, i. e. for instance pressing buttons and displaying numbers on a seven segment display, and is described in Subsection 3.1.8.3.

Basic Configuration The HiFive1 [92] is based around the FE310 SoC [93] that integrates a single RISC-V `RV32IMAC` core with several peripherals and memories. Interrupts are processed by the CLINT and PLIC peripherals following the RISC-V ISA specification. The PLIC supports 53 interrupt sources with 7 priority levels. The RISC-V VP has a configuration, including the corresponding PLIC peripheral, to match this specification. A non-volatile flash memory is provided to store the application code and a small writable data memory to hold the application data. The application data is initialized during the boot process by copying relevant data (e. g. initialized global variables) from the flash memory into the data memory. This initialization code is embedded in the application binary that is placed in the

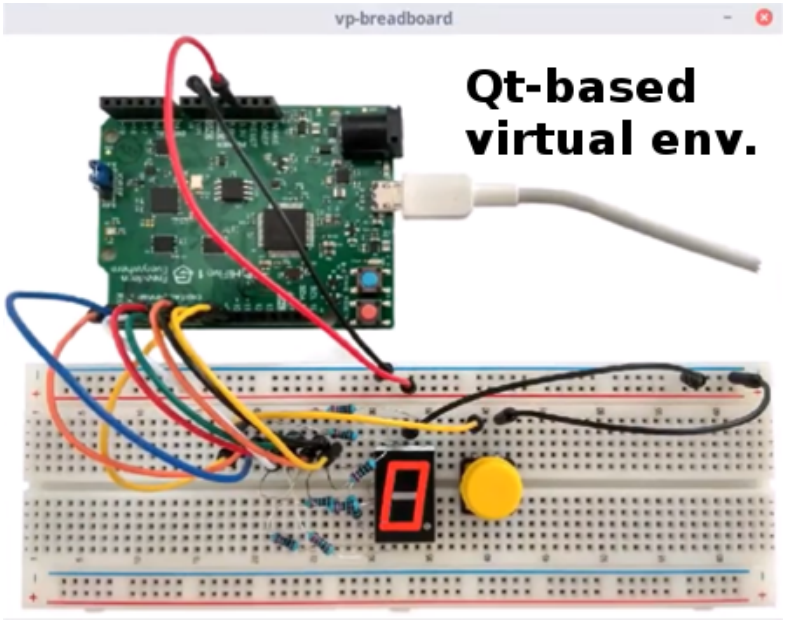


Figure 3.2: Qt-based virtual environment, showing the HiFive1 board with a seven segment display (output) and a button (input), attached to the VP simulation through a TCP connection.

flash memory. GPIOs and an UART are provided for communicating with the environment. Each write accesses to the UART is redirected to the standard output of the host system (usually the console), while the standard input is read and fed into the corresponding registers of the UART peripheral. To speed up the design process, the proposed register modeling layer is used to create these additional peripheral models in SystemC. For the GPIO peripheral, a (re-usable) interface is provided to access an environment model. An introduction on this interface is given in the following, with a full in-depth demonstration in Section 3.2.

GPIO Server Environment Interface As introduced in Section 2.1, GPIO-peripherals are used to connect the embedded system to the outside world. Each GPIO pin can be configured to serve as output or input connection. Input pins can trigger an interrupt when being written.

A GPIO-(TCP)-server is integrated into the FE310-compatible GPIO-interface to provide access to the GPIO pins in order to attach an environment model. The server runs in a separate system thread and hence needs to be synchronized with the SystemC kernel. Therefore, the SystemC `async_request_update` function is used to register an update function from the "outside timing" to be executed on the next update cycle of the SystemC thread. The update function then triggers a usual SystemC event notification to wake up the SystemC GPIO processing thread.

The virtual environment GUI, which is the client-side of the presented GPIO-server, is described in the next section.

Virtual Environment GUI Figure 3.2 shows an example virtual environment implemented in C++ using the Qt framework. To make it more intuitive for the user in the first versions of the GUI, a photo from a "real" HiFive1 board is included, where a seven segment display and a button are attached to the GPIO pins. The virtualization of this environment GUI was done by implementing the functionality to press the button via mouse input (see as Figure 3.2) as well as to show numbers on the seven segment display in Qt, and to perform the corresponding communication with the RISC-V VP via the GPIO server environment interface (cf. previous section).

In a concrete example SW, the display shows the current counter value and the button is used to control the count mode (switch between increment and decrement). The main benefit then is to execute the exact same RISC-V ELF binary on the real HiFive1 board using the same setup as shown in the virtual environment. The functional behavior of the SW on the real board was identical in comparison to the VP. For a more detailed description, please refer to Section 3.2.

3.1.9 VP Evaluation

This section first describes how the RISC-V VP was tested to evaluate and ensure the VP quality, and then presents results of a performance evaluation.

3.1.9.1 Testing

Testing is very important to ensure that the VP is working correctly. The following list describes how the RISC-V VP was tested. In particular, it was used:

1. The official RISC-V ISA tests [94], in particular the `RV32/64IMACFD` tests.
2. The RISC-V Torture test-case generator [95], also targeting the `RV32/64 IMACFD` instruction set.
3. State-of-the-art Coverage Guided Fuzzing (CGF) techniques for test-case generation. For more information on this approach please refer to [96].
4. Several example applications, ranging from bare-metal applications up to examples using the FreeRTOS [68] and Zephyr [69] operating systems. This is tested using observations if example applications behave as expected.

In contrast to the other tests, the RISC-V ISA tests are directed, meannig that are hand-written and already encode the expected result inside the test. Hence, no reference simulation is required for the ISA tests.

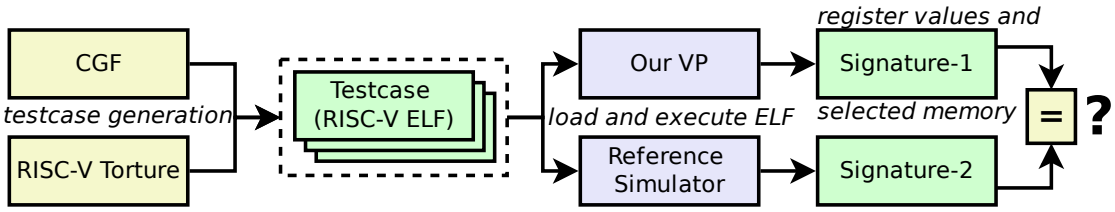


Figure 3.3: Overview on the RISC-V Torture and CGF approach for VP testing.

Figure 3.3 shows an overview for the Torture and CGF approach. They both work by generating a set of tests, i. e. each test-case is a RISC-V ELF binary, which is then executed one after another on the RISC-V VP and (one or multiple) reference simulators. In this evaluation, the official RISC-V reference simulator SPIKE [65] is used. The RISC-V VP was extended to dump the execution result, called *signature*, to be compared with SPIKE. The *signature* contains the register values and selected memory contents. After each execution the dumped signatures are compared for differences. Hence, the reference simulator is considered as a black box and no intrusive modifications are required for this testing.

The RISC-V ISA tests, as well as the torture and fuzzing test generation approaches, primarily focus on testing the execution core. The example applications, on the other hand, also tend to use larger portions of the whole VP platform, specifically the peripheral devices. They further integrate the CLINT and PLIC interrupt controller as well as selected peripherals that are required for the particular application. In the following, more details are provided on some selected applications that demonstrate the applicability of the RISC-V VP for real-world embedded applications.

Zephyr Examples The Zephyr OS is designed around a small-footprint kernel primarily targeting resource constrained embedded systems. Zephyr supports multiple architectures, including RISC-V. Specifically for these tests, support for the HiFive1 board is available for the RISC-V architecture. The HiFive1 board configuration of the RISC-V VP was used to run several Zephyr example applications that extensively use core components of the kernel; including threads, timers, semaphores and message queues. In addition, examples applications are tested that perform *aes128* and *sha256* encryption/decryption schemes using the TinyCrypt library.

FreeRTOS Examples Similar to Zephyr, FreeRTOS is also designed around a kernel component, targets embedded systems and provides support for the RISC-V architecture. Several example applications were built for this evaluation, creating

multiple threads with different priorities, using queues for data passing, with integrated interrupts. In addition, two applications were created that use the File Allocation Table Filesystem (FAT) and User Datagram Protocol (UDP) library extensions of FreeRTOS. The first application formats an SD card by creating a new FAT32 Master Boot Record and writing data to the new FAT partition. The second application sends/receives a set of UDP packets using an Ethernet peripheral to/from a remote computer.

3.1.9.2 Performance Evaluation

In the previous conference paper [72], it was already demonstrated: 1) that the RISC-V VP provides more than 1 000 times faster simulation performance compared to different RTL implementations, and 2) the effectiveness of the presented RISC-V VP simulation performance optimization techniques (between 6.1x and 7.8x improvement on the considered benchmark set). In this subsection, an updated performance evaluation of the RISC-V VP is presented on a more recent and faster simulation host. A set of different applications are used to demonstrate the execution performance, measured in Million Instructions Per Second (MIPS), of the RISC-V VP.

Furthermore, a performance comparison to other RISC-V simulators is provided, in particular: FORVIS, SAIL (the C simulator back-end in particular), gem5, SPIKE and QEMU (which have been introduced in the related work Subsection 3.1.2).

Experimental Setup and Results All experiments are performed on a Fedora 29 Linux system with an Intel Xeon Gold 5122 processor with 3.6 GHz. The memory limit is set to 32 GiB with a timeout of 4 hours (i. e. 14 400 seconds). The RISC-V VP has been compiled with GNU Compiler Collection (GCC) version 8.2.1 with enabled `-O3` optimization.

Table 3.1 shows the results: The first six columns show the benchmark name (column: *Benchmark*), number of executed instructions (column: *#instr*), Lines of Code (LoC) in C (column: *C*) and in assembly (column: *ASM*), the aforementioned MIPS (column: *MIPS*) performance metric and the simulation time (column: *time*) for the RISC-V VP. The remaining five columns show the simulation time for the other RISC-V simulators. All simulation times are reported in seconds. *M.O.* denotes a memory out and *N.S.* that the benchmark is not supported by the simulator.

Benchmarks from different application areas are considered:

- *dhrystone* is a synthetic computing benchmark designed to measure the

general (integer) execution performance of a CPU. 10 000 000 iterations of the algorithm are executed.

- *qsort* is the well known *quicksort* algorithm in a standard implementation to sort an array of 50 000 000 elements.
- *fibonacci* is a small program implemented in assembler that performs 1 000 000 000 iterations and ignores any integer overflow.
- *zephyr-crypto* uses two threads that communicate through a (single-element) message queue. The first thread encrypts ≈ 1 MB of data, using the *aes128* algorithm of the TinyCrypt library, while the second thread decrypts it again.
- *mc-ivadd* is a multi-core benchmark that performs a vector addition and stores the result in a new vector. Each core operates on a different part of the vector. The vector size is set to 4 194 304 and the algorithm performs 30 iterations.

The results are reported for the RISC-V VP with one `RV32` core (four `RV32` cores for the multi-core benchmark). However, similar results are obtained when using an `RV64` core(s).

It can be observed that the RISC-V VP provides a high simulation performance between 42 to 53 MIPS with an average of 46 MIPS on the benchmark set, which demonstrates the applicability of the RISC-V VP to real-world embedded applications. The pure assembler program (*fibonacci*) achieves the highest simulation performance, since it does not require performing memory access operations (besides instruction fetching). It can also be observed that the additional synchronization overhead (in the SystemC simulation) to perform a multi-core simulation has no significant performance impact, although the SystemC kernel is not using a multi-threaded simulation environment, but executes one process (i. e. one core) at a time and switches between the processes. Please note that for the multi-core simulation benchmark, the total MIPS for all four cores are reported.

Comparison with other Simulators Compared to the other RISC-V simulators (right side of Table 3.1), the RISC-V VP shows very reasonable performance results and is located in the front midfield. As expected, the RISC-V VP is not as fast as the high-speed simulators SPIKE and in particular QEMU. The reason for that can be found in the performance overhead of the SystemC simulation kernel and the more detailed simulation of the RISC-V VP. This includes more accurate timing by leveraging SystemC, instruction accurate interrupt handling and the ability to integrate TLM-2.0 memory transactions (cf. the trade-off between simulation time and accuracy in Section 2.2). Furthermore, compared to QEMU, the results do not integrate Dynamic Binary Translation (DBT) or Just-in-Time Compilation (JIT). On the other hand, the RISC-V VP is much faster than FORVIS and SAIL as well as

Table 3.1: Experiment results - all execution times reported in seconds, number of executed instructions (#instr) reported in Billions (B). MIPS = Millions Instructions Per Second. LoC = Lines of Code in C and assembly (ASM). M.O. = Memory Out (32GB limit). T.O. = Time Out (4h = 14400 seconds limit). N.S. = Not Supported.

Benchmark	#instr	LoC		VP		Other Simulators (Time)				
		C	ASM	MIPS	Time	FORVIS	SAIL	gem5	SPIKE	QEMU
dhrystone	4.06B	362	2 212	42.7	94.86	M.O.	10 986.24	1 170.08	13.90	3.53
qsort	2.93B	146	2 279	42.8	68.71	M.O.	T.O.	985.99	11.01	3.04
fibonacci	5.99B	/	17	53.9	111.15	13 954.57	12 830.05	1 447.37	17.19	1.94
zephyr-crypto	2.52B	86	5 407	46.6	54.11	N.S.	N.S.	N.S.	N.S.	6.50
mc-ivadd	2.39B	26	1 798	44.4	53.76	N.S.	N.S.	N.S.	48.11	N.S.

gem5, which arguably is the closest to the RISC-V VP in terms of the intended use-cases. Please note, the *zephyr-crypto* and *mc-ivadd* benchmarks are not supported (*N.S.*) on some simulators due to missing support for the Zephyr operating system and the multi-core RISC-V test environment, respectively. In the following, the results are discussed in more detail.

Compared to QEMU, the performance overhead is most strongly pronounced on the *fibonacci* benchmark. The reason can be found in that the benchmark iterates a single basic block and only performs very simple operations (mostly additions) without memory accesses. This has two implications: This single basic block is pre-compiled once into native code using DBT and then re-used for all subsequent iterations (hence QEMU executes at near native performance). Furthermore, since only simple operations are used, the simulation overhead induced by SystemC has a comparatively strong influence on the overall performance.

On more complex benchmarks such as *qsort* and *zephyr-crypto*, the overhead to QEMU is less strongly pronounced. These benchmarks have a much more complex control flow and hence require to compile several basic blocks and also use more complex instructions that take longer time for (native) execution, compared to a simple addition. Furthermore, *zephyr-crypto* performs several simulated context switches between the Zephyr OS and the worker threads, which has additional impact on the simulation performance of QEMU, since this causes an indirect and non-regular control flow transfer between the basic blocks.

Compared to SPIKE, the performance overhead is mostly uniformly distributed (since SPIKE does not use DBT either) on the single core benchmarks, with SPIKE being around 6.2x to 6.8x faster than the RISC-V VP. The performance on the multi-core benchmark *mc-ivadd* on the other hand is very similar for both SPIKE and the RISC-V VP. Apparently, SPIKE is very strongly optimized for simulation of single

core systems and hence the newly introduced synchronization and context switch overhead is very significant.

Compared to gem5, the RISC-V VP is significantly faster (between 12.3x to 14.4x). The primary use case of gem5 is also not a pure functional simulation (which is the goal of SPIKE and QEMU) but rather an architectural exploration and analysis. Thus, gem5 provides more detailed processor and memory models with complex interfaces which in principle can also be extended for accurate extra-functional properties like the RISC-V VP. Furthermore, gem5 is a large (and aims to be a rather generic) platform which supports different architectures besides RISC-V which causes additional performance overhead.

Compared to FORVIS and SAIL, the RISC-V VP is consistently much faster (up to two orders of magnitude). The reason is that these simulators have been designed with a different use-case in mind (establishing an executable formal representation of the RISC-V ISA) and hence very fast simulation performance is only a secondary goal. Furthermore, FORVIS runs into memory outs (*M.O.*), which may indicate a memory leak in the implementation. SAIL has a time-out (*T.O.*) on the *qsort* benchmark, while a 10x smaller version of *qsort* completed successfully within 700.22 seconds on SAIL. Hence, this time out may also indicate a memory related problem in SAIL since *qsort* requires a significant amount of memory for the large array to be sorted.

In summary, the simulation performance very strongly depends on the simulation technique, which in turn depends on the primary use-case of the simulator. The goal of the RISC-V VP is to introduce an open-source, extendable implementation that leverages SystemC TLM-2.0 into the RISC-V ecosystem to lay the foundation for advanced SystemC-based system-level use cases. Overall, the RISC-V VP provides a high simulation performance with an average of 46 MIPS in the shown benchmark set.

3.1.10 Discussion and Future Work

The RISC-V based VP is implemented in SystemC TLM-2.0 and already provides a significant set of features, which makes the RISC-V VP suitable as foundation for various application areas, including early SW development and analysis of interactions at the HW/SW interface of RISC-V based systems. Nonetheless, the RISC-V VP can still be further improved. In the following, different directions are sketched, that either were not shown in detail in this section, or could be considered for future work.

One direction in which progress was made is the extension of the RISC-V VP with new components and integration of additional RISC-V ISA extensions. Also not discussed in this section are the RISC-V floating-point extensions `F` (single-

precision) and `D` (double-precision) which were implemented in order to support the complete RISC-V common standard ISA [16], including an integration of Memory Management Unit (MMU) to support virtual memory layouts and memory protection. These layouts are described in the RISC-V specification as the *Sv32*, *Sv39* and *Sv48* virtual memory systems [17], that the MMU supports to match different application areas. Specifically, these extensions allow the RISC-V VP to run the Linux OS (see [49]).

For future work, further performance optimizations are also very interesting, in particular for running supervisor mode OSes. Two techniques seem very promising: 1) Integration of DBT/JIT techniques to avoid the costly interpreter loop whenever possible, e. g. translate and cache RISC-V basic blocks. This (dynamic) translation from RISC-V instructions to the simulation host instruction set should also preserve (for example) timing information of the SystemC simulation to avoid losing accuracy in the simulation timing model. 2) Use a real host computer thread for each core in a multi-core simulation. This requires dedicated techniques to synchronize with the SystemC simulation.

While extensive testing was already performed for the RISC-V VP, in particular the core, additional verification techniques could be considered, with a stronger emphasis on verification of peripherals and other IP components. One of these methods can be found in Section 4.1. Furthermore, it may be promising to consider formal verification techniques for SystemC, e. g. [97–99], and also investigate (UVM-based [100]) constrained random techniques for test-case generation [101].

3.1.11 Conclusion

In this section, the first RISC-V based VP was proposed and implemented to further expand the RISC-V ecosystem. The VP has been implemented in SystemC and designed as extensible and configurable platform around a RISC-V `RV32/64IMACFD` (multi-)core with a generic bus system employing TLM-2.0 communication. In addition to the RISC-V core(s), SW debug and coverage measurement capabilities are provided, along with a set of essential peripherals, including the RISC-V CLINT and PLIC interrupt controllers, support the FreeRTOS and Zephyr operating systems, and an example configuration matching the HiFive1 board from SiFive. The existing feature-set in combination with the extensibility and configurability of the RISC-V VP makes the RISC-V VP suitable as foundation for various application areas and system level use-cases, including early SW development and analysis of interactions at the HW/SW interface of RISC-V based embedded systems (e. g. found in [39, 40, 44, 46]). The evaluation demonstrated the quality and applicability to real-world embedded applications as well as the high simulation performance of the RISC-V VP. Finally, the RISC-V VP is fully open source to stimulate further research and development of ESL methodologies.

3.2 Virtual Breadboard - Advanced Environment Modeling GUI

This section includes and extends published material from the conference paper [39] and the subsequent journal extension [40].

It is started by a motivation to this topic in the following paragraph, continued by a discussion of related work (Subsection 3.2.2) and an outline of relevant background information. Next, the VP-driven environment modeling methodology is presented in more detail, including the communication interfaces and configuration features (Subsection 3.2.4). Following up, the rapid prototyping approach using the dynamic Lua scripting language is introduced in Subsection 3.2.5. Then, the modeling case-studies with two different environment configurations are described using the proposed architecture (Subsection 3.2.6.1). Afterwards, the results of the performance evaluation are discussed (Subsection 3.2.6.2). Finally, after an outlook on future work (Subsection 3.2.7), the proposed approach is concluded in Subsection 3.2.8.

3.2.1 Introduction

As stated earlier in Section 2.3, RISC-V [16, 17] is a modern ISA that gained significant momentum in the recent years. A key factor that drives the RISC-V success story is its free and open nature combined with a lightweight and modular architecture. Moreover, RISC-V is designed from the ground up to enable integration of custom instruction set extensions in order to build highly application specific solutions. These properties push the adoption of RISC-V and strengthen its potential to become a game changer in the IoT era. As such, great interest can be observed around RISC-V in industry and academia.

In line with the RISC-V popularity, the extensive RISC-V ecosystem is continuously growing to include a broad set of software and hardware development tools and library. As stated in Section 3.1, a key property of VPs is their binary compatibility with the hardware platform, i. e. from the software perspective the VP provides the same interface as the hardware platform and hence the software can be executed unmodified on the VP and hardware. Beside a functional validation, VPs also enable design space exploration by evaluating different design decisions early in the design flow.

The binary compatibility, however, is not enough for embedded systems: Program control flow usually also depends on values and behavior of input and output devices like sensors and actuators that may not be a part of the SoC. As these devices are often are a critical part of the embedded system as a whole, there

exists the need for design space exploration, testing and validation of the complete system as early as possible.

The RISC-V VP is a representative, advanced open source VP tailored for RISC-V and available at GitHub [49] and has been described in Section 3.1. It provides an extensive feature set, such as support for the 32 and 64 bit RISC-V ISA with all standard instruction set extensions, several operating systems (such as Zephyr and Linux), advanced debugging capabilities and configurations to create different platforms such as the HiFive1 board from SiFive [92]. The main benefit of the RISC-V VP is, however, its ease of expandability: from custom RISC-V instructions with dynamic data-flow analysis extensions [45], over HW-intrinsic [44] or SW-centric [34] visualization, to a symbolic execution engine [90]. However, the RISC-V VP previously was missing an effective methodology to design and integrate models that capture the interaction of the VP *with the environment*, such as other components on a PCB besides the processor chip.

In this section, such an extension is proposed and described to broaden the application domain for virtual prototyping in the RISC-V context. Also, a set of building blocks for the environment is provided which includes buttons, LEDs, and a display. The main idea of the approach is to separate the hardware-model from the world behavior (see Figure 3.4, *VP Environment* vs. *RISC-V VP*). This allows for the parallel development of software and hardware, within the intended environment, speeding up the design process. For visualization of the environment, a GUI using the Qt C++ library is proposed. To ease the environment setup, a configuration-file based approach has been chosen, which enables the designer to specify the desired components and appropriate connections to the VP in a simple way. Additionally, it is possible to instantiate, connect and modify new devices for an interactive user experience. The communication channel between the RISC-V VP and Environment Model GUI is established through a TCP connection, which also enables to distribute the simulation to different computers e. g. a simulation server and a user's desktop PC. Appropriate libraries are also provided to tunnel several hardware communication interfaces such as GPIO, SPI, UART, or Controller Area Network (CAN) (via SPI) on top of the TCP channel. This allows to transparently map these interfaces between the VP, which models the SoC, and the Environment Model GUI, which displays and simulates the behavior of external components. Furthermore, the communication is optimized to avoid performance impacts on the VP simulation. This proposed approach is designed, but not limited, to be integrated with SystemC-based VPs that leverage a TLM communication system. In addition, the setup provides a foundation to even attach external real hardware components to perform a VP-driven hardware-in-the-loop simulation, which is described in more detail in Section 3.4. To facilitate the environment model design, a set of building blocks, such as buttons, LEDs

and an OLED display, are already implemented. Moreover, for rapid prototyping purposes, a modeling layer that leverages the dynamic Lua scripting language is employed to design and integrate components faster with the VP-based simulation. Finally, two case-studies with different virtual environments are provided for evaluation. In all case-studies, the RISC-V VP in the HiFive1 configuration is used, which is a model of the RISC-V HiFive1 board from SiFive [92], which was introduced in Figure 3.2. Beside the two virtual environments, the corresponding two real physical systems were also built for comparison. It could be observed that both the virtual and physical systems behave identically in these case-studies, which demonstrates that the proposed approach provides suitable virtual models to enable early software development in the design flow.

Besides positive experiences in using the RISC-V VP platform for teaching lectures on system-level design and virtual prototyping, also other academic groups are known to leverage the RISC-V VP infrastructure for teaching embedded systems lectures with laboratory sessions in the RISC-V context [42]. This further underlines the applicability of the RISC-V VP platform with environment modeling capabilities for educational purposes. To further spread its adoption, the environment interaction GUI in combination with the case-studies is provided as open source [49].

3.2.2 Related Work

The extensive ecosystem of RISC-V comprises several simulators that differ in their implementation technique and intended purpose in order to cover different use-cases. SPIKE is the reference simulator that is mainly designed for pure CPU simulations with a basic set of peripherals [65]. RV8 is a high-speed simulator that employs just-in-time compilation techniques to boost the execution performance but also mainly covers pure CPU simulations [80]. R2VM also targets CPU simulations by utilizing binary translation techniques [102]. It can switch between fast and accurate simulations in order to cover different use-cases. QEMU enables full system simulation that covers a complete platform and employs advanced binary-level optimization techniques to achieve a high performance [64]. Building on that, [103] proposed an approach to efficiently simulate Translation Lookaside Buffer behaviors in a QEMU setting. Gem5 is also a full-system simulator that puts a stronger emphasis on architectural exploration aspects but has a significantly reduced performance as a trade-off [82, 104]. Going beyond that, the *Renode* simulation system supports multi-node networks of embedded systems in a distributed simulation [83]. Recently, SystemC-based processor simulation solutions have been introduced into the RISC-V ecosystem as well. Besides the RISC-V VP, which was previously covered in Section 3.1, viable alternatives are the DBT-RISE [81]

framework, ETISS [105, 106], and the RISC-V-TLM [107] instruction set simulator which are also designed with a SystemC integration in mind and provide RISC-V support. They lack however a way to model external devices, e. g. via SPI or GPIO. Regarding DBT-RISE, an example VP platform which integrates a RISC-V instruction set simulator and is implemented in SystemC TLM is provided [108]. Another SystemC TLM simulator for RISC-V is RISC-V-TLM [107], which is currently under active development to increase the supported core feature set. A recent approach, that has been build upon the RISC-V VP, proposed visualization of internal VP execution state for debugging purposes [44]. It offers a live view into the execution state of the SystemC peripherals but lacks an interactive modeling platform for the environment interaction. However, the freely available VP-based frameworks for RISC-V are currently missing an effective methodology to design and integrate configurable environment models with extensive graphical capabilities. Advanced environment modeling capabilities in a configurable framework with extensive graphical capabilities, as is described in this section, is not yet available by any of the open RISC-V virtual prototyping approaches. Finally, there are commercial VP tools such as Synopsys Virtualizer [109] that might support RISC-V in combination with extensive environment modeling capabilities, but their implementation is proprietary.

Looking beyond RISC-V devices, existing simulators like `simavr` [110] and `PICSimLab` [111] (using `simavr` in the background) can be cycle-accurate but are limited to a certain family of AVR processors, and are fairly computationally expensive.

In contrast to the proposed approach, which offers an interface to a SystemC VP and hence is able to incorporate custom in-house chips and IPs, the other simulators are not designed with advanced industry-proven SystemC-based virtual prototyping in mind.

3.2.3 Embedded Systems: Components and Interfaces

As briefly introduced in Section 2.1, the GPIO module is in most embedded devices the interface to the outside world. It drives the physical pins of the chip or interprets applied voltage as logic inputs, but can also be configured as an interface to other on-chip peripherals. These peripherals depend on the instance of the SoC and may include digital data interfaces (e. g. UART, SPI, ADCs) or timer-controlled PWM outputs. Aside from active polling, the CPU may also enable hardware interrupts that trigger when, e. g., the input state of a pin changes. This way, the CPU may work on other threads and only be notified via the Platform Local Interrupt Controller (PLIC) on a state change, initiated by the GPIO module.

3.2.4 VP-driven Environment Modeling

In this subsection the proposed approach is presented. First, the main software architecture components of the modified RISC-V VP and the new VP Environment Model GUI (Subsection 3.2.4.1) are introduced. Then, Subsection 3.2.4.2 gives a more detailed look at the communication between SystemC modules and the new GPIO server; and between the two executables. Lastly in Subsection 3.2.4.5, the details of the Environment Model GUI and the currently available, place-able objects are shown.

3.2.4.1 Architecture Overview

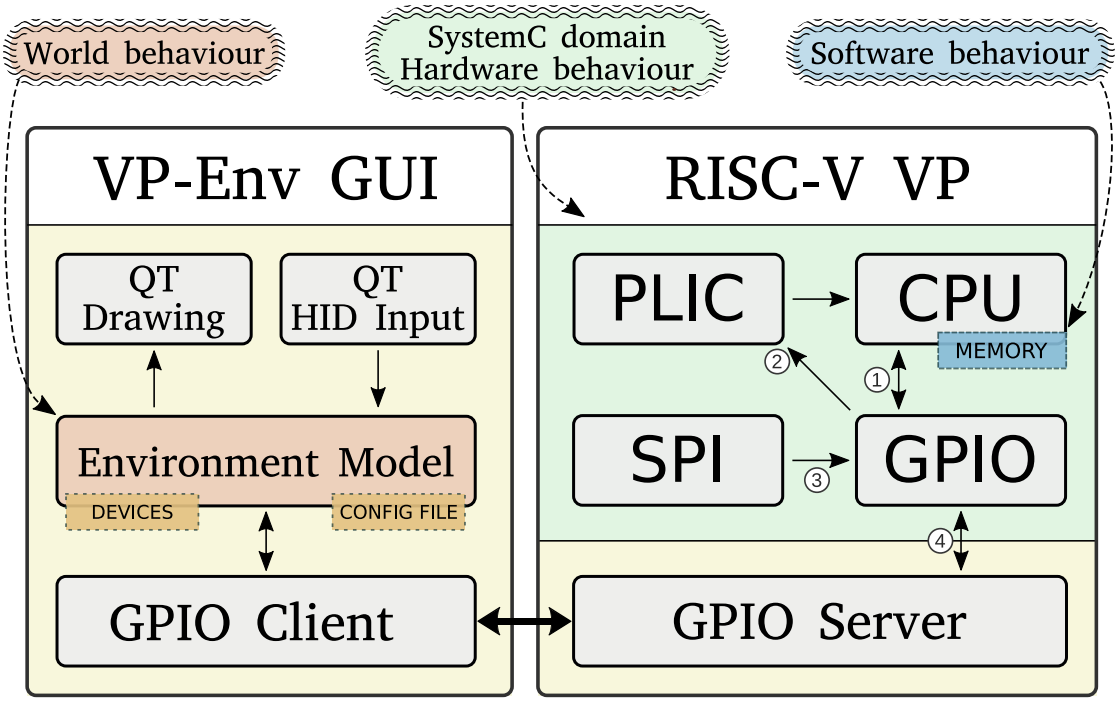


Figure 3.4: Main architecture of the virtual environment system. Elements highlighted in green define the hardware behavior through the SystemC domain language (simplified for readability). The contents of the VP's memory define software behavior, highlighted in blue. On the left side is the VP Environment Model GUI, which provides the interface to interact with the user. The behavior of outside components is combined in the environment model with its configurable set of devices, highlighted in orange.

Figure 3.4 shows an overview on the proposed approach. It consists of two standalone executables: The modified *RISC-V VP* and the new *VP Environment*

Model GUI. The RISC-V VP is organized in the *SystemC domain* (the hardware model residing in simulation time, highlighted in green) and the *GPIO server* (bottom right). The SystemC peripherals relevant to this work and their interactions will be described in more detail in Subsection 3.2.4.2.The GPIO server interfaces between the SystemC GPIO peripheral to the environment model in physical (i. e. wall clock) time, highlighted in yellow.

The VP Environment Model GUI (*VP-Env GUI*, on the left side) consists of the *GPIO client* (bottom left), the environment model and the QT modules for the GUI. The GPIO client is responsible for getting and setting the pin states to the simulated hardware (the RISC-V VP), while the environment model (highlighted in orange) models the world behavior based on the configured devices and user input (see Subsection 3.2.4.5). The protocol between the GPIO client and the GPIO server is described in more detail in Subsection 3.2.4.4. The environment model that manages the devices like buttons, LEDs, etc., and their respective connections to individual pins, is shown in Subsection 3.2.4.5. The QT interface modules handle the drawing functions and distribute keyboard/mouse input events to the respective environment components and is not shown in this work for brevity.

3.2.4.2 VP Peripheral Interfaces

As the GPIO peripheral resides in the user-space scheduled SystemC threading scheme, an interface to the asynchronous "real time" world is needed. This interface is modeled in the form of the *GPIO server*, which accepts TCP connections from a GPIO client to receive and send pin status updates. To minimize the dependency of the VP to the environment simulation, the server does not act (and thus impose an execution overhead) when no client is connected. Furthermore, the *GPIO client* in the environment simulation performs active polling on the server, where it sends changed pins on the environment side and requests the current status on VP-side. This way, the execution overhead on a running VP with environment simulation is minimized at the cost of missed changes that happen between the update cycles. With LEDs and buttons that interact with humans this is usually enough, but this loss of information would disable fast digital transmissions like SPI communication. This is why, when the GPIO peripheral is configured to do so, the GPIO server and client will transmit their respective payload either in a best-effort manner or synchronously. For digital communication interfaces like SPI, the approach abstracts the underlying protocol and directly transmits the payload to enable lossless communication and a faster simulation time.

3.2.4.3 SystemC Peripheral Interface

The GPIO peripheral is accessed through the device's bus (Figure 3.4, ①) where it can be configured and read by the software running on the CPU. It has an asynchronous (with respect to the CPU clock) interrupt line ② to notify the PLIC (and thus the CPU later) when a configured input triggers, and lastly a direct interface from the SPI peripheral ③ where the payload bytes are sent directly into the GPIO interface.

"Outside" pin changes (from the VP Environment Model GUI) are handled by using SystemC `AsyncEvent`s to notify the synchronous (with respect to SystemC simulation time) GPIO thread of pin changes ④, which in turn notifies the PLIC if an interrupt is configured for the corresponding change type.

3.2.4.4 GPIO-Protocol

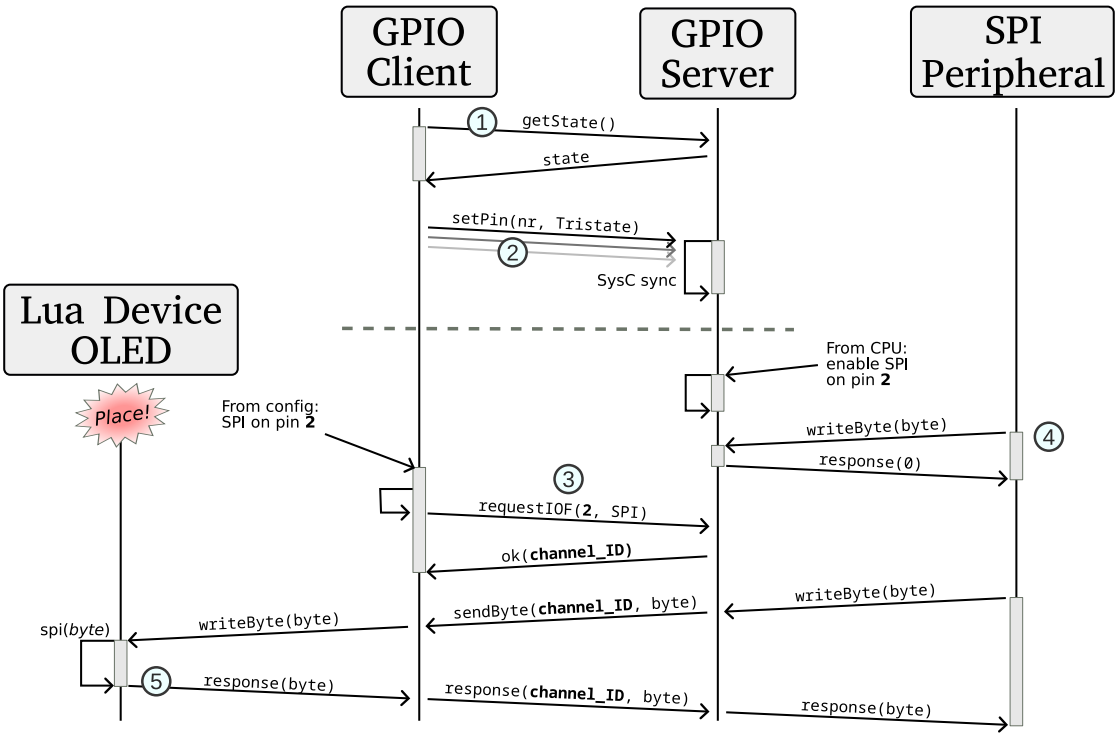


Figure 3.5: Example sequence diagram of the GPIO-Protocol. The dashed line illustrates that the `getState()` and `setPin(...)` functions are continued regularly in the background.

Upon startup, the GPIO server opens a TCP socket and listens to incoming connections from the GPIO client. The GPIO client in the virtual Environment Model GUI then connects to the server and begins polling with a `getState`

command to request all pin states (see Figure 3.5, ①). The response is a list of states for all pins that may assert either direct values (`LOW`, `LOW_WEAK`, `HIGH`, `HIGH_WEAK` and `FLOATING`), or any of the current supported IO-functions: `SPI`, `SPI_NORESPONSE`, `BITSYNC`, `PWM` and reserved future functions like `UART` and `I2C`. If an environment device (such as a button) sets a pin, it is always sent immediately over the channel (Figure 3.5, ②). Note that this happens throughout the execution of the environment in regular intervals, indicated by the dashed line after ②.

The GPIO client also may request an IO-function channel, in which case the GPIO server will generate a unique channel `ID` (Figure 3.5, ③) and opens (if not already existing) a second TCP socket, to which the GPIO *client* will connect (not shown in Figure 3.5 for brevity). If some on-chip peripheral writes to one of the GPIO IO-function inputs, but the corresponding pin is not yet "tracked" by the GPIO client, the GPIO peripheral will discard the message. Additionally, in case of a two-way protocol, it will respond as if no device was connected (e. g. a zero for SPI, Figure 3.5, ④). If the pin has an already registered *channel*, the datum is transmitted to the GPIO client, with the actual response forwarded if needed (Figure 3.5, ⑤).

Note, that in case of SPI, there is also the `SPI_NORESPONSE` mode (*unidirectional*) which can be requested by the GPIO client, in where the response phase is omitted by the GPIO client and the server will continue to respond directly with a zero. This is especially useful for devices that do not implement a response anyway, to further reduce the overall latency.

3.2.4.5 VP Environment Model

In order to support the design process to build an environment model with the accompanying GUI, a set of building block components which can be re-used across different environments is provided, while the GUI supports settings and placement via a configuration file. In the following, both aspects are described in more detail.

C++ Building Blocks As an overview, five ready-to-use C++ components are described as building blocks to support the design engineer in building environment models:

- Button: Simple input area on screen that changes its state by mouse click or button press and sets/resets a pin value.
- LED: A colored light spot that changes its brightness from transparent to a configured color tone, depending on a pin input value. This value can either be binary (on/off) or a PWM ratio sent as float.

