# Memory Mapped I/O Register Test Case Generator for Large Systems-on-Chip

Roni Hämäläinen
*Unit of Computing Sciences*
*Tampere University*
Tampere, Finland
roni.hamalainen@tuni.fi

Henri Lunnikivi
*Unit of Computing Sciences*
*Tampere University*
Tampere, Finland
henri.lunnikivi@tuni.fi
0000-0003-4817-2939

Timo Hämäläinen
*Unit of Computing Sciences*
*Tampere University*
Tampere, Finland
timo.hamalainen@tuni.fi
0000-0002-7867-0800

***Abstract*—This paper addresses automated testing of a massive number of Memory Mapped Input/Output (MMIO) registers in a real large-scale Systems-on-Chip (SoC). The golden reference is an IP-XACT hardware description that includes a global memory map. The memory addresses for peripheral registers are required by software developers to access the peripherals from software.**

**However, frequent hardware changes occur during the HW design process, but the changes might not always propagate to the SW developers and an incorrect memory map can cause unexpected behaviour and critical errors. Our goal is to ensure that the memory map corresponds exactly to the HW description.**

**The correctness of the memory map can be verified by writing software test cases that access all MMIO-registers. Writing them manually is time consuming and error prone, for which reason we present a test case generator. We use a Rust-based software stack, where the generator itself is written in Rust while the generator input is in CMSIS-SVD-format that is generated from IP-XACT. We have used the generator extensively in Tampere SoC Hub Ballast and Headsail SoCs and fixed several errors before the chips manufacturing. The test generator can be used with any IP-XACT based SoCs.**

***Index Terms*—verification, MMIO, SVD, IP-XACT, Rust**

## I. Introduction

The System-on-Chip (SoC) Hub project of Tampere University targets the design and implementation of one large chip per year [19]. The project combines multiple stakeholders' effort to boost SoC design competence from application requirements to chips, software stack and demo applications. To this date, two SoCs, Ballast and Tackle, have been manufactured and tested. The third SoC, Headsail, is taped out in fall 2023. Ballast [14] is a heterogeneous RISC-V based multi-processor SoC (MP-SoC) designed with a subsystem-based architecture and design methodology. Ballast includes three RISC-V based subsystems: SysCtrl, housing a 32-bit single-core RISC-V CPU, *MPC*, also a 32-bit single-core, a 64-bit dual-core *HPC*, a digital signal processor designed using OpenASIP [9], Nvidia NVDLA-based deep learning accelerator [12] and three connectivity subsystems. Subsystems communicate via a multi-level interconnect, consisting of a top-level and subsystem-level interconnects.

To write software that communicates with the various peripherals present in the system, the SW developers need to know which addresses map to which peripherals. These addresses are documented into one or multiple memory maps. Each processor subsystem has a separate memory map and in total the Ballast contains 3 252 Memory Mapped IO (MMIO) registers and each register can contain multiple fields. Due to the multi-level interconnect architecture, a single memory access can contain multiple address translations which makes determining final memory maps complex.

This paper addresses all Ballast's RISC-V subsystems which can all use the same software framework despite their distinct architectures. The software stack for SoC Hub chips supports both bare-metal applications and Linux-kernel and is written in Rust which is a modern memory-safe systems-programming language [16].

Rautakoura et al. discuss the design flow [14] and methodology [15] of Ballast. This paper contributes to the verification phase of the overall development, and more specifically the HW/SW-boundary. Our research question is summarized as follows: how to make sure that the given memory map represents the given SoC correctly for software development? We will focus on the presence, location and some limited behaviour of the MMIO-registers based on the SoC HW description given in IP-XACT as the golden reference. How to fix the HW design errors or manipulate the IP-XACT descriptions in the design flow are out of the scope of this paper. In addition, we focus on simulation-based verification. The method presented in this paper is available as an open source tool [11] and it can be generalized to any platform that is representable with standard IP-XACT.

