Processor Verification using Symbolic Execution: A RISC-V Case-Study

Niklas Bruns
Institute of Computer Science
University of Bremen
Bremen, Germany
nbruns@uni-bremen.de

Vladimir Herdt
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

***Abstract*—We propose to leverage state-of-the-art symbolic execution techniques from the *Software* (SW) domain for processor verification at the *Register-Transfer Level* (RTL). In particular, we utilize an *Instruction Set Simulator* (ISS) as a reference model and integrate it with the RTL processor under test in a co-simulation setting. We then leverage the symbolic execution engine KLEE to perform a symbolic exploration that searches for functional mismatches between the ISS and RTL processor. To ensure a comprehensive verification process, symbolic values are used to represent the instructions and also to initialize the register values of the ISS and processor. As a case study, we present results on the verification of the open source RISC-V based MicroRV32 processor, using the ISS of the open source RISC-V VP as a reference model. Our results demonstrate that modern symbolic execution techniques are applicable to a full scale processor co-simulation in the embedded domain and are very effective in finding bugs in the RTL core.**

I. Introduction

Verification of the processor at the *Register-Transfer Level* (RTL) is crucial since the processor is a key component in every embedded system. Due to its ease of use and scalability, simulation-based methods still form the primary backbone of the verification effort. Moreover, modern design flows for embedded systems rely on *Virtual Prototypes* (VPs) as a reference model for the *Hardware* (HW) development stage. Regarding the processor, the relevant component of the VP is the *Instruction Set Simulator* (ISS), which is an abstract model of the processor and thus fetches, decodes and executes one instruction after another. To generate processor-level input stimuli several test generation techniques have been proposed that improve upon the classical randomized instruction stream generation. A notable direction is the model-based approach that relies on a constraint-based specification to guide the test generation process [1]–[3]. Further optimizations have been proposed by propagating constraints among multiple instructions in a more effective way [4]. However, model-based approaches require significant effort to provide a respective input format specification. A recent trend, that tries to mitigate this issue, is to leverage automated verification techniques from the SW domain and apply them in the HW domain. A particular effective technique in this regard is fuzzing [5], [6]. Modern coverage-guided fuzzer work by mutating randomly created data and are guided by coverage, hence they do not require an input model specification [7], [8]. Fuzzing has been shown very effective for processor-level stimuli generation [9] and by using an ISS as reference model for the RTL processor under test, an effective fuzzing-based processor verification methodology is obtained [10]. However, while being very effective, even a state-of-the-art fuzzing-based approach is still suceptible to miss corner case bugs as it is an inherently incomplete testing approach.

Looking again for inspiration in the SW domain, the working solution to address the issue of finding corner-case bugs efficiently is by using the symbolic execution technique. In contrast to fuzzing, it is a formal verification technique that allows to execute a program using symbolic expression, that efficiently represents sets of concrete values, and thus enables to explore large state spaces more efficiently and comprehensively [11]. State-of-the-art symbolic execution engines, such as KLEE, have been very effective in finding numerous intricate bugs in SW programs [12]–[14]. Following the success story from the SW domain, first reaserch approaches start to leverage symbolic execution in the HW domain. However, the focus is mainly on test-case generation techniques to boost the obtained coverage for general RTL designs [15]–[17]. An effective processor verification methodology using symbolic execution is to the best of our knowledge not yet available.

In this paper, we thus propose to leverage state-of-the-art symbolic execution techniques for processor verification at the RTL and describe an effective open source tool flow that takes the requirements of a modern system design flow into account. In particular, we utilize an ISS, which is readily available as a C++ description in a modern VP-based design flow, and integrate it with the RTL processor under test in a co-simulation setting using a testbench. The testbench is designed to provide the same instructions to the ISS and processor, and compare the register values after execution. To ensure a comprehensive verification process, symbolic values are used to represent the instructions and also to initialize the register values of the ISS and the processor. Following a standard RTL design flow, we assume that the processor description is available in Verilog, which is then translated into a C++ description using the open source verilator tool. As such the RTL processor can be integrated with the C++ ISS in a combined C++ co-simulation. We then leverage the open source C++ symbolic execution engine KLEE to perform a symbolic exploration according to the co-simulation setting. We designed dedicated symbolic execution interfaces in our testbench to enable such an integration and as such we can benefit from the vast symbolic execution optimizations that KLEE provides. As a case study, we present results on the verification of the open source RISC-V based MicroRV32 processor[1]. We use the ISS from the open source RISC-V VP[2] as functional reference model for the MicroRV32 processor. Our results demonstrate that modern symbolic execution techniques are applicable to a full scale

[1]Available at GitHub https://github.com/agra-uni-bremen/microrv32.
[2]Available at GitHub https://github.com/agra-uni-bremen/riscv-vp.

processor co-simulation in the embedded domain and are very effective in finding bugs in the RTL core.