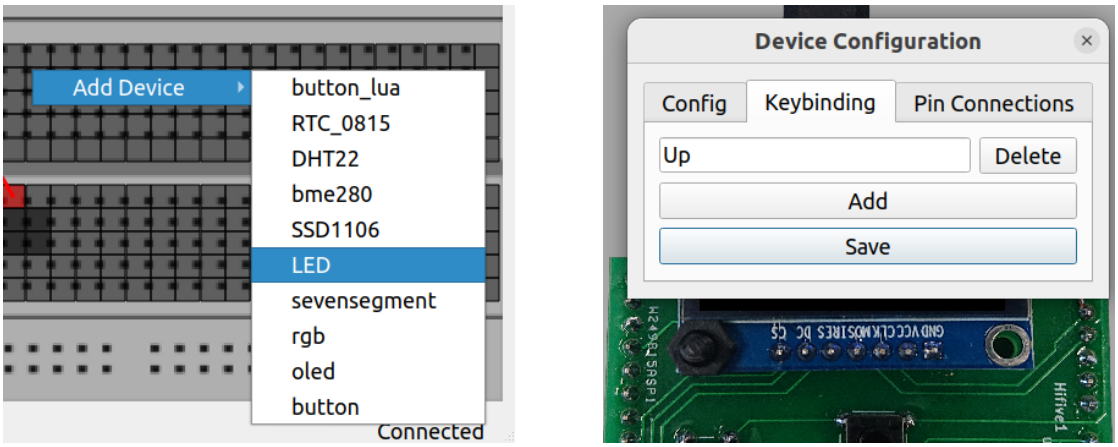
- Red Green Blue (RGB) LED: An extension to the regular LED, for convenience directly mixing the three colors.
- Seven segment display: Arranged color lines with background of a configurable size, connected to up to seven input pins.
- Display: An SSD1306 SPI OLED display with internal state machine, connected to one of the digital SPI inputs and two pins for slave select and data/command switch.

Besides this set of C++-implemented components, the Environment Model GUI architecture is designed to be extensible and, as such, adding additional components to increase the toolbox is fairly easy. For rapid prototyping purposes, a Lua-scripting interface is also provided for modeling devices, as explained in the following Subsection 3.2.5.

3.2.4.6 Drag and Drop

The drag and drop functionality is based on QT's `QTDragEvent`s. For the simulated connections between devices and to the controller board, a visual and an underlying logical representation is kept redundantly. So when adding a device or wire and connecting it to a row in a breadboard (as in Figure 3.9), the (possibly cascaded) path is resolved once and saved as a 1:1 lookup-table, in order to reduce per-frame computational cost. Adding a new device is easily done with a right-click context menu on the screen (see Figure 3.6a) and selecting one of the currently loaded devices. A wire is created by simply starting a drag from one pin-position (on the PCB or breadboard) and releasing it on another. As some devices offer on-click callbacks (cf. Figure 3.7), moving existing devices is only possible in the drag-mode, switched with pressing the space bar on the keyboard. To distinguish the modes, every drag-able object is highlighted in transparent red visually when in drag-mode.

The placed devices also offer a context menu to alter settings, by right-clicking on them in any mode (see Figure 3.6b). These offer all the settings that will be discussed in Subsection 3.2.5. Additionally, when the model does not have a breadboard but "invisible" PCB traces, the pin connections are set in this menu.



(a) Device add (b) Configuration window

Figure 3.6: Context menus of the Environment Model GUI. Figure 3.6a demonstrates a list of devices to be added, and Figure 3.6b shows a settings-window of a button that offers to bind new keys to the button, as well as changing the pin-connections and the device-specific configuration elements.

3.2.5 Rapid Prototyping using Lua Scripting

To increase the usability of the RISC-V VP and Environment Model GUI, a device scripting engine is provided. This allows developers to focus on the actual behavior of devices, without having to understand the whole system, to not have to re-build the framework for each change in a device, and to increase modularity for an easier community-driven library of devices.

Such a scripting engine has to be fast, memory efficient, and easily learnable. Without a particular scientific relevance, Lua was chosen as the driving scripting language; as it is widely used in games and other applications where execution speed and a low memory footprint is key. While the Python language was considered, as is being used widely nowadays in more high level applications, its interpreters for C/C++ programs compare rather laborious and (slightly) slower.

For the interface between Lua and C/C++, it proved to be beneficial keeping the dynamically typed language style, and thus activating offered interfaces in a *"duck typing"* way. This means, if a script is loaded, it is checked whether it implements certain functions that are expected by the framework. These can then be used by the configuration mechanism to enable/connect the following currently implemented functions: **SPI**, **Pin input/output**, **Configuration change**, **Button/Mouse input**, and **Graphics**. In Figure 3.7, a brief overview of the interface registry is shown. On the right side, the *Lua* tab, the functions to be implemented are grouped by their interfaces (colored). Highlighted in bold are the necessary

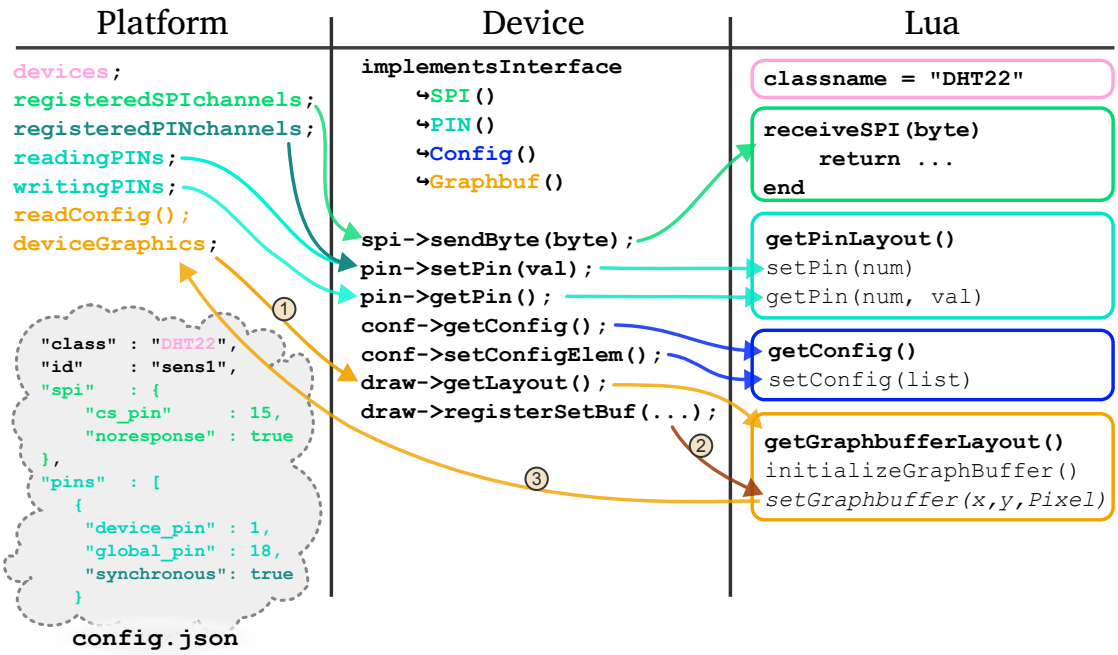


Figure 3.7: Available device interfaces for Lua scripts. In the Lua tab, highlighted in bold, are the minimum necessary functions for each interface. Not shown is the **Button/Mouse input** interface with the functions `onClick(active)` and `onKeypress(code, active)` for better readability.

functions for each interface to be recognized by the *Device* wrapper (central tab). Note that the *Graphics* interface is special, which will be discussed in the following paragraphs.

Upon instantiation, the device wrapper will check for existence of these functions and add the corresponding interfaces to the C++-world. The *Platform* then instantiates and stores all devices listed in the *configuration file* (bottom left). For each of the interfaces, a central registry is held for all devices, to speed up the lookup each frame.

A Lua scripted device may implement a set of functions and at least have a member `classname` which is used to identify and instantiate this device (see Figure 3.7). For *pin input/output*, it has to implement at least the function `getPinLayout()`, in where it defines the number of input or output pins during the instantiating setup phase (see also Figure 3.8).The host system will then periodically call `getPin(num)` (if implemented) to request updates, and `setPin(num, val)` if registered and connected pins are updated from outside the device. A pin in updated asynchronously (i. e. only when the Environment Model GUI updates, see Subsection 3.2.4.4) per default, unless the environment configuration sets it as `synchronous`. This is to save bandwidth and performance. Normal, asyn-

chronously sensitive, pins are registered in the `reading-` and `writingPINs` data structures, while synchronous pins are handled in the `registeredPINchannels` data structure, as it is considered as an IO-function internally. SPI connections are handled by implementing `receiveSPI(byte)` and is called synchronously when the device is connected to an SPI port and received something. Note, that the function may return a value that is passed back to the processor, if not configured in `SPI_NORESPONSE` mode.

If the device implements the `setConfig(list)` functions it may receive configuration updates in form of a key-value list during setup from the json config file. Additionally, it may implement `getConfig()`, from where the (default) settings may be viewed and reconfigured in the GUI. For GUI interactions (*Button/Mouse input*), the device may implement `onKeypress(keycode, press_release)` or `onClick(press_release)`. Note that for `onClick`, a graphical representation is needed.

The interface for *Graphics* is a bit more interesting, as the Environment Model GUI offers functions *to* the device once it defines the `getGraphbufferLayout()` function. During setup (Figure 3.7, ①), the GUI calls this function and reserves a memory region with the requested image size and format (currently only `RGBA8888`), and inserts the callback function(Figure 3.7, ②) `get-` and `setGraphbuffer(x,y,Pixel)` which directly access the internal image buffer. A `Pixel` is a custom data type that combines red, green, blue and alpha values. These functions may be called by the device during all callbacks (Figure 3.7, ③).

Due to technical reasons, all scripted devices run in one single Lua interpreter state as scoped chunks for the best memory and execution speed[2]. This means that a script is loaded into a table, in where it only may access pre-defined global functions without access to the other script's functions. All devices may call `setGraphbuffer(..)`, but they may only access their own buffer. To enable this, this approach uses prefixed global[3] C functions (e. g., `button1_setGraphbuffer(..)`, Listing 3.8, Line 10). These are inaccessible for the scoped device scripts (*chunks*), until it is inserted into the respective Lua meta-table (Listing 3.8, Line 21), and without the prefix.

3.2.5.1 Configuration

The Environment Model GUI loads a `json`-formatted configuration file on start-up for ease of customizing the user interface. An example is shown in Subsec-

[2]Early tests have shown that instantiating one Lua-state per device results a prohibitively high memory usage already in small numbers of devices, and also significantly reduces the execution speed.

[3]This is a technical limitation of the used *LuaBridge3*, in where C functions may only be global.

```
1  template<typename FunctionFootprint>
2  void LuaDevice::Graphbuf_Interface::registerGlobalFunctionAndInsertLocalAlias(
3      const string name, FunctionFootprint fun) {
4    if(m_deviceId.length() == 0 || name.length() == 0) {
5      cerr << "[Graphbuf] Error: Name '" << name << "' or prefix '"
6          << m_deviceId << "' invalid!" << endl;
7      return;
8    }
9
10   const auto globalFunctionName = m_deviceId + "_" + name;
11   luabridge::getGlobalNamespace(L)
12     .addFunction(globalFunctionName.c_str(), fun)
13   ;
14
15   const auto global_lua_fun =
16     luabridge::getGlobal(L, globalFunctionName.c_str());
17   if(!global_lua_fun.isFunction()) {
18     cerr << "[Graphbuf] Error: " << globalFunctionName  << " is not valid!" <<
       ↪ endl;
19     return;
20   }
21   m_env[name.c_str()] = global_lua_fun;
22 };
```

Listing 3.8: Mechanism for unique global C-functions that are inserted into the Lua-script's metatable `m_env` as prefix-less references.

tion 3.2.5.1. In the `window` section, a background image (Line 3) and a desired window size (Line 4) can be defined (which defaults to the background image size).

After that, all implemented / loaded device classes may be referenced and instantiated in the `devices` section (Line 6). A `device` entry must have a `class` and an `id` (Lines 9 and 10). The `class` references the building blocks `classname` (see Subsection 3.2.5), while the `id` must be a unique name to the instance. Further items depend on the implemented interface of the specific building block (see Figure 3.7). For example, a Lua-implemented button `button_lua` offers the `graphics` (Line 11), `onKeypress` (Line 16), and `pin` (Line 17) interface. The OLED device (Line 44) was implemented in both Lua and C++, with the latter being instantiated in this example.

```
1  {
2    "window" : {
3      "background" :
       ↪ ":/img/oled_shield.jpg",
4      "windowsize" : [470, 750]
5    },
6    "devices" :
7    [
8     {
9        "class" : "button_lua",
10       "id" : "button_up",
11       "graphics" :
12       {
13         "offs" : [77, 625],
14         "scale" : 2
15       },
16       "keybindings" : ["DOWN"],
17       "pins" :
18       [
19         {
20           "device_pin" : 1,
21           "global_pin" : 19
22         }
23       ]
```

```
24      },
25      {
26        "class" : "button_lua",
27        "id" : "button_down",
28        "graphics" :
29        {
30          "offs" : [220, 580],
31          "scale" : 2
32        },
33        "keybindings" : ["DOWN"],
34        "pins" :
35        [
36          {
37            "device_pin" : 1,
38            "global_pin" : 4
39          }
40        ]
41      },
42      [...] // other buttons
43      {
44        "class" : "SSD1106",
45        "id"    : "display",
46        "spi"   :
47        {
48          "cs_pin" : 15,
49          "noresponse": true
50        },
51        "pins"  :
52        [
53          {
54            "device_pin" : 1,
55            "global_pin" : 16,
56            "name": "dc_pin",
57            "synchronous" : true
58          }
59        ],
60        "graphics" :
61        {
62          "offs" : [105, 308],
63          "scale" : 2
64        }
65      },
66      {
67        "class" : "LED",
68        "id"    : "led1_down",
69        "pins"  :
70        [
71          {
72            "device_pin" : 1,
73            "global_pin" : 3,
74            "name": "led_pin"
75          }
76        ],
77        "graphics" :
78        {
79          "offs" : [435, 655],
80          "scale" : 2
81        }
82      }
83    ]
84  }
```

Listing 3.9: Excerpt of an example configuration file for a PCB with an OLED display, used in Figure 3.10b.

3.2.5.2 Scoping layers

To increase modularity in the whole HW stack from device to SoC peripheral, the *Environment Model* consists of four layers: The *device layer*, the *environment layer*, the *platform layer*, and the *GPIO layer* in the GPIO peripheral of the VP (see Figure 3.8). The *device layer* is scoped to every individual device, which define the pin and other protocol descriptions according to their respective interfaces (see Subsection 3.2.5). In the *environment layer*, all instantiated devices are connected to the global pin identifiers. This would normally be done in a prototyping breadboard or a PCB. It is allowed to not connect pins. The pins between the labeled "global" connectors of a platform (like the HiFive 1) into the chip's GPIO register offsets are translated in the *platform layer*. Lastly, in the GPIO module that resides in the VP, the actual pin states are set/read according to Subsection 3.2.4.4 and can either contain per-pin managed digital levels (see Subsection 3.2.4.3) or pass through to an IO-function like SPI.

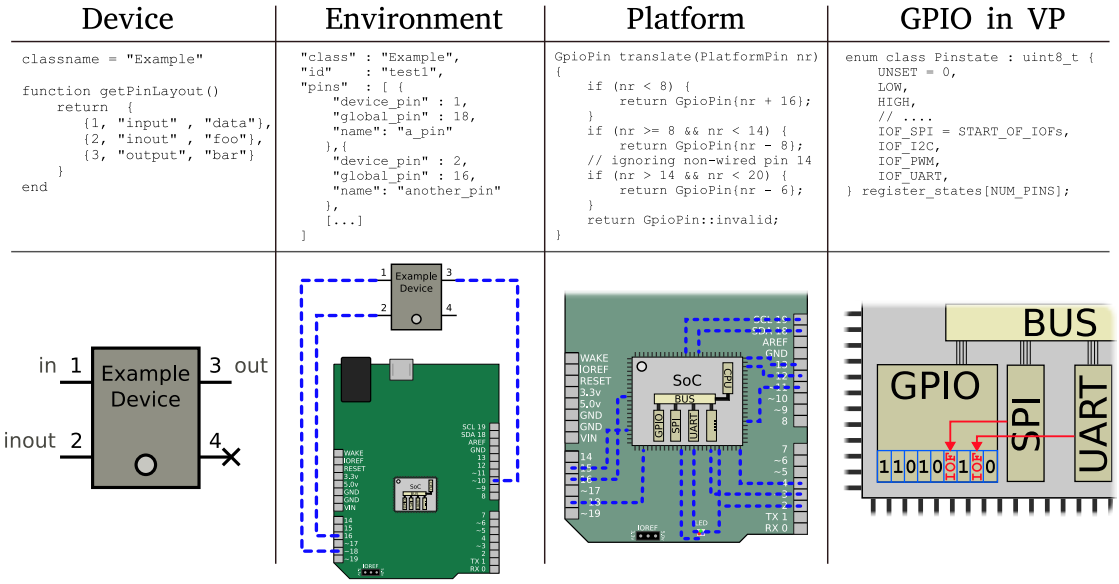


Figure 3.8: The four layers of scoping for connections from the individual environment device to the actual register contents of the GPIO peripheral in the VP.

3.2.5.3 Example Devices

To explain the concept better, the following paragraphs show two of the currently implemented devices in more detail: A simple red LED (see Listing 3.10) and a more complex OLED display (Listing 3.11).

LED The LED implementation in Listing 3.10 uses the `pin`, `config`, and `graphic` interfaces. In Lines 3 to 6, the module defines only one pin with the number `1` as an input pin and the description string of `led_on`. Lines 8 to 11 request only one color pixel from the graphics system, which is accessed later via `setGraphbuffer(0,0, ...)` on Lines 32 and 34. Lines 13 to 15 define local variables for the displayed color, which are set or read by the configuration file or during runtime via `getConfig()` and `setConfig(conf)` in Lines 17 and 23, respectively. The actual display action happens if the input pin (`1`) is changed (Lines 29 to 37). The call supports multiple pins, so `setPin(...)` includes the pin number and the (boolean) value if it is `HIGH` or `LOW`.

SSD1103 OLED Display In Listing 3.11, a more sophisticated example is given. It implements the already known `pin` interface, but also the `SPI` interface with the function `receiveSPI(byte_in)` (Lines 62 to 89) Note that, for brevity, some of the internal logic is omitted (Lines 50, 64, 83). In Lines 38 to 47, the most common operator bytes are defined. The omitted function `getMask(op)` determines the value

```
1  classname = "LED"
2
3  function getPinLayout()
4    -- number, [input | output | inout], name
5    return  {1, "input", "led_on"}
6  end
7
8  function getGraphBufferLayout()
9    -- x width, y width, data type
10   return {1, 1, "rgba"}
11 end
12
13 local r = 255
14 local g = 10
15 local b = 0
16
17 function getConfig()
18   return {{"r", r},
19           {"g", g},
20           {"b", b}}
21 end
22
23 function setConfig(conf)
24   r = conf["r"] or r
25   g = conf["g"] or g
26   b = conf["b"] or b
27 end
28
29 function setPin(number, val)
30   if number == 1 then
31     if val then
32       setGraphbuffer(0, 0, graphbuf.Pixel(r, g, b, 255))
33     else
34       setGraphbuffer(0, 0, graphbuf.Pixel(r, g, b, 0))
35     end
36   end
37 end
```

Listing 3.10: Simple one-pixel LED model with Lua

bits of an input command byte, which is then used by the `match(cmd)` function (Lines 53 to 60) to decode incoming raw bytes. Lastly, in `receiveSPI(byte_in)` (Lines 62 to 89), the actual drawings to the frame buffer are done when the incoming SPI byte is detected as `data` (if the `data_command` pin was set `HIGH`). In Lines 63 to 71, the translation from 1-bit-pixel rows to the pixelwise frame buffer is done, including the increment of the current column pointer. Some of the command handling is shown in Lines 71 to 87, where internal state variables are changed.

```
classname = "SSD1106"

function getPinLayout ()
  -- number, [input | output | inout], name
  return  {1, "input", "data_command"}
end

local width = 132
local height = 64

function getGraphBufferLayout()
  return {width, height, "rgba"}
end

local isData
local state = {
  column     = 0,
  page       = 0,
  contrast   = 255,
  display_on = true
}

function setPin(number, val)
  if number == 1 then
    isData = val
  end
end

-- optional
function initializeGraphBuffer()
  for x = 0, width-1 do
    for y = 0, height-1 do
      setGraphbuffer(x, y, graphbuf.Pixel(0,0,0, 255))
    end
  end
end

operators = {
  COL_LOW             = 0   ,
  COL_HIGH            = 0x10,
  PUMP_VOLTAGE        = 0x30,
  DISPLAY_START_LINE  = 0x40,
  CONTRAST_MODE_SET   = 0x81,
  DISPLAY_ON          = 0xAE,
  PAGE_ADDR           = 0xB0,
  NOP                 = 0xE3
}

function getMask(op)
  [...]
end

function match(cmd)
  for key, op in pairs(operators) do
    if ( (cmd ~ op) & getMask(op) ) == 0 then
      return op, cmd & (~getMask(op))
    end
  end
  return operators.NOP, 0
end

function receiveSPI(byte_in)
  if isData then
    [...]
```

```
65      for y = 0,7 do
66        if (byte_in & 1 << y) > 0 then pix = 255 else pix = 0 end
67        setGraphbuffer(state.column, (state.page*8)+y,
68                       graphbuf.Pixel(pix,pix,pix, state.contrast))
69      end
70      state.column = state.column + 1
71    else
72      op, payload = match(byte_in)
73      if op == operators.DISPLAY_START_LINE then
74        return 0
75      elseif op == operators.COL_LOW then
76        state.column = (state.column & 0xf0) | payload
77      elseif op == operators.COL_HIGH then
78        state.column = (state.column & 0x0f) | (payload << 4)
79      elseif op == operators.PAGE_ADDR then
80        state.page = payload
81      elseif op == operators.DISPLAY_ON then
82        display_on = payload
83      [...]
84      else
85        print("unhandled operator " .. byte_in)
86      end
87    end
88    return 0
89  end
```

Listing 3.11: Simple SPI OLED driver model with Lua

3.2.6 Evaluation

In this section, some use-cases for the Environment Model GUI are featured by modeling two example environments along with their interacting software (Subsection 3.2.6.1), give a performance evaluation of different modeling strategies (comparing to the baseline RISC-V VP, Subsection 3.2.6.2), and lastly a short demonstration on how it was used in lectures (Subsection 3.2.6.3).

3.2.6.1 Modeling Case-Studies

The implementation of the proposed approach for VP-driven environment modeling and interaction in the RISC-V context was done using the previously described RISC-V VP as foundation (cf. Section 3.1). To demonstrate the effectiveness of the proposed approach in building feature-rich environments, two example environments were designed in combination with different firmware applications as a case-study. In the following, both case-studies are presented in more detail (Subsection 3.2.6.1 and Subsection 3.2.6.1).

Breadboard Environment To demonstrate the usability of the proposed approach as a rapid prototyping methodology, a custom breadboard environment is featured with a configured button, an LED and a seven segment display, as well as the built-in RGB-LED of the HiFive1 board. An excerpt of the corresponding

configuration file can be found in Subsection 3.2.5.1. The corresponding graphical display of the environment is shown in Figure 3.9. Beside the already mentioned components, the environment also displays the connection between the respective GPIO pins of the HiFive1 and the breadboard. During the VP-driven simulation, the Environment Model GUI is updated accordingly to reflect the current execution state of the RISC-V VP.

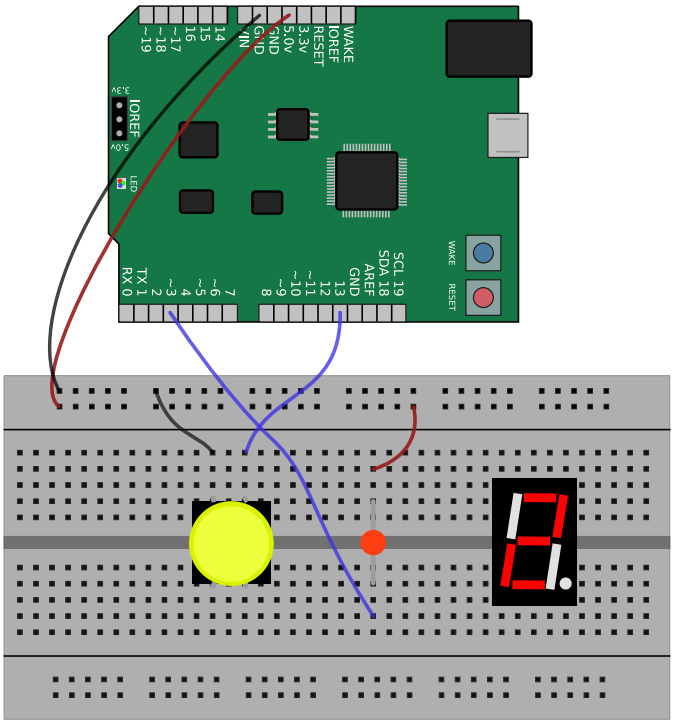
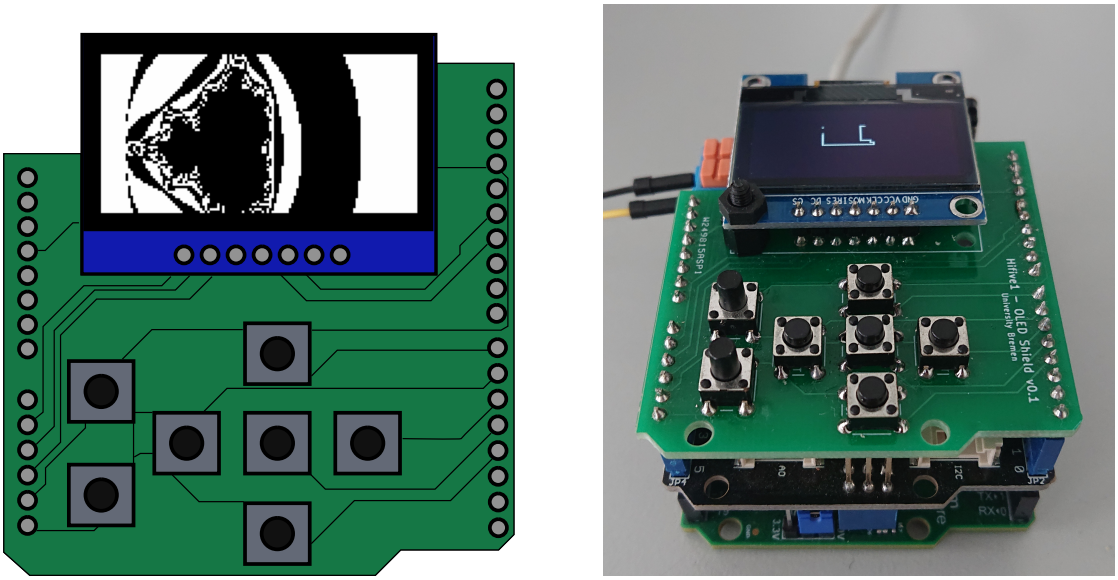


Figure 3.9: Image of the virtual breadboard environment with a button, a red LED and a seven segment display on the breadboard, and the builtin RGB-LED on the HiFive1. The connections to the seven segment display are omitted for readability reasons.

The firmware is held simple in this example: It counts a number in seconds using the CLINT timer, and renders it to the seven segment display. Whenever the button is pressed, the count direction is reversed accordingly. The single LED is changed every second. Due to the built-in RGB LED segments being always connected to certain GPIO pins of the seven segment display, its color changes and mixes as well.

OLED Display Shield with Buttons For a more sophisticated example, a handheld "gaming" device with seven input buttons and a 64 by 128 pixel wide OLED screen is presented here. An overview of this system is shown in Figure 3.10, with the left side showing the virtual environment and the right side showing the corresponding real physical device. The screen is connected via SPI and demonstrates

the bytewise I/O functions of the GPIO and environment model. The buttons are connected to ground and require a pull-up resistor on the input pins to work, while the OLED screen is interfaced via an SSD1306 [112] compatible protocol, consisting of the usual SPI pins MISO, Clock (CLK), and a Data Command (DC) input. This interface also demonstrates the requirement of synchronicity between the abstracted byte-wise SPI transmissions and the GPIO-handled (software driven) Data/Command (DC) pin[4], where a small transmission jitter of the data vs. DC pin would already result in a glitchy or inoperable display (see also Section 3.3). An excerpt of the Lua-implementation can be found in Listing 3.11.
To test the interaction between user input and output, a demo snake game was implemented that listens to the up, down, left and right buttons in an interrupt routine and draws a gaming field on the screen. With the key mapping of the Environment Model GUI, it can be played by clicking on the on-screen buttons or via the arrow-keys on the keyboard. As the RISC-V VP is binary compatible to the HiFive1 board, the same program could be used and played on the real board, after the PCB has been manufactured (see Figure 3.10b). For demonstration purposes, another firmware beside the snake game was built, that displays a *Mandelbrot* set visualization on the same device (as shown on Figure 3.10a).



(a) Simulated (b) Real

Figure 3.10: The OLED display shield with an SSD1306 driver and seven buttons, running the demos from Subsection 3.2.6.2 (Fig. 3.10b) and Subsection 3.2.6.1 (Fig. 3.10a), respectively.

[4]To optimize the communication protocol, some SPI devices use a separate input pin to incoming bytes as data or commands.

3.2.6.2 Performance Evaluation

For performance evaluation purposes, designed a non-interactive test program was built that first calculates 40 frames of the *Mandelbrot* set visualization (see Figure 3.10a) which uses software floating-point arithmetic to render the fractal. It then fades the display using the background illumination command, and draws 1000 characters of predefined, randomized text to the screen. After this, the program exits with a special RISC-V exit sequence that is handled in the RISC-V VP. This is, of course, not handled by the real processor. While the *Mandelbrot* set visualization is computationally intensive, as every pixel is calculated individually, the text stream part is only limited by the SPI-bandwidth as it uses lookup-buffers for the font and addresses the native 8-pixel-rows per byte.
Also, the SPI OLED display was implemented differently three times: **1.** In the SystemC VP, communicating directly with the SPI device over TLM, sharing only the screenbuffer over memory-mapped I/O to the GUI; **2.** in the Environment Model GUI as a C++-Device, using the proposed GPIO protocol; and **3.** in the Environment Model GUI as a Lua-Device, using the proposed GPIO protocol (see Listing 3.11).
The results of this experiment can be found in Table 3.2. The first column describes the **Test** type: *Baseline* (unmodified RISC-V VP with non-functional mock-up GPIO peripheral); *Disconnected* (the modified RISC-V VP with the display modeled in SystemC, but no connection to the environment); and *GUI connected* (the modified RISC-V VP with the connected Environment Model GUI actively displaying the execution state). The connected tests are built in four different set-ups: *SystemC-Device*, where the OLED display driver is directly connected to the SPI peripheral in SystemC sharing the screenbuffer with the Environment Model GUI; *Bidirectional C++-Device* where the driver is modeled in the Environment Model GUI and the SPI peripheral awaits the answer byte via the protocol; *Unidirectional C++-Device* where the device's answer is discarded for speedup; and finally *Unidirectional Lua-Device* where the logic of the display driver is modeled in the Lua scripting engine.
The next column, **Time**, reports the real time as reported by the GNU-binutils program `time` of the whole simulation with an already started GUI (if applicable). # **Exec. Instr.** refers to the number of native machine instructions (not pseudo-instructions) executed until test end. Note, that the number differs slightly for the same binary due to different behavior when the GPIO memory-mapped region is either mock-up memory (*Baseline*), correct but disconnected (*Disconnected*), or responding to actual SPI devices (*GUI connected*). Lastly, the amount of MIPS is calculated, to offer a comparison to other simulation approaches. All tests were conducted on a desktop grade AMD Ryzen 3700G processor with 32 GiB Random

Table 3.2: Performance overhead test results. GPIO register accesses (read/write): 3025/946. SPI words transmitted: 58 678 (in connected tests).

Test	Time	# Exec. Instr.	MIPS
Baseline	27.312s	79 392 401	2.907
Disconnected	27.617s	79 390 408	2.875
GUI connected			
– SystemC-Device	27.917s	79 390 486	2.844
– Bidirectional C++ Device	32.118s	79 390 437	2.472
– Unidirectional C++ Device	28.789s	79 390 491	2.758
– Unidirectional Lua Device	28.654s	79 390 408	2.771

Access Memory (RAM), and outperformed the real HiFive1 setup; especially in memory intensive tests[5], which is usually not possible with RTL models.
As can be observed, a connected and running VP environment has a minimal impact on the execution speed of the RISC-V VP. Besides the asynchronous communication scheme, the minimal overhead could be achieved through the use of a multi-core processor, as the RISC-V VP uses the single-threaded SystemC reference implementation. Thus, the RISC-V VP and the Environment Model GUI can be executed in parallel with little to no interference. Secondly, it can be noted that the implementation of a high-troughput device (like the OLED display) in the *Lua* scripting language does *not* add a significant run-time overhead to the simulation speed, as long as the response is discarded. Note however, that the refresh rate of the Environment Model GUI drops slightly[6], as Lua devices accesses to the framebuffer are generally slower because of the C-wrapper (see Figure 3.7). The overall impact of the proposed approach on execution speed can be observed against a baseline version of the GPIO peripheral, where any accesses to the memory-mapped IO-interface are ignored (pass-through to memory). This reveals only a 2.2% runtime overhead on average for the benefit of a functioning, interactive GPIO interface.

3.2.6.3 Educational Tool for Teaching

Among others, a system-level design lecture has been given that also covers programming embedded systems. During the corona pandemic, there was no possibility for the students to interact with physical prototype boards like the Sifive Hifive1. As the students covered implementing their own small VPs, it was easy

[5]Note that the RISC-V VP has the feature to lock the CLINT's internal timer to either simulation- or wall clock time.
[6]Refresh rate in all tests varied between 10 and 20 Hz, limited to 20 Hz.

to show them the principles of the more complex RISC-V VP. The students then could use and program the digital version of the Hifive1 board to understand the basic concepts of interrupt handling and how embedded systems interact with their environment. As the RISC-V VP can be analyzed using normal software-based debuggers like GDB, the detailed steps of different control flows during run-time could be shown, and how software and hardware modules interact between each other. The small exercises were laid out in incremental steps to program an interrupt-triggered blinking LED while reacting to button presses. One year, the final lectures could be held in person, where the students could test their own programs on real Hifive1 boards supplied by the university.
Overall, it could be noted that the RISC-V VP along with the Environment Model GUI, while posing an initial learning curve, was very helpful during remote-teaching and still nice to have in in-person teaching as every student could test and build their programs at home without having to supply real hardware. It can be supposed that it is also be beneficial for more practical-focused embedded programming courses as in [42]; especially when using hardware that is either too costly or complex to be supplied to every student or hardware that requires special programming devices.

3.2.7 Discussion and Future Work

The evaluation demonstrates the applicability of the proposed approach in building advanced VP-driven environment models for embedded systems efficiently to enable a full-platform simulation early in the design flow. To further boost this approach, promising future work can be seen in several aspects:

1. Extend the GUI to build the environment configuration live in an interactive way instead of using a static configuration file. Moreover, development has already started in investigating to enable dynamic modifications of the environment configuration at runtime in order to facilitate the rapid prototyping process or for debugging purposes.
2. Combine this approach with advanced VP-based debugging techniques like the ones mentioned in Section 3.3, that enable to present additional internal VP run-time information alongside the environment model state.
3. Leverage a VP-driven hardware-in-the-loop integration (as proposed in Section 3.4) that allows attaching real physical HW objects to the VP-based simulation. This would allow mixing virtual and physical environment objects in an *extended reality* setting. In addition, it allows a step-wise approach to refine models and specifically debug certain physical objects by providing virtual wrappers for the others.

4. Add a monitor that records the devices behavior and compares it against wanted behavior. Together with a headless simulation and checkpointing, this can be used for automated integration tests; essentially closing the loop for accurate embedded SW testing in Continuous Integration / Continuous Deployment (CI/CD) systems.
5. Look into techniques to boost the simulation performance of the VP-driven simulation. In particular integration of just-in-time compilation techniques seems very promising but requires special attention to be integrated with a SystemC-based simulation in combination with the environment model communication.
6. Investigate integration of extra-functional models with the VP-driven simulation to enable fast and accurate estimation of extra-functional properties like timing behavior and power consumption in a full platform setting. This requires appropriate interfaces and dedicated techniques for measurement and synchronization between the VP and environment models.

3.2.8 Conclusion

This section presented an effective methodology for advanced environment modeling and interaction for VPs in the RISC-V context to enable the design of advanced embedded system early in the design flow. It described a library with a set of building blocks and support for several hardware communication interfaces. For visualization purposes of the environment, an interactive GUI was designed which communicates to the VP through TCP connections. The environment model is specified through a configuration file or can be created on the fly using the GUI controls to ease the setup and improve the user experience. For rapid prototyping purposes, a modeling layer was proposed that leverages the dynamic Lua scripting language to design components and integrate them with the VP-based simulation. The evaluation with two different case-studies demonstrated the applicability of the proposed approach in building virtual environments effectively and correctly. To advance the RISC-V community and stimulate further research, the complete framework including all case-studies is provided publicly in [48]. The combined VP platform has also proven to be very beneficial for education purposes in lectures, as shown in [42] and in the author's experience with own lectures, and was featured as the first entry of the *Chip Industry's Technical Paper Roundup: October 2022* [43].

3.3 Minimally Invasive SW/HW Co-debug Live Visualization on Architecture Level

The following section includes published material from the conference paper [44]. It is started by a recapitulation of the topic's motivation in Subsection 3.3.1, directly followed by an overview into architecture of the proposed approach.
Next, the approach is put into perspective to related work in Subsection 3.3.2, followed by preliminaries needed to fully understand the approach and the case-study in Subsection 3.3.3 with building blocks of the proposed tool and the relevant concepts. More details of RISCview's architecture and implementation are then given in Subsection 3.3.4, while Subsection 3.3.5 describes the case-study. Finally, contributions are summarized with an outlook to promising directions of future work in Subsection 3.3.6.

3.3.1 Introduction

As mentioned earlier, VPs [38, 113] are important tools for hardware/software co-design. This section considers VPs typically modeled on the TLM layer, written in a high-level language like SystemC [4], which abstracts from implementation details of the hardware. It models the hardware to a level of detail such that it can execute software that is supposed to run later on the developed hardware. This way, VPs allow writing software for a target system before the actual hardware is finalized and produced, resulting in a shorter time-to-market. Additionally, it also enables effective debugging early in the design process, in particular of the often complex interplay between hard- and software.
The usage of VPs is not limited to modeling the capabilities of the later designed hardware, however. This section's approach leverages the possibilities of a *SW* simulation system by gaining insights with a GUI into the simulated HW devices that would only be possible on real HW with incredibly expensive equipment on IC scale. Moreover, the developers are able to focus on the internal states of the individual devices of interest, with the proposed fast and easy-to-use framework called RISCview. It offers a configurable HW/SW co-visualization with a minimal impact on the existing code-base by leveraging the model-view principle.
The main architecture is sketched in Figure 3.11. The HW debugging GUI (left side) is connected via TCP to the executable (right side) consisting of the virtual prototype and user-defined views of the hardware structure. For establishing the connection to the debugging GUI, RISCview provides a *visualization interface* that is linked into the VP executable. A number of instantiated *views* (green) define the data of interest of the existing system's *modules* (blue). This model/view scheme

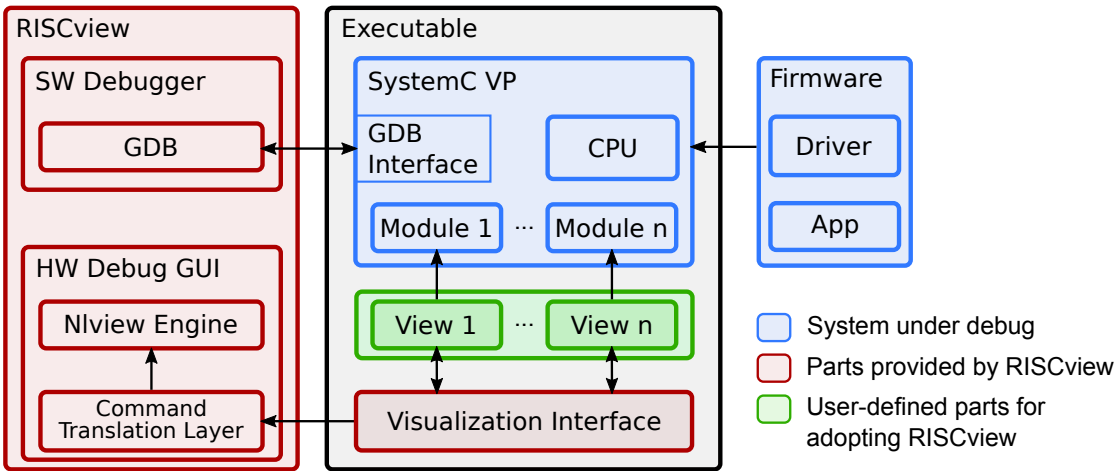


Figure 3.11: Architecture of RISCview (in red) together with a system under debug (in blue). Highlighted in green are the user-defined parts that are necessary for the adoption.

separates the VP and the displayed information, minimizing the impact of adding the proposed framework to a virtual prototype. It also reduces the interference with other automated testing systems. Additionally, areas of interest can flexibly be highlighted by dynamic reconfiguration of the views without modifying the VP. The VP exhibits a debugging interface (top left) such that a standard software debugger like GDB [114] can be used to inspect and manipulate the internal state of the software that is currently executed on the VP. It allows monitoring variables, to set breakpoints, etc.
The combination of the HW debugging GUI with a GDB instance for the executed software gives the user deep insights into and control of the interplay between HW and SW. RISCview can be used, e. g., to aid the integration process of new peripherals and matching software drivers into VPs, to visualize the existing architecture at run-time, and to analyze interrupt and timer correlations.
The proposed tool is evaluated by debugging a HAL for a newly designed OLED-screen shield for the RISC-V processor board HiFive1 [91] (cf. Subsection 3.1.8.3). The results show clearly that RISCview allows to find bugs in the HW/SW interaction more efficiently than the available alternatives.