This paper is structured as follows. The Ballast SoC and its software stack is described in section II. The verification problem is described in detail in section III, after which we present related work in section IV. Our approach is described in section V, followed by the results presented in section VI. Discussion of the results is presented in section VII and conclusion with future ideas in section VIII.

Funded by Business Finland and SoC Hub

979-8-3503-3757-0/23/$31.00 ©2023 IEEE

## II. SoC Hub chips

In the following we take Ballast as an example of the SoC Hub chips. The architecture is presented in Figure 1 [14]. Each Ballast subsystem conforms to a subsystem architecture template which includes an independent clock domain with a Phase-Locked Loop (PLL) and a standardized interface to the rest of the system, including an AXI-interface, a clock domain crossing, status-signals, interrupts and clock and reset control signals. Subsystems enable a hierarchical ASIC design flow, enabling the subsystems to be verified separately and in parallel.

The top-level interconnect contains three fully-connected crossbars with independent clock domains and is divided to 64-bit and 32-bit wide data regions. Processor subsystems communicate with other subsystems and system-level peripherals, e.g. the global interrupt router, via the top-level interconnect, interrupts and shared memory. Processor subsystems communicate with subsystem-level peripherals, e.g. timers, via their respective subsystem-level interconnect. Since the processor subsystems were designed to be as autonomous as possible, each of them provides their own JTAG and memory for verification, debug and chip validation. Peripherals are exposed to the processor cores via MMIO-registers by normal memory access instructions. The assignments of peripherals' registers are documented in the memory map.

The SW stack is depicted in Figure 2. A peripheral access crate (PAC) is generated from the memory map. It provides an interface to the MMIO-registers which is then used by the hardware abstraction layer (HAL). RISC-V-specific routines are provided by a µ-architecture crate *riscv* which is adapted to Ballast with *ballast-riscv*. A minimal runtime for bare-metal applications is provided by *riscv-rt* which is again adapted to Ballast with *ballast-rt*. Bare-metal applications can be built on top of these components as presented in the figure where a horizontal dotted line separates software that supports bare-metal applications from software designed to support operating systems (OS). The boot ROM loads a program called *hpc-loader* which configures the system for *rustsbi* [17] which eventually boots the Linux-kernel using U-Boot [22].

Since each processor subsystem uses its own memory map, each subsystem also has its own PAC. The HAL depends on the PAC but with compile-time configuration, the same HAL can be used with all processor subsystems. The µ-architecture crates and runtimes are adapted to corresponding processor subsystem. Changes to the memory map directly change the corresponding PAC. The HAL needs modification if the PAC's symbolic names change. Since the µ-architecture crate depends on the processor ISA, which is in this case standardized by RISC-V, it is unlikely to be affected by changes in the memory map. However, changes to memory regions affect the runtime's linker script which is used to determine the placement of the SW sections.

## III. Overview of the verification problem

During system design, the hardware team selects what components are used and how they are connected based on the requirements and the specification [6]. Typically both new and reused components are included. Each IP block has its own memory addresses, and after instantiation an SoC specific memory map is created. SoC Hub-project uses the Kactus2-program to create, modify and integrate IPs into SoCs [19] [20]. Internally, Kactus2 uses IP-XACT-format which is an XML-based standard to create and exchange IPs in computer-readable format [8]. IP-XACT can also contain the memory map of the system with registers, their locations and fields [8]. Kactus2 is used by the designers to create, modify and view the memory map in a graphical user interface [20]. Kactus2 can also detect and warn the designer about overlapping memory space assignments and verify that the MMIO-registers presented in IP-XACT are physically reachable from the other components. An example of a memory map of a hypothetical system, presented using Kactus2, is depicted in Figure 3.

The memory map is integrated to the SW stack as presented in Figure 2. A Kactus2 plugin transforms IP-XACT into CMSIS-SVD-format [20] which is also XML-based and similar to IP-XACT but focuses on the programmer's view of the device [2] instead of the HW designer's. The *svd2rust* [4] [21] transforms the SVD-file into Rust-bindings, i.e., the PAC.