II. Related Work

As mentioned in the introduction, symbolic execution techniques, which have been very effective in the SW domain, are increasingly leveraged to tackle problems in the HW domain. One important research direction is to automatically generate test cases that improve the coverage of the HW under test. [18] present an approach based on static analysis in combination with symbolic execution techniques applied on execution traces of the RTL design to systematically drive up the branch coverage. [15] propose applying symbolic execution tools from the SW domain to generate test vectors for RTL designs and evaluate their approach using a floating point unit. [16] present a control flow graph assisted approach that enables to guide the symbolic execution engine in uncovering specific remaining coverage targets. [17] describe further optimizations to boost scalability by avoiding overlapping searches involving multiple coverage targets. Beside test vector generation to improve the coverage, another research direction that employs symbolic execution techniques at the RTL is the security evaluation of the design. [19] present a method for exploits generation by using a backward symbolic execution approach based on a set of security-critical invariants. [20] describe an approach that works by adding security-critical assertions into the RTL design and then show that leveraging symbolic execution is effective in detecting assertion violations. However, an effective processor verification methodology using symbolic execution at the RTL in combination with a VP-based reference model, as proposed in our paper, is to the best of our knowledge not yet available.

Classic formal verification approaches targeting RTL designs rely on HW model checking techniques that verify the design against a set of properties specified in temporal logic, e.g. [21]–[23]. The most notable approach for formal RISC-V processor verification is RISCV-formal. It is a framework for formal verification based on *Bounded Model Checking* (BMC) [24]. For the verification, RISC-V-formal needs complex formal models as a reference, and our approach only requires a C++ description already used in Virtual Prototypes. Virtual Prototypes can be used for the rapid development of new functionality and debugging. Furthermore, RISCV-formal is incapable to find implementation mismatches between an ISS and a RTL processor implementation. These mismatches complicate debugging and can lead to subtle HW/SW interaction errors that may lead to security vulnerabilities. And last but not least, RISCV-formal does not support Control and Status Registers (CSRs). Consequently, our symbolic execution based approach, is complementary to classical formal verification approaches.

III. Background on RISC-V

RISC-V is an open and royalty-free Instruction Set Architecture (ISA). It has a modular design and is popular in industry and academia. The RISC-V specification is administered by the non-profit RISC-V International association, which was founded in 2015 [25]. The RISC-V specification is divided into two volumes. The first part is the unprivileged specification [26]. The heart of RISC-V is the instruction set I that is available in 32bit, 64bit, and 128bit versions. RISC-V has additional instruction set extensions like Multiply/divide (M) or Compressed instruction (C). The second volume is called the privileged architecture [27]. It contains all components needed for the hardware-software interactions like Control and Status Registers (CSRs) for hardware identification, trap handling, and performance measurement.

IV. Processor Verification using Symbolic Execution

In this section, we present our proposed processor verification approach, which is based on co-simulation and symbolic execution. We start with a general overview and then describe the relevant parts in more detail.

A. Overview

Fig. 1 shows an overview on our approach. The complete flow starts with a SpinalHDL processor description and a C++ ISS description (left side of Fig. 1). SpinalHDL is an open-source high-level hardware description language that is based on Scala. It aims to give the hardware designers a tool to design a new abstraction level to create reusable code [28]. Popular open source processor implementations that are based on SpinalHDL are VexRiscV [29] and MicroRV32 [30]. Next, the SpinalHDL processor description is configured based on the configuration description and translated using the *Scala Build Tool* (SBT) into an RTL core. Likewise, the C++ ISS description is configured into a C++ ISS. Because the RTL core and the C++ ISS are configured based on the same processor configuration description, the RTL core and the C++ should behave in the same way at the functional level. Next, the RTL core is transcompiled (*verilated*) into a cycle-accurate C++ equivalent using the open source tool named verilator [31]. The transcompiled RTL core (C++) and ISS (C++) are combined with the co-simulation main and compiled into bytecode using the LLVM toolchain to create the processor co-simulation (right side of Fig. 1). The co-simulation contains the processor and ISS bindings, symbolic execution interface, instruction stream, symbolic memory & registers, and a voter that compares the functionality of the ISS and the RTL core. The symbolic processor co-simulation is executed on the C++ symbolic execution engine KLEE to generate test vectors in order to find bugs and create a high coverage test set. The essential components of the co-simulation are described in the following subsection in detail.