3.3.2 Related Work

While debugging tools for later design phases exist both on the software side and on the hardware side, a lucid and easy-to-use hardware visualization tool for early virtual prototypes is not available yet. For instance, [115, 116] offer debugging tools for SoCs at gate level later in the design process. [115] emulate CPU and IPs

by implementing a GDB interface to an FPGA simulator, while [116] proposes a debug controller that can be integrated in SoCs on the final silicon chip.
Both Rogin et al. [117] and Große et al. [118] propose SystemC IDEs for low-level interactions with a focus on the signal layer. These IDEs are incompatible with transaction-level models and do not offer a live view of the system at run-time.
Since a virtual prototype is a software implementation – in this case using the C++ class library SystemC –, it is also possible to attach a software debugger like GDB [114] directly to the virtual prototype and to step this way through the software model of the hardware logic. Compared to the proposed tool, this approach has the severe drawback that it shows the variables of the SystemC implementation, but not a direct view of the modeled hardware; not to mention of the software that is running on the VP.

In summary, the existing solutions are either too late in the design process, provide no live view, or are not appropriate for debugging the hardware/software interaction. The main contributions of this section can be listed as follows:

1. An implementation-agnostic HW/SW visualization,
2. Early visual debug parallel to existing software tools,
3. A live view of the system's state during the debugging session,
4. A case-study showing that the proposed approach is well suited for finding bugs in the interaction of hard- and software.

3.3.3 Preliminaries

In this subsection, the core concepts used in the proposed tool and the following case-study are explained. It starts with the base hardware board *HiFive1* that is simulated, introduces the protocol SPI that is used for device communication, and gives an overview of NLview™, which is the visualization engine.

Sifive Hifive1 Prototype Board For a case-study, the open-source RISC-V VP (cf. Section 3.1) in its *HiFive1* mode was extended. This mode emulates the tinkering board HiFive1 of the company SiFive. The processor board comes with peripherals such as buffered SPI, DMA, and UART, which are all modeled in the virtual prototype. The VP offers two ways to debug the system: A GDB connection to the simulated CPU (software side) and a GDB session over the SystemC executable itself (hardware side). While it is possible to access the hardware IPs through a GDB session, the effort to gain information of interest is disproportionate because one has to access variables through the SystemC kernel with its user-space scheduling. The software GDB module inside the VP, however, is usable as if the RISC-V binary was executed locally.

SPI In this case-study, again the Serial Peripheral Interface (SPI) is used which was briefly introduced in Section 2.1. This protocol operates on three or four wires for data transmission between a master device and one or multiple slave devices. The bus master starts a transmission by activating the CS line of the target device, starting a clocking signal on the *CLK* line in sync with its MOSI line. Eight bits can be transferred for each burst. Depending on the use-case, the MISO line may be used to transmit data from the slave fully duplex. The master device has to actively poll slaves if no additional interrupt bit lines are used.

Nlview NLVIEW™ [44] is a commercial state-of-the-art library by Concept Engineering GmbH for creating schematic diagrams for electronic systems at different abstraction levels, ranging from transistor level via gate and RTL-level to system level. It is compatible with different GUI frameworks like Tcl/Tk, Qt, WxWidgets, and HTML5 canvas. Nlview provides Application Programming Interfaces (APIs) in C, Tcl, Java, Perl, and Python. The automatically generated schematic layout can be modified and controlled both by the APIs and by human intervention. Interactive circuit exploration is supported by Nlview's incremental schematic generation technology.
RISCVIEW uses the Nlview Tcl/Tk widget to render views of the hardware modeled in the VP together with simulation data (see Figure 3.12 for an example view). Nlview's incremental navigation features thereby allow to interactively explore the hardware views and hide irrelevant parts.

3.3.4 Implementation

To extend an existing SystemC VP, the system designer needs to add *views* for every IP module that shall be a part of the visualization. Views are abstract representations of modules containing the relevant information with high control over the module's layout. These representations act independently of the actual SystemC behavior, separating the view from the model as much as possible. The views are automatically collected by the *visualization interface*. The visualization interface translates the instantiated views and their data into a live stream of commands for the debugging GUI via TCP. During the simulation of the SystemC VP, the interface extracts updated information via the registered views asynchronously.
In the GUI, the *command translation layer* receives the commands from the visualization interface and generates appropriate API calls of the visualization engine to render the model in a graphical representation. This additional translation layer offers the flexibility of using different visual styles or levels of detail. Adding other visualization engines requires only implementing a different translation layer. In

the following case study, the industry-proven Nlview engine [44] was chosen, as it allows creating structure components and connections via Tcl/Tk commands. The combination allows an interactive exploration of the underlying model, offering an auto routing of individual nodes and a partial exploration to limit the view to the relevant parts at run-time.
As already stated, there are two GDB interfaces that can be used simultaneously: A GDB session of the simulated CPU (software side) and the SystemC executable itself (hardware side). The RISC-V binary can be loaded with GDB as a remote target to the SystemC VP. The virtual CPU inside the VP then can be halted with breakpoints and the virtual memory can be explored. Additionally, the actual VP including its numerous IP models are written in C/C++ and thus can also be debugged with the native GDB. Due to the visualization interface running in an asynchronous thread, the hardware can be inspected in real-time with both methods.

3.3.4.1 Symbols and Connections

A view has to implement at least two functions: `getSymbol()` and `update()` (e. g. see Listing 3.12). In `getSymbol()`, the view's layout such as size, shape, location of attribute fields and input/output pins is defined. This function is only called once during instantiation of the views. The actual values for the attributes are generated in the `update()` function, which is periodically called by the *visualization interface* (see Sect. 3.3.4.2). It may update the attributes of its instance and the values of all connected pins. To display useful information, the view needs a reference to the module it describes. For convenience, an auto-generation compiler macro is supplied for trivial views (non-templated models and no extra functions) to speed up the design process. How the view accesses its model is up to the designer and available interfaces; this case study passes the structure's reference in the SystemC main function during elaboration time. But it is also possible accessing the information directly over class pointers, indirect over function calls, or any other way that C/C++ allows. Lastly, a *Connection* is a meta-element to connect two or more pins and can display relevant data, which can be set by any symbol that has connected pins to it.

```
1  const Symbol GPIOView::getSymbol() {
2    Rect size = default_box;  //100x100 units
3    riscview::Pin bus{"BUS", Direction::INOUT,
4      PinLocation{Orientation::left, Point{0,1*size.y/5}}
5    };
6    riscview::Pin o12{"12", Direction::OUT,
7      PinLocation{Orientation::right, Point{size.x,1*size.y/5}}
8    };
9    [...]
10   std::map<std::string, Attribute> attrs {
11     //name, init value, lower left alignment, margin, size
12     {"regs", {"", Locator::ll, {default_attrtextsize,
       ↪ size.y-default_attrtextsize}, default_textsize/3}},
13   };
14   return Symbol("GPIO", {bus, o12, [...]}, size, attrs);
15 };
16 void GPIOView::update() {
17   std::string text = "VAL: " + toBin(model.value, 3);
18   instance.getPin("16")->getConnection()->setText(
19       model.port & (1 << 10) ? "1" : "0");
20   instance.setAttribute("regs", text);
21 }
```

Listing 3.12: Example view building pins and attributes of a general purpose I/O (GPIO) hardware module (cf. the resulting symbol in Figure 3.12). This C++ description is translated into Tcl/Tk commands that are then streamed to the renderer.

3.3.4.2 Visualization Interface

The visualization interface provides a library of usable layout objects (e. g. `Symbol`, `Direction`, `Orientation`, etc.), a registration function for all views, and an own update thread. At program start-up, the interface tries to connect to the *command server* of the debugging GUI over a TCP connection. If no connection is possible, all further view-related function calls are ignored and the SystemC program continues as normal. Otherwise, the registered layout objects are serialized into individual commands and sent to the command server.
Every module and connection needs to be defined in an elaboration phase. This definition allows setting the size of the module, location and names of input or output pins, and the layout of attributes along with a unique identification (see Listing 3.14). These properties cannot be changed after the instantiation to allow the visualization engine to place modules in a space-efficient manner. When the SystemC simulation starts, the update thread starts polling all registered instances periodically and checks for changed attributes. All changed attributes can be updated via their respective identification strings and are then serialized and sent to the command server (see Sect. 3.3.4.3).
Note that the update thread is independent of the SystemC simulation, and thus does neither affect nor is affected by the simulation time. Since the used

```
#define GEN_DEFAULT_VIEW(CLASS)
struct CLASS##View : public Viewable {
  static const Symbol symbol;
  Instance instance;

  static const Symbol getSymbol();

  CLASS &model;
  CLASS##View(CLASS &model, string name = #CLASS);

  void update() override;
};
```

Listing 3.13: Trivial class structure of a default view. Trailing slashes are omitted for readability.

implementation of Accellera SystemC [60] is single-threaded, the impact of the proposed debugger on the simulation speed can be neglected when run on a multi-core system.

```
GPIO gpio0("GPIO0", INT_GPIO_BASE);  // SysC HW-Model
RV_DEF_AND_ADD(GPIOView, gpio0);     // View
SPI spi1("SPI1");
RV_DEF_AND_ADD(SPIView, spi1);
SS1106 oled([...]);
RV_DEF_AND_ADD(SS1106View, oled);

riscview::Connection gpio_oled_dc("GPIO-OLED-DC");
gpio_oled_dc.connect(gpio0_v.instance.getPin("16"));
gpio_oled_dc.connect(oled_v.instance.getPin("DC"));
riscview.add(gpio_oled_dc);
[...]

if(!riscview.connect()) exit(-1); // connect with GUI
std::thread updater([&riscview]{  // start RISCview thread
  while(true) {
    ViewableRegistrar::updateAll(riscview);
  }
});
sc_core::sc_start();              //start SysC thread

if(!nlv.init()) exit(-1);
if(!nlv.show()) exit(-2);
std::thread updater([&nlv]{
while(true)
{
  ViewableRegistrar::updateAll(nlv);
  usleep(250000);
}
```

Listing 3.14: Excerpt of an initialization list of HW-modules and their views. `RV_DEF_AND_ADD()` is a compiler macro that instantiates and registers a view, naming it with the suffix `_v`.

3.3.4.3 Debugging GUI

The debugging GUI is responsible for collecting visualization commands and drawing appropriate structures. For interchangeability of the graphical representation, the visualization commands are based on Tcl/Tk. The GUI opens a server at start-up and listens for incoming commands from the visualization interface. These commands are then translated by the *command translation layer* into API-calls to the graphics engine. The Nlview visualization engine includes a placement algorithm to minimize the needed screen size and concisely routes the connections between the components.

3.3.5 Case Study

As a case study, an OLED display was implemented as a HW module into the existing open source RISC-V VP (see Section 3.1) along with a software driver to interface the display. The VP is able to model the SiFive HiFive1 processor board including some of the most used peripherals (i. e. UART, SPI, timers). The SS1106 OLED display driver is a multi-protocol driver (SPI 3-wire, SPI 4-wire, I^2C and others) supporting monochrome displays with up to 64 by 132 pixels resolution. The SPI 4-wire connection was chosen, because it has the fastest net transmission capabilities. To show the real-world comparability, also a PCB was designed with an OLED display and seven buttons. The PCB was build so that it can be stacked on top of the HiFive1 board.

3.3.5.1 Display HW Model

The display-driver model was developed according to the corresponding datasheet [112] and connected to the HiFive's SPI peripheral in the RISC-V VP (see Subsection 3.3.3). The SPI 4-Wire mode requires a differentiation of `command` and `data` bytes via a dedicated pin connected from the GPIO module to the display. Commands may consist of one to three bytes and expect up to two trailing value bytes. For instance, to set the display's contrast, the command line has to be set low, and the byte `0x81` for *set contrast* along with the value (encoded in one byte) has to be sent over SPI. Issuing multiple data bytes after a `PAGE_ADDR` command are interpreted as consecutive pixel values, incrementing the internal pixel pointer state. To aid with the design process, a view of display driver was also created showing sent data, the last command, and an excerpt of internal state. The implementation of this view took only 23 lines of code (see Listing 3.15).

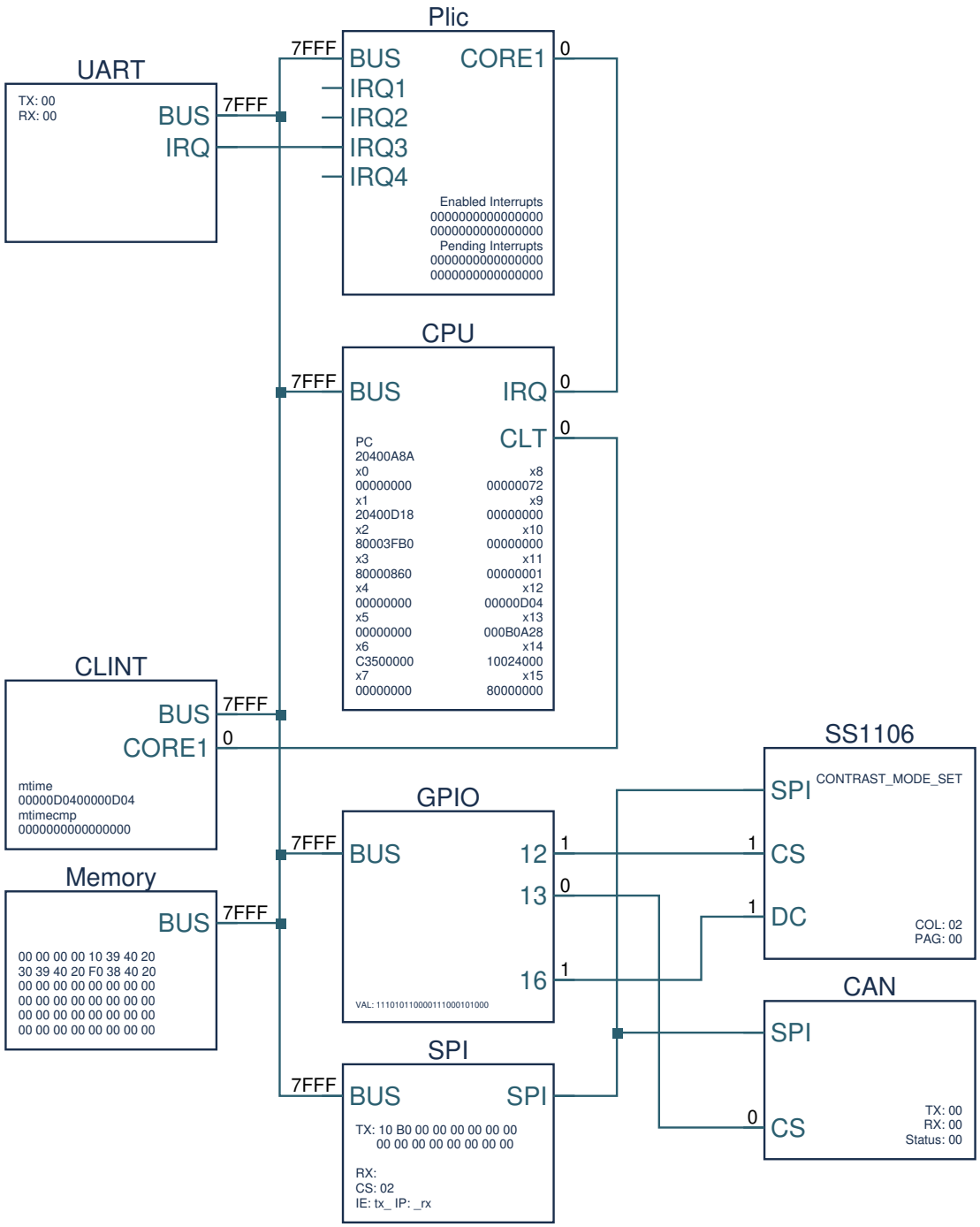


Figure 3.12: Screenshot of the architecture view in RISCview.

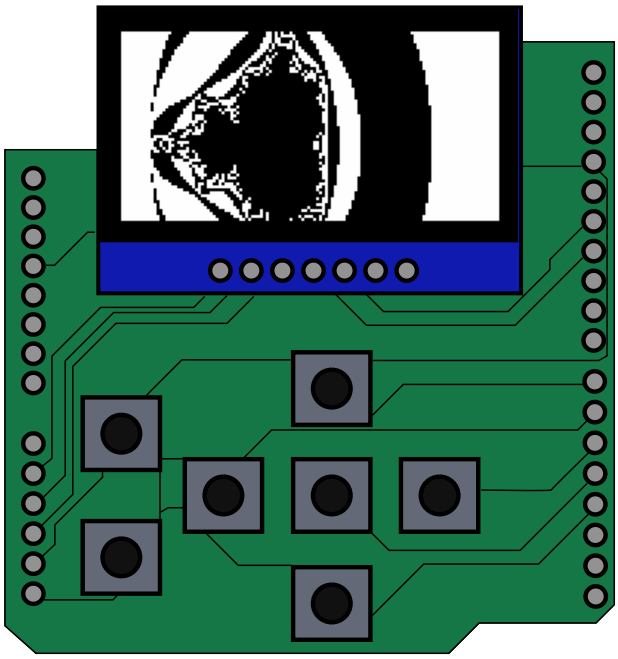


Figure 3.13: Screenshot of the VP simulation with an active OLED Display running an example program.

```
1  GEN_DEFAULT_VIEW(SS106);
2  const Symbol SS1106View::getSymbol() {
3    Rect size = default_box;
4    std::vector<nlv::Pin> pins = {
5      nlv::Pin {"SPI", Direction::INOUT,
6      PinLocation{Orientation::left, Point{0,  size.y/5}}},
7      nlv::Pin { "CS", Direction::IN,
8      PinLocation{Orientation::left, Point{0,  size.y/2}}},
9      nlv::Pin { "DC", Direction::IN,
10     PinLocation{Orientation::left, Point{0,4*size.y/5}}},
11   };
12   std::map<std::string, Attribute> attrs {
13     {"command", {"", Locator::lr, {size.x-default_attrtextsize,
       ↪ 1.5*default_textsize}, default_attrtextsize}},
14     {"regs", {"", Locator::lr, {size.x-default_attrtextsize,
       ↪ size.y-default_attrtextsize}, default_attrtextsize}},
15   };
16   return Symbol("SS1106", pins, size, attrs);
17 };
18
19 void update() {
20   std::string text = "COL: " + toHex(model.state->column) +
21   "\nPAG: " + toHex(model.state->page);
22   instance.setAttribute("command", ~model.last_cmd.op);
23   instance.setAttribute("regs", text);
24 };
```

Listing 3.15: Code to generate a view for the SS1106 Controller (cf. Figure 3.12).

3.3.5.2 Display SW Driver

The SW driver offers a set of high-order functions like *set pixel at position* x and *draw line from point* x *to* y and translates them to series of low-level commands for the display. It also manages the values for GPIO-Pins and handles the SPI peripheral interface, both over memory-mapped I/O.

3.3.5.3 Debugging

During development of the software driver, undefined behavior of the display could be noted during operations with a high pixel-throughput. Sometimes, the display glitched in a way that the image was distorted or showed random artifacts (see Figure 3.14).

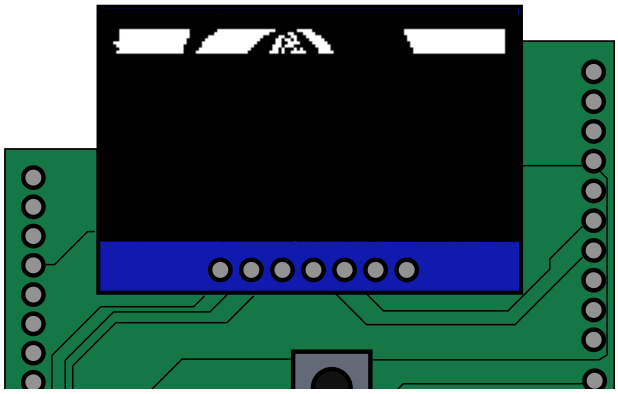


Figure 3.14: Glitched display showing only a partial image and distorted lines. This simulation behaves exactly like the real HiFive1 board with the custom PCB.

```
1  void mode_data(void) {
2    setPin(OLED_DC, 1);
3  }
4  void mode_cmd(void) {
5    setPin(OLED_DC, 0);
6  }
7  void setContrast(uint8_t contrast) {
8    mode_cmd();
9    spi(0x81);   //Command: next byte is contrast value
10   spi(contrast);
11 }
12 void oled_init() {
13   spi_init();
14   // Initial setup
15   // Enable RESET and D/C Pin
16   GPIO_REG(GPIO_OUTPUT_EN) |= (1 << mapPinToReg(OLED_RES) | 1 <<
       ↪ mapPinToReg(OLED_DC));
17   setPin(OLED_DC, 0);
18
19   // RESET
20   setPin(OLED_RES, 0);
```

```
21    sleep_u(10);  // at least 10us
22    setPin(OLED_RES, 1);
23    sleep(100);   // at least 100ms
24    // Initialize display to desired operating mode.
25    [...]
26    setChargePumpVoltage(0b10);
27    setContrast(0xff);
28    // Clear screen (overwrite entire memory with zeroes)
29    oled_clear();
30    setDisplayOn(1);
31  }
```

Listing 3.16: Part of the original SS1306 display software driver.

The first approach to finding this bug was starting the simulation with a breakpoint on the software side in the display driver routine that handles the SPI transfers. However, this did not yield any results, because the simulation did not show any false behavior as long as the breakpoint was active. Also, printing out the SPI bytes over the serial monitor suppressed the undefined behavior. The second approach was to set a breakpoint in the display module (hardware side) at the command interpretation state machine. It could be noticed that the display driver got invalid command bytes that were not implemented in the software driver. Also, the display got too many consecutive data bytes, thus writing out of bounds of its page buffer. The execution of the SystemC executable was then paused with a breakpoint, instructing to halt when the display detected an invalid command. By inspecting the RISCview window (Figure 3.14), it could be seen that the transmit 'X' (TX) queue of the SPI module still contained command bytes, but the D/C-line was already set high (`data mode`). In this state, the display's state machine still expected a second command byte for the contrast value (`CONTRAST_MODE_SET`). This observation led us to the idea that the switch between data and command mode did not wait until the whole SPI transmit queue was emptied. It also explained why a debug print in the software driver suppressed the problem; the time it takes to send text through the comparatively slower UART was enough for the SPI TX queue to run empty.

The fix itself required only a few lines to change: Before switching between data- and command mode, wait for the lower SPI transmit watermark (`SPI_IP_TXWM`) to indicate an empty transmit queue (see Listing 3.17).

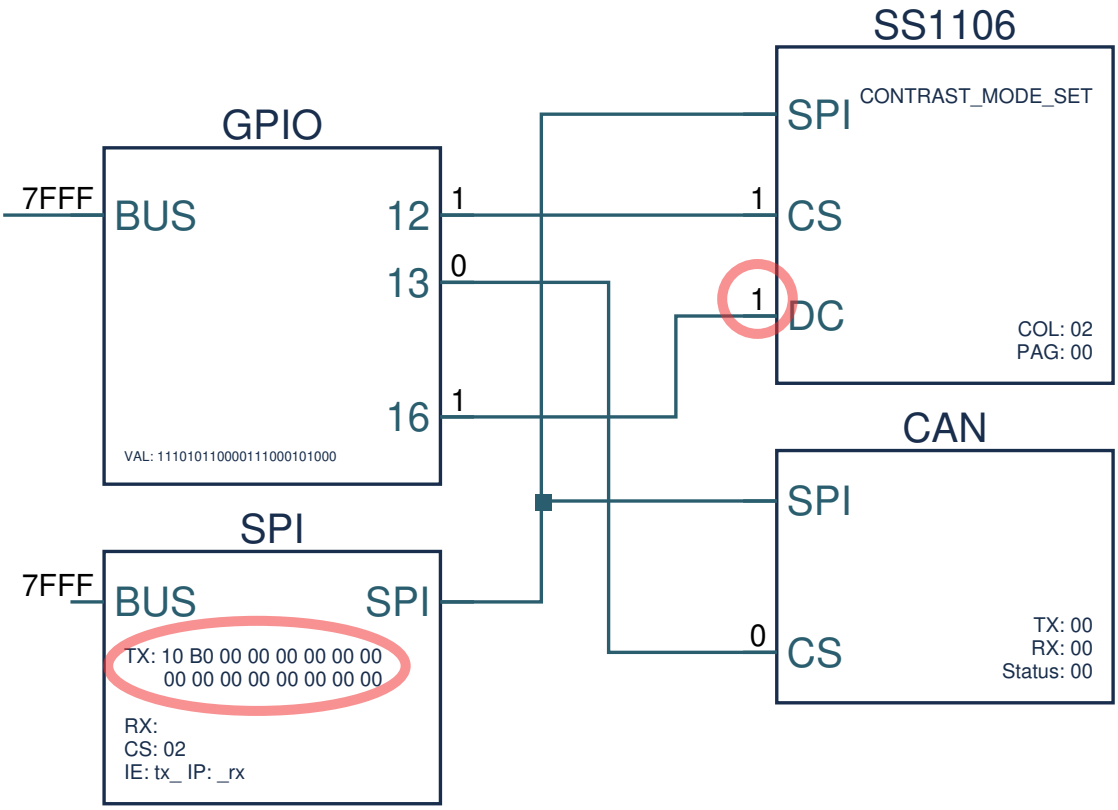


Figure 3.15: Snapshot of a still command-populated TX queue, although Data/-Command line just toggled to data mode. Note the populated TX buffer in the SPI peripheral, where the top left byte is the first to be transmitted. The first two are still commands: `0x10` for the contrast value and `0xB0` for the charge pump voltage. Following bytes are all zeroes to clear the screen. Additional status flags indicate that the receive 'X' (RX) queue is empty, CS is set to device `02` (SS1106), and the TX interrupt is enabled but not pending.

```
1  void spi_complete() {
2    // Wait for interrupt condition.
3    while (!(SPI1_REG(SPI_REG_IP) & SPI_IP_TXWM))
4      asm volatile("nop");
5    // TX-Watermark is set while byte is still in transit
6    // One byte at 8KBit/s is one microsecond
7    sleep_u(1);
8  }
9
10 void mode_data(void) {
11   // not already in data mode
12   if(!getPin(OLED_DC)) {
13     // wait for SPI to complete before toggling
14     spi_complete();
15     setPin(OLED_DC, 1);
16   }
17 }
18
```

```
19 void mode_cmd(void) {
20   // not already in command mode
21   if(getPin(OLED_DC)) {
22     // wait for SPI to complete before toggling
23     spi_complete();
24     setPin(OLED_DC, 0);
25   }
26 }
```

Listing 3.17: Fixed part of the software driver.

3.3.5.4 Evaluation

If one had used just the normal GDB debugger, the underlying problem would not have been clear. When the program is halted at the memory interface of the display module, the access to the state of the SPI module is hidden behind the stack-frames of the different user-space threads of SystemC. The encountered bug was also noticeable in the real hardware, which shows the accuracy of the provided case-study.

3.3.6 Conclusion and Future Work

This section has presented a novel system for hardware/software co-debugging that is applicable in an early stage of the development with a minimal impact on design-time. Using a transaction-level virtual prototype of the hardware, written in SystemC, it provides a live view on the internals of the hardware design, while stepping through the executed software using a state-of-the-art software debugger like GDB. The integration into a project requires little adaptation to the code-base with a flexible view on the hardware. A case study with a modeled OLED-Display operated by a RISC-V processor demonstrated the usefulness of the proposed visualization for finding bugs related to hardware-software interactions.
While the proposed visualization system works as intended, it also opens up possible future work, including:

- Combining the system with a dynamic flow analysis framework like [45] to visualize security policy violations and data flow in real-time;
- Adding a static code analysis based on re-occurring SystemC class patterns, which would enable automated visualization of modules at the expense of displaying possibly irrelevant information;
- Implementing a hardware-version of the visualization interface to permit hardware debugging with the real hardware in the same style as its VP.

3.4 Hardware-In-The-Loop Framework to Bridge the VP/RTL Design-Gap

The following section contains unpublished material that was submitted to the *CODES+ISSS Embedded Systems Week 2023*. It starts by an extended introduction and motivation in Subsection 3.4.1 and an overview of related work in the field of HWITL in Subsection 3.4.2. The main approach with the protocol and the bridge implementations is described in Subsection 3.4.3. Following is an experimental setup for the evaluation and case-studies in Subsection 3.4.4 where different aspects of the VPIL approach are evaluated. Subsequently, the lessons-learned and general considerations are discussed in Subsection 3.4.5. Lastly, further improvements and future applications are summarized in the outlook in Subsection 3.4.6.

3.4.1 Introduction

Modern SoC development is driven by the ever rising demand of a faster time-to-market for highly integrated and complex designs which are subject to phenomena known as the design gap and the verification gap [1, 2]. Briefly, the design gap describes the discrepancy between an increasing capability of chip manufacturers, making more transistors per chip available, while the design engineers can not make use of that large amount of available transistors in a design [3]. Similarly, the verification gap is characterized by the increasing need for verification of chip designs versus the available state-of-the-art technology for verification [119]. Solving these issues has been the subject of research in the past decade.
esign methodologies like ESL offer additional abstractions help to accelerate the design and verification cycles [120–123]. One of the abstractions within the ESL methodology is TLM. TLM basically abstracts away the multiple events required for a communication (like synchronizing transmitter / receiver, bit-encoding of instructions, and bus communication itself) into single transactions.
To further enhance the process of ESL based systems development a central technique is the VP. Unlike pure ISSs, VPs model the structural and behavioral interaction between the processing units, the bus system and peripherals. When VPs are modeled with the help of the TLM abstraction, they can gain high simulation speeds that enables booting operating systems, while analyzing the hardware interaction and exploring the design space efficiently. Through VPs, the SoC design flow is improved and allows the development of the SW in parallel to the HW. As a consequence of the parallel SW / HW development, the time-to-market is improved and verification strategies can be employed early in the

development process. Furthermore, this allows the development team to spend more time in developing the USP of the product.
The use of VPs and TLM are two key elements of the ESL methodology and are considered industry proven for the purpose of bridging the design and verification gap. With VPs representing executable models of the specification, HW and SW engineers both have well-defined interfaces with a given reference model to work with. While the VP will be binary compatible with the final HW (*behavioral* model), the actual HW usually is designed with the help of RTL abstractions. These RTL models can be synthesized automatically to the final layout of the transistors. While the design and verification gap can be bridged with the ESL methodology, VPs and TLM, a gap between the TLM and RTL emerges as a visible challenge. Although there exist efforts in academia and industry to automatically generate RTL code from TLM representations, these techniques are mostly proprietary and not widely prevalent. Instead, most of the implementation is done through manual RTL coding or with the aid of High-Level Synthesis. On the road to a gap-free design and verification methodology, so called cross-level techniques have been proven to be a viable option. Cross-level techniques that emerged in recent years combine TLM *and* RTL representations for design and verification to leverage the advantages of the ESL methodology, specifically TLM and VPs. In this section, a technique is presented to augment VPs with HWITL to add to these techniques and lessen the TLM/RTL gap.
Between the available VP and the final RTL model, usually parts of the system can already be considered to be ready for test and verification, or even signed-off as complete. But without the full RTL model available, though, these parts cannot be used and tested on the real HW. With the proposed methodology, the available, synthesizeable RTL modules can be integrated to run on an FPGA while the rest of the system is still running in the VP. Through this technique, the TLM/RTL gap is bridged, as incremental results can be tested early in an integrated setup running with the real SW applications. This reduces the time-to-market further, as integration efforts for the final RTL can be reduced and the focus can be shifted to the *USP* of the system (e. g. a specific HW accelerator). Thus, as modern SoC often contain an extensive library of readily available IP, the development of the USP can become the point of interest.
Concluding, this section proposes a HWITL methodology for the SoC development flow. It connects TLM VPs with memory-mapped RTL components (specifically the USP) while the remainder of the system's HW is still in development. The integration of the RTL component is realized with a FPGA commonly used for HW prototyping. For the SW running on the SoC VP, and the HW itself, it is fully transparent, such that integration tests can already start before the complete system is modeled in synthesizable RTL. A brief overview of the approach is

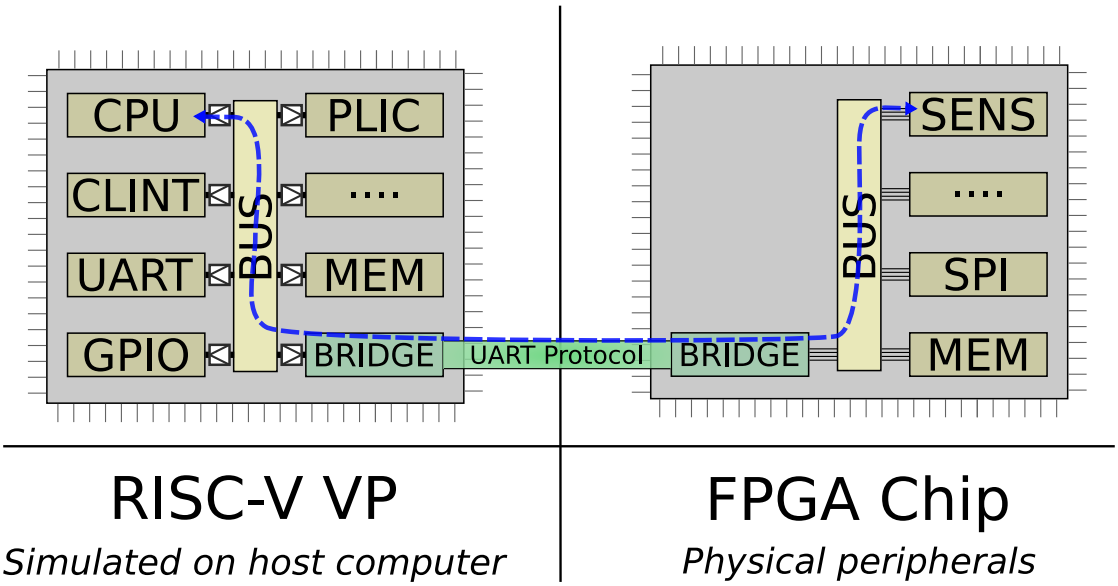


Figure 3.16: Architecture level overview of the proposed Virtual Peripheral in-the-Loop approach. On the left side is the TLM virtual prototype with a memory-mapped bridge (in green) as the *initiator*. The right side represents the real hardware with the *responder* bridge handling the bus accesses. In blue is plotted a possible data flow path from the virtual CPU to a real sensor RTL implementation.

illustrated in Figure 3.16. A VP model (left side) is interconnected with the FPGA based prototype (right side) through the proposed bridge (green). The VPIL bridge allows the FPGA based prototype to work without an own CPU, as this is part of the VP in this example. The proposed VPIL enables a HW/SW co-design boost to close the TLM/RTL gap with an easily extensible protocol and an IP library of existing and well-established VP components. VPIL allows system designers to focus on their USP from the design space exploration phase to early system integration tests, as well as aiding the verification with the possibility of cross-level model verification. Additionally, it offers using the SW-based debugging capabilities for the TLM models, even if the synthesized chip will not have them.
In a case-study, the RISC-V VP [38] is utilized and extended with a peripheral bridge, allowing TLM bus transactions to be executed on a RTL model residing in an FPGA (see Figure 3.16). The experiments demonstrate how a lightweight and extensible protocol can be employed to establish a bridge between the VP and the FPGA HW. Considerations are discussed that arise in the implementation of the proposed bridge and what parts of the bridge can be adapted in order to fit various requirements (fast setup time, robustness, high throughput / low latency, etc.). Finally, to stimulate further research, the proposed HWITL bridge along

with the implementation and test results of the case-study is made available as open-source [51] as most of the work in this thesis.

3.4.2 Related Work

HWITL is a dynamic testing method that combines real HW, simulated environments, and integrated software [124], where usually the real hardware (once available) and placed in a simulation loop where the outside environment is simulated [125, 126]. The strengths of this methodology enables early integration, higher quality tests as developed HW can partially be used together with the simulation environment. As a method, HWITL has experienced a broad range of industrial and academic interest in the modeling of electronics [125, 127–130], automotive [131, 132], aerospace [133] and other multi-disciplinary fields [134–136]. In [125] the authors present a survey of past HWITL approaches and the challenges in respective approaches. This work mostly focuses on works of electrical engineering, in which control algorithms are simulated and HW is controlled in a loop. In [128–130, 132, 133], HWITL is utilized such that the SW integrated in the final HW is simulated (e. g. the control algorithm for a plant) in the loop with prototypes of the HW. Such approaches are common and allow for SW and control engineers to explore the design parameters for the control algorithm and validate and verify the control algorithms. [131] shows an industrial approach for virtual HWITL for automotive use, that focuses on higher level models to be integrated easily in HWITL environments. [136] present a methodology that offers simulation and automatic optimization for distributed Cyber-Physical Systems (CPS) that utilizes FPGA-based HWITL.
[127] describe an approach different from the mentioned ones as here the SoC is the HW and the environment (e.g. sensor and actuator data processing are handled in a simulated environment). [135] follow an approach similar to [127] but addresses real-time challenges regarding data exchange and synchronization.
While none of these works directly mention a HWITL methodology for the SoC development process, they address challenges and approaches that are common across HWITL. Often, the SW part of the embedded system to simulate with HWITL is designed either as a control system (e. g. with MATLAB/Simulink), or emulated otherwise. In these fields, a complex embedded system is commonly tightly coupled with a CPS, consisting of sensors, actuators and mechatronic systems. The simulation of the CPS often utilizes complex simulation systems (e. g. with MATLAB/Simulink), such that either the embedded system (e. g. a SoC) or the physical system are modeled with HW and respective HW prototypes. Through this effort, the modeling complexity within the realm of simulations can

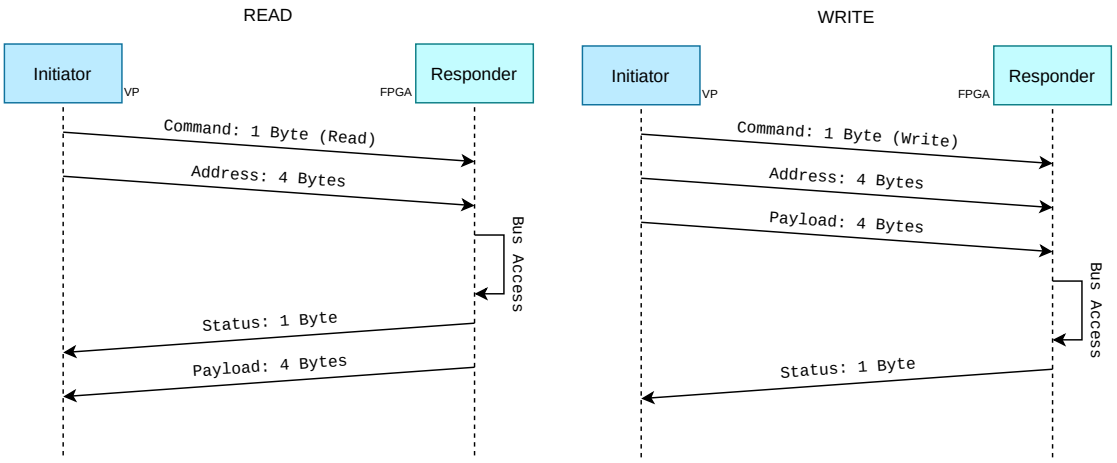
be decreased and engineers can focus on design, integration, test and verification tasks.
With technological advancements and the increase in complexity of embedded systems (specifically SoCs), industry and academia have observed the aforementioned design and verification gaps. Naturally, engineers and researchers refine existing the design and verification processes and establish new methodologies to anticipate the gaps. Previously, there existed no peripheral-centric HWITL approach for SoCs. One of the reasons for this may be the lack of full system hardware simulations, that only recently have been established by academia and industry. For full system hardware simulation VPs can be considered as an accepted methodology for complex SoC design and verification.
Due to the usage of VPs in this work, a short overview of related works to VPs will be given as well.