The HW evolves during the design process due to refinement and optimization and when functionality and architecture changes. Unfortunately, it is possible that the changes to the HW do not always propagate to the memory map even if quality assurance measures are taken in the process. The HW team can accidentally forget to add the IPs to the memory map, use the wrong IPs or IP configurations in the memory map, assign IPs to wrong memory spaces or make mistakes in the documentation of the assignments and the IP's registers. Possible sources of failure include following scenarios:

1) one or more of the registers of the peripheral are missing from the memory map,
2) an address in the memory map to the wrong register, e.g., the memory map claims that an address points to register X of peripheral A, when it actually points to register Y or peripheral B,
3) an address in the memory map points to a non-existent register and
4) memory map has incorrect information about a register, e.g., the memory map claims that an address is read-write when it actually is read-only.

Using an incorrect memory map can compromise the system's ability to perform its intended task since the system behaves unexpectedly. To avoid this situation, the memory map should be verified.

Ballast was verified in a hierarchical manner where subsystems were verified as separate entities and top-level verification focused on verifying the integration of the subsystems and on verifying top-level functionality such as the global ad-

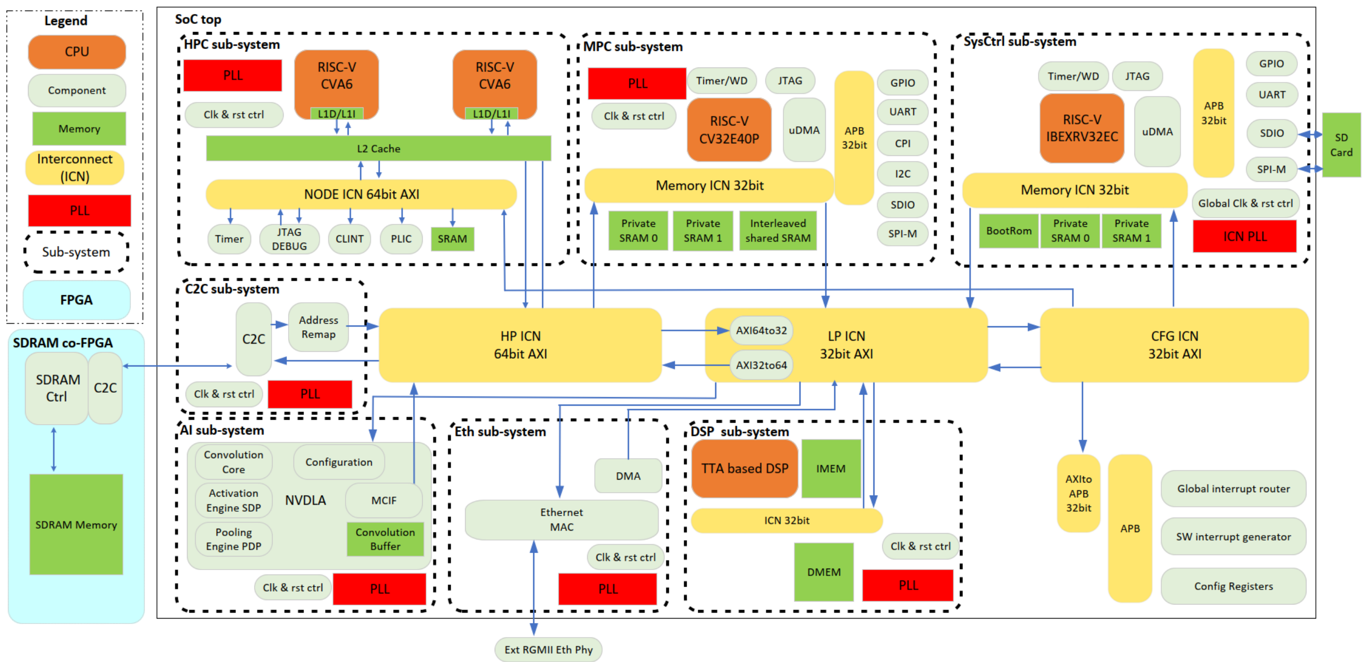


Fig. 1. Structure of the Ballast SoC [14].