B. Symbolic Co-Simulation

In the following, we describe the essential parts of the co-simulation. The co-simulation main loop consists of the following parts: the initialization of the symbolic memory and sliced symbolic registers, the handling of the *Data Bus* (DBus) and *Instruction Bus* (IBus) of the RTL Core and the connection to the symbolic memory, the monitoring of the RTL core execution behavior, including the determination of the complete initialization, execution of ISS steps and comparison of the execution results (Voter), and the enforcement of runtime limitations (Execution Controller). Next, we describe the symbolic execution interface, the symbolic memory, and the sliced symbolic registers. Last but not least, we describe the voter and execution controller.

C. Symbolic Memory & Registers

In the following, we describe our symbolic memory and registers. The IBus and DBus are separated in many processor implementations to avoid performance bottlenecks. Therefore the symbolic memory consists of the symbolic instruction and the symbolic data memory. The symbolic execution interface is used to connect the symbolic memory and the symbolic execution engine. The symbolic interface in our approach contains the functions *klee_make_symbolic* and *klee_assume* of the symbolic execution engine KLEE. The function *klee_make_symbolic* is used to mark a variable as symbolic. The function arguments are the variable address, the variable size in bytes, and a freely choosable name

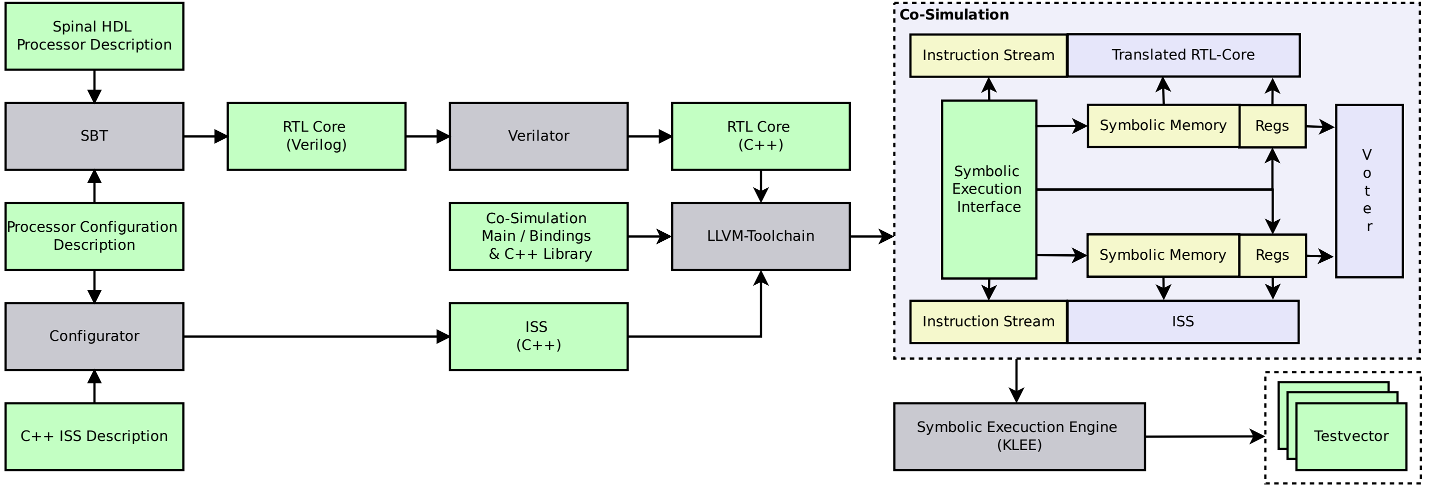

Fig. 1. Overview on our proposed processor verification using a symbolic co-simulation

for the variable [32]. The function *klee_assume(condition)* is used to constrain the symbolic variables and adds the condition to the current path constraints [33]. Next, we describe the *symbolic instruction memory*, the *symbolic data memory* and last but not least the *sliced symbolic registers*.

1) Symbolic Instruction Memory: The symbolic instruction memory is the memory that provides the instructions for the co-simulation. The instruction memory is read-only and shared between the RTL core and the ISS. It is connected via the IBus to the RTL core and is directly connected to the ISS. If the RTL core wants to fetch an instruction at a specific memory address, it sets the signal *IMem_address* and enables the signal *IMem_fetchEnable*. The co-simulation main redirects the request to the symbolic instruction memory. The symbolic instruction memory checks if the instruction at this specific address was already generated and cached. This behavior guarantees that the RTL and ISS are always supplied with the same instructions to prevent false mismatches. A new instruction is generated using the symbolic execution engine if the instruction was not already generated. Therefore, the new instruction variable is marked as symbolic using the function klee_make_symbolic. Depending on the test scenario, klee_assume is used to constrain the instruction generation. Next, the co-simulation main writes the generated instruction into the signal *IMem_instruction* and enables *IMem_instructionReady*. The symbolic co-simulation main loop must disable the signal *IMem_instructionReady* i.e., after one clock cycle, to comply with the bus protocol. Next, we describe the symbolic data memory.