3.4.3 Approach Overview

As stated earlier, the key idea of the proposed approach is to enable developing individual RTL models without the need of designing the complete SoC first. Following Figure 3.16, early RTL development may already start with existing VP TLM models (left side). By leveraging the proposed VPIL protocol with the reference implementations for the RISC-V VP and a generic FPGA, individual peripherals can directly implemented in RTL (right side). Bus accesses to external peripherals, initiated by the CPU, DMA, or other bus masters on the RISC-V VP, are directly mapped via the virtual bus bridge (see Subsection 3.4.3.2) and the UART protocol (Subsection 3.4.3.1). Additionally, these forwarding memory ranges can be easily changed in the RISC-V VP via program options, allowing simultaneously existing SystemC models and on-FPGA implementations of the same IP. This enables a fast and easy way of behavioral cross-level testing and debugging.
The following subsections present the details of the proposed approach. Starting in Subsection 3.4.3.1, an introduction is given into the serial protocol between the VP and a remote hardware that allows mapping these accesses to devices on the FPGA. Subsection 3.4.3.2 then continues on a high abstraction level by explaining how virtual peripherals are connected to the VP and how bus accesses are routed transparently. Finally, in Subsection 3.4.3.3, the implementation of the protocol decoder and bus handling on the FPGA is demonstrated.

3.4.3.1 Protocol

The proposed VPIL protocol is a lean and hardware-parsing-friendly protocol between an *initiator* (the VP) and a *responder* (the FPGA implementation). The initiator will always start an interaction with a `Command` and an `Address` in network-order endianess. Depending on the command (e. g. a *read*), the initiator will also transmit an `Address` (cf. Figure 3.17b). The responder will always respond with a `Status` that contains an acknowledgment field and a flag whether an interrupt is pending or not. In the case of a *read*, it will also contain a 4 byte `Payload` in network order.
Besides payload data handling, a `Command` may also poll for an interrupt, reset the FPGA, and initiate other actions that are reserved for future-use (see Listing 3.18, Lines 5 to 12). In case of *reset* and *getPendingIRQs*, no payload is sent in the request. The *reset* command forms a special case; as the responder immediately resets itself, no `Address` is sent in the request, and no response is given. All other commands will carry the 4-byte `Address` field and expect at least one byte of `ResponseStatus`.



(a) A *read* request. The 4-Byte *Payload* is always sent, including in error conditions.

(b) A *write* request.

Figure 3.17: Flow diagrams of two requests from the *initiator* to a *responder*. All individual fields are encoded in *little endian* network order.

```
1  typedef uint32_t Address;
2  typedef uint32_t Payload;
3
4  struct __attribute__((packed)) Request {
5    enum class Command : uint8_t {
6      reset = 0,
7      read = 1,
8      write,
9      getPendingIRQs,
10     setTime,
11     exit
12   } command;
13   Address address;
14 };
15
16 struct __attribute__((packed)) ResponseStatus {
17   /*
18    * Ack: bits 0 to 6
19    * irq_waiting: bit 7
20    */
21   enum class Ack : uint8_t {
22     never = 0,
23     ok = 1,
24     not_mapped,
25     command_not_supported
26   } ack : 7;
27   bool irq_waiting : 1;
28 };
29
30 struct ResponseRead {
31   ResponseStatus status;
32   Payload payload;
33 };
34
35 struct ResponseWrite {
36   ResponseStatus status;
37 };
```

Listing 3.18: Exerpt of the protocol data types. This is used by the initiator and the mock-up responder host programs.

3.4.3.2 Peripheral Bridge

The peripheral bridge is the SystemC TLM implementation of the *initiator*. This bridge acts as a common memory-mapped bus-slave peripheral in the RISC-V VP. Every access, however, is forwarded transparently through the communication protocol (see Subsection 3.4.3.1) to a connected *responder* bridge. At this stage, the initiator bridge is unaware whether the responder is implemented in an FPGA via a serial connection, or just simulated in a separate host process. This simplifies the testing and debugging of the protocol, and also allows extensions for future applications.

```
1  void VirtualBusMember::transport( tlm_generic_payload &trans, sc_core::sc_time
   ↪ &delay) {
2    tlm_command cmd = trans.get_command();
3    unsigned addr = trans.get_address();
4    auto len = trans.get_data_length();
5
6    hwitl::Payload temp = 0;
7    hwitl::Payload* data = &temp;
8    const bool unaligned = len != sizeof(hwitl::Payload);
9    if(!unaligned) {
10     data = trans.get_data_ptr();
11   } else {
12     if(cmd == TLM_WRITE_COMMAND)
13       memcpy(data, trans.get_data_ptr(), len);
14   }
15
16   if (cmd == TLM_WRITE_COMMAND) {
17     const auto response = bus_bridge.write(base_address + addr, *data);
18     switch(response) {
19       case hwitl::ResponseStatus::Ack::ok:
20         break;
21       case hwitl::ResponseStatus::Ack::not_mapped:
22         trans.set_response_status(TLM_ADDRESS_ERROR_RESPONSE);
23         break;
24       default:
25         trans.set_response_status(TLM_GENERIC_ERROR_RESPONSE);
26     }
27     delay += m_write_delay;
28   } else if (cmd == TLM_READ_COMMAND) {
29     const auto response = bus_bridge.read(base_address + addr);
30     if(!response) {
31       trans.set_response_status(TLM_GENERIC_ERROR_RESPONSE);
32       return;
33     }
34     switch(response->getStatus()) {
35       case hwitl::ResponseStatus::Ack::ok:
36         *data = response->getPayload();
37         break;
38       case hwitl::ResponseStatus::Ack::not_mapped:
39         trans.set_response_status(TLM_ADDRESS_ERROR_RESPONSE);
40         break;
41       default:
42         trans.set_response_status(TLM_GENERIC_ERROR_RESPONSE);
43     }
44     if(unaligned)
45       memcpy(trans.get_data_ptr(), data, len);
46
47     delay += m_read_delay;
48   }
49 }
```

Listing 3.19: The simplified transport function of a virtual bus member using the initiator bridge. Read and write accesses are mapped through the TLM-agnostic `bus_bridge`.

The implementation is written in SystemC TLM, as can be seen in Listing 3.19. Read and write accesses to the peripheral are mapped through the initiator bridge, while most of the code is just for possible alignment of the four bytes of the protocol's payload (Lines 6 to 14, and Lines 44 to 45) and error handling (in the switch-cases). The protocol handling, including the byte order packing, is completely wrapped

by the virtual bus protocol implementation `bus_bridge`. Not shown, for brevity, is the SystemC thread that periodically polls the *responder* via the `bus_bridge` for pending interrupts that are then forwarded to the PLIC.

3.4.3.3 FPGA Implementation

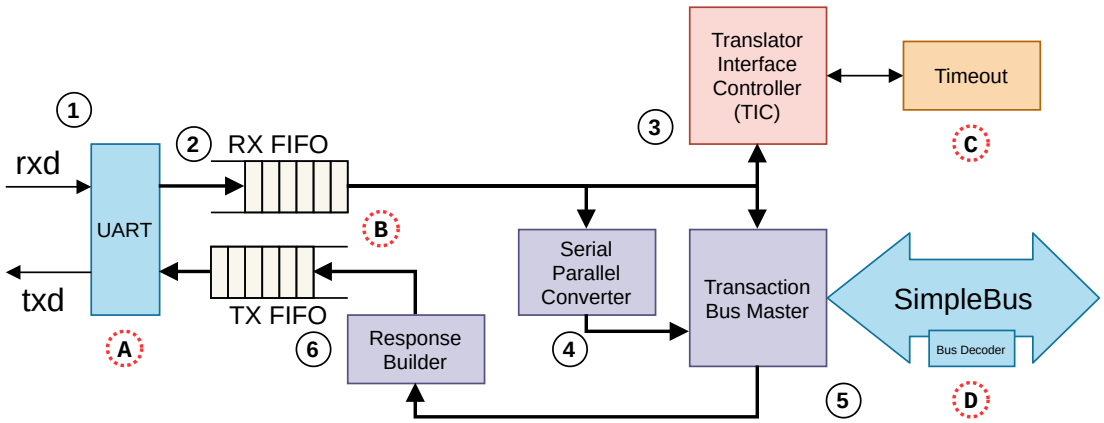


Figure 3.18: The FPGA implementation of the responder bridge. Modules in blue are for interfaces, models in purple represent internal modules handling communication between interfaces, and red / orange modules are for orchestration and control. The response buildup time is in the proposed implementation always under one millisecond.

Figure 3.18 shows a block diagram of the internal hardware architecture of the translation bridge. The explanation will follow along the enumeration from ① to ⑤ in Figure 3.18 for the hardware modules and provide further information following the enumeration from (A) to (D). Additionally, the modules are grouped in colors to differentiate them easily for the reader. Modules with the color blue are for interfaces, internal modules in purple are handling the bytes between interfaces, modules for orchestration and control are red and orange.
The UART interface (①, blue, left side) provides the byte wise serial communication with the host executing the VP. This interface was chosen exemplary for this case study, but any interface that will provide raw bytes to the internal First-In-First-Out (FIFO) buffer can be used. Next the received bytes are stored in a FIFO ②, from where the bytes are directed respectively according to the protocol as discussed in Subsection 3.4.3.1. At the heart of the hardware implementation is the Translator Interface Controller (TIC) (③, red, top right). The TIC orchestrates and parses the bytes according to the defined protocol, relays transactions coming from the VP through the bus master (④, purple, center), and handles errors like unmapped address responses and exception cases in the protocol. Upon too much

delay or other unaccounted exceptions that could stall the hardware through the interfaces, a timeout will return the system, initiated by the TIC, into a defined initial state. As addresses and data arrive in bytes a converter (④, purple, center), pre-processes them into chunks of 32 bit words for the transaction bus master (⑤, purple, center). The transaction bus master is attached to the SoC bus, onto which the hardware peripherals designed as RTL modules are attached. After processing the received transaction, the response builder (⑥, purple, center) generates a protocol conform response. The TIC handles the different cases (including errors) and can instruct the response builder to generate appropriate packets. Generated packets are passed byte-wise into a FIFO, as the serial interface transmits the data at a different rate than the packets are generated at.

A: UART was chosen as the serial interface, as it is a readily available physical layer protocol that can be extended and replaced if the requirements demand for more speed, robustness or other modes of operations. The hardware for the serial interface can be configured for various baudrates (e. g. 115 200 baud), bit modes and additionally provide RS-232 conforming control flow signals to allow further robustness already.

B: The FIFOs for the receiving and transmitting end are designed to be configurable for the requirements that stem from the different data rates on the serial interface and the internal processing speed.

C: For additional robustness and a configurable timeout is included. By default, it is set to 2 ms. If no event (such as incoming bytes, change of bus state, etc.) occurs, the timeout instructs the TIC to reset the systems state to the initial values to provide a clean and defined start.

D: The internal bus interface for the case study provides an easy to use and extensible bus configuration.

Listing 3.20 provides an example how through the abstractions of SpinalHDL new peripherals can be easily included on the bus with a respective bus address range. In Line 2 a list is declared, that will hold tuples of references to peripheral bus interfaces, the respective select signal and a bus mapping. In Lines 4 to 6, a peripheral is added by appending the list with its bus interface, select signal and respective memory mapping. The bus mapping (e. g. `MaskMapping(0x50000000l, 0xffffff0l)`) uses a base address and a respective mask to allow the decoder to check if a requested address maps to the peripherals base address. This is repeated between Lines 8 to 19 for further peripherals and finally the list is passed in Line 22spinal-interconnect:decoder-end into the bus decoder. The bus decoder will interconnect the bus master and the list of bus interfaces according to the respective bus mapping. This additional abstraction, made available through the features of SpinalHDL, provides an easy extension mechanism and makes the HDL code easy to follow.

```
1  // ******** Peripherals *********
2  val busMappings = new ArrayBuffer[(SimpleBus,(Bool, MaskMapping))]
3
4  val gpio_led = new GPIOLED() // onboard LEDs
5  busMappings += gpio_led.io.sb -> (gpio_led.io.sel,
     ↪ MaskMapping(0x500000001,0xffffff01))
6  io.leds := gpio_led.io.leds
7
8  val gpio_bank0 = new SBGPIOBank() // GPIO for IO switches
9  busMappings += gpio_bank0.io.sb -> (gpio_bank0.io.sel,
     ↪ MaskMapping(0x500010001,0xffffff01))
10
11 val gpio_bank1 = new SBGPIOBank() // GPIO for LEDs, etc.
12 busMappings += gpio_bank1.io.sb -> (gpio_bank1.io.sel,
     ↪ MaskMapping(0x500020001,0xffffff01))
13
14 val uart_peripheral = new SBUart() // uart 9600 baud
15 busMappings += uart_peripheral.io.sb -> (uart_peripheral.io.sel,
     ↪ MaskMapping(0x500030001,0xffffff01))
16 uart_peripheral.io.uart <> io.uart0
17
18 val gcd_periph = new SBGCDCtrl()
19 busMappings += gcd_periph.io.sb -> (gcd_periph.io.sel, MaskMapping(0x500040001,
     ↪ 0xFFFFFF001))
20
21 // ******** Master-Peripheral Bus Interconnect *********
22 val busDecoder = SimpleBusDecoder(
23     master = busMaster.io.sb,
24     decodings = busMappings.toSeq
25 )
```

Listing 3.20: SpinalHDL digest of top level peripheral bridge. Digest shows how new peripherals can be easily added to the bus infrastructure.

3.4.4 Evaluation / Case-Study

For the case-study of the proposed approach we, envision two scenarios. First, an incremental development in which modules are added throughout development stages, to aid the general development process for SW and HW developers (as real HW descriptions become available but are not existing yet as a full system). Second, a focused development/refinement of the USP of a new system (e. g. a specialized HW accelerator). State-of-the-art systems rely on a backbone of a rich and well-tested IP library for the common, reoccurring modules. This circumstance enables engineers to design more complex and powerful systems offering USPs that competing chips do not offer (e. g. specialized accelerators for cryptography, artificial intelligence, etc.). Putting the focus to the USP and making it available to the SW developers and verification engineers, who then can design and verify the respective firmware much earlier in the design process.
For the evaluation study, we use a HX8K FPGA from Lattice Semiconductor on a respective development board (see Figure 3.19). The development board features on-board LEDs (lower part of the FPGA's PCB) and many mappable

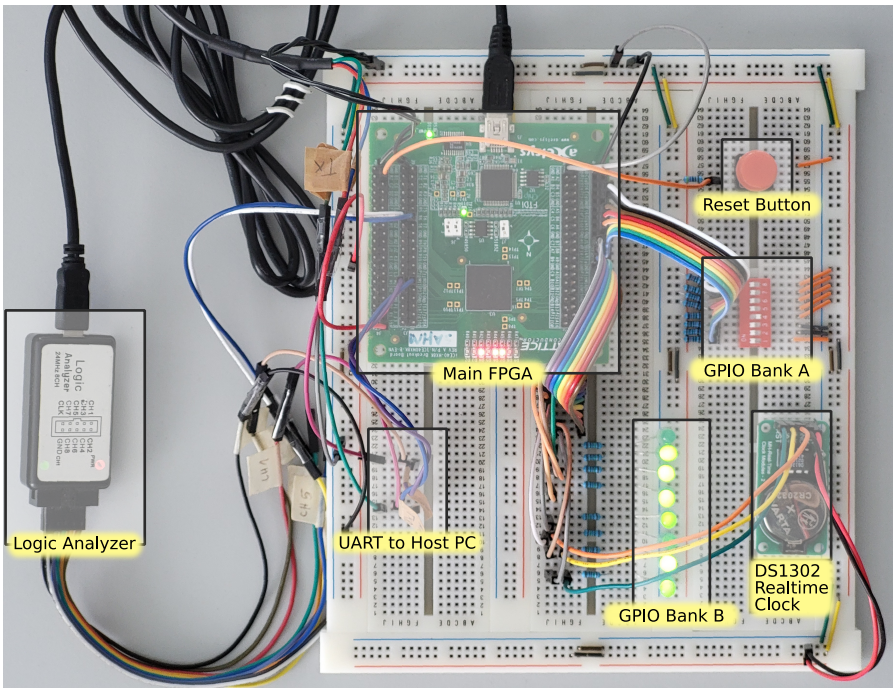


Figure 3.19: Annotated image of the experimental breadboard setup. The USB-connections not shown are connected to the host PC.

I/O connections for prototyping. For the experiments, these are two GPIO banks and an internal Greatest Common Divisor (GCD) peripheral. GPIO bank **A** is connected to a switch array, while GPIO bank **B** is connected to LED and a DS1302 real-time clock. The VPIL protocol is routed through the *UART to Host PC* connection. The protocol and some internal pins can be monitored through the connected logic analyzer on the left. The used HX8K FPGA is supported through vendor tool chain as well as a open source tooling. Additionally, this model was chosen to make the approach accessible and open sourced, thus stimulating further research. However, the choice for the HX8K FPGA also sets a limit on the available resources, so the determined area and memory usages, as well as the maximum operating frequency f_{max}, are more assessable. For the host computer executing the VP, an Intel i5-8520U @ 1.60 GHz with Fedora 37 is utilized.
To determine the quality and effectiveness of the approach, the two scenarios will be subject to measurements of execution times, FPGA resource utilization and additional metrics. In order to determine the time dependent behavior (e. g. protocol overhead, time per transaction, etc.), for the GPIO case-study the transaction were recorded with the logic analyzer. This allows measuring the duration for the read and write transactions respectively. Additionally, the measurement allows for an estimation of the delay contributed by the FPGA environment. Furthermore, for

each case-study the FPGA resource utilization was measured in terms of area (as Logic Cells (LCs)) and memory (as Block RAM (BRAM)). The performance and usability of the FPGA design can be estimated through the maximum operating frequency f_{max} and the time it takes for the various designs to be synthesized and processed by the place & route (PNR) to obtain a bitstream for the FPGA. For the case-study with the GCD accelerator, the execution time of the VP executing SW version is compared with the HWITL approach utilizing the RTL implementation.

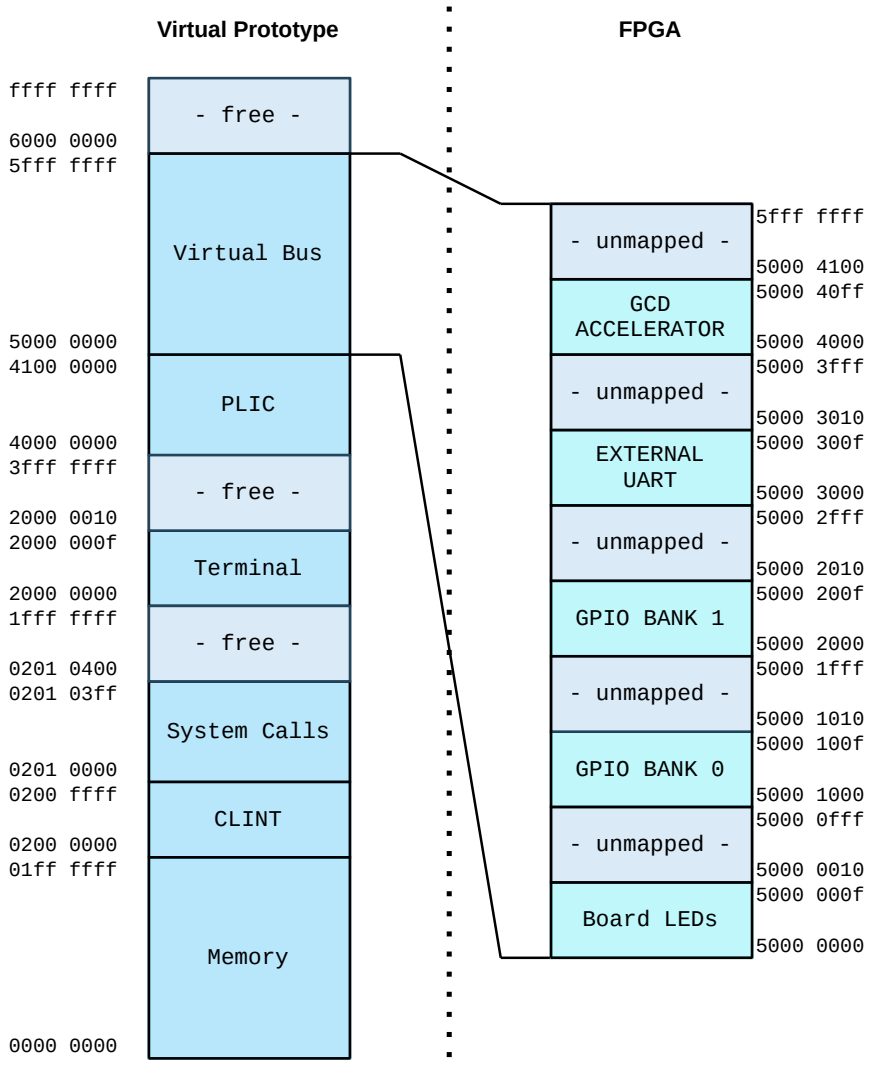


Figure 3.20: Memory map implemented for the case-study. The simulated SoC is on the left side, while the RTL HW implementations are on the right side.

Figure 3.20 shows the memory map implemented for the various peripherals and the exemplary accelerator. Addresses are denoted at the sides in hexadecimal starting from the bottom (e. g. Memory from `0x0000 0000` to `0x01ff ffff`). On

the left side the memory map with its simulated peripherals inside the VP is shown. For the HW implementation on the FPGA the right side shows the respective memory map. The Virtual Bus peripheral inside the VP on the left side (`0x5000 0000` to `0x5fff ffff`) is mapped transparently through the proposed protocol to the peripheral bridge on the FPGA. In the FPGA, the memory map is implemented such that it matches the VP's address range. This is not a requirement, though, as the VPIL SystemC peripheral may re-map addresses transparently.

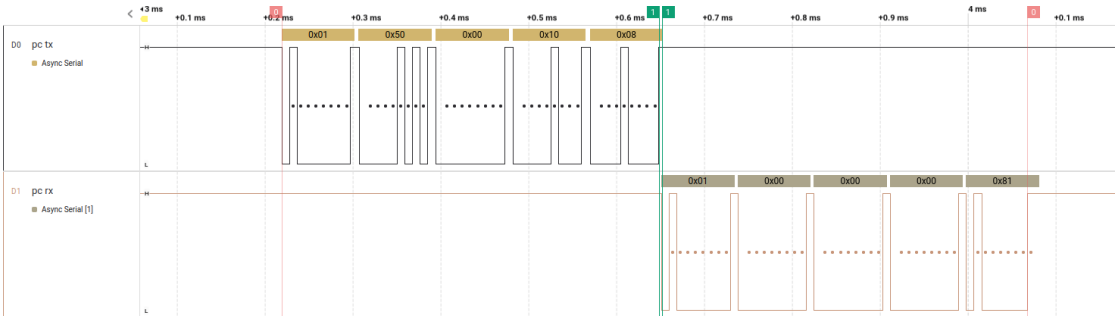
3.4.4.1 GPIO Bank

```
1  typedef uint32_t BUS_BRIDGE_TYPE;
2  static volatile BUS_BRIDGE_TYPE * const INTERNAL_LED = (BUS_BRIDGE_TYPE * const)
   ↪ 0x50000000;
3  static volatile BUS_BRIDGE_TYPE * const GPIO_BANK_A  = (BUS_BRIDGE_TYPE * const)
   ↪ 0x50001000;
4  static volatile BUS_BRIDGE_TYPE * const GPIO_BANK_B  = (BUS_BRIDGE_TYPE * const)
   ↪ 0x50002000;
5
6  struct MRV32_GPIO {
7    volatile uint32_t direction;
8    volatile uint32_t output;
9    volatile uint32_t input;
10 };
11 struct MRV32_INTLED {
12   volatile uint32_t val;
13 };
14
15 static struct MRV32_INTLED* const INT_LEDs = (struct MRV32_INTLED*) INTERNAL_LED;
16 static struct MRV32_GPIO*   const SWITCHES = (struct MRV32_GPIO*)   GPIO_BANK_A;
17 static struct MRV32_GPIO*   const EXT_LEDs = (struct MRV32_GPIO*)   GPIO_BANK_B;
18
19 volatile static uint8_t internal_led_state = 0;
20 void timer_irq_handler() {
21   INT_LEDs->val = internal_led_state++;
22   set_next_timer_interrupt();
23 }
24
25 int main() {
26   SWITCHES->direction = 0x00;
27   EXT_LEDs->direction = 0xff;
28   //[...]
29   while(!(SWITCHES->input & 0b10000000)) {  // main loop
30     if(SWITCHES->input & 0b00000001)
31       sweepLED();
32     else
33       countLED();
34   }
35   return 0;
36 }
```

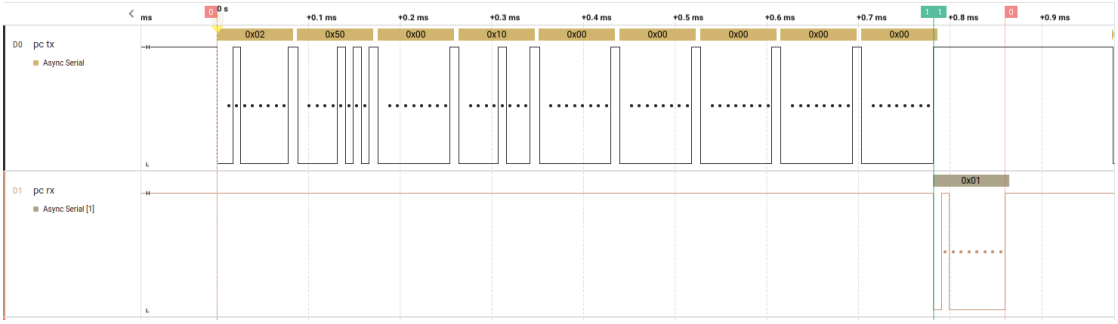
Listing 3.21: Simplified implementation of the GPIO bank interaction demonstration running on the VP. The GPIO banks are memory-mapped and behave the same as if they were implemented on the VP.

Listing 3.21 shows an excerpt of the basic interaction test that reads from GPIO bank **A** connected to a switch array, and writes data to the GPIO bank **B** which is

connected to LEDs (cf. Figure 3.19). The global memory map is defined in Lines 1 to 4, with the actual peripheral interfaces defined in Lines 6 to 17. The actual read/write interaction is done through `struct` accesses (e. g. in Line 26). Based on the value of a physical, external switch (read in Line 30), the external LEDs are driven in a different pattern to demonstrate the ability of interacting with the external environment. If the switch on the Most Significant Bit (MSB) is unset, the program terminates (Line 29).
The GPIO bank peripheral was taken from the open-source *MicroRV32* [89] that offers a set of SpinalHDL models, including a set of basic I/O peripherals. The GPIO peripheral offers three basic 32 bit registers (see Listing 3.21, Lines 6 to 10). The direction register determines whether a physical pin should be used for input (`0`) or output (`1`). The `input` register contains the corresponding state if input is enabled in the `direction` register, while the `output` register sets the physical pin state respectively.



(a) Read request.



(b) Write request.

Figure 3.21: Read (3.21a) and write (3.21b) transactions with annotated timing information and decoded serial communication. This is the UART implementation of the proposed protocol (cf. Figure 3.17), and the response buildup time, in both cases, is under one millisecond (green marker *1*).

While the functional test succeeded, the serial communication was also recorded between the VP and the FPGA with a logic analyzer. For both recorded communications in Figure 3.21, the top portion shows the transmitted bytes from the VP to the FPGA, while the bottom portion shows the bytes received from the FPGA as response. The top measurement (Figure 3.21a) shows a read (`0x01`) to the address `0x5000 1008`, with an acknowledging response (`0x01`) and the read data `0x0000 0081`. For the whole transaction, the marker pair *0* (red) indicates a time of 848.25 µs, while the internal processing on the FPGA is measured by marker pair *1* (green) and takes 3.25 µs. The bottom measurement (Figure 3.21b) shows a write (`0x02`) to the address `0x5000 1000` with the write data `0x0000 0000` and the acknowledging response (`0x01`). For the whole transaction the marker pair *0* (red) measures a time of 859.75 µs, the internal processing on the FPGA is measured by marker pair *1* (green) and takes 0.626 µs.
In this configuration, the mean protocol latency was measured as just under one millisecond at 115 200 baud. This is a promising result, as the FPGA implementation itself only needs less than 4 µs and the UART speed can be further increased when required.

3.4.4.2 GPIO Bit-Banging SPI

This GPIO experiment focuses on the general latency of the protocol. In this experiment, the SPI function to interface with an DS1302 real-time clock is not implemented on the FPGA but instead bit-banged via the `MRV32_GPIO` bank, as introduced in Subsection 3.4.4.1. The relevant pins of the DS1302 real-time clock are *CE* (chip enable), *I/O* (bi-directional data port), and *SCLK* (clock input for chip). These can be used to clock-in control bytes, which are either a read- or a write command followed by an address. With this scheme, HW-registers can be read or written. In the case of the DS1302, the registers contain the current time in a certain format.
For implementation, a readily available *Arduino* library was used. As it references only four functions of the Arduino framework (`void digitalWrite(PinNumber pin, LogicLevel level)`, `LogicLevel digitalRead(PinNumber pin)`, `void pinMode(PinNumber pin, PinDirection dir)`, and `void delayMicroseconds(Duration_us duration)`), the functions could be implemented quickly to interface with the `MRV32_GPIO` bank. Basically, the SPI / 3-wire protocol is implemented in SW by setting and reading the pins, combined with accurate delays in-between. As the `delayMicroseconds(...)` function depends on a measure of time (through the RISC-V CLINT), a host-time locked CLINT in contrast to the usual simulation time CLINT was used in this experiment. This is needed, as the interfacing DS1302 device resides in the "real" time that needs to be synchronized.

The case-study concluded successfully as the absolute time, managed in the DS1302 chip, could be read and written over the time span of several days.

3.4.4.3 GCD Calculation

To demonstrate application area for developing accelerators, a GCD implementation in both SW and SW were timed against each other. GCD was chosen because of the comparatively simple implementation, while still being not easy to pipeline because the length of the data-path heavily depends on the input combination. The SW and HW implementation both use Euclid's algorithm to find the GCD (see Listing 3.22). For the experiments, a separate executable for the two implementations was build to run on the RISC-V VP. The SW implementation does not use the proposed VPIL bridge but implements the algorithm purely in SW (Listing 3.22, Lines 1 to 9), while the HW executable interfaces with the FPGA's memory map (Lines 10 to 16) tunneled through the VPIL bridge.

```
1  uint32_t sw_GCD(uint32_t a, uint32_t b) {
2    while(a != b) {
3      if(a > b)
4        a -= b;
5      else
6        b -= a;
7    }
8    return a;
9  }
10 uint32_t hw_GCD(uint32_t a, uint32_t b) {
11   GCD_ACCEL->a = a;
12   GCD_ACCEL->b = b;
13   GCD_ACCEL->valid = 1;
14   while(!GCD_ACCEL->ready){};
15     return GCD_ACCEL->res;
16 }
```

Listing 3.22: SW and memory-mapped HW implementation of the `gcd(a,b)` algorithm.

Table 3.3: Test results for GCD-implementations `gcd(a,b)` on SW and a memory-mapped RTL implementation, both using Euclid's algorithm. The timings include the startup- and shutdown overhead of the RISC-V VP.

A	B	SW [s]	HW [s]
10154	3	0.19	0.17
101654	3	0.73	0.17
1051654	3	6.09	0.23
10512654	3	55.35	0.74
36546	1051654	0.14	0.17

Table 3.3 shows the results of five different tests, with an increasing imbalance between the parameters **A** and **B**. As can be seen, the SW run-time increases faster with a, due to the more efficient implementation on the FPGA. The protocol overhead becomes negligible even in the sub-second execution time (with $a = 1\,051\,654$ and $b = 3$), although the HW implementation uses active polling on the FPGA peripheral.

3.4.4.4 Synthesis Results

For the aforementioned case-studies we measured the resource utilization (area in terms of LCs, memory in terms of BRAM), the maximum operating frequency f_{max} and respective synthesis and PNR times. As the PNR process is heuristic driven, results for the frequency and the tool run times vary for each run. We choose to average the results over ten randomly seeded runs and provide each result with their respective standard deviation.

Table 3.4: Synthesis and Place & Route parameters for evaluated designs attached to responder bridge. Each design refers to an evaluated configuration of peripherals. Measured frequencies and times are averaged over ten runs with respective standard deviation. Area and memory utilization are shown as absolute (#) and relative (%) to their available resources, which were 7680 LCs and 32 BRAM units, with a target frequency of 12 MHz.

Description [unit]	**Peripheral Configuration**				
	GCD Acc.	**LED**	**LED+2xGPIO**	**LED+2xGPIO +UART**	**LED+2xGPIO +UART+GCD**
LCs [#]	1001	568	706	943	1432
LCs [%]	13	7	9	12	18
BRAM [#]	2	2	2	3	3
BRAM [%]	6	6	6	9	9
f_{max} [MHz]	96.86 ± 4.19	116.58 ± 5.62	113.23 ± 5.7	100.47 ± 3.8	94.97 ± 3.76
Synthesis time [s]	5.3 ± 0.08	3.93 ± 0.04	4.74 ± 0.08	6.13 ± 0.05	7.48 ± 0.08
Place & Route time [s]	2.22 ± 0.25	1.24 ± 0.16	1.61 ± 0.22	2.09 ± 0.38	3.41 ± 0.12

Table 3.4 shows the results of the synthesis and place & route for the utilized HX8K FPGA. The table is split into two parts. On the left side each description for the value is shown. For the LC and BRAM their respective available resources on the FPGA are shown next to their description. For the maximum operating frequency f_{max}, we configured the PNR with the target frequency of 12 MHz. On the right side, the five columns show at first the accelerator configuration itself (second

column) and then the incremental integration of additional peripherals, starting from only LEDs to a configuration with four peripherals and one accelerator.
For each hardware configuration (i. e. responder bridge plus respective peripherals) we collected the logic area in terms of LCs and memory BRAM both in absolute and relative numbers in respect to the maximum (max. 7680 LCs, 32 BRAM). It should be noted, that the design with the responder bridge proves to be lightweight, as even on a small FPGA such as the HX8K the area resource utilization is small (starting with the LEDs configuration at 7 %). This result emphasizes the lightweight property of the proposed HWITL bridge. With this, many peripherals can be attached and the integration process can be carried on for a long time into the development process to aid the engineers. Naturally, with the incremental addition of RTL modules, the maximum frequency f_{max} decreases.

3.4.5 Discussion

During development of the protocol, an appropriate focus should be given to endianess conversion. The C/C++ data structures are read / written via Unix file sockets and are thus in the host endianess domain. As the testing and validation programs for the *initator* and *responder* functionalities were mainly used on `x86_64` machines (*little* endian) and a certain object-oriented programming style was targeted, a part of the integration workload needed to be focused on synchronizing the exact byte-order between host computer and FPGA.
Another consideration should be given to the simulation vs. wall-clock time synchronization. As the simulation may be faster or slower than the outside (or wall-clock) time, interfacing with actual devices may either require simulation-time locking (as done in Subsection 3.4.4.2) or clock synchronization from the VP (as is provisioned into the protocol commands, see Listing 3.18, Line 10).
Furthermore, the case study utilized a specific physical layer (UART) with a fixed data rate (115 200 baud). The reasoning behind this was the fast setup and prototyping time, providing a proof of concept for the proposed methodology. Switching to other protocols and techniques (e. g. I^2C, Ethernet, PCIe, etc.) will drastically improve the speed, but requires additional prototyping. Moreover, as the designed showed an already high f_{max} (around 100 MHz) on a small FPGA family (Lattice Semiconductor HX8K), an additional presumption is a boost in higher operation frequencies for bigger and faster FPGA families (e. g. Xilinx Virtex, Kintex or Artix families). These two possible enhancements can reduce the aforementioned phenomenon of synchronization, as the overhead in communication and processing can further be reduced.
Even though the case studies were implemented with UART as the physical layer and a HX8K FPGA at 12 MHz, the results demonstrate that the proposed

methodology is a lightweight approach with adaptability for design needs towards even better speed or response times.

3.4.6 Conclusion and Future Work

In conclusion, this paper proposed a novel HWITL strategy called VPIL that is focused on combining transaction- and register transfer layer models, effectively placing RTL models on FPGAs "in-the-loop" of TLM VPs. It leverages the existing RISC-V VP infrastructure and enables RTL designers to focus development on their Unique Selling-Point with a minimal design evaluation cost. The contribution includes the serial communication protocol and the respective bridge implementations in SystemC TLM for the initiator and SpinalHDL for the FPGA responder. The proposed approach was evaluated in separate case-studies that included modeled peripherals like GPIO banks and a GCD accelerator. To stimulate further research, the proposed tool and the case-studies will be are publicly available on GitHub [51].
While already proven practical, the proposed approach also opens up future work to improve the efficiency and expand the application range of VPIL:

- Use of high performance FPGAs, integrated through a *PCIe* inteface, allowing for a high speed communication interface to development boards with Xilinx Virtex-7 or Artix 7 FPGAs that offer *PCIe* in an M.2 form factor. This would add a convenient development method on fast and high performance FPGAs, that are commonly used for artificial-intelligence accelerator development.
- Utilization of FPGAs containing full SoCs (e. g. Xilinx Zynq-7000 SoC series with ARM Cortex-A9), that combine configurable FPGA fabric and a commercial SoC. Through such FPGAs, the VP can be executed on the accompanying SoC in a lightweight Linux environment, communicating via the VPIL protocol on the FPGA fabric. This could enable a flexible MVP strategy to scale the complexity between the prototype- and small batch production phases.
- Support faster interrupts besides polling by using interface mechanisms (e. g. data-ready (DTR) signal from FTDI-compatible UART devices), to improve the protocol latency for interrupts. If an interrupt controller is implemented on the HW (i. e., RISC-V's CLINT or PLIC), this would allow a more efficient execution.
- Add efficiency-improving commands to read or write at the same address again or the next higher data word. These would be used by buffering the target address in HW to reduce protocol overhead. While

`{read,write}_again` would just re-use the last accessed address for improving polling the same remote register (e. g. a UART receive register), `{read,write}_consecutive` would increment the address by the register width (4 bytes) to speed up read / write accesses spanning larger address spaces (e. g. filling a memory block with encrypted data).

Chapter 4
Verification

As established in Chapter 1, it is not enough being able to *build* embedded systems, but it is also critical to thoroughly *verify* these systems. It was shown in the previous chapter that SystemC models have the main advantage of being modular and offering many abstraction layers; which renders them ideal as an early evaluation and reference model for the following HW/SW co-design stages. The engineering benefits are not the whole aspect, however: Being C/C++ models, the internal structure of SystemC VPs can be taken to an advantage, as the computer science community has a strong set of tools for such models. These tools can be categorized into two main categories from the viewpoint of this thesis: HW- and SW-centric verification approaches.

Hardware-centric verification approaches focus on the hardware modules (or the system as a whole), usually by extending the SystemC internals or by leveraging the module's SystemC interfaces for static and dynamic analysis techniques. *Software-centric* verification, on the other side, extends or modifies the hardware modules themselves (or the environment, for that matter) to gain additional knowledge about the SW's behavior.