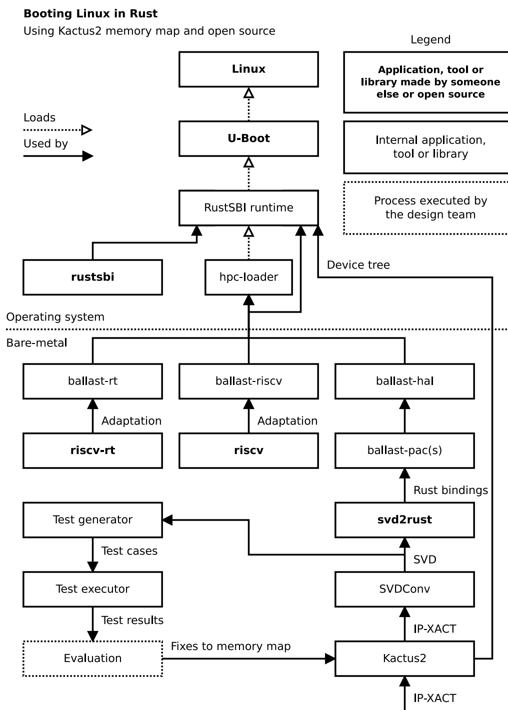


Fig. 2. Overview of Ballast's software stack.

dress map [14]. Verification methodologies included Universal Verification Methodology (UVM) and HW-SW co-simulation.

Co-simulation was used to functionally verify the design by first writing stimulus programs, executing these programs in the system and comparing the system's response to expected response. The register-transfer level (RTL) code of Ballast

Fig. 3. An example memory map visualized in Kactus2.

and stimulus programs were simulated using Siemens' QuestaSim [18]. Same co-simulation test cases could also be used with FPGA prototyping and ASIC sample testing [14].

Often these stimulus programs test one or few functionalities of the system. E.g., one program could test that UART's registers are accessible and that the UART behaves as expected and another could test GPIO. Since not all functionalities use all MMIO-registers, it is possible that a subset of the registers are not tested at all. Writing multiple stimulus programs manually to test all registers can be a tedious and error-prone task. If the HW changes during the design process, then some programs might need a rewrite. Misunderstanding the changing documentation about the memory map and peripheral's behavior can lead to errors in the test programs. Like Ballast, a single system can contain multiple memory maps which all must be tested. One way to avoid writing register test cases manually is to generate them from the SVD's memory map as described more thoroughly in section V.

## IV. Related work

Generation of software from IP representation formats is not a new idea. Model-Driven Development (MDD) has been popular with its idea that correct implementations are generated from models at higher abstraction levels. Ideally with perfect code generators which produce code that is ”correct by construction”, the need for verification is reduced. Instead of comparing the implementation against the specification, more attention is spent on the generators. However, an argument against MDD is that the modeling effort is high and generator flows are seldom bi-directional. In practice, verification is still needed to make sure that specifications and implementations match each others. Test generation is as relevant as the idea of code generation. Ecker et al. described in 2009 generation of HW/SW interface and behavioral code using IP-XACT and behavioral models such as TLM with the goal of eliminating manual implementation of low-level SW [23].

Hunsinger, Francois and Jerraya presented generation of SW test programs from COLIF-format [3] for functional verification of SoCs in 2003 [7]. Their approach comprises generation of high-level test programs and the subsequent refinement of these test programs to low-level test programs. The high-level test programs are generated using a global test plan, component specific test plans, expected test results from behavioral simulation and the COLIF SoC representation. The low-level test programs use a custom OS generator which is able to generate API, OS and HAL layers. The OS provides an API for high-level test programs which can enable reuse of the test programs on different HW platforms and more complex test scenarios involving parallelism and scheduled tasks.

Lins and Barros presented a HAL generator from IP-XACT for SoC functional verification in 2010 [10]. This HAL generator is able to create C-functions that manipulate registers and their fields at a high abstraction level. They call this method *Processor Driven Testing* (PDT). They applied the generator to test functionality of a UART serial port of Leon2 processor architecture.