Both verification targets can then use simulation-based verification (e. g. [137, 138]) and the more abstract formal techniques (e. g. [139, 140]). Simulation-based verification is based around inputs, that may or may not be generated automatically, and the assurance that the system behaves correctly under these stimuli. Formal techniques, inversely, start with an abstract specification and prove or disprove that the underlying Device under Verification (DUV) behaves *equivalently*. While formal techniques have the reputation of being more complete, they are also said to have a high initial formalization hurdle to overcome and to not map to real world applications. This is not completely true; nonetheless they require a more rigorous description of the target behavior in formal specification languages (e. g. [141–143]) with the trade-off being between a highly abstracted description that is easy to assume correct and then prove, and a detailed description that maps better

to reality but may contain errors itself. Simulation-based verification approaches usually deliver quicker results with less focus on a highly abstract description, making them more attractive for large-scale problems; especially in projects with fast-changing requirements. Their completeness differs, ranging from little (e. g. fuzzing techniques) to possibly complete (e. g. symbolic execution) depending on the actual technique and how much time is given to the verification runs.
Regardless of the concrete verification tools used, a considerable effort in the industry is undertaken to verify hardware models as early as possible [140] to detect specification [138], design [144], and implementation flaws [90] before the first hardware revision is manufactured. This speed is crucial; the later such problems are found, the more the respective fix-up will cost [29]. Additionally, in a world where consumer products are churned out in ever faster product cycles, a late-found issue is very likely to be either ignored or just fixed with the smallest effort possible, leaving deployed systems vulnerable.
As stated earlier, the first section proposes an effective approach for verification of real-world SystemC TLM peripherals using modern C++ symbolic execution tools (Section 4.1). The previously missing verification opportunities in the early VPs inhibit the use of intermediate models as *golden reference models*, which this approach aims to solve. It features a lightweight SystemC peripheral kernel (*Peripheral Kernel (PK)*) that enables an efficient integration with the modern symbolic execution engine Klee and acts as a drop-in replacement for the normal SystemC kernel on pre-processed TLM peripherals. The pre-processing step essentially replaces context switches in SystemC threads with normal function calls which can be handled by Klee. The following experiments, using a publicly available RISC-V specific interrupt controller, demonstrate the scalability and bug hunting effectiveness of the approach.
Section 4.2 extends on this idea to include the functionality needed for RTL-models that have been elevated to SystemC. In contrast to the previous TLM-only implementation of the *PK*, this includes the communication overhead of SystemC signals and sockets. With this added compatibility, actual RTL models can be cross-level verified against their TLM counterparts or directly be verified with the use of regular test-benches.
Section 4.3 finally presents a novel approach that enables early and accurate Dynamic Information Flow Tracking (DIFT) of binaries targeting embedded systems with custom peripherals. As avoiding security vulnerabilities is critical for embedded systems (as introduced in Section 1.1), a lot of different verification methods exist. This section focuses on Dynamic Information Flow Tracking, which is a powerful technique to analyze SW with respect to security policies in order to protect the system against a broad range of security related exploits. However, existing DIFT approaches either do not exist for VPs or fail to model complex

hardware/software interactions. Leveraging the SystemC framework, the proposed DIFT engine tracks accurate data flow information alongside the program execution to detect violations of security policies at run-time, enabling an early security policy evaluation for high-quality VPs. The effectiveness and applicability of the proposed approach is shown by extensive experiments, including a buffer-overflow attack suite and the analysis of an Advanced Encryption Standard (AES) encryption peripheral.

4.1 Verifying SystemC TLM Peripherals using Modern C++ Symbolic Execution Tools

This section includes and extends published material from the conference paper [46]. It starts with an introduction into the need of verification techniques for HW models in Subsection 4.1.1, and gives an overview over related work and approaches worthy of note in Subsection 4.1.2. Following in Subsection 4.1.4, the main approach is presented and explained in detail, including the thread-to-function translation in Subsection 4.1.4.2, the proposed replacement kernel in Subsection 4.1.4.3, and the usage of symbolic execution in Subsection 4.1.4.4. The experimental setup is described in Subsection 4.1.5, including an overview into the different test classes (Subsection 4.1.5.1), and is evaluated in Subsections 4.1.5.2 and 4.1.5.3. A second case-study about a sensor peripheral, not published in the original conference paper, is featured in Subsection 4.1.5.4. The subsection is discussed and concluded in Subsection 4.1.6 with a lookout to future work.

4.1.1 Introduction

As noted in Section 2.2, SystemC in combination with the TLM style has become an industrial standard for creating advanced VPs. A VP is essentially an abstract executable model of the entire hardware platform which is leveraged for early software development and acts as a reference model for the subsequent hardware design flow steps. Early and thorough verification of SystemC-based VPs is very important to avoid propagation of errors and the associated costly iterations for fixing them. Beside the instruction set simulator, which is an abstract model of the processor, TLM peripherals, such as an interrupt controller, are a central part of VPs. TLM peripherals rely on common modeling standards to describe the register interface, according to a device memory map, and provide a TLM interface to implement (software-driven) read and write accesses. The actual functionality is implemented through SystemC threads that leverage the event driven semantics of the SystemC kernel for synchronization. Application of formal verification techniques on TLM peripherals is very challenging as it needs to support the intricate TLM periperhal modeling semantics in combination with the simulation semantics of the SystemC kernel. Existing methods commonly rely on formal intermediate representations to capture the TLM peripheral semantics, which require significant effort to derive, do not scale to advanced SystemC TLM peripherals, or do not support core features of the SystemC kernel (see Subsection 4.1.2)
To mitigate these issues, in this section proposes an effective approach for verification of real-world SystemC TLM peripherals by using modern C++ symbolic

execution tools. The proposed approach consists of three main parts: Firstly, a SystemC thread transformation pre-processing step to enable replacement of context switching in threads with normal function calls, which is the main reason why an unmodified SystemC is incompatible with KLEE. Secondly, a new SystemC library called *Peripheral Kernel* (PK) that essentially acts as a drop-in replacement for the normal SystemC kernel on the pre-processed TLM peripherals. It implements all necessary interfaces which are used by advanced SystemC TLM peripherals. At the same time, the *PK* is much more lightweight by focusing only on relevant interfaces and integrating optimization procedures tailored to support symbolic execution engines. Thirdly, an existing state-of-the-art symbolic execution tool like KLEE [145] to verify (symbolic) properties specified for the TLM peripheral by means of assertions and assumptions embedded in a test-bench. As a case-study, verification results for a RISC-V specific PLIC [16] are reported. The PLIC is used in the open source virtual prototyping environment of the RISC-V VP (see Section 3.1) for the SiFive FE310 SoC [92]. The PLIC provides interrupt handling capabilities supporting several operating systems such as Zephyr and FreeRTOS. The proposed approach has been scalable and proven to be effective in verification measures. It found new, previously unknown bugs in the PLIC, as well as injected faults causing intricate bugs, which were detected in a short time. To stimulate further research, the proposed *PK* together with the experimental setup is openly available [47] as always.

4.1.2 Related Work

Formal verification of SystemC [4] designs is both important and also challenging [140, 146]. Therefore, it has received significant attention from the research community. Early efforts, for example [147–150], have very limited scalability or do not model the SystemC simulation semantics thoroughly [151]. Furthermore, they are mostly geared towards RTL signal-based communication.
More recent approaches are specifically targeting high-level SystemC designs that are in general suitable to capture the TLM semantics [5]. As a result, a set of SystemC verification tools have emerged. KRATOS [98] employs a model checking algorithm based on symbolic lazy abstraction and accepts an intermediate C input language with simple assertions. SCIVER [152] operates on sequential C models and leverages high-level induction techniques to check temporal properties [153]. SDSS[154] formalizes the semantics of SystemC designs in terms of Kripke structures and then applies a bounded model checking algorithm. In a follow-up work [155], the approach has also been optimized with state space reduction techniques based on Partial Order Reduction (POR). SISSI [156] defines the Intermediate Verification Language format and employs stateful symbolic

simulation techniques in combination with POR to deal efficiently with cyclic state spaces. For optimization purposes, native execution techniques have been leveraged [157]. STATE [158] translates SystemC designs to timed automata and verifies properties formulated on the timed automata using the UPPAAL model checker. In the context of these approaches, an extensive set of academic SystemC benchmarks is available. However, from a practical perspective, these approaches are still limited since due to their employed intermediate formalizations, and thus are not easily applicable to real-world VPs.

Other recent approaches have attempted to tackle this challenge: A first attempt has been made in [159], where the successful application of [160] on a simplified ARM AHB TLM-2.0 model is reported. In a follow-up work [161], slicing-based techniques are investigated to improve scalability and results on the verification of a FIFO-queue and a packet switch are reported. However, the specific modeling challenges of TLM peripherals have not been considered.

Another recent approach [162] addresses this real world application issue specifically. The authors propose a *XIVL* formal intermediate representation that bridges the modeling gap of TLM peripherals with the formal language employed by the SISSI verification tool. While the approach has shown promising results in verifying formal properties on an interrupt controller, it still requires significant effort to (manually) transform a SystemC model into the XIVL. In contrast, the proposed approach operates directly on the C++ code and can thus also benefit from recent advances in modern symbolic execution engines tailored for C++.

Also worth considering is the approach of Yang et al. [163], which leverages the Klee symbolic execution engine to generate test cases for SystemC modules that provide a high (branch) coverage. The approach also needs to integrate a customized scheduler to cover the SystemC simulation semantics and has reported very promising results in testing different SystemC designs. However, only the high-level synthesizable subset [164] of SystemC is supported by that approach. Moreover, it only supports static sensitivity to a single clock edge and does not allow the use of `sc_event`s, which is a common modeling requirement for TLM peripherals. Therefore, this approach does not support the verification of TLM peripherals as considered in this case-study.

In contrast, this section proposes a modular verification platform featuring freely available off-the-shelf symbolic execution frameworks to speed up the verification process in the early design phases using re-usable test-benches, suitable for CI/CD pipelines.

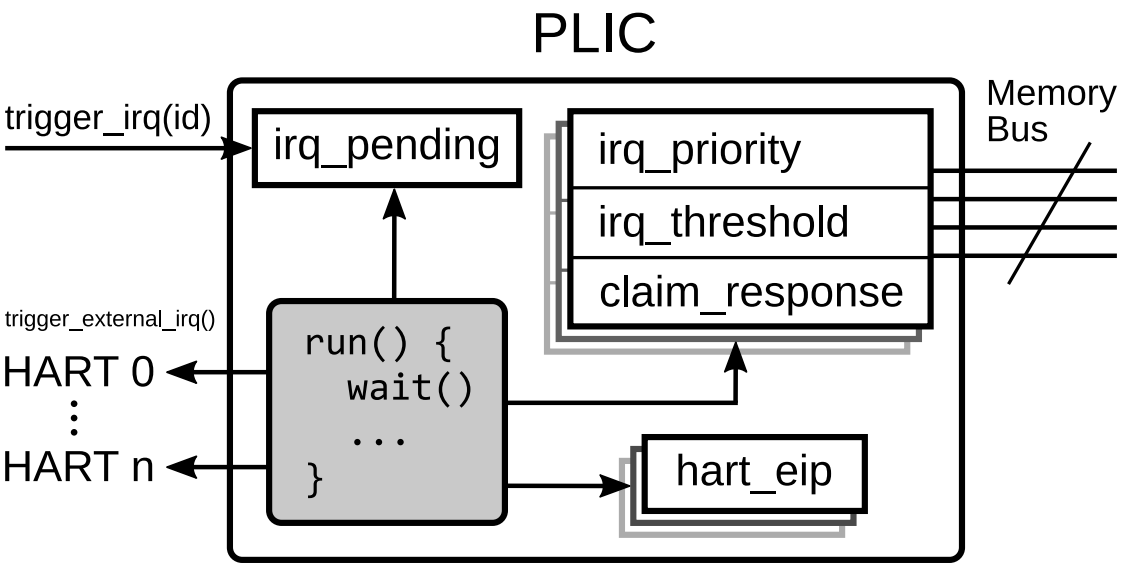


Figure 4.1: I/O Ports of the Platform Level Interrupt Controller. Elements with sharp corners are registers, managed by logic in the main `run()` method. The *external interrupt pending* (`hart_eip`) registers are private variables used for suppressing interrupt re-triggers and exist for every HART. Priority, threshold and the claim/response registers are duplicated for every interrupt.

4.1.3 Preliminaries - PLIC

This subsection provides relevant background information on the RISC-V specific PLIC implementation.

The Platform Level Interrupt Controller is specified by the RISC-V instruction set architecture [16]. It manages incoming, *global* interrupts and notifies the Hardware Threads (HARTs), i. e. the individual processor cores. It contains a set of registers for each HART where the processor can assign a priority and a notification threshold for each interrupt (see Figure 4.1). When an external interrupt fires, it sets an *interrupt pending* bit to the corresponding position in an internal register. Then, the PLIC will decide, based on the interrupt's assigned priority and its threshold, if a notification is passed to the individual HARTs (via `trigger_external_irq()`).

After an interrupt notification, a HART may check pending interrupts in the claim/response register via the memory-mapped interface. The HART finishes the completion of the interrupt by writing back the corresponding interrupt ID to the claim/response register. If other interrupts of less priority are pending, the PLIC will re-trigger all HARTs based on their individual threshold after that. Citing the official specification: "A priority value of 0 is reserved to mean *never interrupt* and effectively disables the interrupt. Priority 1 is the lowest *active* priority while the

maximum level of priority depends on PLIC implementation. Ties between global interrupts of the same priority are broken by the interrupt ID; interrupts with the lowest ID has the highest effective priority" [165].

4.1.4 TLM Peripheral Verification via Symbolic Execution

In this subsection, the proposed approach for verification of TLM peripherals via symbolic execution is presented in detail.

4.1.4.1 Overview

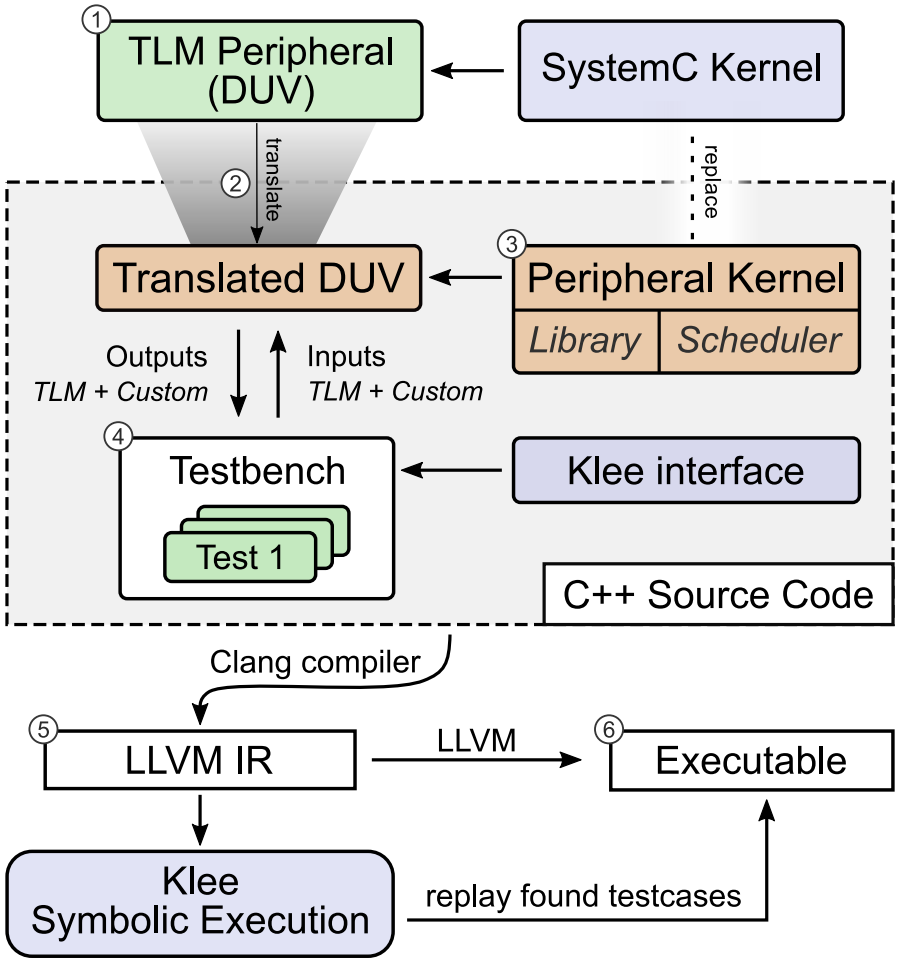


Figure 4.2: Overview of the verification approach using the proposed *PK*. Highlighted in green are the user-defined parts, in brown are the provided elements, and blue are existing tools.

Figure 4.2 shows an overview over the verification process with the proposed approach. Starting-point is a SystemC *TLM peripheral* ① which is the DUV. It

provides a TLM interface to communicate with other devices embedded in a virtual prototyping environment, and interacts with the SystemC kernel. The complexity of the SystemC kernel makes it very difficult to apply symbolic execution techniques for verification purposes of the DUV directly. In particular, the SystemC thread scheduling mechanism that relies on context switches and heavyweight data structures, such as floating point based `sc_time` implementations, lead to significant performance bottlenecks in symbolic execution tools, up to the point of being unsupported. Therefore, in a first pre-processing step, the DUV is *translated* ② by transforming its userspace-scheduling styled threads into classic function calls. In addition, a Peripheral Kernel (*PK*, ③) is provided as a drop-in replacement for the SystemC kernel on the translated DUV, with a compatible library and an optimized scheduling mechanism. The proposed *PK* provides all necessary interfaces which are used by advanced SystemC TLM peripherals. At the same time, it is much more lightweight by focusing only on relevant interfaces and integrating optimization procedures tailored to support symbolic execution engines. More details on the translation step and the *PK* can be found in Subsection 4.1.4.2 and Subsection 4.1.4.3, respectively.
test-benches ④ are user-provided for verification purposes. They interact with the translated DUV using the standard TLM interface (e. g. to read/write TLM registers) as well as custom interface functions (e. g. an interrupt line in an interrupt controller). Assumptions and assertions can now be embedded in the test-bench to specify symbolic input parameters and check the output behavior, respectively. This setup enables verification engineers to write very fine-grained yet generalized tests to enable a broad coverage and search for previously unknown corner-cases via symbolic execution. This case-study leverages KLEE, a state-of-the-art symbolic execution engine for C/C++, which provides a set of interface functions to declare and reason about symbolic expressions.
Each test-bench is compiled together with the translated DUV, *PK* and KLEE interface into a single LLVM Immediate Representation (IR, ⑤) using the Clang C++ compiler. The LLVM IR is analyzed using the KLEE symbolic execution engine. KLEE performs a symbolic state space exploration searching for errors on the symbolic execution paths. An error may be an assertion evaluated to *false*, an invalid memory access (segmentation fault, array-out-of-bounds), a software trap such as a division by zero, or an unhandled exception. For every error, a counterexample, i. e. concrete assignment for symbolic inputs, is generated by KLEE. It allows to reproduce the error and replay the test-bench execution for debugging purposes. For convenience, the IR bytecode can be compiled into a machine-native *Executable* ⑥ so that a classical debugger can be attached to analyse the counterexamples.

In the following, more details on the *translation step* ② and the proposed *PK* ③ will be given in subections 4.1.4.2 and 4.1.4.3, respectively.

4.1.4.2 Thread to Function Translation

As already mentioned in Section 2.2, SystemC incorporates optimized user-space scheduling as implementation of the thread/method model. This user-space scheduling however, despite its benefits, throws off analysis tools like valgrind and especially hinders symbolic execution engines. Typical behavior of these tools is then to either silently drop possible paths (under-approximation) or try out all possibilities by brute force (prohibitive time consumption). A *thread to function* translation is the key idea in enabling the symbolic execution through KLEE, as the SystemC userspace-scheduling implementations are incompatible with KLEES interpreter.

```
1  void run() {
2    while (true) {
3      sc_core::wait(e_run);
4      for (unsigned i = 0; i < NumberCores; ++i) {
5        if (!hart_eip[i]) {
6          if (hart_has_pending_enabled_interrupts(i)) {
7            hart_eip[i] = true;
8            target_harts[i]->trigger_external_interrupt();
9          }
10       }
11     }
12   }
13 }
```

Listing 4.1: Original SystemC `run` process of the PLIC from the open source RISC-V VP. The `e_run` event is used for synchronization with a new incoming interrupt. The function on Line 6 implements the priority calculation.

Unlike SystemC, the context of a yielding process is *not* saved in this return-based scheduling scheme. Thus, the translation essentially works by moving local into static variables to preserve them across function calls and embedding *Finite State Machine* (FSM) logic with goto statements to interrupt and resume the function at the right position on each context switch. This translation allows to preserve the execution context across multiple function calls and thus models the SystemC thread semantic. For illustration, Listing 4.1 shows a SystemC thread (from the PLIC TLM peripheral) called `run` and Listing 4.2 the resulting thread function after the translation process. The translated function consists of a header (Lines 2 to 14) and body (Lines 16 to 33) part. The header consists of `goto` statements to dispatch execution according to the context switch semantic. The current position in the thread function is stored in the newly introduced static `position` variable,

which is an enum of type `Label` (Line 7). A label is provided for the first execution (`init`) and each wait function call (`lbl1` in this example). The body is a copy of the SystemC thread body where each wait function is annotated with appropriate context switch logic. It saves the current position (Line 20) before exiting the function (Line 21). A corresponding label is added for this position (Line 5). To support the translation process, a Python script was built that automates these transformation steps for the DUV threads.

```
1  void run() {
2    //--[ header begin ]-----
3    enum class Label {
4      init,
5      lbl1,
6    };
7    static Label position = Label::init;
8    switch (position) {
9      case Label::lbl1:
10       goto LBL1;
11     default:
12       break;
13   }
14   //--[ header end ]-----
15
16   //--[ body begin ]-----
17   while (true) {
18     // context switch (i.e. wait) transformation
19     sc_core::wait(e_run);
20     position = Label::lbl1;
21     return;
22 LBL1:
23     // unmodified logic of the original run thread
24     for (unsigned i=0; i<NumberCores; ++i) {
25       if (!hart_eip[i]) {
26         if (hart_has_pending_enabled_interrupts(i)) {
27           hart_eip[i] = true;
28           target_harts[i]->trigger_external_interrupt();
29         }
30       }
31     }
32   }
33   //--[ body end ]-----
34 }
```

Listing 4.2: Translated SystemC `run` process of the PLIC.

4.1.4.3 Peripheral Kernel

The Peripheral Kernel (*PK*) is designed to be used as a drop-in replacement for the actual SystemC kernel. Figure 4.3 shows an overview of the *PK* architecture and integration. It consists of a SystemC compatible library (top left of Figure 4.3), matching wrapper macros (top of Figure 4.3), and the *PK* scheduler (bottom left of Figure 4.3) itself. It incorporates a pre-processor macro wrapper that maps

SystemC macros like `SC_HAS_PROCESS` to automatically register to the replacement kernel. As SystemC modules are designed to be modular and interact with the environment via defined functions and interfaces, the proposed *PK* library can connect to these with custom, lightweight, SystemC classes that the DUV in the test-bench will link to. Symbolic execution engines typically save and re-start the execution context of individual branches of the program, so the slimmed down *PK* library enables faster spawning of states. Especially the `sc_time` calculation routines need to be re-designed to use integer arithmetic wherever possible, to both speed up the symbolic execution and expand the possibilities for symbolic propagation. This is necessary, as KLEE currently does not support floating-point operations and concretizes these values.
As in SystemC, macros like `SC_HAS_PROCESS()` are used to register threads or processes to the simulation context of the proposed *PK* scheduler. The scheduler keeps track of waiting processes, scheduled events and the simulation time. E.g., when a translated process waits for a specified time or an event, it will be placed into a *wakelist*. These waiting processes are managed in a sorted list. Every simulation step advances the global time by the maximum amount possible without skipping a waiting event, calling all threads that are scheduled for that time. As the SystemC scheduler is non-deterministic [4], the *PK* scheduler does not need to incorporate a special order within multiple threads waiting for the same simulation event.
In summary, the *PK* is a lightweight implementation focusing on relevant interfaces and integrating well-designed and optimized data-structures for SystemC process scheduling. It serves as foundation to enable an efficient symbolic verification process.

4.1.4.4 Symbolic Execution

For the symbolic execution, the source code is compiled into *LLVM IR*, which is an intermediate language understood by KLEE, and can be further compiled into the specific host machine code for native execution. The test-bench uses KLEE to elevate certain parameters to a symbolic level, but would be valid for "normal" usage with arbitrary, predefined, values without it. By declaring a parameter as *symbolic*, KLEE will check all possible branches in the program flow for reachability during execution. If, within the exploration space and the collected conditions for symbolic variables, an error occurs, a counterexample with concrete parameter values is logged. An error may be an assertion evaluated to *false*, an invalid memory access (segmentation fault, array-out-of-bounds), a software trap such as a division by zero, and an unhandled exception. In the case-study's configuration, KLEE will only terminate after having visited every reachable path.

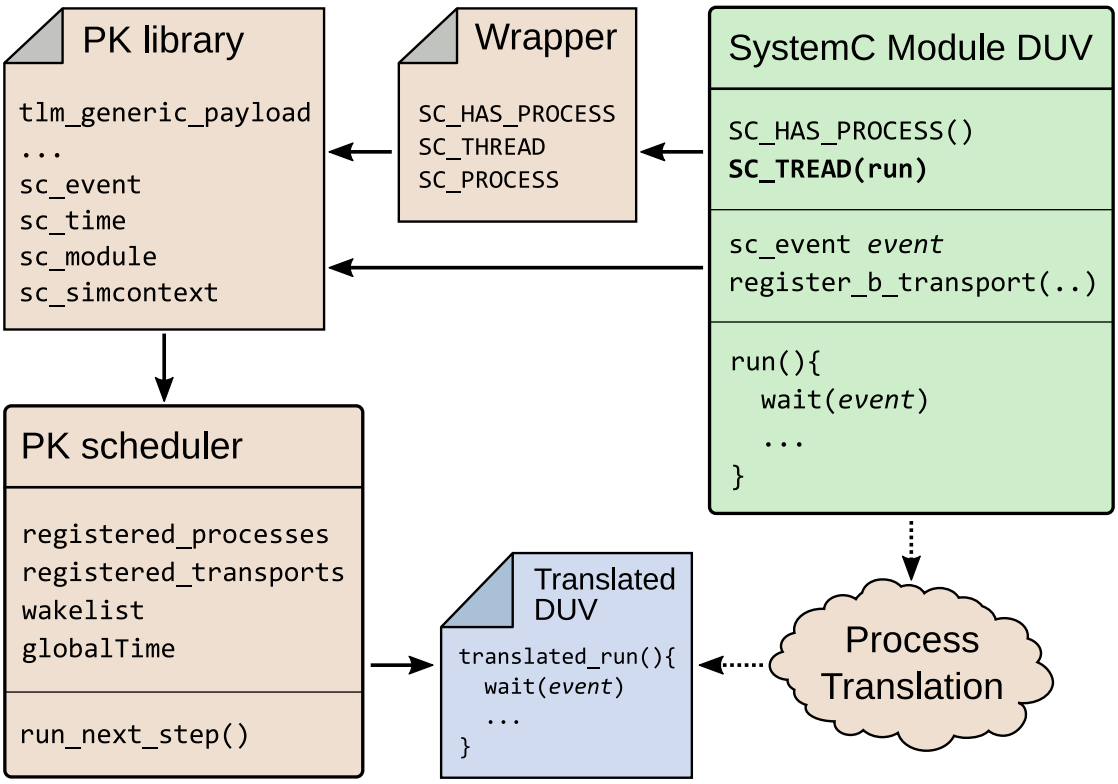


Figure 4.3: *PK* architecture overview with different interfaces for connecting to the (translated) DUV. Shown are the different interfaces of the SymSysC framework connecting to the DUV via the proposed wrappers.

4.1.5 Experiments

The proposed approach was implemented for TLM peripheral verification in this case-study. For evaluation purposes, the PLIC from the open source RISC-V VP is considered (see Section 3.1). In particular, the FE310 configuration of the PLIC which is based on the respective FE310 SoC from SiFive [92]: One HART, 51 interrupt sources with 32 priority levels. Implementation-wise, this PLIC uses a dynamic, synchronous `run`-method that is sensitive to an `sc_event` which in turn is triggered when new interrupts arrive, or on TLM register state changes.
For evaluation, a set of symbolic unit tests is given in the following sections to assess the PLIC against behavior, timing, and conformance to interface specifications. In addition to testing the original PLIC with SystemC version 2.3.3 and the *PK*, a fault-injection evaluation is also performed to further demonstrate the ability of the approach in finding intricate TLM peripheral bugs very efficiently. A comparison with the tools mentioned in related work is not possible because the tools are either not available publicly, only support custom, intermediate representations of the system, need a high manual effort to implement the compatibility, or do not

support the needed instruction set of SystemC TLM Peripherals. All experiments have been performed on a Linux Fedora 31 with an Intel Xeon 5122 with 3.6 GHz, and KLEE in version 2.2 with the Satisfiability Modulo Theories (SMT) solver STP. In the following, the layout of the symbolic test-benches (Subsection 4.1.5.1) are described. Then, the obtained results in testing the original PLIC (Subsection 4.1.5.2) as well as the PLIC with faults injected (Subsection 4.1.5.3) are presented, including an additional experiment with the *simple sensor* peripheral that is described in Chapter 3, Subsection 3.1.8.1.

4.1.5.1 Tests

In total, there are five symbolic tests. Each test feeds symbolic input data through the standard TLM interface in order to access the TLM registers of the PLIC, or triggers interrupts for processing using a custom interface function. Assertions are placed in each respective test to check correct output behavior and (internal) state changes of the PLIC. In addition, KLEE also searches for generic errors such as buffer overflows or null pointer dereferences.
In the following, more details are provided on the five symbolic tests chosen to verify the sanity of the in- and output interface and the interrupt sequence assumption mentioned in Subsection 4.1.3.
T1 performs a basic interaction test. It triggers a symbolic interrupt and checks if the correct interrupt is fired within the specified latency, the corresponding `pending_interrupt`-bit is set, claimable through a TLM transaction, and cleaned up afterwards.
T2 performs an interrupt sequence test. For illustration purposes, an excerpt of this test is shown in Listing 4.3. It configures two symbolic (but different) interrupt lines (Lines 8 to 14) with symbolic priorities (Lines 16 to 17) and triggers them simultaneously in zero (simulation) time (Lines 19 to 20). After that preparation, it advances the time to the next event and checks if the interrupt with the higher priority was fired first (Line 32). If they have the same priority, the one with lower interrupt ID shall fire first. The test goes then on to check the second, lower prioritized interrupt for integrity, which is omitted in this listing for readability reasons.
T3 performs an interrupt masking test. It configures a symbolic interrupt line with a symbolic priority and sets the `consider_threshold` to a symbolic value. It checks if the interrupt is only fired if its priority is both not zero and above the configured threshold.
T4 performs a TLM read interface test. It triggers an interrupt and starts a TLM *read*-transaction at a symbolic address using a symbolic length parameter. This test

allows to check that the TLM periperhal can handle generic TLM read transactions and is not missing the handling of specific address ranges.
T5 is similar to T4 but performs a TLM write interface test. It also triggers an interrupt but then starts a TLM *write*-transaction at a symbolic address using a symbolic length parameter and writes up to 1000 bytes of symbolic data.

4.1.5.2 Test Results: Original PLIC

Table 4.1: Test results for the original PLIC. For a *Fail*ed result, the number of detected failures by that test is given in parentheses.

Test	Result	# Exec. Instr.	Time [s]	Paths	Solver
T1	Fail (1)	4 330 418	1293	64	98.02 %
T2	Pass	8 975 783	78 755	3162	98.82 %
T3	Pass	7 027 481	66 576	967	98.62 %
T4	Fail (3)	38 062 265	67	1168	74.17 %
T5	Fail (4)	102 992 556	93 383	62 017	97.58 %

Table 4.1 shows an overview on the test results for the original PLIC. The first column reports the performed test. The second column provides the test result. Each test can either *Pass* with no errors or *Fail* with at least one detected error. In case of a *Fail*, the number of detected errors by the respective test is given in parentheses. Please note, KLEE does not terminate after the first error is found but completes the symbolic state space exploration[1]. The next column # *Exec. Instr.* provides the overall number of executed LLVM bytecode instructions. The remaining columns show the total execution time in seconds (column: *Time*), the number of explored symbolic execution paths by KLEE (column: *Paths*) and how much of the overall execution time was spent in the SMT solver engine of KLEE to process SMT queries (column: *Solver*). It can be observed that the solver time vastly dominates the overall execution time in most tests. Only in T4 the solver queries are less complex performance-wise, thus resulting in a (symbolic) execution speed of up to 568 000 instructions per second. The overall runtime varies between 67 seconds for T4 and around 26 hours for T5. Please note that this is the time required to perform the complete state space exploration, errors are typically found much faster, which will be discussed further in Subsection 4.1.5.3. Also, the proposed peripheral kernel has been very important to achieve these runtime results. When using the normal SystemC kernel, the initialization phase could be completed, but right at the first scheduling event, KLEE crashed with a segmentation fault right

[1] Only the single execution path that triggers the error is terminated.

```
1  void functional_test_itr_priority(
2    PLIC<1, numberInterrupts, maxPriority>& dut) {
3
4    Simple_interrupt_target hart(dut);  // mock-up hart
5    // connecting interrupt line plic -> hart
6    dut.target_harts[0] = &hart;
7
8    uint32_t i = klee_int("i interrupt");
9    uint32_t j = klee_int("j interrupt");
10
11   // generate two valid different interrupt ids
12   klee_assume(i < numberInterrupts && i > 0);
13   klee_assume(j < numberInterrupts && j > 0);
14   klee_assume(i != j);
15
16   uint32_t lower_itr = i < j ? i : j;
17   uint32_t highr_itr = i > j ? i : j;
18
19   dut.trigger_interrupt(i);
20   dut.trigger_interrupt(j);
21
22   pkernel_step(); //advance time to next event
23
24   // PLIC should have triggered an external interrupt
25   assert(hart.was_triggered &&
26          "HART interrupt was not triggered");
27
28   // Is correct Interrupt claimable?
29   uint32_t first_itr = hart.claim_interrupt();
30
31   // Was the itr with the highest prio chosen first?
32   assert(first_itr == lower_itr &&
33          "Wrong interrupt was fired first");
34
35   assert(hart.was_cleared &&
36          "Interrupt was not cleared after claim");
37
38   pkernel_step(); //advance time to next event
39
40   //the step should trigger an external interrupt
41   assert(hart.was_triggered &&
42          "HART interrupt was not triggered a second time");
43
44   // Is correct Interrupt claimable?
45   uint32_t second_itr = hart.claim_interrupt();
46
47   //Was the itr with the lowest prio chosen now?
48   assert(second_itr == highr_itr &&
49          "Second interrupt was fired in wrong order");
50   assert(hart.was_cleared &&
51          "Interrupt was not cleared after claim");
52 }
```

Listing 4.3: Part of the *interrupt priority test* (**T2**). This test contains multiple logic checks in the form of assertions.

```
1  struct Simple_interrupt_target : public external_interrupt_target {
2    bool was_triggered = false;
3    bool was_cleared = false;
4    PLIC<1, numberInterrupts, maxPriority>& dut;
5
6    Simple_interrupt_target(PLIC<1, numberInterrupts, maxPriority>& dut) :
       ↪ dut(dut){};
7
8    void trigger_external_interrupt() {
9      assert(!was_triggered &&
10            "interrupt triggered more than once");
11     was_triggered = true;
12     was_cleared = false;
13   };
14
15   void clear_external_interrupt() {
16     assert(!was_cleared && "interrupt cleared more than once");
17     was_cleared = true;
18   };
19
20   uint32_t claim_interrupt() {
21     assert(was_triggered &&
22            "tried to claim untriggered interrupt target");
23     was_triggered = false;
24
25     sc_core::sc_time delay;
26     tlm::tlm_generic_payload pl;
27     uint32_t interrupt = 0;
28     pl.set_read();
29     pl.set_address(0x200004); //claim_response register
30     pl.set_data_length(sizeof(uint32_t));
31     pl.set_data_ptr(reinterpret_cast<unsigned char*>(&interrupt));
32
33     dut.transport(pl, delay);
34
35     assert(interrupt > 0 &&
36            "interrupt was triggered, but no interrupt in register");
37
38     unsigned idx = interrupt / 32;
39     unsigned off = interrupt % 32;
40     assert(((dut.pending_interrupts[idx] & (1 << off)) == 0) &&
41            "pending interrupt shall be reset after read");
42
43     pl.set_write();
44     dut.transport(pl, delay);   // clear interrupt
45
46     assert(was_cleared || was_triggered &&
47            "interrupt was either cleared or triggered for another prio");
48
49     return interrupt;
50   }
51 };
```

Listing 4.4: Interrupt target used in the tests **T1-T3**. The target itself contains a number of assertions already.

after a `mprotect()` syscall in the *quickthreads* implementation. Even after manually patching the kernel without this syscall, a successful context switch could not be performed, rendering this approach unsuccessful.
Based on the five introduced tests, six errors in total could be found. They are described in the following:
F1 is a forgotten assertion in the `trigger_interrupt` routine. This assertion checks if the passed interrupt id is valid; i. e. between one and the maximum number of interrupts. However, this assertion throws an unhandled error that terminates the program which does not fit into a production grade environment. Also, when built in release mode, such assertions would not be checked and thus the program would produce a segmentation fault.
F2 describes a failed assertion checking the 4-byte alignment of a TLM register access. The correct way to handle failed assertions would be to return a TLM error state instead of terminating the program. This way, a transaction initiator like a processor can handle this with a correct exception handler.
F3 defines a failed assertion, similarly to F2, that checks the existence of a TLM register mapping that can handle the required address.
F4 characterizes a failed assertion, similarly to F2, checking the TLM target register is registered as writeable in case of a write transaction.
F5 is an unhandled memory access in which a TLM read transaction was accepted by a register mapping if the address matched a register with a 4-byte aligned transaction size, that could exceed the actual register boundaries. This leads to a `memcopy()` with the source exceeding valid memory addresses.
F6 labels a failed assertion inside the TLM transport register access callback that was previously thought never to be false. In this case, the address was set to the interrupt `claim_response` register. Normally, a interrupt target writes to the register only after being notified. In this case however, the test initiated the transaction just after triggering the interrupt before the periodic PLIC thread was scheduled. This race condition was previously not found in normal operation because of the high PLIC thread frequency compared to the processor.
The faults could be found respectively by: **F1** with T1; **F2** to **F4** with T4; and **F3** to **F6** with T5. In the following, more details are provided on how fast each error was found, and the results on finding additional injected faults that represent other common and intricate TLM peripheral errors are presented.

4.1.5.3 Test Results: PLIC with Injected Faults

For further evaluation purposes, six additional common (TLM peripheral) bugs were injected into the PLIC: **IF1** to **IF6** (see also Table 4.2). These include off-by-one faults (**IF1**, **IF6**), selectively dropping functional parts (**IF2**, **IF4**, **IF5**) and a race-

condition (**IF3**). Of these, **IF1**, **IF3** and **IF6** have been present in earlier versions of the PLIC, as can be observed in the GitHub logs in [49]. In the following, more details are given on these six bugs and then show how fast they are detected with the proposed approach:

Table 4.2: Classification of the faults injected or found in the PLIC.

Type	Instance	№
Original	`trigger_interrupt` input assertion	F1
	TLM map 4-bit alignment assertion	F2
	TLM map register mapping assertion	F3
	TLM map read-only assertion	F4
	Out-of-bounds memory read	F5
	Interrupt claim race condition	F6
Injected	`trigger_interrupt` buffer overflow	IF1
	Interrupt 13 not triggering	IF2
	Missing re-trigger consecutive interrupts	IF3
	Interrupt notification too late (position-dependent)	IF4
	Interrupt 16 not clearing	IF5
	Priority zero interrupts considered	IF6

IF1 changes a check for the highest allowed interrupt number from `irq_id < NumberInterrupts` to `irq_id <= NumberInterrupts`, resulting in a buffer overflow in the array storing pending interrupts.
IF2 explicitly drops the notification of interrupts with the id 13 after writing to the correct pending interrupt register.
IF3 skips a necessary re-trigger for another simultaneously waiting interrupt after claiming the first one. This behavior is particularly hard to debug without well-suited unit tests.
IF4 artificially increases the event notification for the main thread if interrupt number is over 32. This shall emulate an error or wrong specification in the timing model of the Device under Test (DUT).
IF5 returns the interrupt clear routine early if a specific interrupt is being cleared.
IF6 originates in a misinterpretation of the specification that checks if a pending interrupt priority is greater or equal to the configured threshold, while it shall be strictly greater than the threshold.
Table 4.3 shows how fast the errors in the original PLIC (F1 to F6) and the PLIC with injected faults (IF1 to IF6) have been found by the respective tests. It can be observed that all original bugs are found in less than 3 hours with most bugs being found in just a few minutes or even less than a minute. The efficiency can

Table 4.3: Overview on how fast the errors in the original PLIC (F1 to F6) and the PLIC with injected faults (IF1 to IF6) have been found by the respective tests. The runtime is given in minutes (except for IF3, given in hours) and rounded to the next highest integer.

	F1	F2	F3	F4	F5	F6	IF1	IF2	IF3	IF4	IF5	IF6
T1	1m	–	–	–	–	–	1m	3m	–	19m	4m	–
T2	–	–	–	–	–	–	–	60m	24h	–	105m	–
T3	–	–	–	–	–	–	–	–	–	–	–	7m
T4	–	1m	1m	1m	–	–	–	–	–	–	–	–
T5	–	–	1m	1m	16m	147m	–	–	–	–	–	–

be explained by KLEE's symbolic exploration heuristics, which attempt to solve the most promising paths first and by tracking extensive symbolic constraints among these paths. The results demonstrate the effectiveness of the proposed approach in finding relevant bugs in real-world TLM peripherals quickly.

4.1.5.4 Test Appendix: Simple Sensor Peripheral

As a second case-study, not included in the original publication [46], the default *simple-sensor* peripheral (first described in Subsection 3.1.8.1) was analyzed with a single test-bench. This peripheral is comparatively simple, so the complete test-bench could be fully explored by KLEE in under five seconds (see Table 4.4). The test-bench (Listing 4.5) contains the following number of different checks:

1. Correctly triggering specified interrupt (Listing 4.5, Line 11).
2. General read-accesses to all possible addresses and sizes (Line 28).
3. Correctly returned value range as specified (Lines 40 and 41). Here, KLEE is especially potent, as the issued `rand()` is implemented to return symbolic values.
4. General 4 byte write-accesses to all possible addresses in Line 51.

```
1  int main()
2  {
3    uint32_t interrupt = klee_int("Interrupt");
4    SimpleSensor dut(interrupt);
5    test_interrupt_gateway tig;
6    dut.plic = &tig;
7
8    pkernel_step(); // 0
9    pkernel_step(); // 40ms
10   //Test 1: Is triggered interrupt correct?
11   assert(interrupt == tig.triggered_irq);
12
13   sc_core::sc_time delay;
14   tlm::tlm_generic_payload pl;
15   uint32_t address = klee_int("address_read");
16   uint32_t length = klee_int("length_read");
17   const uint32_t max_len = 1000;
18
19   // limit possible length
20   klee_assume(length <= max_len);
21   pl.set_read();
22   pl.set_address(address);
23   pl.set_data_length(length);
24   uint8_t* buffer = new uint8_t[max_len];
25   pl.set_data_ptr(buffer);
26   // Test 2: Read at any address for any given length.
27   // Is there any unexpected output?
28   dut.transport(pl, delay);
29
30   pkernel_step(); // 80ms
31
32   //Test 3: Are returned values in specification?
33   pl.set_read();
34   pl.set_address(0);
35   pl.set_data_length(1);
36   uint8_t rand_value;
37   pl.set_data_ptr(&rand_value);
38   dut.transport(pl, delay);
39
40   assert(rand_value > 48);  // In default filter, values shall range in
41   assert(rand_value < 58);  // [48 + rand() % 10]
42
43   // Test 4: Write at any address.
44   // Is there any unexpected output?
45   pl.set_write();
46   pl.set_address(klee_int("address_write"));
47   pl.set_data_length(4);
48   uint8_t write[4];
49   klee_make_symbolic(write, 4, "write data");
50   pl.set_data_ptr(write);
51   dut.transport(pl, delay);
52   [...]
53 }
```

Listing 4.5: Part of the *simple sensor* test-bench.

The results of the symbolic execution can be found in Table 4.4. In total, 12 individual faults could be identified, of which 8 were original bugs and 4 were injected manually (denoted with a ‡). They can be grouped into the following types: *Specification-*, *Timing-*, and *Function* faults. As can be seen, the proposed approach is able to find all of these in a minimal amount of execution time.

Table 4.4: *Simple sensor* fault categories and number of individual occurrences. ‡ indicates a previously unknown fault. Complete path exploration time: 4.46 s.

Type	Instance	# faults
Specification	Virtual memory out-of-bounds	2‡
	Forgotten assertions	3‡
	Internal data accessible	1
	Invalid return value at undefined addresses	2‡
	Invalid sequence of triggering interrupt	1
Timing	Response time too long (depending on data)	1
Function	No or incorrect interrupt triggered	1
	Measured data outside requested bounds	1‡

4.1.6 Conclusion

This section proposed an effective approach for verification of real-world SystemC TLM peripherals by using modern C++ symbolic execution tools. The foundation of the proposed approach is a lightweight *PK* that acts as drop-in replacement for the SystemC kernel and is tailored for enabling the symbolic execution of TLM peripherals. The *PK* combines optimized data structures with a simplified function-based scheduling mechanism that relies on a thread to function transformation process. As a case-study, verification results are reported for a RISC-V specific PLIC and an example sensor peripheral that is used in the open source virtual prototyping environment for the SiFive FE310 SoC. New, previously unknown bugs could be found in both the PLIC and the sensor peripheral, while it also could be demonstrated that other intricate bugs can be detected by means of fault-injection very quickly using KLEE, a state-of-the-art symbolic execution engine for C/C++. To stimulate further research, the *PK* together with the experimental setup, is available as open source in [47] as always.
For future work, it could be promising to investigate additional optimizations of the *PK* to further boost symbolic execution performance; and to evaluate the approach beyond TLM peripherals, both for verification of other SystemC IP components such as a co-processor and the feasibility to verify whole SystemC projects with a high number of individual components. Also, it is worth investigating on how the system behaves and scales if lower-level SystemC primitives like signals are added to the *PK*, as described in Section 4.2.

4.2 Towards Cross-Level Equivalence Testing of Peripherals using Symbolic Execution Tools

This section contains unpublished material in the form of an extended abstract, as the experimental evaluation is still ongoing at the time of this writing. This section starts with an introduction into the key idea of this approach, as well as related work. In Subsection 4.2.2, the improved verification process is explained in more detail, along with the extended architecture of the *PK*. In Subsection 4.2.3, the initial experimental setup is described. Finally, the conclusion and future work is presented in Subsection 4.2.4.

4.2.1 Introduction

While the feasibility of symbolic execution of TLM SystemC peripherals have been shown in Section 4.1, the previous approach still lacks the possibility to efficiently execute RTL models. This misses the chance of early cross-level verification approaches during the development of HW RTL models. Existing methods either focus on formal specifications and TLM models [139] which are not applicable to lower level HDL Verilog models, or rely on heuristic-based simulation equivalence checking [166] which is both not complete enough and relies on existing RTL models.
Besides using VPs as a source of quality in terms of verification level, it is also possible to lift RTL-HDL models to a SystemC compatible description using tools like *Verilator* [167]. This technique enables both *equivalence checking*, as well as *symbolic execution* by re-using the existing SystemC test-benches established in Subsection 4.1.5. This is in contrast to existing work that either relies on TLM *golden reference models* or a formal description of the desired behavior.
While symbolic execution tools are used in classic SW verification [168], the SystemC kernel is hard to execute symbolically because of the scheduling model, *x86* assembler instructions, and floating-point arithmetic [46]. Existing approaches either focus on feature-oriented design methodology [169], which effectively abstracts away the model's SystemC interfaces, or ignore the communication of models altogether [170]. This challenge of actual SystemC kernel execution hinders further advances in the area of TLM peripheral IP verification.
In this section, an extension to the replacement kernel of SystemC called *Peripheral Kernel* (*PK*, first introduced in Section 4.1), is proposed. The extension focuses to add support of the SystemC RTL layer system, resulting in the compatibility of mixed TLM/RTL models. In combination with *Verilator*, a tool to lift RTL models written in HDLs to SystemC, this enables the symbolic execution of such

models. With symbolic execution engines like KLEE [145], these models can then be verified with the existing test-benches, or against already verified TLM models. To stimulate further research, the proposed *PK* together with the experimental setup is made openly available [47].

4.2.2 RTL Peripheral Verification via Symbolic Execution

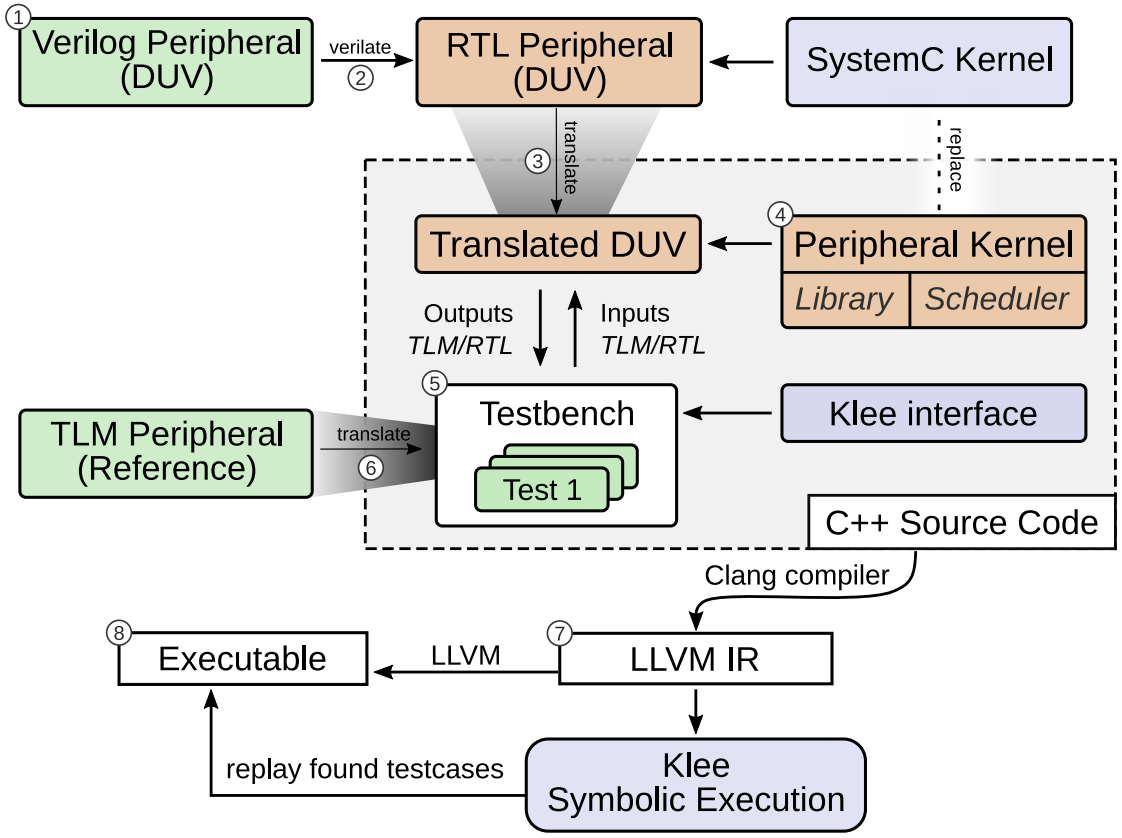


Figure 4.4: Overview of the verification process flow using the proposed *PK*. Highlighted in green are the user-defined parts, in brown are the provided elements, and blue are existing tools.

The main process flow is depicted in Figure 4.4: Compared to the previously published approach in [46], this includes another translation step. It starts with the Verilog model (①, on the left) that is *verilated* ② into a SystemC-compatible RTL model. This model, containing interface features like signals and ports, then is translated ③ via the *thread-to-process* method described in Subsection 4.1.4.2. Through the newly adapted and extended *PK* ④, it then may be interfaced by the test-benches created by the verification engineers. If a TLM reference model exists,

it can then also be translated ⑥ and used for cross-level verification in the test-benches. Next, the test-benches are compiled into LLVM IR ⑦ and symbolically explored using KLEE. As stated in Subsection 4.1.4, KLEE performs a symbolic state space exploration searching for errors on the symbolic execution paths. An error may be an assertion evaluated to *false*, an invalid memory access (segmentation fault, array-out-of-bounds), a software trap such as a division by zero, or an unhandled exception. For every error, a counterexample, i. e. concrete assignment for symbolic inputs, is generated by KLEE. It allows to reproduce the error and replay the test-bench execution in the machine-native executable ⑧ to pin-point the cause of the errors.

4.2.2.1 Peripheral Kernel

To support SystemC RTL models, the improved *PK* implements the SystemC thread/method sensitivity model. In the following, the group of threads and methods is referred to as *processes* if both are meant.

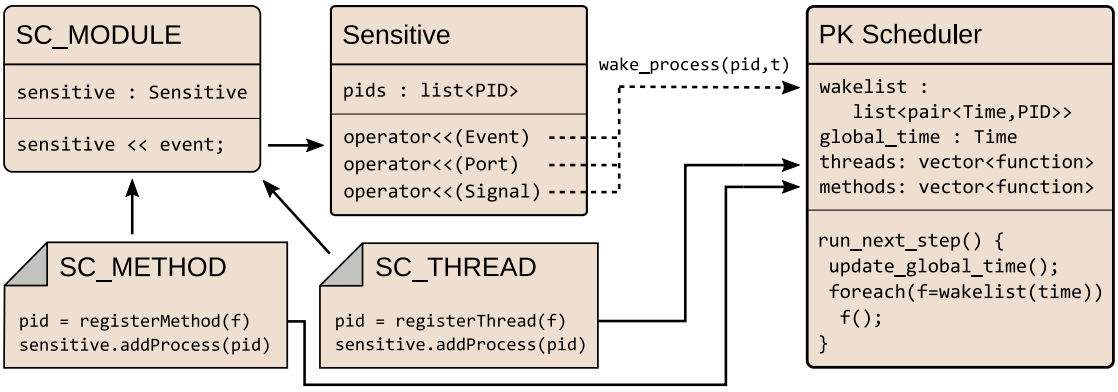


Figure 4.5: The module bring-up phase of the extended *PK*. Rounded boxes are classes, while the folded boxes represent macros. The dashed arrows (as in `wake_process(pid, t)`) indicate that this operation occurs only after the bring-up phase, during run-time. Some functions are omitted for clarity, especially the actual functions of `SC_MODULE`.

When a module (Figure 4.5, left side) defines methods or threads, it may register them to the sensitivity list of `event`s, `port`s, and `signal`s. As the *PK* scheduler is function-based to maintain compatibility to symbolic execution engines, the `SC_METHOD` and `SC_THREAD` macros register the function as C++ function bindings. In contrast to the old *PK*, this is done in one central `vector` per type in the *PK* scheduler (Figure 4.5, right) with a unique process id called `PID`. This `PID` is referenced in the following sensitivity operators in the module's constructor to keep the *one-to-many* association intact. At the end of the bring-up phase,

all registered events contain a list of processes they are sensitive to; allowing them during run-time to wake up all processes (`wake_process(pid, time)`, cf. Figure 4.6).

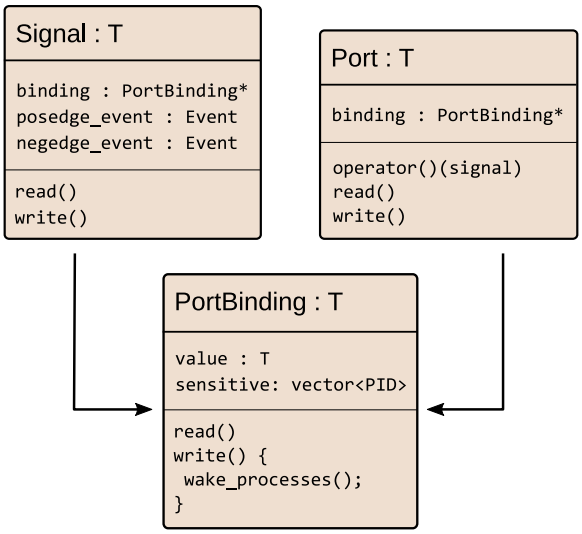


Figure 4.6: Association of the classes `Signal` and `Port` in the *PK* library. A connection, made in the bring-up phase, will result in a `PortBinding`. All classes here are templated based on the signal base type `T`.

The connections between modules is done through `PortBinding` structures, that are created when a SystemC *Signal* is connected to a *Port* (see Figure 4.6). Every thread or method, that is declared `sensitive` to the port, is registered with its `PID` in the sensitive list of the corresponding `PortBinding`. Upon a `write()` on the signal during run-time the value is changed and sensitive processes are woken up in *zero time*, i. e. in the current `global_time` (see Figure 4.5) in accordance to the SystemC specification.

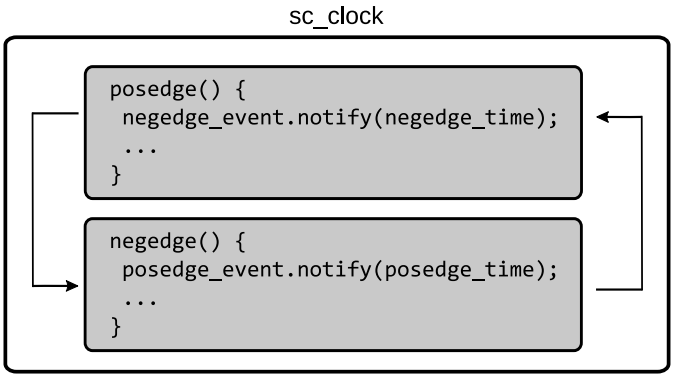


Figure 4.7: `sc_clock` update mechanism in the *PK* library.

To support the SystemC event system fully (including multiple waiting processes), the `event` class contains a list of sensitive `PID`s just like `PortBinding` (Figure 4.6). Especially `sc_clock` relies on that behavior (see Figure 4.7) as it re-sets the `posedge_event` and `posedge_event` in a loop. Each `notify(time)` inserts the sensitive processes into the *PK*'s `wakelist` at the corresponding time.

4.2.3 Experimental Setup

For the proposed experimental setup, the RTL implementation of the *MicroRV32* [89, 171] called `RVPLIC` is used. It is written in the HDL *SpinalHDL*, which is synthesizable into Verilog and VHDL. As it is based on the RISC-V's TLM basic PLIC, it also should conform to the RISC-V ISA specification [16] which is the subject to this experimental verification.

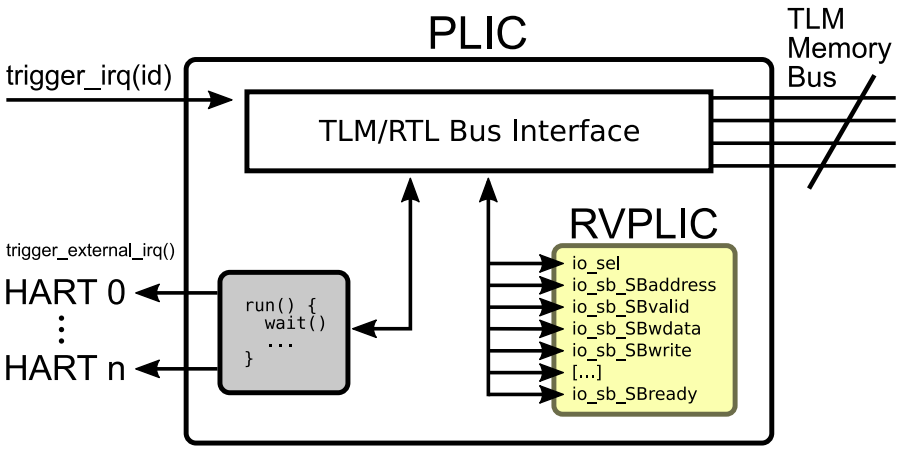


Figure 4.8: The structure of the DUT. TLM transactions are translated by the Bus Interface to RTL signals for the verilated `RVPLIC` (yellow).

For the integration into the verification framework, the original `RVPLIC` is first verilated into SystemC compatible C++ code, which is then converted by the *thread-to-function* method described in Subsection 4.1.4.2. This is already enough for RTL test-bench verification. For a TLM equivalence verification against the previously verified RISC-V PLIC, it also needs a TLM/RTL translation (see Figure 4.8). The common interface of direct interrupt triggering (top left), the HART notification (bottom left), and the TLM bus (top right) are left unchanged, while the actual logic is translated by the TLM/RTL Bus Interface (middle). This clocks in the signals needed to stimulate the RTL `RVPLIC` on each memory transaction (top right), and during normal operation via the `run()` function (bottom left).

4.2.4 Conclusion and Future Work

In this section, an approach to improve the verification methods for Verilog HDL models was presented. By adapting the verification process flow, previously published in [46], and extending the *PK* and its libraries, the possibility of symbolically executing RTL peripherals was enabled. This allows for a complete verification of such systems either by implementing dedicated test-benches or by cross-level equivalence testing via previously verified TLM models.
For future work, the proposed experimental setup promises to find interesting behavioral and interface-level errors once it is conducted in a more thorough case-study. It is furthermore worth investigating whether this previously unpublished approach is applicable to a broader range of experiments for evaluation. It also remains an open question whether the added complexity of fine-grained steps during the signal update phases result adds a significant impact of verification run-time, which could be addressed in further studies.

4.3 Dynamic Information Flow Tracking for Early Security Policy Validation

This section includes and extends published material from the published conference paper [45]. The structure is as follows:
The main motivation and the rationale behind security policies in general is introduced in Subsection 4.3.1, followed by Subsection 4.3.2 that discusses related work and mentions similar approaches in different fields. The definition of a security policy, declassification schemes, and the threat model in is given and introduced in Subsection 4.3.3. Then, in Subsection 4.3.4 the proposed VP-based DIFT approach for early and accurate verification of binaries targeting embedded systems with peripherals is presented. Finally, Subsection 4.3.6 describes the experimental results and Subsection 4.3.7 concludes the section with a discussion and inspirations for future work.

4.3.1 Introduction

As introduced in Section 2.1, embedded systems are small application-specific devices with a broad range of applications, from the highest requirements of safety in automotive or aerospace domains, over medical control systems to consumer electronics. All of them integrate several peripherals (devices) alongside the CPU core and extensively rely on embedded SW for configuration as well as complex functionality and communication. Following the trend of relying human life on an ever-growing number of embedded systems, avoiding security vulnerabilities in the embedded SW and HW is crucial to prevent leaking sensitive information or compromising safety. Besides functional verification on chip level and on the highest SW level, the verification of the SW/HW interaction level was previously either too late or not extensive enough.
Dynamic Information Flow Tracking [172, 173] is a powerful technique to analyze and protect software against a broad range of security related exploits by tracking and checking the information flow between inputs and outputs alongside the SW execution. Therefore, the DIFT engine is configured according to a security policy that essentially specifies the *classification* of input data, the rules of propagation (Information Flow Policy, IFP) and what kind of information is allowed to leave the system at which output interfaces (*clearance*) [174]. A *security policy* enables the specification of several fine-grained Access Control Models (ACMs) including *confidentiality* (secret data must not leak to untrusted places) as well as *integrity* (untrusted data must not influence sensitive registers/data).
While several SW- and HW-based approaches for DIFT have been proposed,

they suffer from deficiencies if SW targeting embedded systems is considered: i) SW-based approaches do not consider the HW in sufficient details and thus are susceptible to miss complex HW/SW interactions, e. g. due to interrupts, memory-mapped peripheral access as well as DMA controllers, and ii) HW-based approaches can only be used once the HW is available, hence the development and validation of security policies has to wait until then. At the same time, the security policy has influence on the SW development and HW design, hence it is important to consider security policies *early* in the design flow to avoid costly iterations.
In this section, a novel approach is presented that enables early and accurate DIFT of SW binaries targeting embedded systems. The approach works by integrating the DIFT engine in combination with the security policy into the VP (cf. Section 3.1) of an embedded system. As a recap, VPs are essentially executable SW models of the entire HW platform, and they are pre-dominantely implemented in IEEE-1666 SystemC [4] employing TLM [5] for abstract communication, and hence very fast simulation. Therefore, VPs are heavily used for early SW development and design space exploration [56, 175]. This section's approach extends the VP use-cases to early development and validation of security policies. Leveraging the VP, the proposed DIFT engine can track information flow on the embedded binary taking fine-grained HW/SW interactions into account. As SystemC is a C++ class library, the C++ features of templates and operator overloading can be leveraged to enable a transparent and virtually non-intrusive integration into the VP. The effectiveness and applicability of this approach will be demonstrated in several RISC-V experiments. This includes the development of a security policy for a car engine immobilizer, the detection of code injections, as well as the evaluation of the performance overhead.To stimulate further research, this implementation is published as open source in [50].
Summarizing, the major contributions of this section are:

- VP-based DIFT on embedded binary taking fine-grained HW/SW interactions into account
- Early development and validation of security policies, before the HW is available
- Transparent and virtually non-intrusive integration in the RISC-V VP
- Moderate performance overhead using VP-based DIFT

4.3.2 Related Work

Several HW-based DIFT approaches have been proposed. For example [176–179] focus on integration of DIFT into processor cores. There are also some approaches for extending DIFT support to the whole SoC [180–182]. Finally,

several approaches consider DIFT at RTL and gate-level in general [183, 184]. HW-based DIFT is complementary to the RISC-V VP-based DIFT, since the proposed approach enables early development and validation of security policies before the HW is available. In addition, requirements for the HW mechanisms can be derived. There also exist various SW-based DIFT approaches, e. g. [185–187], and methods based on static analysis and symbolic execution focusing on security validation, e. g. [188–190]. However, due to the source-level abstraction it is very challenging to provide accurate models for peripherals and to consider complex HW/SW interactions such as interrupts and DMA accurately. [172] integrated a DIFT engine into the Bochs x86 emulator to enable DIFT of SW binaries with full platform support. [191] is conceptually similar but uses QEMU. However, these approaches only target very specific security aspects (integrity-based validation [172] and malware detection [191]) instead of generic security policies, and only offer limited support for data flows outside of the CPU which are necessary to track fine grained HW/SW interactions. In addition, they do not support SystemC-based VPs, which is an industry-proven modeling standard (IEEE-1666, [4]).
Finally, an approach for SoC security validation using VPs has been proposed in [192]. However, the approach targets to find security vulnerabilities in the VP model, i. e. the HW. In [193] a dynamic VP-based information flow tracking method for security validation has been introduced. However, the approach only supports a much simpler security policy and threat model compared to this work. Overall, a VP-based generic binary-level DIFT approach specifically tailored for embedded SW binaries was previously not available.

4.3.3 Preliminaries: Security Policies and Threat Model

This subsection explains a more practical definition of security policies and how they can be applied to real-world systems. It shows that arbitrary security schemes like confidentiality and integrity can be simultaneously mapped to a single *security lattice*. This explanation is followed by a discussion about declassification issues and how to handle them. Finally, it concludes with the considered threat model of this approach.

4.3.3.1 Security Policy

A *security policy* consists of three parts: (i) the **classification** which assigns security classes to data that enters the system, (ii) the **Information Flow Policy** which is a lattice of security classes that describes the allowed information flow in the system and how the combination of differently labeled data is computed when the data propagates through the system, and (iii) the **clearance** which assigns

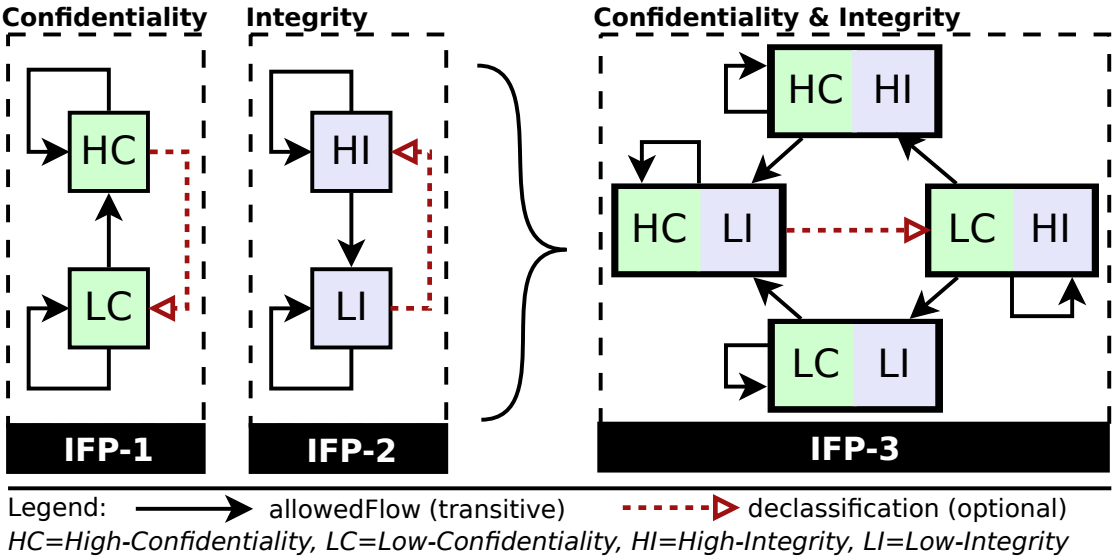


Figure 4.9: Three example IFPs. IFP-1 and IFP-2 show a simple policy that models confidentiality and integrity, respectively. IFP-3 is a natural combination of IFP-1 and IFP-2, thus modeling confidentiality and integrity together.

allowed security classes to system outputs and execution units. Recall that output/execution to/of data labeled with a certain security class is allowed iff the flow of the given security class X to the output/execution security class Y is allowed according to the IFP, i. e. there is a (transitive) connection from X to Y (denoted as `allowedFlow(X,Y)`).
Security policies enable the specification of several ACMs including *confidentiality* (secret data must not leak to untrusted places) as well as *integrity* (untrusted data must not influence sensitive registers/data).
In the following, an example is provided to demonstrate the principles of IFPs.

Example 1. *Figure 4.9 shows three IFPs. IFP-1 (see left side of Figure 4.9) has two security classes:* ***High-Confidentiality*** *(HC) and* ***Low-Confidentiality*** *(LC). Data flow is allowed from LC to HC but not the opposite way, i. e. confidential information is not allowed to leave the system through an output interface without appropriate clearance. IFP-2 (see middle of Figure 4.9) only allows data flow from a* ***High-Integrity*** *(**HI**) to a* ***Low-Integrity*** *(**LI**) security class, i. e. untrusted data (LI security class) is not allowed to influence sensitive data (HI security class).*
It is possible to consider confidentiality and integrity together as shown in IFP-3 (right side of Figure 4.9). IFP-3 is a natural combination of IFP-1 and IFP-2 by combining the individual security classes (hence, IFP-3 has 4 security classes) and allow a flow iff the individual flows are allowed in IFP-1 and IFP-2.

An important operation on an IFP (lattice) is the Least Upper Bound (LUB) operation. Essentially, the LUB of two security classes A and B denotes the next security class C that has equal or more restrictive clearance than both A and B. LUB is used to compute the resulting security class when applying operations (like addition, shift, etc) on data with different security classes. For example, in IFP-3 the LUB of A=(LC,LI) and B=(HC,HI) is C=(HC,LI) which essentially means that the resulting data becomes untrusted (as specified in A) but stays confidential (as specified in B).

4.3.3.2 Declassification

Another important concept is *declassification* [174, 194, 195]. It allows introducing fine-grained exceptions to the IFP by selectively changing the security class of specific data at run-time (cf. red dashed arrows in Figure 4.9). Typically, only trusted HW peripherals are allowed to declassify data to reduce the risk that an attacker exploits the declassification mechanism.
The main use case for declassification is to ensure that a system operating with confidential information can actually interact with the environment. It is a tight trade-off between strong security properties and controlled release of information, which may lead to unwanted attack vectors. A concrete example is changing the data classification to non-confidential after it has been encrypted. This is needed, as otherwise no encrypted information could be sent out on a public output interface because it depends on a secret key, even though in practice the secret key is sufficiently protected with getting only access to the encrypted data. Another example is a login prompt that provides a very small information about the internal (secret) information about the password with every attempted login and thus would be blocked by a strict security policy without declassification.
Thus, declassification is an important concept to ensure that a system operating with confidential information can interact with the environment. In a practical view, it heavily depends on the security requirements how declassification can work in an automated way. Hence, the design engineer is expected to add declassification requests manually at appropriate places. This can happen in an iterative way, starting with a strict policy without declassification and only adding declassification when necessary (i. e. a policy violation is detected by the proposed approach, which the design engineer deems to be too strict). Since the SW is considered to be untrusted and contain potentially malicious/erroneous code, only HW peripherals may declassify data to a lower/different security class in the case-studies. However, since RISC-V is an easily extensible ISA, custom CPU instructions can be added to handle declassification from SW. In this case, it could be done by, e. g., exectuing the instructions only in a privileged mode like

supervisor mode which can be combined with RISC-V's physical memory protection scheme [17].

4.3.3.3 Threat Model

In the following, a threat model is assumed where an attacker can write arbitrary (malicious) data at every input port of the embedded system. The goal of the attacker is, for example, to obtain confidential information or destroy the integrity of the system. The primary attack vector is to exploit functional SW bugs as well as accidentally included information flows, for example indirect/implicit information flow or an unsecured debug- or logging port. However, side-channels (e. g. timing and power) related attacks are not considered in this section.
In this proposed approach it is assumed that the HW is trusted and only the HW can perform declassification.
The security policy of the system is specified by the (security) engineer. How the RISC-V VP-based DIFT approach works, and can be used to validate the security policy, is presented in the next subsection.

4.3.4 DIFT for Embedded Binaries using VPs

The RISC-V VP-based DIFT approach tracks information flow on the binaries for embedded systems with peripherals. This is performed taking *fine-grained HW/SW interactions* into account, i. e. the flow is also tracked within the peripherals and the way back to the SW. The proposed DIFT engine benefits from the SystemC/C++ features of templates and operator overloading to enable a transparent and virtually non-intrusive integration into the VP.
The following subsections start with an overview of the proposed approach (Subsection 4.3.4.1), then present more details on the proposed DIFT engine and VP integration (Subsection 4.3.4.2), and finally present an example scenario in Subsection 4.3.4.4 to illustrate the specification and encoding of a security policy.

4.3.4.1 Approach Overview

The proposed approach is centered around a VP that represents the target SoC. An overview of the proposed approach is shown in Figure 4.10. A DIFT engine is integrated into the VP that enables DIFT at the VP level (see center of Figure 4.10), along with specified security policies that are encoded into the VP and checked alongside the SW execution. Please recall from Subsection 4.3.3.1 that a security policy consists of three components that reason about security classes: 1) classification, 2) IFP, and 3) clearance.

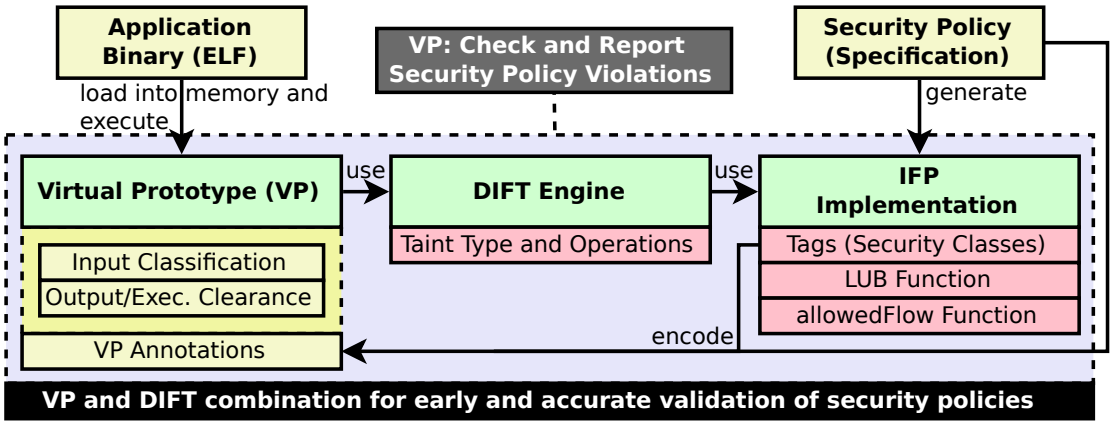


Figure 4.10: Overview of the RISC-V VP-based approach for early and accurate validation of security policies.

Security classes in the DIFT engine are represented as (integer) *tags* by simply mapping each security class of the IFP to a unique tag (see first red box on the right side of Figure 4.10 below IFP implementation). Tags are assigned to input data (for example a secret key stored in memory or the data generated by a sensor peripheral) and output interfaces (e. g. the output port of a UART) according to the classification and clearance mappings, respectively (see left side below Virtual Prototype in Figure 4.10). In addition, *execution clearances* are needed to be specified by assigning tags to specific execution units in the CPU. The concept of execution clearance is discussed later in Subsection 4.3.4.3 in more detail. To implement the specified IFP, LUB and `allowedFlow()` functions are needed that operate on tags according to the IFP semantics (bottom red boxes on the right side of Figure 4.10). Based on these two functions, the DIFT engine propagates and checks the tags, triggering a runtime error upon violation.

4.3.4.2 DIFT Engine

Modeling Security Policies With a given security policy (see Subsection 4.3.3.1), the modeled system is divided into different security classes and a list of allowed flows between them. To integrate this abstract information into the proposed tracking extension, firstly the user has to map the classes to unique integer ids in a central configuration file. This integer ID (also called *taint*) refers to its specific security class and is propagated along its data. Secondly, to model the lattice of a give information flow policy, a combination operator and a relation operator are needed [196]. To account for the combination operator, the user has to define relations of allowed flows between security classes in a function. This combination function is called whenever an operation is applied on two or more operands (e. g. an `add`-

instruction) and specifies the security class of the outcome. The relation operator is called `allowedFlow()` and is used to determine if an information of a certain security class may flow to another. It returns `true` if a path between two security classes exists, otherwise `false`. While the functions are free to map a user's security layout, it is important that the relation operator is reflexive, transitive and anti-symmetric to hold the security assumptions mentioned in Subsection 4.3.3.1.

```
1  static Taint combine(const Taint a, const Taint b) {
2    if (a == b) {
3      return a;
4    } else {
5      MergeStrategy am = static_cast<MergeStrategy>(a & mergeMask);
6      MergeStrategy bm = static_cast<MergeStrategy>(b & mergeMask);
7
8      if(am == MergeStrategy::forbidden || bm == MergeStrategy::forbidden)
9      {
10       throw(TaintingException("merging forbidden by policy"));
11       return 0;
12     }
13     switch (am) {
14       case MergeStrategy::lowest:
15         switch(bm)
16         {
17         case MergeStrategy::lowest: //low: low
18           return a < b ? a : b;
19         case MergeStrategy::highest: //lowest and highest, choose highest
20           return b;
21         case MergeStrategy::none:  //lowest and none: demote to none
22           return 0;
23         default:
24           break;
25         }
26       break;
27       case MergeStrategy::highest:
28         switch(bm)
29         {
30         case MergeStrategy::lowest: //lowest and highest, choose highest
31         case MergeStrategy::none: //highest and none: promote to highest
32           return a;
33         case MergeStrategy::highest: //highest: highest
34           return a > b ? a : b;
35           default:
36           break;
37         }
38       break;
39     }
40   }
41   throw(TaintingException("invalid merging policy"));
42   return 0;
43 }
44
45 static bool allowedFlow(const Taint to, const Taint from)
46 {
47   if (to == from) {
48     return true;
49   } else{
50     MergeStrategy tom = static_cast<MergeStrategy>(to & mergeMask);
51     MergeStrategy frm = static_cast<MergeStrategy>(from & mergeMask);
52     if(frm == MergeStrategy::forbidden || tom == MergeStrategy::forbidden)
53       return false;
54     switch (frm) {
55     case MergeStrategy::lowest:
```

```
56        //this includes to = none = 0 < from
57        return tom == MergeStrategy::highest ? true : from > to;
58      case MergeStrategy::highest:  //high/none to lowest forbidden
59      case MergeStrategy::none:
60        return tom == MergeStrategy::lowest ? false : from < to;
61      default:
62        break;
63      }
64    }
65    return false;
66  }
```

Listing 4.6: The two functions necessary to model a security flow policy: The combination operator and the relation operator.