The scope of above related work is larger than ours, but follows the similar pattern of taking some HW description as the golden reference and performing defined operations on it. However, we did not find publications or open source tools exactly for our problem on systematic testing of MMIO-registers by generating test cases from the IP representation format.

## V. Methods

Based on the memory map we know what peripherals exist, what registers exist, what memory address points to what register and details about each register. These details include register’s bit width, fields, reset values, access rights and so on. We can test that a memory address:

1) is readable if it should be,
2) is writable if it should be,
3) contains an expected value, e.g., a specific value on reset, called *reset value* and
4) the bit width of the address is as expected.

Testing is limited to these items since SVD can not express how to interpret the register values. This kind of higher level functional verification is thus left to other methods.

Checking the readability and the writability of a register requires that the processor generates an exception when a non-existent memory address is accessed. This requirement is specified by, e.g., the RISC-V standard where physical memory attributes are checked for all accesses to physical memory and violations generate access fault exceptions [1]. If the access rights to memory addresses can be configured during runtime, then they must be configured appropriately before running the test cases. Some registers react to accesses with side-effects, e.g., by clearing the register after it is read. These side-effects can [2] and should be documented in the SVD-file to properly generate the test cases.

A single test case is dedicated to test one MMIO-register and can consist of reading a value from a memory address, checking if the received value is expected value, writing another value to the memory address and then writing back the original value. This test case checks that the given memory address is readable, contains the expected value and that it is writable. Expected value can be, e.g., the reset value of the register. The content of a test case depends on the information parsed from the input as explained in Subsection V-B.

Overview of the SW components involved are presented in Figure 4. The generator is executed through a command line interface and is responsible for parsing the input and generating the test cases from the internal model. The test case executor is responsible for executing test cases on the target hardware and is compiled with the test cases and a test runtime. The test results are reported as Rust enumerations and can be printed out in a platform dependent way, e.g., by using the serial interface provided by the platform runtime.

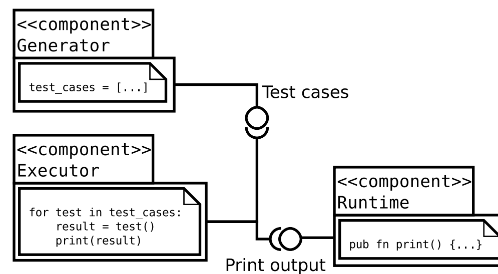


Fig. 4. Overview of the test generator components.

Rust was chosen as the implementation technology because it is already used internally by the SoC Hub SW team for the development of composable hardware dependent software, and because it offers convenient code generation facilities that allow the production of Rust-language source files that can be compiled directly to target assembly. This is in contrast to more traditional C-based code generation flows which often rely on a higher level language, such as Python, to generate text that is then interpreted by a C-compiler to target assembly.

Additionally, the Rust compiler guarantees the program's memory-safety without garbage collection [16] which is not provided by C. Thus, programs written in Rust have one risk mitigated by design with little performance cost.

The generator itself in its current realization contains about 1 600 lines of code. The default executor without the runtime contains about 160 lines of code. Ballast SoC for example contains about 57 000 lines of RTL code including test benches. For SysCtrl-processor, 984 read test cases were generated with about 17 000 lines of code in about 3.3 seconds. The test executable size with all test cases was about 4.5 MB with optimizations enabled. The executable size can be controlled by filtering testable subsystems and test types. The results for other subsystems can be seen from Table I.

### A. Input parsing and modeling

The first step is to parse the SVD-file using a XML parser and transform its contents into an internal model which contains all registers with their hierarchical locations and attributes such as word sizes, access rights and fields. The generator can validate the input's conformance to SVD-format. It also suggests potential improvements such as missing definitions that could make the test case generation more accurate.

### B. Test case generation

The generator iterates through the registers in the model and decides how to test each register. Test case is built based on given detailed information about the register. E.g., read-write registers are read and written to, read-only registers are only read, registers with reset values can be reset, read values are compared to expected values and so on. Generated test cases are finally written to an output file. An example of a test case which reads a value from a register and compares it to a expected value is presented in Listing 1.

```
pub fn test_register1_0x2000() -> Result<()> {
    let reg_ptr: *mut u32 = 0x2000 as *mut u32;
    let read_value = unsafe { ptr::read_volatile(reg_ptr) };
    if read_value != 0x0u32 as u32 {
        return Err(Error::ReadValueIsNotResetValue {
            read_val: read_value as u64,
            reset_val: 0x0u32,
            reg_uid: "peripheral3-register1",
            reg_addr: 0x2000,
        });
    }
    Ok(())
}
```

Listing 1: Test case example.

### C. Test case execution

The next steps are to compile the final binary, execute it and process the results. Our solution has a test case executor that includes the source file with the test cases. It is up to the implementer of the executor to decide what subsystems are tested, what registers are tested, in what order and what to do with the test results. Our executor iterates through each test case, executes it and provides the results via UART.

A possible problem has been found by the test, if is causes an access fault exception or returns an error as a result value. Access fault exception during register access can imply that the register does not exist or that the access rights are implemented or documented incorrectly. The user then documents the memory address, the expected register in that address and values read from or written to the register. The overall behavior and state of the system can also be documented, e.g., is the system responding to external interrupts and states of the various configuration registers. The HW team is then consulted and if the problem is indeed related to the memory map or IPs, the HW team can then proceed to correct the issue.

Continuous Integration [5] (CI) pipelines were used in the SoC Hub project to provide automatic building and regression testing of the hardware design with the test case generator integrated to the workflow. After a change is pushed to the remote repository, the CI-pipeline is activated. First it creates a container with necessary configuration and source code and compiles the RTL-files to be ready for an RTL-simulator. Test cases are generated and built with the executor and transformed into a bitstream which is then written to the RTL memory. Finally, the test cases are executed in parallel and the results are printed out. With Headsail, a typical time for configuring and running the test cases took from 5 to 10 minutes for each job where a single job tested some part of the processor subsystem's full memory map. A successful example output is presented in Listing 2. A failed example output is presented in Listing 3. In this case, the test case times out due to the hanging of the simulated AXI-bus, which signals an error in register accessibility with regards to what was expected. This timeout could be avoided if the platform supported trapping PMA violations as precise access fault exceptions as recommended by the RISC-V standard [1]. Since the address of the violating instruction is known precisely, the execution of test cases can continue starting from next instruction after the address and access method that caused the exception are recorded in the exception handler. Then. after all of the test cases have been executed, the program sets a bit in an agreed upon memory address to signal completion. The test bench driver polls this memory address, notices the set bit and proceeds to finish the simulation. HW team then gets notified about the test results.

```
[STDOUT] All subsystems activated, test count: 2
[STDOUT] UART-test_register1-0x0
[STDOUT] > OK
[STDOUT] UART-test_register1-0x4
[STDOUT] > OK
[STDOUT] [ok]
```

Listing 2: CI-pipeline job output example.

```
[STDOUT] All subsystems activated, test count: 29
[STDOUT] DMA-test_register1-0x0
ERROR: Job failed: execution took longer than 15m0s seconds
```

Listing 3: Failed CI-pipeline job.

## VI. Results

The register test generator has been applied in the verification effort of SoC Hub's Ballast and Headsail SoCs.

Multiple cases were detected where changes to the HW did not propagate to the memory map. The results for Ballast and Headsail SoCs and their subsystems are presented in Table I.

TABLE I
RESULTS

| SoC | Processor subsystem | #-of test cases | Lines of code generated (kLoC) | #-of actionable issues found |
|---|---|---|---|---|
| **Ballast** | SysCtrl | 984[a] | 16.8 | 20 |
| | MPC | 984[a] | 16.8 | 65 |
| | HPC | 984[a] | 16.8 | 25 |
| **Headsail** | SysCtrl | 605 | 14.6 | 11 |

[a] Ballast's SysCtrl, MPC and HPC subsystems have access to same peripherals.