Implementation Sketch The main ingredient of the proposed DIFT approach is a custom `Taint` data type with a template parameter *T* for the to be tainted data. Listing 4.7 shows the main code excerpts of the `Taint` struct: *value* stores the data (Line 4) and *tag* captures the assigned security class (Line 5).
This data type is used to represent CPU and peripheral registers as well as memory bytes. More precisely, the open-source RISC-V VP (proposed in Section 3.1, [38, 72]) is chosen as a representative example, with the following three modifications in the SystemC VP model:

1. Replace the register types to use the `Taint<int32_t>` data type instead of the native int32_t. With the `Taint` operator overloading (see Line 33 and following), the RISC-V instruction execution, e. g. an addition `regs[RD] = regs[RS1] + regs[RS2]`, works without any further modification, but now also performs the tainting with respect to the given security policy (Line 35 shows the addition and Line 36 shows the taint update based on the least upper bound of both arguments, respectively).
2. Integrate execution clearance checks at appropriate locations (primarily to handle implicit information flows, more details follow in the next section).
3. Adapt the memory interface, which is responsible to translate load/store instructions into TLM transactions, to support tainted values. To ensure compatibility with TLM transactions, the `Taint` data type provides the `to_bytes` (Line 12) and `from_bytes` (Line 18) functions that convert any `Taint` (e. g. `Taint<uint32_t>`) to and from an array of tainted bytes (i. e. `Taint<uint8_t>`), respectively. Casting the `Taint<uint8_t>` array into a char pointer allows to transparently embed the `Taint` data array into a TLM transaction and route it as usual through the bus. The receiving HW peripheral obtains the `Taint<uint8_t>` (array) pointer by casting the char data pointer of the TLM transaction back.

```
typedef uint8_t Tag;
template <typename T>
class Taint {
  T value;  // data
  Tag tag; // security class

  Taint(const T value, const Tag tag) {
    this->value = value;
    this->tag = tag;
  }
  // convert instance to and from a Taint byte array
  void to_bytes(Taint<uint8_t> ar[sizeof(T)]) const {
    for (uint8_t i=0; i<sizeof(T); i++) {
      ar[i].value = ((uint8_t*)(&value))[i]; // copy each byte
      ar[i].tag = tag; // use the same tag for each byte
    }
  }
  void from_bytes(Taint<uint8_t> ar[sizeof(T)]) {
    tag = ar[0].tag;
    for (uint8_t i=0; i<sizeof(T); i++) {
      tag = LUB(tag, ar[i].tag); // combine all tags
      ((uint8_t*)(&value))[i] = ar[i].value; // copy each byte
    }
  }

  void check_clearance(uint8_t required_tag) const {
    if (!allowedFlow(tag, required_tag))
      throw ClearanceException();
  }

  // Operator overloading to perform regular operation
  // according to data of type T _and_ tainting
  Taint<T> operator+(const Taint<T>& other) {
    // apply operation and merge tags according to IFP
    Taint<T> ans(value + other.value);
    ans.setTag(LUB(tag, other.tag));
    return ans;
  }
  //...other operators implemented similarly...
}
```

Listing 4.7: Code excerpts of custom `Taint` data type using overloaded operators.

```
1  struct SimpleSensor : public sc_core::sc_module {
2    tlm_utils::simple_target_socket<SimpleSensor> tsock;
3    // memory-mapped data frame
4    std::array<Taint<uint8_t>, 64> data_frame;
5
6    // security tag for the generated data
7    uint8_t data_tag = Taint::LowConf;
8
9    // register SystemC thread and TLM transport function
10   SC_HAS_PROCESS(SimpleSensor);
11   SimpleSensor(sc_core::sc_module_name) {
12     tsock.register_b_transport(this, &SimpleSensor::transport);
13     SC_THREAD(run);
14   }
15   void run() {
16     while (true) {
17       sc_core::wait(25, sc_core::SC_MS); // 40 times per second
18       // fill with random printable data
19       for (auto &n : data_frame) {
20         // generate data of the specified security class
21         n = Taint<uint8_t>(rand() % 96 + 128, data_tag);
22       }
23       // notify interrupt controller (IC) about new sensor data
24       IC->trigger_interrupt(2 /*IRQ NUMBER*/);
25     }
26   }
27
28   // the VP bus routes transactions to this function
29   void transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &delay) {
30     auto addr = trans.get_address();
31     auto cmd = trans.get_command();
32     auto len = trans.get_data_length();
33     auto ptr =
34         reinterpret_cast<Taint<uint8_t>*>(trans.get_data_ptr());
35     if (addr <= 63) {
36       // access data frame
37       assert(cmd == tlm::TLM_READ_COMMAND);
38       assert((addr + len) <= data_frame.size());
39       // return last generated random data at requested address
40       memcpy((void *)ptr, &data_frame[addr],
41           sizeof(Taint<uint8_t>) * len);
42     } else {
43       if (cmd == tlm::TLM_READ_COMMAND) {
44       // the configured security class is not confidential
45         *ptr = Taint<uint8_t>(data_tag, Taint::LowConf);
46       } else if (cmd == tlm::TLM_WRITE_COMMAND) {
47         data_tag = *ptr;
48       } else {
49         assert(false && "invalid access);
50       }
51     }
52   }
53 };
```

Listing 4.8: Implementation of a sensor peripheral using SystemC TLM and the proposed DIFT approach.

Besides the CPU of the VP, also some adaptations in the HW peripherals were done. Listing 4.8 shows a sensor peripheral implementation (other peripherals are implemented similarly). The sensor contains a memory-mapped 64 byte *data frame* (Line 4) using the custom `Taint` data type to store a tag alongside the value. To allow the sensor to send confidential or nonconfidential data, an 8 bit `data_tag` register (Line 7) was added, in where the security level of the produced data can be defined from SW. The sensor periodically generates new data in the SystemC `run` thread using the configuration as given by the *data_tag* (Lines 19 to 22). By this, depending on the concrete application, differently classified sensor sources can be modeled. SW read/write accesses are routed by the VP's bus via TLM transactions to the *transport* function. The TLM `generic_payload` provides the transactions data and size. Based on the transaction type, either a read or a write access is handled in the sensor peripheral.
To extend the original version of the sensor, only 6 lines of code needed to be changed (see highlighted lines in Listing 4.8). In Lines 4 and 41 the modifications were straight forward from `uint8_t` to `Taint<uint8_t>`. Line 34 casts the transport data pointer to an array of tainted bytes instead of the original char buffer. This convention needs to be adapted in every peripheral that uses TLM transactions. In Line 21, tagged random data (the sensor's source) is generated using the `Taint` constructor with the tag as the second argument. Note, that Line 47 does not have to be changed; this is due to the overloaded conversion routine of the `Taint` class. This implicit cast to its underlying type (here `uint8_t`) requires by default a low confidentiality (*LC*) tag, throwing an error otherwise.
In summary, the integration of the DIFT engine into the VP (including peripherals) only affected 6.81 % of lines of code of the original VP, of which 58.7 % are type-conversions (as seen e. g. in Listing 4.8, Line 4).

4.3.4.3 Execution Clearance

Beside direct information flow from computational instructions and clearance checks at output interfaces, the DIFT engine has also to consider implicit information flow (*confidentiality* specific aspect) and protection of internal resources (*integrity* specific aspect). Three operations in the CPU core can be identified that are relevant in this context: 1) branch execution, 2) instruction fetching, and 3) memory access. These operations are handled by assigning each of them an execution clearance (i. e. a security class represented as tag). For example, the instruction fetch unit performs a clearance check based on its own security class *A* and the security class *B* of the fetched instruction, i. e. it requires `allowedFlow(B, A) == true`. For branch instructions the clearance check is performed on the branch condition and for memory access operations on the address. The execution

clearance is configurable to let the engineer select the most suitable configuration (it is specified in the security policy). Furthermore, fine-grained exceptions to the execution clearance are possible by using declassification (recall Subsection 4.3.3.1) to selectively change the security class of specific data (e. g. one specific branch condition) at runtime. Only trusted HW peripherals are allowed to declassify information, to reduce the risk that an attacker exploits the declassification mechanism. The rationale behind this execution clearance problem is discussed in the following paragraphs for the three operations in the CPU core in more detail:

Branch Execution Observing the control flow can implicitly reveal confidential information. Consider for example a branch `if(secret == 1) then public = 1` with a confidential condition. The control flow dependence of *public* with *secret* allows to infer information about the value of *secret* by outputting *public*. Specifically, either it has the new or the previous value; even though no direct data flow dependence exists between *secret* and *public*. Therefore, control flow dependencies need to be considered alongside data flow dependencies by the DIFT engine. However, in the presence of an attacker that may be able to inject code (by exploiting SW bugs), their computation is very challenging. Requiring an *LC* clearance on the branch condition is a safe approximation to avoid leaking sensitive information. Please note, the same clearance is used to check the interrupt/trap handler address.

Instruction Fetch Similar to branches, instruction fetching/decoding can also leak sensitive information. For example; consider a confidential memory word fetched by the CPU. In case the word is an illegal instruction, a jump to the (SW error) trap handler is performed. The trap handler may write to public variables, hence posing a risk of leakage. Also, the behavior of the system changes based on the fetched instruction which may provide an additional attack surface. This includes timing and power related side-channel attacks, though this is not the focus of this work. Again, requiring an *LC* clearance on the fetched instruction is a safe approximation to avoid leaking sensitive information.
In addition, to reduce the risk of code injection by exploiting SW bugs, it makes sense to also use a *HI* clearance for instruction fetching. This prevents execution of data from external untrusted sources. However, it still cannot fully prevent code injection, since an attacker might be able to exploit bugs in the embedded SW to inject malicious code by re-using trusted code from memory.
Because of this possibility, the SystemC module of the CPU is not able to jump based on a non-zero tainted register. In Listing 4.9, a part of the branching instructions are shown. They use the overloaded arithmetic functions defined in

Listing 4.11. Due to the implicit calls to the proposed conversion functions, no further adaptions have to be made to the existing instruction set simulator of the CPU.
A comparison of two registers, e. g. `if (regs[instr.rs1()] == regs[instr.rs2()]` first calls the overloaded `operator==` function (see 4.11, Line 23). This function returns a boolean that is elevated into the right confidentiality level or domain, depending on the merging strategy. The tainted boolean is then converted into the native underlying boolean, calling the conversion operator on Line 45. The conversion function either returns the actual boolean value, or throws a run-time tainting exception if the value is on any confidentiality level other than 0.

```
1  [...]
2  case Opcode::BEQ:
3     if (regs[instr.rs1()] == regs[instr.rs2()])
4        pc = last_pc + instr.B_imm();
5  break;
6
7  case Opcode::BNE:
8     if (regs[instr.rs1()] != regs[instr.rs2()])
9        pc = last_pc + instr.B_imm();
10 break;
11
12 case Opcode::BLT:
13    if (regs[instr.rs1()] < regs[instr.rs2()])
14       pc = last_pc + instr.B_imm();
15 break;
16
17 case Opcode::BGE:
18    if (regs[instr.rs1()] >= regs[instr.rs2()])
19       pc = last_pc + instr.B_imm();
20 break;
21 [...]
```

Listing 4.9: The part of the VP CPU instruction set simulator handling branching instructions. An implicit demotion is attempted in the arithmetic functions of the tainted registers.

Memory Access A memory access with confidential address can also leak information. For example consider `Mem[secret] = public`. Then, the value of *secret* may be inferred by querying the memory, e. g. `public2 = Mem[i]` and check `public == public2` for $i = [0, \ldots, \texttt{int\_max}]$. Even if the value of `Mem[secret]` is confidential too, an inference of the *secret* address is still possible by writing `Mem[0]`, `Mem[1]`, etc., to a public output interface and observe if an error is raised (due to insufficient clearance in case `Mem[i]` is confidential). Using an *LC* clearance on the memory address prevents these attacks.

4.3.4.4 Example Scenario: System Description and Security Policy

This subsection describes an exemplary cross-section of a system that uses the proposed DIFT approach. It shall help with understanding the data flow between HW and SW.

Scenario Description Consider an SoC with an input sensor, an input/output UART and a designated memory storing secret data. Intuitively, the security policy specifies that the secret data is neither leaked nor modified. Hence, as IFP is possible to use IFP-3 (defined in Subsection 4.3.3.1) and use the security class (*LI*,*LC*) to classify sensor input data as well as clearance for the output UART. The secret key is defined as (*HI*,*HC*). IFP-3 has four security classes, hence it uses four tags in the DIFT engine to distinguish them: $tag(LI,LC) = 0$, $tag(HI,LC) = 1$, $tag(LI,HC) = 2$, and $tag(HI,HC) = 3$. More details on the SW side is shown in the next paragraph, with VP side to be described in the paragraph after that.

SW Side Listing 4.10 shows example SW to be executed on the VP. The SW reads sensor data into a local buffer (Lines 27 to 28) and writes the result to the UART peripheral (Lines 32 to 37). The sensor applies a filter that returns values in a specific range (Line 21). The program registers an interrupt handler for the sensor (Line 16) to detect when new sensor data is available.
The peripherals (i. e. sensor, UART and the designated memory) are accessed using memory-mapped I/O. These memory-mapped I/O accesses are routed to the corresponding peripheral in the VP and processed there. For example, Line 28 is a read access for the local address `i` from the sensor peripheral that is mapped to the memory address `0x20000000` in the VP. The secret memory returns data with $tag(HI,HC)=3$ while the sensor returns data with $tag(LI,LC)=0$. A write to the UART TX register address queues the data for printing. Before actually printing the data to console, the output interface checks that the data has a permitted tag according to the IFP and its clearance $tag(LI,LC) = 0$.
In this SW example, a security policy violation is detected in the UART, initiated by the memory-mapped write access in Line 36. The reason is that in the last iteration of the for loop, i. e. `i = BUF_SIZE`, the array access `buf[i]` overflows beyond the array bounds resulting in a read of the secret variable, that is placed right after the buf array on the stack in this example (Line 17). The occurrence of the error depends on the sensor configuration (Line 21) and hence it is important to also consider peripherals to obtain accurate results.

VP Side As a recap, Listing 4.8 shows the relevant parts of the corresponding SystemC TLM sensor peripheral implementation (counterpart for the SW example

in Listing 4.10 – the other peripherals are implemented similarly). The sensor has a memory-mapped 64 byte data `frame` (mapped to local address 0 to 63) and a 32 bit `filter` register (mapped to local address 64 to 67). Both `frame` and `filter` use the custom `Taint` data type to store the tag alongside the value.
The sensor periodically generates new data in the SystemC `run` thread and processes it based on the filter value (Lines 19 to 22) which is configurable by the SW. The data is tagged with *tag*(*LI*,*LC*) (Line 21) using the sensor data tag (Line 7).
SW read/write accesses are routed by the VP bus via TLM transactions to the `transport` function (see also Section 2.2). The TLM `generic_payload` provides the transaction data and size. Based on the transaction, either a read or write access is handled in the sensor peripheral. The transaction data pointer `uint8_t*` is *packed* into the `Taint` data type to pass along the data tag(s) for the TLM transaction. Here, the `to_bytes`/`from_bytes` functions (briefly introduced in Subsection 4.3.4.2, Line 12) are *implicitly* used to convert data to and from the tainted byte array. This allows to implement read/write accesses in a natural way using the access functions of the `Taint` data type.
The VP side of this example illustrated how to integrate tainting support into SystemC-based TLM peripherals.

```
1  #define NORMAL_RANGE_FILTER 1
2  #define SPECIAL_RANGE_FILTER 2
3  #define BUF_SIZE 64
4  // memory-mapped register (used as input/output)
5  volatile const uint8_t*  UART_TX_ADDR       =  (uint8_t*) 0x10000000;
6  volatile const uint8_t*  SENSOR_DATA_ADDR   =  (uint8_t*) 0x20000000;
7  volatile const uint32_t* SENSOR_FILTER_ADDR = (uint32_t*) 0x20000040;
8  volatile const uint32_t* SECRET_MEM_ADDR    = (uint32_t*) 0x30000000;
9  bool sensor_has_data = 0;
10 void sensor_irq_handler() {
11   sensor_has_data=1;
12   // [...]
13 }
14
15 int main() {
16   register_interrupt_handler(2 /*SENSOR IRQ NUMBER*/, sensor_irq_handler);
17   uint32_t key = *SECRET_MEM_ADDR; // tag(key) = 3
18   uint8_t buf[BUF_SIZE];
19
20   // config: apply post-process filter to return values in range [0..63]
21   *SENSOR_FILTER_ADDR = NORMAL_RANGE_FILTER;
22   // wait for sensor input
23   while (!sensor_has_data) {
24     asm volatile ("WFI");//WFI=Wait For (any) Interrupt
25   }
26
27   for (int i=0; i<BUF_SIZE; ++i) {
28     buf[i] = *(SENSOR_DATA_ADDR+i); // tag(buf[i]) = 0
29   }
30
31   // security policy violation due to (read) buffer overflow (loop condition
     ↪ should be: i<BUF_SIZE)
32   for (int i=0; i<=BUF_SIZE; ++i) {
```

```
33      if (buf[i] > 127) // stop on large values
34        break;
35      // buffer overflow (on the stack) into the secret key on last iteration
36      *UART_TX_ADDR = buf[i]; // error (will be detected in the UART): buf[BUF_SIZE]
         ↪ tag (tag(HI,HC)=3) incompatible to UART_TX output tag (tag(LI,LC)=0), i.e.
         ↪ allowedFlow(3,0) is false
37    }
38    return 0;
39  }
```

Listing 4.10: Example SW to illustrate the approach for checking security policies by introduction, propagation and checking of tags.

4.3.4.5 Branches with Confidential Conditions

An unintentional data flow may happen through conditional branching based on a confidential value. Imagine an adversary that leaks confidential data just by comparing a secret variable with known values, e. g. `if(secret == 1) then notsecret = 1` (see Subsection 4.3.6). With this procedure, the *program counter* would implicitly leak information about the secret, even when no assignments would happen in the conditional branch (e. g. `sleepSeconds(secret)`). Because of this possibility, the SystemC module of the configured CPU is not able to jump based on a non-zero tainted register. In Listing 4.9, a part of the branching instructions are shown. They use the overloaded arithmetic functions defined in Listing 4.11. Due to the *implicit* calls to the conversion functions, no further adaptions have to be made to the existing instruction set simulator of the CPU. A comparison of two registers, e. g. `if (regs[instr.rs1()] == regs[instr.rs2()]` first calls the overloaded `operator==` function (see Listing 4.11, Line 23). This function returns a boolean that is elevated into the right confidentiality level or domain, depending on the merging strategy. The tainted boolean is then converted into the native underlying boolean, calling the conversion operator on Line 45. The conversion function either returns the actual boolean value, or throws a run-time tainting exception if the value is on any confidentiality level other than *LC*.

4.3.5 SystemC TLM-2.0 Compatible Tainting Engine for Virtual Prototypes

To have fine-grained domain separation, the underlying data types of CPU registers were expanded to contain one byte of tainting information per byte of usable data by implementing new generic data types. These data types implement all arithmetic functions as well as conversions to their underlying base-type to maintain full compatibility to the existing functions. This allows a minimal adaption to the SystemC model.

```
1  static Taint mergeTaintingValues(const Taint a, const Taint b) {
2    if (a == b) {
3      return a;
4    } else {
5      MergeStrategy am = static_cast<MergeStrategy>(a & mergeMask);
6      MergeStrategy bm = static_cast<MergeStrategy>(b & mergeMask);
7      if (am != bm) {
8        throw(TaintingException("combination of different merging policies"));
9        return 0;
10     }
11     switch (am) {
12     case MergeStrategy::forbidden:
13       throw(TaintingException("merging forbidden by policy"));
14       return 0;
15     case MergeStrategy::highest:
16       return a > b ? a : b;
17     default:
18       throw(TaintingException("invalid merging policy"));
19     }
20   }
21 }
22 [...]
23 Taint<bool> operator==(const Taint<T>& other) {
24   Taint<bool> ret(value == other.value);
25   ret.setTaintId(mergeTaintingValues(getTaintId(), other.getTaintId()));
26   return ret;
27 }
28
29 Taint<bool> operator==(const T& other) {
30   Taint<bool> ret(value == other);
31   ret.setTaintId(getTaintId());
32   return ret;
33 }
34 [...]
35 T demote(Taint level) const {
36   // if forbidden merge policy, the merge throws. If highest, the highest ID may
       ↪ only be lower or equal
37   uint8_t max = mergeTaintingValues(getTaintId(), level);
38   if (level < max) {
39     throw TaintingException("Invalid demotion of ID " +
         ↪ std::to_string(getTaintId()) + " (allowed: " + std::to_string(level) + ")");
40   }
41   return value;
42 }
43
44 operator T() const {
45   return demote(MergeStrategy::none);
46 }
```

Listing 4.11: The part of the VP tainting mechanism handling the merging of variables, two arithmetic operators and the implicit demotion to the underlying type.

The minimal set of adapted modules are the CPU registers and the internal bus. To save memory overhead, only parts of the internal RAM may accept tainted data types, if necessary. When other peripherals like the DMA, random number generator etc. are left unchanged, the default conversion automatically assumes a low-confidentiality domain *LC*. Any higher-security data flow would fail to these devices, if not otherwise stated in the peripherals program code. If a conversion fails due to the security policy, a run-time error is thrown which may be handled by the software running on the VP. To simplify the policy enforcement, any merging strategies between different classified data are defined at a single file that concentrates all tainting-specific source code.

4.3.6 Experimental Evaluation

The proposed VP-based DIFT approach for early development and validation of security policies has been implemented by integrating the DIFT engine into the open-source SystemC TLM RISC-V VP, proposed in Section 3.1. The proposed approach is evaluated in three steps. First, Subsection 4.3.6.1 presents a case-study on developing and validating the security policy for an ECU of a car engine immobilizer. Then, the effectiveness of the proposed approach in detecting code injection is shown in Subsection 4.3.6.2. Finally, the performance overhead of the proposed DIFT engine is evaluated in Subsection 4.3.6.3.

4.3.6.1 Security Policy Evaluation: Car Engine Immobilizer

In the first experiment, an ECU of a car engine immobilizer is considered as case-study . The immobilizer holds a secret key (Personal Identification Number (PIN)) in memory which is used for a challenge-response protocol together with the engine's ECU for authentication. Therefore, the engine sends a challenge (random number) and the immobilizer returns a response (challenge encrypted by a PIN using an AES peripheral). The engine holds the same PIN as the immobilizer and checks the response by performing the same encryption. The communication channel between the ECUs is established by reading and writing to a CAN peripheral. Please note, that in this authentication process the PIN is never exchanged on the CAN bus in plain-text.
The goal is that the PIN is neither leaked (to prevent unauthorized access to the car) nor modified (to keep the car operational). Thus, the security policy uses IFP-3 (see Subsection 4.3.3.1) and classifies the key as (*HC,HI*) and use (*LC,LI*) clearance on all input and output devices (including the CAN peripheral). In addition, the AES peripheral has (*HC,HI*) clearance and performs declassification,

i. e. all encrypted data has (*LC*,*LI*) classification, so it can be sent out on the CAN bus.
By running a manually written test-suite, it could be observed that the security policy is violated because the immobilizer can be instructed to perform a complete memory dump (including the secret key) on the UART (which was previously implemented for debugging purposes). This security vulnerability could easily be fixed by correcting the debug function to exclude the memory region of the key.
For further evaluation purposes, the immobilizer's SW was further extended to include common attack scenarios: 1) directly or indirectly (through an intermediate buffer or buffer overflow) write the PIN to an output interface, 2) use control flow instructions that depend on the PIN, and 3) override the PIN in memory with external data. All attack scenarios have been detected using the proposed approach successfully.
However, further testing revealed another attack scenario that is still not covered by the security policy yet. While the current security policy prevents overwriting the PIN with external data (i. e. *LI*), it does not protect against overwriting with trusted data (i. e. *HI*). Thus, according to the security policy it is still possible to e. g. overwrite byte 2, byte 3, etc. of the PIN with byte 1. This significantly reduces the encryption entropy (all bytes in the PIN are equal) and hence enables a brute-force attack (by trying 256 possibilities) to obtain the PIN byte by byte from the encrypted response on the CAN bus. This issue can be fixed by modifying the security policy to use a separate security class for each byte of the PIN, hence further reducing the risk of a security vulnerability.

4.3.6.2 Code Injection Protection

In the second experiment, effectiveness of the proposed approach in detecting code injection is evaluated. Therefore, it uses the Wilander-Kamkar buffer overflow attack suite [197] which has been ported for RISC-V by [177], though some attacks are not applicable (N/A) in the RISC-V environment, primarily due to differences in the calling convention [177]. Table 4.5 shows an overview of the attacks. The suite features several attack patterns that exploit buffer overflows on the stack or the Heap/BSS/Data segment (column: *Location*) to target e. g. the return address, base pointer, function pointer or longjmp buffer (column: *Target*). The buffer is either accessed directly or indirectly through a pointer (column: *Technique*). All attacks try to inject and execute a pre-defined malicious code payload which is a serious security breach and may gain the attacker complete access to the system.
To protect against code injection, a security policy based on IFP-2 was chosen. The memory holding the program is classified as *HI* during program loading, and the instruction fetch unit in the CPU is also set to *HI* clearance, i. e. it will raise an

Table 4.5: Modified Wilandar-Kamkar buffer overflow test-suite results.

Atk #	Location	Target	Technique	Result
1	Stack	Function Pointer (param)	Direct	N/A
2	Stack	Longjmp Buffer (param)	Direct	N/A
3	Stack	Return Address	Direct	Detected
4	Stack	Base Pointer	Direct	N/A
5	Stack	Function Pointer (local)	Direct	Detected
6	Stack	Longjmp Buffer	Direct	Detected
7	Heap/BSS/Data	Function Pointer	Direct	Detected
8	Heap/BSS/Data	Longjmp Buffer	Direct	N/A
9	Stack	Function Pointer (param)	Indirect	Detected
10	Stack	Longjump Buffer (param)	Indirect	Detected
11	Stack	Return Address	Indirect	Detected
12	Stack	Base Pointer	Indirect	N/A
13	Stack	Function Pointer (local)	Indirect	Detected
14	Stack	Longjmp Buffer	Indirect	Detected
15	Heap/BSS/Data	Return Address	Indirect	N/A
16	Heap/BSS/Data	Base Pointer	Indirect	N/A
17	Heap/BSS/Data	Function Pointer (local)	Indirect	Detected
18	Heap/BSS/Data	Longjmp Buffer	Indirect	N/A

error when fetching instructions with *LI* classification. All other information in the system (including data coming from the serial console) is classified as *LI*. Because the test-suite features a well-defined function as a representation for malicious code, this function was specifically classified as *LI* before conducting the tests. In a real world scenario, this code would be inserted by external components (e. g. the terminal) and thus also have an *LI* security class. With this security policy all applicable attacks were detected which demonstrates the effectiveness of the proposed approach in detecting code injection attacks.

4.3.6.3 Performance Overhead Evaluation

To evaluate the performance overhead of the DIFT engine, the execution times of the proposed approach (denoted *VP+*) are compared against the original RISC-V VP (denoted *VP*). All benchmarks are executed on a Linux machine with Fedora 29 and an Intel™ i5-8250U processor.
Table 4.6 shows the results. The first three columns report the benchmark name, the number of executed instructions (column: *#instr. exec.*) and number of assembler opcodes (column: *LoC ASM* in the final binary (which includes linked

Table 4.6: Results on the performance overhead of the proposed DIFT approach.

Benchmark	#instr. exec.	LoC ASM	Sim. Time VP	Sim. Time VP+	MIPS VP	MIPS VP+	Ov.
qsort	430,719,182	17,052	11.6	18.3	37.1	23.5	1.6x
fibonacci	5,999,999,997	14	136.1	191.4	44.1	31.3	1.4x
dhrystone	1,370,010,911	17,158	39.1	60.1	35.1	21.1	1.6x
primes	7,114,988,890	16,793	186.3	390.0	38.1	18.2	2.1x
sha512	7,578,047,617	17,862	251.6	441.5	30.1	17.1	1.8x
simple-sensor	1,393,000,060	2,970	67.6	83.0	20.6	16.7	1.2x
freertos-tasks	5,937,843,750	11,146	141.6	411.5	41.9	14.4	2.9x
immo-overflow	931,081,431	17,191	26.1	49.1	35.6	18.9	1.8x
immo-fixed	931,083,025	17,188	26.1	46.9	35.6	19.8	1.8x
– average –	3,536,527,633	14,309	103.4	207.3	33.2	17.0	2.0x

libraries). The remaining columns compare the simulation time (in seconds) and MIPS for the RISC-V VP (VP) and VP+, and the resulting performance overhead of VP+ (column: *Ov.*). The last row summarizes the results by providing average values for all benchmarks. The following programs are used as benchmarks: *qsort* from the *newlib* C-library, a recursive *fibonacci* implementation written in RISC-V assembler, a standard *dhrystone* implementation, a prime number generator, the hash sum function *sha512*, a simple-sensor application that copies randomly generated data from a sensor to a UART peripheral, a FreeRTOS application SW scheduling two interleaved tasks, and the different car immobilizer SW images (see the previous section).
It can be observed that VP+ is in average a factor of two times slower (worst and best case at 2.9 times and 1.2 times, respectively) than the original VP on the benchmark set, which is a very reasonable performance overhead.

4.3.7 Conclusion and Future Work

In this section, a VP-based DIFT approach for embedded binaries was introduced, taking fine-grained HW/SW interactions into account. This approach supports a wide range of security policies which can be fully configured by the user. Moreover, since this approach leverages SystemC-based VPs security policies can be developed and validated early, i. e. before the HW is available. In addition, the benefits offered by SystemC and C++ could be utilized; in particular templates and operator overloading, to design a taint data type that enables a straightforward

integration of the DIFT engine into the VP platform. Extensive RISC-V experiments demonstrated the effectiveness of this approach.
For future work, automatic test-case generation methods could be investigated that consider the SW as well as the VP level (e. g. combined with [96, 198]) and/or can be tailored for stress-testing security policies. To support the development and validation process, the performance could be optimized further by leveraging DBT/JIT instead of interpretation-based execution in the CPU core.
To test (and iteratively refine) the existing confidentiality policies and their encoding, including the declassification mechanisms, a (comprehensive) set of test-cases is required. To help in this process, automatic coverage-driven test-case generation approaches could be worth investigating. In particular fuzzing and constrained-random generation techniques are very promising. A combination of SW and VP (branch) coverage can be used to guide the test generation process. To close remaining coverage gaps, this method could also be combined with symbolic execution engines such as SymSysC (described in Section 4.1).
Currently, the proposed system reports a policy violation in case control flow depends on confidential data. This solution is a safe (over-)approximation to ensure that no confidential information is leaked. Worth investigating are thus techniques to provide a more accurate solution to reduce this over-approximation. A promising idea is to perform a lookahead-based analysis that runs ahead for a certain number of instructions (based on the current program counter when reaching a branch instruction with a confidential condition) and analyzes these instructions (based on the current execution state of the CPU core) to reconstruct at run-time the control flow of the immediate surroundings (even in the presence of an attacker that is able to inject code at run-time). Beside using a look ahead limit, the analysis would also terminate when for example detecting a store instruction that writes memory close to the current program counter or when the program-counter is loaded with a register value to ensure that the analysis stays local and no code is injected in-between.
As a first step, the proposed technique could be used to detect regular control flows (e. g. resulting from an if-then-else construct) and then mark all dependent assignments (i. e. inside the `if` and `else` branch) to be confidential as well. This is inherently difficult, however, as side-channels such as interrupt routines can throw off this estimation.

Chapter 5
Conclusion

Complexity of digital systems, combined with an ubiquitous number of devices, impacts security and safety of human lives on an ever-increasing level. While the use of embedded systems and ICs in general improved the way of living considerably, actual [25] and fatality-causing [26] incidences have shown the necessity of improving the quality of these systems. Especially, the need emerged to not only rely on the "quality" of care given by individual developers or designers, but to actually improve the design processes to a level of trust that meets modern standards.
In this thesis, an efficient and high-quality design process for complex systems leveraging architecture-level VPs has been presented. It features several novel approaches for a fast and thorough design space exploration by simulating and visualizing on- and off-chip devices for a reliable system specification reducing the possibility of late, and thus costly, design-reiterations. It further enhances existing verification techniques like DIFT and symbolic execution on continuous levels along the system design process, from the earliest stages on. This reduces the need to rely on single individuals to ace the development, but instead increases the reliability and repeatability of complex systems.
The different contributions can be summarized into two main fields: **Modeling** to expand the possibilities of building new digital computer systems, and **Verification** to expand the trust and to improve the correctness of these systems. The following individual contributions to the overall process, described in this thesis, are:

Modeling The first contribution speeds up the system design with an open source RISC-V VP implemented in SystemC TLM. It is able to run multiple operating systems, including embedded systems like Zephyr and common desktop-grade systems like Linux. It offers not only the core with the RISC-V instruction set extensions `I`, `M`, `A`, `F` and `D` in both 32 and 64 bit with *Sv32*, *Sv39* and *Sv48*

virtual memory translation systems, but also contains a rich set of peripherals to run whole system simulations. This set includes RISC-V compliant PLIC and CLINT implementations, as well as ethernet- filesystem- and graphics devices, and debugging capabilities with the GDB suite. Together with a comparatively outstanding execution speed compared to other architecture level VPs, it allows a fast and accurate design space exploration and performance evaluation. It is also used as the basis for a huge number of consecutive publications, including the ones of this thesis.
The second contribution further improves the system design process in the form of a novel, virtual Environment Model GUI which enables emulating off-chip devices such as sensors, displays and actuators in graphical representation with mouse and keyboard interaction input. It features a rich set of devices that can both be modeled in C++ (for execution speed) and the scripting language Lua (for flexibility and fast bring-up). It has been proven successful in several experiments for designing PCBs and educational uses. Combined with a further extended RISC-V VP to incorporate the protocol by a newly created GPIO-interface, complete systems can be simulated, analyzed and debugged; including the Sifive HiFive1 embedded development board with an OLED shield and buttons.
The third contribution in this area improves the HW/SW co-design introspection by allowing a unique view on hardware states of a virtual prototype. RISCVIEW offers an easy-to-use visualization of SystemC peripherals with a minimal impact on the existing code-base through leveraging the *Model-View* principle. Its HW/SW co-debugging system is applicable from the earliest stages of the design process and provides a live view on the internals, intended to be used alongside existing SW debugging tools like GDB, which was shown in a case-study by finding an intricate race-condition that could not be found by traditional debugging techniques.
Finally, the fourth contribution closes the TLM/RTL gap by implementing a HWITL system that is focused on combining transaction- and register transfer layer models. The proposed tool *VPITL* leverages the existing RISC-V VP infrastructure by translating memory-mapped I/O accesses to synthesized RTL peripheral implementations to an FPGA transparently, virtually placing the RTL peripherals into the VP. The approach enables RTL designers to focus development on their USP with a minimal design evaluation cost, as the minimally required set of RISC-V specific HW does not have to be brought up in HDL first. Additionally, it offers cross level testing possibilities of RTL models against their SystemC counterparts, and *mixed reality* system evaluation approaches; thus also improving the integration test phases of the system design process.

Verification The first contribution in this field adds a unique verification approach for early HW TLM models through a framework that enables the symbolic execution of previously too complex SystemC TLM models. This verification approach leverages a different scheduling scheme to the SystemC's user-space scheduling, which is incompatible with modern C++ symbolic execution tools like KLEE. Instead, it offers a conversion script to generate intermediate models that instead rely on return-based function scheduling. This scheduling scheme is implemented in an alternative SystemC kernel, called *Peripheral Kernel*, *PK*. The proposed *PK* has proven to be very effective in the evaluation by enabling real-world TLM models to be thoroughly verified with common symbolic execution engines because of its sleek structure and optimized architecture. In the experimental application of the approach, several known and previously unknown faults could be located in the RISC-V VP's implementation of the FE310 PLIC and other peripherals.
The second contribution extends the previous contribution to the RTL, enabling early cross-level verification of RTL/TLM peripheral models, and directly testbench driven verification on the RTL. It extends the *PK* to incorporate a more detailed simulation of signals, slots, and other SystemC RTL principals, while still using the return-based function scheduling approach. Preliminary experiments with a HDL PLIC, written in SpinalHDL and converted to SystemC RTL using Verilator, use a TLM translation bus to interface with the *SimpleBus* interface of the SoC MicroRV32.
The third and final contribution in the verification field enables a DIFT-based security policy evaluation in the early system design stages, allowing for a security-focused DSE with VPs. The framework was applied to the RISC-V VP with a minimal impact on the existing code-base, and is intended to be used continuously from the earliest phases in the design process, starting in the specification phase. It supports arbitrary security lattice concepts, including the simpler integrity- and security models, throughout the complete SoC, including DMA and UART peripherals in different clearance and source level classes. It was used successfully to analyze different SW images *while running on the HW* based on the conformity to example security lattices, as well as to evaluate security concepts and their implications and restrictions they impose on the system.
All of the mentioned approaches have been implemented, evaluated, thoroughly discussed and made publicly available (excluding RISCVIEW due to licensing reasons). In conclusion, these contributions have significantly improved the complex system design process by offering easy, early and continuous verification approaches, as well as speeding up the early system design phases with several modeling techniques.

Outlook While the proposed approaches have already shown a multitude of benefits along the system design process, they can also serve as the foundation for new investigations and extensions besides the discussed fields of future studies in the individual sections. In particular, there are overlapping and process-wide ideas that can be worth investigating; and the following directions seem very promising to further improve the complex system design process in different stages:

1) Investigate further techniques on verifying SW on VPs, especially based on constrained random / concolic testing techniques in a combination of the RISC-V VP and [32, 33, 199]. Combining this with the proposed DIFT framework could enable a thorough verification of the software regarding its conformity to security policies, including the hardening against security flaws. Especially if a complete path exploration can be achieved, the same system could be used to both explore and evaluate security policies in the earliest stages, as well as function as a driver for SW verification in the last acceptance test phase.

2) Use the existing techniques for DIFT to analyze different aspects of the HW/SW data flow. The collected data on variables of interest could be used for post-mortem analysis tools to track their paths through all relevant parts of the HW/SW system, enabling a unique look at aspects on which the data depends on. This also can be used to boost automatic test-case generation tools such as fuzzers with an in-depth data-flow centric coverage metric, including, but not limited to, security policies.

3) For further improvement of the initial design space exploration phase, it might be worth investigating on how to automatically generate both TLM-based HW memory maps and their interfacing SW drivers based on the documentation or specification. This tool could start from just the named register locations up to an initial state-machine in the HW peripheral that can be interfaced by the stub SW driver. This approach can speed up the bring-up time even further besides using the proposed RISC-V VP.

4) While maintaining a specialized, but compatible SystemC *PK* has proven a successful approach in verifying TLM/RTL SystemC peripherals using symbolic execution, it may be worth investigating to get the SystemC kernel out of the symbolic execution interpreter altogether. With tools like Angr [200], it may be possible to leverage *selective symbolic execution* to focus only on the functional part of a SystemC run-time system. This might speed up the execution process or enable the verification of the full-system interaction, but with the remaining challenge of proving that the approach is still functionally complete.

5) Another modeling speedup could be achieved by adding scriptable mock-up peripherals in the RISC-V VP. These could be interpreted by a Python or Lua (as in the Environment Model GUI) interpreter and drive an even faster design space exploration phase and further expand the RISC-V IP ecosystem.

6) Further adding to the IP ecosystem, it might be worth investigating the use of machine learning techniques to deduce a peripheral's behavior model based on SW accesses. Combining this with proposed symbolic execution methods as in Section 4.1, program path exploration based on symbolic peripheral registers in memory-mapped I/O might enable effective reverse-engineering of underlying hardware models based on given SW executable binaries.