Next we discuss some example problems found and fixed using this tool during verification of the Headsail SoC. In one case, the test cases detected that most of SysCtrl subsystem's registers' reset values were incorrect. The source of the error was determined to be that the subsystem's base address was assigned to a different address in the HW design compared to what was claimed in the memory map. The HW team was informed about the issue and the base address was corrected in the IP-XACT model. In another case, deprecated µDMA [13] registers were incorrectly left into the memory map while the hardware IP was updated. When the test cases attempted to access these registers, the processor generated an access fault exception. The deprecated registers were removed from the IP-XACT model. At one point of the design process, no test cases were generated for SysCtrl subsystem. When the reason was investigated, it was determined that SysCtrl's memory map had vanished. The hardware team was informed and the memory map was restored. In another case, the test cases detected that the processor could not access certain subsystems, including both Headsail's DMAs. The common denominator with these issues was determined to be the AXI-interconnect. The connection from AXI to these subsystems was then restored.

## VII. DISCUSSION

We present an MMIO-register test generator that generates test cases from IP representation formats such as CMSIS-SVD. Test cases are used to partially verify that given memory map represents given system correctly and they can solve some of the error scenarios listed in section III as presented in Table II.

TABLE II
ERROR SCENARIOS

| | Scenario | Detected by the test cases? |
|---|---|---|
| 1. | A register is missing from the memory map. | No. |
| 2. | An address points to the wrong register. | Sometimes. |
| 3. | An address points to a non-existent register. | Yes. |
| 4. | Incorrect register metadata. | Yes. |

Since the test cases are generated from a given memory map, the test cases can not detect peripherals which are not defined in the memory map and scenario 1 remains unsolved. Scenario 2 can be solved if the wrong register has different access rights or reset value compared to the correct register. Even without the previous assumptions, the error can still be detected at peripheral-level. Assume that the memory map claims that a memory space A should belong to a peripheral B with registers $B_1$, $B_2$, but it actually contains a peripheral C with registers $C_1$, $C_2$. Even if the registers $B_1$ and $C_1$ behave identically, it is possible that the registers $B_2$ and $C2$ or later behave differently. If this difference is detected, it can be noticed that the memory map claims the memory space to a wrong peripheral. If the processor generates an access fault exception when accessing a non-existent memory address, then the test cases can detect registers which are claimed to exist by the memory map but do not exist in the hardware, solving scenario 3. If the processor can also generate an access fault exception when reading a write-only or writing a read-only register, then the test cases can detect registers that are wrongly defined in the memory map, solving scenario 4.

Presented method can be generalized to other platforms where the HW can be presented with IP-XACT or SVD and which are supported by the Rust-compiler, which are in-effect, the targets supported by the LLVM compiler and toolchain technologies. These constraints can change if more input and output formats are to be supported. Due to dependence on the information provided in the IP representation format, some error sources are not eliminated. E.g., if the memory map claims that a read-write register is read-only, then writing is not tested. One solution is to test reading and writing with all registers and check which registers behave as expected.

## VIII. CONCUSIONS AND FUTURE WORK

Currently a subset of SVD's features such as register arrays, write constraints, read actions and access rights are supported. Some registers or their features are not tested. The generator can not distinguish if a register can only be written once, if the original value of the register must be written back after test write, and similar sequential events. Also the possible side-effects of accesses are not utilized. Verification resolution could be increased by enabling the test cases to e.g., check if the value is set to zero after reading if it should.

Currently the test case generator utilizes only register-level information. Since a single register can have multiple fields, it is possible that different fields have e.g., different access rights, read-only and read-write fields. If the test resolution is the whole register, then writing to the whole register is prohibited if one field does not allow it. Support for field-level test cases is added in the future. More input and output formats can be supported. Support for IP-XACT is underway.

The presented MMIO-register test generator has been successfully used in SoC Hub chips, and it helped to detect several critical errors before the manufacturing of the chips. One future work idea is to try the tool with other platforms as well. We learned that automatically generated MMIO-register test cases can be used to partially check that given memory maps represent the given MP-SoC correctly. The generator is available as an open source tool [11] and it can be used with any SoC design that inludes the IP-XACT HW description.