Acronyms

ACM Access Control Model. 144, 147

ADC Analog-to-Digital Converter. 16, 60

AES Advanced Encryption Standard. 118, 162

API Application Programming Interface. 84, 88

ASIC Application-specific Integrated Circuit. 4, 23

AT Approximately Timed. 19, 20

BRAM Block RAM. 107, 112, 113, 182

CAN Controller Area Network. 58, 162, 163

CGF Coverage Guided Fuzzing. 50, 51, 178

CI/CD Continuous Integration / Continuous Deployment. 80, 121

CISC Complex Instruction Set Computer. 5

CLINT Core-Local Interruptor. 24, 28, 31, 32, 48, 51, 56, 75, 78, 110, 114, 168

CLV Cross-Level Verification. 7

CPS Cyber-Physical Systems. 98

CPU Central Processing Unit. 1, 16, 30–32, 34, 35, 38–40, 45–47, 53, 59, 60, 63, 82, 83, 85, 97, 99, 144, 146, 148, 150, 152, 155–157, 160, 162, 163, 166, 180, 185

CS Chip Select. 17, 84, 93, 180

CSR Control and Status Register. 16, 21, 22, 30, 31, 35, 36, 38, 40, 41, 47

DAC Digital-to-Analog Converter. 16

DBT Dynamic Binary Translation. 53, 54, 56, 60, 166

DIFT Dynamic Information Flow Tracking. 11, 13, 117, 118, 144–146, 149, 150, 152, 154–156, 158, 162, 164–167, 169, 170, 183, 185

DMA Direct Memory Access. 16, 31, 42, 83, 99, 145, 146, 162, 169

DMI Direct Memory Interface. 39, 42

DRAM Dynamic Random Access Memory. 4

DSE Design Space Exploration. 25, 26, 169

DSP Digital Signal Processor. 15

DUT Device under Test. 134, 142, 181

DUV Device under Verification. 116, 123, 124, 126–128, 181

ECU Electronical Control Unit. 1, 162

EDA Electronic Design Automation. 1

ELF Executable and Linking Format. 32, 47, 48, 50, 51

ESL Electronic System Level. 23, 28, 56, 95, 96

FAT File Allocation Table Filesystem. 52

FIFO First-In-First-Out. 103, 104, 121

FPGA Field-Programmable Gate Array. 3, 4, 23–26, 83, 96–101, 103, 105–108, 110–114, 168, 180

GCC GNU Compiler Collection. 52

GCD Greatest Common Divisor. 106, 107, 111, 112, 114, 182

GCOV Gnu Coverage Tool. 30, 32, 38

GDB GNU Project debugger. 28, 30, 46–48, 79, 82, 83, 85, 94, 168

GPIO General Purpose Input/Output. 16, 17, 24, 48–50, 58, 60–64, 70, 71, 75–78, 86, 88, 91, 106, 108–110, 112, 114, 168, 179, 182, 184, 185

GPU Graphics Processing Unit. 1

GUI Graphical User Interface. 24, 25, 48, 50, 58, 59, 61–68, 74–82, 84, 86, 88, 168, 171, 179

HAL Hardware Abstraction Layer. 16, 82

HART Hardware Thread. 122, 128, 142, 180

HDL Hardware Description Language. 9, 23, 104, 138, 142, 143, 168, 169

HW hardware. 1–9, 11, 13, 15, 16, 21, 23, 25–28, 33, 34, 55, 56, 58, 70, 79, 81–83, 87, 88, 95–98, 105, 107, 108, 110–112, 114, 116, 119, 138, 144–146, 148, 149, 152, 155, 156, 158, 165, 168–170, 178, 180, 184, 185

HWITL Hardware-in-the-Loop. 9, 13, 25, 26, 95–99, 107, 113, 114, 168

I²C Inter-Integrated Circuit. 17, 88, 113

IC Integrated Circuit. 1, 36, 81, 167

IDE Integrated Development Environment. 47, 48, 83

IFP Information Flow Policy. 144, 146–150, 158, 162, 163, 181

IoT Internet of Things. 1, 6, 7, 23, 27, 57

IP Interlectual Property. 5, 7, 25, 56, 60, 82–85, 96, 97, 99, 105, 137, 138, 171

IR Immediate Representation. 124, 127, 140

ISA Instruction Set Architecture. 5, 14, 16, 21–23, 27, 29, 40, 45, 48, 50, 51, 55–58, 142, 148, 182

ISP In-System Programmer. 17

ISS Instruction Set Simulator. 5, 23, 24, 27, 95

JIT Just-in-Time Compilation. 53, 56, 166

LC Logic Cell. 107, 112, 113, 182

LED Light Emitting Diode. 17, 18, 25, 48, 58, 62, 64, 65, 71, 72, 74, 75, 79, 105, 106, 109, 112, 113, 176, 179, 184

LoC Lines of Code. 52, 164

LT Loosely Timed. 19, 20

LUB Least Upper Bound. 148, 150

MIPS Microprocessor without Interlocked Pipelined Stages. 5

MIPS Million Instructions Per Second. 52–55, 77, 78, 165, 182

MISO Master-In-Slave-Out. 17, 76, 84

MMU Memory Management Unit. 56

MOSFET Metal-Oxide Semiconductor Field-Effect Transistor. 17

MOSI Master-Out-Slave-In. 17, 84

MSB Most Significant Bit. 109

MVP Minimum Viable Product. 3, 114

OLED Organic Light Emitting Diode. 25, 59, 65, 69–71, 75–78, 82, 88, 90, 94, 168, 179, 184

OS Operating System. 3, 16, 19, 38, 51, 54, 56

PCB Printed Circuit Board. 18, 58, 65, 70, 76, 88, 91, 105, 168, 179, 184

PIN Personal Identification Number. 162, 163

PK Peripheral Kernel. 117, 120, 123–128, 137–143, 169, 170, 181

PLIC Platform Level Interrupt Controller. 24, 28, 32, 35, 48, 51, 56, 60, 63, 103, 114, 120, 122, 123, 125, 126, 128–130, 133–135, 137, 142, 168, 169, 182, 183, 185

PNR place & route. 107, 112

POR Partial Order Reduction. 120, 121

PWM Pulse Width Modulation. 17, 18, 60, 64

RAM Random Access Memory. 4, 77, 162, 174

RGB Red Green Blue. 65, 74, 75, 179

RISC Reduced Instruction Set Computer. 5, 16, 21

RSP Remote Serial Protocol. 47

RTL Register Transfer Layer. iii, 2, 9, 13, 19, 20, 23–28, 52, 78, 84, 95–97, 99, 104, 107, 111, 113, 114, 117, 120, 138–140, 142, 143, 146, 168–170, 180–182

RX receive 'X'. 93, 180

SDV Software Driven Verification. 7

SMT Satisfiability Modulo Theories. 129, 130

SoC System-on-Chip. 1, 4, 5, 11, 13, 14, 21, 25, 26, 48, 57, 58, 60, 70, 82, 83, 95, 96, 98, 99, 104, 107, 114, 120, 128, 137, 145, 146, 149, 158, 169, 180

SPI Serial Peripheral Interface. 17, 24, 58, 60, 62–66, 68, 70, 72, 74–78, 83, 84, 88, 91–94, 110, 180, 182, 184

SW software. 1–5, 7–9, 11, 13, 15, 16, 21, 23–25, 27, 28, 30–35, 40, 41, 46–48, 50, 55, 56, 58, 80–83, 91, 95–98, 105, 107, 110–112, 116, 117, 138, 144–146, 148, 149, 155, 156, 158–160, 163, 165, 166, 168–171, 178, 182, 184, 185

TCP Transmission Control Protocol. 24, 47–49, 58, 62–64, 80, 81, 84, 86, 178

TIC Translator Interface Controller. 103, 104

TLM Transaction Level Modeling. 2, 9, 11, 13, 14, 19, 24–31, 33, 40, 42, 45–47, 53, 55, 56, 58, 60, 77, 81, 95–97, 99, 101, 102, 114, 117, 119–121, 123–125, 128–130, 133, 135, 137–139, 142, 143, 145, 152, 155, 158, 159, 162, 167–170, 180, 181, 185

TX transmit 'X'. 92, 93, 158, 180

UART Universal Asynchronous Receiver / Transmitter. 16, 17, 32, 38, 49, 58, 60, 83, 88, 92, 99, 103, 104, 109, 110, 112–115, 150, 158, 163, 165, 169, 180

UDP User Datagram Protocol. 52

USP Unique Selling-Point. 25, 96, 97, 105, 114, 168

VP Virtual Prototype. iii, 1, 2, 4, 5, 7–11, 13, 14, 19, 24–30, 32–36, 38, 45–51, 55–63, 70, 71, 74, 75, 77–85, 88, 90, 94–97, 99, 100, 103, 106–108, 110, 113, 114, 116–119, 121, 138, 144–146, 149, 152, 155, 157–159, 161, 162, 165–170, 178, 179, 184–186

VPIL Virtual Peripheral in-the-Loop. 25, 95, 97, 99, 100, 106, 108, 111, 114, 180

List of Figures

List of Tables

Listings

Bibliography

[1] B. Menhorn and F. Slomka, "Confirming the design gap," in *Advances in Computational Science, Engineering and Information Technology*, D. Nagamalai, A. Kumar, and A. Annamalai, Eds., Heidelberg: Springer International Publishing, 2013, pp. 281–292, ISBN: 978-3-319-00951-3.

[2] Semiconductor Industry Association. "2008 international technology roadmap for semiconductors (ITRS)." (2008), [Online]. Available: https://cseweb.ucsd.edu/classes/wi09/cse242a/itrs/ORTC.pdf (visited on 2022-11).

[3] J. Henkel, "Embedded computing - closing the SoC design gap," *Computer*, vol. 36, no. 9, pp. 119–121, 2003-09. DOI: 10.1109/mc.2003.1231200.

[4] *IEEE standard for standard systemc language reference manual*, 2012, pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[5] *Osci tlm-2.0 language reference manual*, OSCI, 2009.

[6] E. Sotiriou-Xanthopoulos, S. Xydis, K. Siozios, G. Economakos, and D. Soudris, "Rapid prototyping and design space exploration methodologies for many-accelerator systems," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–2. DOI: 10.1109/FPL.2015.7293990.

[7] D. Große and R. Drechsler, *System-Level Verification*. Springer Netherlands, 2010, ISBN: 978-90-481-3631-5. DOI: 10.1007/978-90-481-3631-5_3. [Online]. Available: https://doi.org/10.1007/978-90-481-3631-5_3.

[8] R. Leupers *et al.*, "Virtual platforms: Breaking new grounds," in *2012 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2012, pp. 685–690. DOI: 10.1109/DATE.2012.6176558.

[9] O. Hagenbruch, *Taschenbuch Mikroprozessortechnik*. Hanser Verlag, 2004, ISBN: 978-3446220720.

[10] F. Vahid, "What is hardware/software partitioning?" *SIGDA Newsl.*, vol. 39, no. 6, p. 1, 2009, ISSN: 0163-5743. DOI: 10.1145/1862900.1862901. [Online]. Available: https://doi.org/10.1145/1862900.1862901.

[11] Mirabilis Design Inc. "Hardware-software partitioning in system-on-chip (soc)." (2022), [Online]. Available: https://www.mirabilisdesign.com/hardware-software-partitioning-in-system-on-chip-soc/ (visited on 2022-11).

[12] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 1–12. DOI: 10.1109/HPCA.2013.6522302.

[13] S. B. Furber, *ARM system-on-chip architecture*. Pearson Education, 2000, ISBN: 978-0201675191.

[14] U. Degenbaev, *Formal specification of the x86 instruction set architecture, Formelle spezifizierung von dem x86-befehlssatz*, 2012. DOI: http://dx.doi.org/10.22028/D291-26338.

[15] C. Domas, "Breaking the x86 isa," *Black Hat*, 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-ISA.pdf.

[16] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*, RISC-V Foundation, December 2019.

[17] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, RISC-V Foundation, December 2019.

[18] "RISC-V calling convention." (2018), [Online]. Available: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf (visited on 2018-05).

[19] "Alibaba cloud unveils chip development platform to support developers with risc-v based high-performance socs." (2022), [Online]. Available: https://www.alibabacloud.com/de/press-room/alibaba-unveils-risc-v-chip-development-platform (visited on 2022-08).

[20] "Die risc-v-basierten ssd-controller kommen." (2020), [Online]. Available: https://www.golem.de/news/seagate-und-western-digital-die-risc-v-basierten-ssd-controller-kommen-2012-152690.html (visited on 2022-08).

[21] R. Leupers *et al.*, "Virtual platforms: Breaking new grounds," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 685–690. DOI: 10.1109/DATE.2012.6176558.

[22] L. Bossuet and G. Gogniat, "Chapter 5: Hardware security in embedded systems," *Communicating Embedded Systems*, 2010-01. DOI: 10.1002/9781118557624.ch5.

[23] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015, pp. 145–152. DOI: 10.1109/PST.2015.7232966.

[24] HiPEAC. "New startup machineware enables ultra-fast risc-v simulation." (2022), [Online]. Available: https://www.hipeac.net/news/6996/new-startup-machineware-enables-ultra-fast-risc-v-simulation/ (visited on 2022-11).

[25] Jo Vanwell. "10 iot security incidents that make you feel less secure." (2021), [Online]. Available: https://conosco.com/industry-insights/blog/iot-security-breaches-4-real-world-examples (visited on 2022-11).

[26] CisoMag Authors. "10 iot security incidents that make you feel less secure." (2020), [Online]. Available: https://embeddedartistry.com/fieldatlas/historical-software-accidents-and-errors/ (visited on 2022-11).

[27] Semiconductor Industry Association. "2015 international technology roadmap for semiconductors (ITRS)." (2015), [Online]. Available: https://www.semiconductors.org/wp-content/uploads/2018/06/0_2015-ITRS-2.0-Executive-Report-1.pdf (visited on 2022-11).

[28] Y. Xu *et al.*, "Towards developing high performance risc-v processors using agile methodology," *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1178–1199, 2022.

[29] S. D. Anthony. "3 ways to fail cheap." (2009), [Online]. Available: https://hbr.org/2009/03/why-focusing-on-innovation-suc (visited on 2022-11).

[30] V. Herdt and R. Drechsler, "Advanced virtual prototyping for cyber-physical systems using risc-v: Implementation, verification and challenges," *Science China Information Sciences*, vol. 65, 2022-01. DOI: 10.1007/s11432-020-3308-4.

[31] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Efficient cross-level processor verification using coverage-guided fuzzing," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, ser. GLSVLSI '22, Irvine, CA, USA: Association for Computing Machinery, 2022, pp. 97–103, ISBN: 9781450393225. DOI: 10.1145/3526241.3530340. [Online]. Available: https://doi.org/10.1145/3526241.3530340.

[32] S. Tempel, V. Herdt, and R. Drechsler, "An effective methodology for integrating concolic testing with systemc-based virtual prototypes," in *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 218–221. DOI: 10.23919/DATE51398.2021.9474149.

[33] S. Tempel, V. Herdt, and R. Drechsler, "SISL: Concolic testing of structured binary input formats via partial specification," in *Automated Technology for Verification and Analysis*, A. Bouajjani, L. Holík, and Z. Wu, Eds., Cham: Springer International Publishing, 2022, pp. 77–82, ISBN: 978-3-031-19992-9.

[34] J. Zielasko, S. Tempel, V. Herdt, and R. Drechsler, "3D visualization of symbolic execution traces," in *Forum on Specification and Design Languages*, 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925664.

[35] V. Herdt, D. Große, S. Tempel, and R. Drechsler, "Adaptive simulation with virtual prototypes in an open-source risc-v evaluation platform," *Journal of Systems Architecture*, vol. 116, p. 102 135, 2021, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102135. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762121001016.

[36] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20, Virtual Event, China: Association for Computing Machinery, 2020, pp. 101–106, ISBN: 9781450379441. DOI: 10.1145/3386263.3406899. [Online]. Available: https://doi.org/10.1145/3386263.3406899.

[37] N. Bruns, V. Herdt, and R. Drechsler, "Unified HW/SW coverage: A novel metric to boost coverage-guided fuzzing for virtual prototype based HW/SW co-verification," in *Forum on Specification and Design Languages, FDL 2022, Linz, Austria, September 14-16, 2022*, IEEE, 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925661. [Online]. Available: https://doi.org/10.1109/FDL56239.2022.9925661.

[38] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101 756, 2020, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101756.

[39] P. Pieper, V. Herdt, and R. Drechsler, "Advanced environment modeling and interaction in an open source RISC-V virtual prototype," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, ser. GLSVLSI '22, Irvine, CA, USA: Association for Computing Machinery, 2022, pp. 193–197, ISBN: 9781450393225. DOI: 10.1145/3526241.3530374.

[40] P. Pieper, V. Herdt, and R. Drechsler, "Advanced embedded system modeling and simulation in an open source RISC-V virtual prototype," *Journal of Low Power Electronics and Applications*, vol. 12, no. 4, 2022, ISSN: 2079-9268. DOI: 10.3390/jlpea12040052. [Online]. Available: https://www.mdpi.com/2079-9268/12/4/52.

[41] F. Böseler, J. Walter, and B. R. Perjikolaei, "A comparison of Virtual Platform Simulation Solutions for timing prediction of small RISC-V based SoCs," in *Forum on Specification and Design Languages*, 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925667.

[42] M. Koenig and R. Rasch, "Digital teaching an embedded systems course by using simulators," in *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, 2021, pp. 1–7. DOI: 10.1109/WCAE53984.2021.9707146.

[43] L. Christensen. "Chip Industry's Technical Paper Roundup: Oct 18." (2022), [Online]. Available: https://semiengineering.com/semiconductor-industrys-technical-paper-roundup-oct-18 (visited on 2022-12-20).

[44] P. Pieper, R. Wimmer, G. Angst, and R. Drechsler, "Minimally invasive HW/SW co-debug live visualization on architecture level," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 321–326, ISBN: 9781450383936. DOI: 10.1145/3453688.3461524.

[45] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218494.

[46] P. Pieper, V. Herdt, and R. Drechsler, "Verifying SystemC TLM peripherals using modern C++ symbolic execution tools," in *2022 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1–6. DOI: 10.1145/3489517.3530604.

[47] P. Pieper. "Symbolic SystemC kernel framework." (2022), [Online]. Available: https://github.com/agra-uni-bremen/symsysc (visited on 2022-12-20).

[48] P. Pieper. "Virtual breadboard GUI." (2022), [Online]. Available: https://github.com/agra-uni-bremen/virtual-breadboard (visited on 2022-12-20).

[49] P. Pieper, V. Herdt, S. Tempel, K. A. Rudkowski, S. Ahmadi-Pour, and N. Bruns. "RISC-V virtual prototype." (2021), [Online]. Available: https://github.com/agra-uni-bremen/riscv-vp (visited on 2022-12-20).

[50] P. Pieper. "Dynamic information flow analysis with the RISC-V VP." (2022), [Online]. Available: https://github.com/agra-uni-bremen/riscv-dfa (visited on 2022-12-20).

[51] P. Pieper. "Virtual peripheral in-the-loop: Protocol." (2023), [Online]. Available: https://github.com/agra-uni-bremen/virtual-bus (visited on 2023-03-24).

[52] P. Pieper, S. Ahmadi-Pour, and R. Drechsler, "Virtual-peripheral-in-the-loop: A hardware-in-the-loop strategy to bridge the VP/RTL design-gap," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '23, Hamburg, Germany: Association for Computing Machinery, 2023. DOI: 10.1145/-pending-.

[53] M. Barr. "Embedded systems glossary." (), [Online]. Available: https://barrgroup.com/embedded-systems/glossary (visited on 2022-12-20).

[54] M. Jimnez, R. Palomera, and I. Couvertier, *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. Springer Publishing Company, Incorporated, 2017, ISBN: 1493944282.

[55] F. Kesel, *Modellierung von digitalen Systemen mit SystemC, Von der RTL- zur Transaction-Level-Modellierung*. München: Oldenbourg Wissenschaftsverlag, 2012, ISBN: 9783486718959. DOI: doi:10.1524/9783486718959.

[56] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, 2014.

[57] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[58] S. Swan, "SystemC transaction level models and RTL verification," in *43rd ACM/IEEE Design Automation Conference*, 2006, pp. 90–92. DOI: 10.1145/1146909.1146937.

[59] M. Goli and R. Drechsler, "Scalable simulation-based verification of SystemC-based virtual prototypes," in *Euromicro Conf. on Digital System Design (DSD)*, IEEE, 2019, pp. 522–529. DOI: 10.1109/DSD.2019.00081.

[60] "Accellera SystemC distributions." (2018), [Online]. Available: https://www.accellera.org/downloads/standards/systemc (visited on 2023-04).

[61] L. Steiner, M. Jung, F. S. Prado, K. Bykov, and N. Wehn, "DRAMSys4.0: A fast and cycle-accurate SystemC/TLM-Based DRAM simulator," Springer, 2020, pp. 110–126.

[62] Y. Liu, K. Ye, and C.-Z. Xu, "Performance evaluation of various risc processor systems: A case study on arm, mips and risc-v," in *Cloud Computing – CLOUD 2021*, K. Ye and L.-J. Zhang, Eds., Springer International Publishing, 2022, pp. 61–74, ISBN: 978-3-030-96326-2.

[63] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, "A reconfigurable convolutional neural network-accelerated coprocessor based on risc-v instruction set," *Electronics*, vol. 9, no. 6, 2020, ISSN: 2079-9292. DOI: 10.3390/

electronics9061005. [Online]. Available: https://www.mdpi.com/2079-9292/9/6/1005.

[64] "RISCV-QEMU." (2018), [Online]. Available: https://github.com/riscv/riscv-qemu (visited on 2022-04).

[65] "Spike." (2018), [Online]. Available: https://github.com/riscv/riscv-isa-sim (visited on 2022-04).

[66] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[67] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[68] "FreeRTOS." (2022), [Online]. Available: https://www.freertos.org/ (visited on 2022-04).

[69] "Zephyr Project." (2022), [Online]. Available: https://www.zephyrproject.org/ (visited on 2022-04).

[70] "RIOT OS." (2022), [Online]. Available: https://www.riot-os.org/ (visited on 2022-04).

[71] Concept Engineering GmbH, *Nlview 7.3.11*, https://www.concept.de, 2021-04.

[72] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *Forum on Specification and Design Languages*, 2018, pp. 5–16.

[73] R. Leupers *et al.*, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.

[74] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[75] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich, "ESL power and performance estimation for heterogeneous mpsocs using SystemC," in *FDL*, 2011, pp. 1–8.

[76] K. Grüttner *et al.*, "CONTREX: Design of embedded mixed-criticality CONTRol systems under consideration of extra-functional properties," *Microprocessors and Microsystems*, vol. 51, pp. 39 –55, 2017.

[77] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schürmans, "Black box ESL power estimation for loosely-timed TLM models," in *SAMOS*, 2016, pp. 366–371. DOI: 10.1109/SAMOS.2016.7818374.

[78] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation - a case study," in *FDL*, 2016, pp. 1–8.

[79] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[80] "RV8." (2018), [Online]. Available: https://rv8.io (visited on 2022-04).

[81] "DBT-RISE." (2021), [Online]. Available: https://github.com/Minres/DBT-RISE-Core (visited on 2022-04).

[82] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011-08, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718.

[83] "Renode." (2022), [Online]. Available: https://renode.io/ (visited on 2022-04).

[84] "Forvis: A formal RISC-V ISA specification." (2020), [Online]. Available: https://github.com/rsnikhil/RISCV-ISA-Spec (visited on 2022-04).

[85] "GRIFT - galois RISC-V ISA formal tools." (2020), [Online]. Available: https://github.com/GaloisInc/grift (visited on 2022-04).

[86] "Riscv sail model." (2020), [Online]. Available: https://github.com/rems-project/sail-riscv (visited on 2022-04).

[87] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7.

[88] "GCC, the GNU compiler collection." (1987), [Online]. Available: https://gcc.gnu.org/ (visited on 2023-04).

[89] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research," *Journal of Systems Architecture*, vol. 133, p. 102 757, 2022, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102757. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122002429.

[90] S. Tempel, V. Herdt, and R. Drechsler, "Symex-vp: An open source virtual prototype for os-agnostic concolic testing of iot firmware," *Journal of Systems Architecture*, vol. 126, p. 102 456, 2022, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102456. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000480.

[91] "HiFive1." (2022), [Online]. Available: https://www.sifive.com/boards/hifive1 (visited on 2022-04).

[92] "SiFive FE310-G000 manual." (2020), [Online]. Available: https://sifive.cdn.prismic.io/sifive%2F500a69f8-af3a-4fd9-927f-10ca77077532_fe310-g000.pdf (visited on 2020-09-17).

[93] "Sifive fe310-g002 datasheet v1p2." (), [Online]. Available: https://starfivetech.com/uploads/fe310-g002-datasheet-v1p2.pdf.

[94] "RISC-V ISA tests." (2020), [Online]. Available: https://github.com/riscv/riscv-tests (visited on 2022-04).

[95] "RISC-V torture test generator." (2022), [Online]. Available: https://github.com/ucb-bar/riscv-torture (visited on 2022-04).

[96] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *Design, Automation and Test in Europe*, 2019.

[97] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, 2018, ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2846638.

[98] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.

[99] M. Y. Vardi, "Formal techniques for SystemC verification," in *DAC*, 2007, pp. 188–192.

[100] M. F. Oliveira *et al.*, "The system verification methodology for advanced tlm verification," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '12, Tampere, Finland: Association for Computing Machinery, 2012, pp. 313–322, ISBN: 9781450314268. DOI: 10.1145/2380445.2380497. [Online]. Available: https://doi.org/10.1145/2380445.2380497.

[101] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[102] X. Guo and R. D. Mullins, "Accelerate cycle-level full-system simulation of multi-core RISC-V systems with binary translation," *CoRR*, vol. abs/2005.11357, 2020. [Online]. Available: https://arxiv.org/abs/2005.11357.

[103] X. Guo and R. D. Mullins, "Fast tlb simulation for risc-v systems," *ArXiv*, vol. abs/1905.06825, 2019.

[104] "GEM5." (2021), [Online]. Available: https://gem5.googlesource.com/public/gem5 (visited on 2022-04).

[105] "ETISS (extendable translating instruction set simulator)." (2022), [Online]. Available: https://github.com/tum-ei-eda/etiss (visited on 2022-04).

[106] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *2017 International Symposium on Rapid System Prototyping (RSP)*, 2017, pp. 79–84.

[107] "RISC-V-TLM." (2022), [Online]. Available: https://github.com/mariusmm/RISC-V-TLM (visited on 2022-04).

[108] "HIFIVE1-VP." (2022), [Online]. Available: https://git.minres.com/VP/HIFIVE1-VP (visited on 2022-04).

[109] "Synopsis virtualizer." (2021), [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html (visited on 2022-04).

[110] "SimAvr RISC-V isa simulator." (2020), [Online]. Available: https://github.com/buserror/simavr (visited on 2022-04).

[111] "PICsimLab - programmable ic simulator laboratory." (2020), [Online]. Available: https://github.com/lcgamboa/picsimlab (visited on 2022-04).

[112] "Data-sheet of the SH1106 OLED display driver." (2019), [Online]. Available: https://www.velleman.eu/downloads/29/infosheets/sh1106_datasheet.pdf (visited on 2023-04).

[113] M. Holzer, B. Knerr, P. Belanović, M. Rupp, and G. Sauzon, "Faster complex SoC design by virtual prototyping," in *Int'l Conf. on Cybernetics and Information Technologies, Systems and Applications (CITSA)*, 2004, pp. 305–309.

[114] R. M. Stallman, R. Pesch, S. Shebs, *et al.*, *Debugging with GDB: The GNU Source-Level Debugger*, 10th. GNU, 2020. [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb.pdf.

[115] R. Willenberg and P. Chow, "Simulation-based HW/SW co-debugging for field-programmable systems-on-chip," in *Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, IEEE, 2013, pp. 1–8.

[116] K. Lee, A. Su, Long-Feng Chen, Jia-Wei Jhou, J. Kuo, and M. Liu, "A software/hardware co-debug platform for multi-core systems," in *IEEE Int'l Conf. on ASIC*, 2011, pp. 259–262. DOI: 10.1109/ASICON.2011.6157171.

[117] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An integrated SystemC debugging environment," in *Embedded Systems Specification and Design Lan-*

guages, ser. Lecture Notes in Electrical Engineering, vol. 10, Springer, 2008, pp. 59–71.

[118] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs," in *FDL*, ECSI, 2003, pp. 646–658.

[119] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017. DOI: 10.1109/mdat.2017.2735383.

[120] A. Adamov, K. Mostovaya, I. Syzonenko, and A. Melnik, "Electronic system level models for functional verification of system-on-chip," in *2007 9th International Conference - The Experience of Designing and Applications of CAD Systems in Microelectronics*, IEEE, 2007. DOI: 10.1109/cadsm.2007.4297576.

[121] S. Rigo, R. Azevedo, and L. Santos, Eds., *Electronic System Level Design*. Springer Netherlands, 2011. DOI: 10.1007/978-1-4020-9940-3.

[122] S. Rigo, B. Albertini, and R. Azevedo, "Transaction level modeling," in *Electronic System Level Design*, Springer Netherlands, 2011, pp. 25–36. DOI: 10.1007/978-1-4020-9940-3_3.

[123] L. Santos, S. Rigo, R. Azevedo, and G. Araujo, "Electronic system level design," in *Electronic System Level Design*, Springer Netherlands, 2011, pp. 3–10. DOI: 10.1007/978-1-4020-9940-3_1.

[124] ISTQB, *Istqb glossary*, https://glossary.istqb.org/en_US/term/hardware-in-the-loop-2.

[125] F. Mihalič, M. Truntič, and A. Hren, "Hardware-in-the-loop simulations: A historical overview of engineering challenges," *Electronics*, vol. 11, no. 15, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11152462. [Online]. Available: https://www.mdpi.com/2079-9292/11/15/2462.

[126] P. Nissimagoudar, V. Mane, G. H M, and N. C. Iyer, "Hardware-in-the-loop (hil) simulation technique for an automotive electronics course," *Procedia Computer Science*, vol. 172, pp. 1047–1052, 2020, 9th World Engineering Education Forum (WEEF 2019) Proceedings : Disruptive Engineering Education for Sustainable Development, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.05.153. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920314836.

[127] C. Köhler, *Enhancing Embedded Systems Simulation*. Vieweg+Teubner, 2011. DOI: 10.1007/978-3-8348-9916-3.

[128] A. Wu, J.-F. Mao, and X. Zhang, "An adrc-based hardware-in-the-loop system for maximum power point tracking of a wind power generation system," *IEEE Access*, vol. 8, pp. 226 119–226 130, 2020. DOI: 10.1109/ACCESS.2020.3045015.

[129] Z. Jiang, R. Leonard, R. Dougal, H. Figueroa, and A. Monti, "Processor-in-the-loop simulation, real-time hardware-in-the-loop testing, and hardware validation of a digitally-controlled, fuel-cell powered battery-charging station," in *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, IEEE. DOI: 10.1109/pesc.2004.1355471.

[130] N. Pătrăscoiu, A. M. Tomus, E. Angela, and S. Vali, "Creating hardware-in-the-loop system using virtual instrumentation," in *2011 12th International Carpathian Control Conference (ICCC)*, 2011, pp. 286–291. DOI: 10.1109/CarpathianCC.2011.5945865.

[131] V. Reyes, *Virtual hardware "in-the-loop": Earlier testing for automative applications*, https://www.synopsys.com/cgi-bin/proto/pdfdla/docsdl/virtual_hardware_wp.pdf, 2013.

[132] R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems," *Control Engineering Practice*, vol. 7, no. 5, pp. 643–653, 1999, ISSN: 0967-0661. DOI: https://doi.org/10.1016/S0967-0661(98)00205-6.

[133] H. Szolc and T. Kryjak, "Hardware-in-the-loop simulation of a UAV autonomous landing algorithm implemented in SoC FPGA," in *2022 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, IEEE, 2022. DOI: 10.23919/spa53010.2022.9927847.

[134] M. D. Signore, V. Krovi, and F. Mendel, "Virtual prototyping and hardware-in-the-loop testing for musculoskeletal system analysis," in *IEEE International Conference Mechatronics and Automation, 2005*, IEEE, 2005. DOI: 10.1109/icma.2005.1626579.

[135] J. Reitz, A. Gugenheimer, and J. Rosmann, "Virtual hardware in the loop: Hybrid simulation of dynamic systems with a virtualization platform," in *2020 Winter Simulation Conference (WSC)*, IEEE, 2020. DOI: 10.1109/wsc48552.2020.9383963.

[136] M. Lukasiewycz, S. Shreejith, and S. A. Fahmy, "System simulation and optimization using reconfigurable hardware," in *2014 International Symposium on Integrated Circuits (ISIC)*, IEEE, 2014. DOI: 10.1109/isicir.2014.7029545.

[137] D. Große, M. Groß, U. Kühne, and R. Drechsler, "Simulation-based equivalence checking between systemc models at different levels of abstraction," in *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, 2011, pp. 223–228.

[138] N. Bruns, D. Große, and R. Drechsler, "Early verification of isa extension specifications using deep reinforcement learning," in *30th ACM Great Lakes Symposium on VLSI (GLSVLSI). ACM Great Lakes Symposium on VLSI (GLSVLSI-2020), Beijing, China*, 2020.

[139] M. Goli, J. Stoppe, and R. Drechsler, "Automatic equivalence checking for systemc-tlm 2.0 models against their formal specifications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 630–633.

[140] M. Y. Vardi, "Formal techniques for SystemC verification," in *DAC*, 2007.

[141] J. Gladigau *et al.*, "Testfallgenerierung für SystemC-Designs mit abstrakten Modellbeschreibungen," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, (Berlin), 2009-03-02/2009-03-04, pp. 157–166.

[142] A. Habibi and S. Tahar, "Design and verification of systemc transaction-level models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 1, pp. 57–68, 2006. DOI: 10.1109/TVLSI.2005.863187.

[143] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on systemc designs," in *DAC Design Automation Conference 2012*, 2012, pp. 327–333.

[144] A. Fin, F. Fummi, and D. Signoretto, "The use of systemc for design verification and integration test of ip-cores," in *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No.01TH8558)*, 2001, pp. 76–80. DOI: 10.1109/ASIC.2001.954676.

[145] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California, 2008, pp. 209–224.

[146] D. Karlsson, P. Eles, and Z. Peng, "Formal verification of systemc designs using a petri-net based representation," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1, 2006, pp. 1–6. DOI: 10.1109/DATE.2006.244076.

[147] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Lussy: An open tool for the analysis of systems-on-a-chip at the transaction level," *ACSD*, vol. 10, no. 2-3, pp. 73–104, 2005.

[148] D. Karlsson, P. Eles, and Z. Peng, "Formal verification of systemc designs using a petri-net based representation," in *DATE*, 2006, pp. 1228–1233.

[149] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM semantics in promela and its possible applications," in *SPIN*, 2007, pp. 204–222.

[150] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *CODES+ISSS*, 2008, pp. 131–136.

[151] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *MEMOCODE*, 2005.

[152] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.

[153] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman, "A temporal language for SystemC," in *FMCAD*, 2008, pp. 1–9.

[154] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.

[155] C. Chou, C. Chu, and C. Huang, "Conquering the scheduling alternative explosion problem of SystemC symbolic simulation," in *ICCAD*, 2013.

[156] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 38, no. 7, pp. 1359–1372, 2019.

[157] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, 52:1–52:8.

[158] P. Herber, M. Pockrandt, and S. Glesner, "State – a SystemC to timed automata transformation engine," in *HPCC-CSS-ICESS*, 2015.

[159] M. Pockrandt, P. Herber, and S. Glesner, "Model checking a SystemC/TLM design of the AMBA AHB protocol," in *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, 2011, pp. 66–75.

[160] P. Herber, M. Pockrandt, and S. Glesner, "Transforming SystemC Transaction Level Models into UPPAAL timed automata," in *Ninth ACM/IEEE MEMPCODE 2011*, 2011, pp. 161–170.

[161] T. Liebrenz, V. Klös, and P. Herber, "Automatic analysis and abstraction for model checking HW/SW co-designs modeled in SystemC," *Ada Lett.*, vol. 36, no. 2, pp. 9–17, 2017-05.

[162] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models - a case study," in *2016 DATE*, 2016, pp. 1160–1163.

[163] B. Lin, Z. Yang, K. Cong, and F. Xie, "Generating high coverage tests for systemc designs using symbolic execution," in *2016 21st ASP-DAC*, 2016, pp. 166–171. DOI: 10.1109/ASPDAC.2016.7428006.

[164] P. Coussy, A. Takach, M. McNamara, and M. Meredith, "An introduction to the systemc synthesis subset standard," 2010-10, pp. 183–184. DOI: 10.1145/1878961.1878993.

[165] A. Chang *et al.* (2023), [Online]. Available: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc (visited on 2022-04).

[166] N. Bombieri, F. Fummi, and G. Pravadelli, "Rtl-tlm equivalence checking based on simulation," in *Proceedings of IEEE East-West Design & Test Symposium (EWDTS'08)*, 2008, pp. 214–217. DOI: 10.1109/EWDTS.2008.5580149.

[167] "Verilator compiler." (2004), [Online]. Available: https://www.veripool.org/verilator/ (visited on 2022-04).

[168] D. Currie, X. Feng, M. Fujita, A. Hu, M. Kwan, and S. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *International Journal of Parallel Programming*, vol. 34, pp. 61–91, 2006-03. DOI: 10.1007/s10766-005-0004-8.

[169] T. Li, J. Ye, and Q. Tan, "Towards functional verifying a family of systemc tlms," *Frontiers of Computer Science*, vol. 14, 2019-03. DOI: 10.1007/s11704-018-8254-y.

[170] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for systemc," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8. DOI: 10.1145/2966986.2967016.

[171] S. Ahmadi-Pour and V. Herdt. "Microrv32 - github." (2022), [Online]. Available: https://github.com/agra-uni-bremen/microrv32 (visited on 2023-03).

[172] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 85–96.

[173] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security - Tools for Analysis and Verification*, 2012, pp. 319–347.

[174] D. E. Robling Denning, *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982, ISBN: 0-201-10150-5.

[175] Automotive Working Group, *Automotive virtual prototyping platform (white paper)*, edacentrum, 2019.

[176] C. Song *et al.*, "Hdfi: Hardware-assisted data-flow isolation," in *Security and Privacy*, 2016.

[177] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for iot applications," in *2018 IEEE HPEC*, 2018-09.

[178] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *ISCA*, 2007, pp. 482–493.

[179] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *DSN*, 2009, pp. 105–114.

[180] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "Pagurus: Low-overhead dynamic information flow tracking on loosely coupled accelerators," *IEEE TCSDI*, 2018.

[181] J. Porquet and S. Sethumadhavan, "Whisk: An uncore architecture for dynamic information flow tracking in heterogeneous embedded socs," in *ISSS*, 2013.

[182] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni, "Tainthls: High-level synthesis for dynamic information flow tracking," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, pp. 798–808, 2019.

[183] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation and Test in Europe*, 2017.

[184] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[185] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *ACSAC*, 2006, pp. 463–472.

[186] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO*, 2006.

[187] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *ISSTA*, 2007, pp. 196–206.

[188] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, "Verifying information flow properties of firmware using symbolic execution," in *Design, Automation and Test in Europe*, 2016.

[189] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik, "Lazy self-composition for security verification," in *CAV*, 2018.

[190] A. Danese, V. Bertacco, and G. Pravadelli, "Symbolic assertion mining for security validation," in *DATE*, 2018, pp. 1550–1555.

[191] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *CCS*, 2007.

[192] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[193] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it-Information Technology*, vol. 61, no. 1, pp. 45–58, 2019.

[194] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[195] H. Mantel and D. Sands, "Controlled Declassification based on Intransitive Noninterference," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 129–145.

[196] R. Denning and D. Elizabeth, *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982, ISBN: 0-201-10150-5.

[197] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *NDSS*, 2003.

[198] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *Design Automation Conf.*, 2019, 188:1–188:6.

[199] S. Tempel, V. Herdt, and R. Drechsler, "SymEx-VP: An open source virtual prototype for os-agnostic concolic testing of iot firmware," *Journal of Systems Architecture*, vol. 126, p. 102 456, 2022, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102456. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000480.

[200] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.