# REFERENCES


[1] Andrew Waterman, Krste Asanović, and John Hauser, RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. December 2021.

[2] Arm Ltd. CMSIS-SVD Description (*.svd) Format. https://www.keil.com/pack/doc/CMSIS/SVD/html/svd_Format_pg.html. [Online, accessed 2023-04-12].

[3] Wander O Cesário, Gabriela Nicolescu, Lovic Gauthier, Damien Lyonnard, and Ahmed A Jerraya. Colif: A design representation for application-specific multiprocessor SoCs. *IEEE Design & Test of Computers*, 18(5):8–20, 2001.

[4] Embedded devices Working Group. svd2rust. https://github.com/rust-embedded/svd2rust. [Online, accessed 2023-04-12].

[5] GitLab B.V. GitLab CI. https://about.gitlab.com/topics/ci-cd/. [Online, accessed 2023-04-12].

[6] Soonhoi Ha and Jürgen Teich. *Handbook of Hardware/Software Codesign*. Springer, 2017.

[7] F Hunsinger, Sebastien Francois, and Ahmed Amine Jerraya. Definition of a systematic method for the generation of software test programs allowing the functional verification of system on chip (soc). In *Proceedings. 4th International Workshop on Microprocessor Test and Verification-Common Challenges and Solutions*, pages 11–16. IEEE, 2003.

[8] IEEE. IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. *IEEE Std 1685-2022 (Revision of IEEE Std 1685-2014)*, pages 1–750, 2023.

[9] Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, and Heikki Berg. *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*, pages 147–164. Springer International Publishing, 2017.

[10] Tiago Lins and Edna Barros. The development of a hardware abstraction layer generator for system-on-chip functional verification. In *2010 VI Southern Programmable Logic Conference (SPL)*, pages 41–46. IEEE, 2010.

[11] Henri Lunnikivi and Roni Hämäläinen. MMIO Test Generator. https://github.com/soc-hub-fi/mmio-test-generator. [Online, accessed 2023-09-28].

[12] NVIDIA Corporation. The NVIDIA Deep Learning Accelerator. http://nvdla.org/. [Online, accessed 2023-08-23].

[13] Antonio Pullini, Davide Rossi, Germain Haugou, and Luca Benini. µDMA: An autonomous I/O subsystem for IoT end-nodes. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[14] Antti Rautakoura, Timo Hämäläinen, Ari Kulmala, Tero Lehtinen, Mehdi Duman, and Mohamed Ibrahim. Ballast: Implementation of a large mpsoc on 22nm asic technology. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 276–283. IEEE, 2022.

[15] Antti Rautakoura and Timo Hämäläinen. Does soc hardware development become agile by saying so: A literature review and mapping study. *ACM transactions on embedded computing systems*, 22(3):1–27, 2023.

[16] Rust Foundation. Rust. https://www.rust-lang.org/. [Online, accessed 2023-04-12].

[17] RustSBI development team. RustSBI. https://github.com/rustsbi/rustsbi. [Online, accessed 2023-08-23].

[18] Siemens. Questa advanced simulator. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. [Online, accessed 2023-08-24].

[19] SoC Hub. SoC Hub. https://sochub.fi/. [Online, accessed 2023-04-12].

[20] Tampere University System-on-Chip Research Group. Kactus2. https://github.com/kactus2/kactus2dev. [Online, accessed 2023-04-12].

[21] The resources team. Appendix A: Glossary. https://docs.rust-embedded.org/book/appendix/glossary.html. [Online, accessed 2023-08-08].

[22] The U-Boot development community. The U-Boot Documentation. https://u-boot.readthedocs.io/en/latest/. [Online, accessed 2023-08-23].

[23] Wolfgang Ecker and Wolfgang Müller and Rainer Dömer. *Hardware-dependent Software: Principles and Practice*. Springer Science, 2009.