2) Symbolic Data Memory: The symbolic data memory is the memory that provides the data for the cross-processor verification. The data memory can be read and written and is separated for the RTL core and the ISS. It is connected via the DBus to the RTL core and directly to the ISS. The memory sizes of both are the same and they are initialized with the same symbolic values in order to prevent false mismatches. The data memory interface of the ISS has dedicated functions to load *byte*, *ubyte*, *half*, *uhalf* and *word*, and to store *byte*, *half* and *word*. For example, the ISS load_byte function has the goal to load a signed byte at the address of the symbolic memory. It loads a signed 8bit value and sign extends it into a signed 32bit value. The signed extension for the RTL core is handled by the RTL core itself. The DBus interface is based on a strobe logic that is a known logic and used e.g: by AXI that is a ARM Bus standard [34] or by the open Wishbown bus standard [35]. Another example is the native bus interface of the RISC-V processor named PicoRV32 [36]. If the RTL core wants to load data at a memory address, the core enables the signal *DMem_enable* and writes the requested address into the signal *DMem_address*. Additionally, the core sets the signal *DMem_wrStrobe*. The strobe value identifies which of the 32-bytes at the address should be accessed. Valid strobe values are 0001, 0010, 0100 and 1000 to access a byte, 0011 and 1100 to access a half word and 1111 to access a full word. The symbolic co-simulation main loop redirects the request to the strobe based interface of the symbolic data memory. The strobe based interface has a load and a store function. As arguments, this two functions have a combination of memory address and strobe. In opposite to the ISS binding, the strobe based interface does not handle the signed/unsigned convertation. Next, the main loop writes the data into the signal *DMem_readData* and enables the signal *DMem_dataReady*. Again, the symbolic co-simulation main loop must disable the signal *DMem_dataReady* i.e., after one clock cycle to comply with the bus protocol. In the following, we describe the sliced symbolic registers.

3) Sliced Symbolic Registers: For processor verification, every instruction should be tested with arbitrary values to cover all processor instruction functionalities. Because RISC-V is a load/store architecture, most instructions receive their values exclusively from the registers. Only the load and store instructions can access the memory. Thus, it would be sufficient to fill only the memory with symbolic values because symbolic values can be loaded from the symbolic memory into the registers, and from there, these symbolic values can be propagated further. However, this would have the disadvantage that the length of the instruction trace would have to be at least two to test all instruction functionalities. A minimal instruction trace length of two would unnecessarily increase the state space of the verification problem. To solve this problem, we fill the registers with symbolic values. However, simply filling the bank register would also unnecessarily increase the verification problem because the register bank in RISC-V contains 32 32bit values. In order to keep the verification state space as small as possible, we slice the registers into three parts. The first slice of RISC-V contains the register x0. According to the specification, the x0 register is hardwired to zero. Thus this register cannot be symbolic in order to keep the verification approach *sound*. The next slice consists of the symbolic registers. These registers are initialized with arbitrary values to verify every case of the processor instructions. After

the initialization, the register can be read and written like regular registers. This part should be sufficiently large but not larger than needed to keep the state space small. In the last slice are the regular registers, which can be written and read normally. For a processor that implements RV32I, it is perfectly adequate to have only two symbolic registers and fill the rest of the symbolic register bank with regular registers because no RV32I instruction has more than two source registers. Next, we describe the voter and the execution controller of our symbolic co-simulation.

D. Voter & Execution Controller

The voter is based on the *RISC-V Formal Interface* (RVFI) [37]. RFVI is part of the riscv-formal [24] framework for formal verification of RISC-V processors, which allows to observe the execution state of the processor at runtime. After the RTL core has executed an instruction, the *rvfi_valid* signal is enabled. The other signals deliver the execution results as long as this signal is enabled. The results contain values like the actual and old PC and the value of the target register of the executed instruction. Next, the ISS follows up and executes the current instruction. After the execution, the voter compares the execution results of the RTL core and the ISS. If there was an execution mismatch, the voter throws an exception and quits the simulation. If no mismatches were found, the simulation would be terminated after the instruction limit or clock cycle limit was reached. After the simulation, the model will be cleaned up, and the memory is freed. In the following, we describe the evaluation of our processor verification approach.

V. Evaluation

This section presents our case study and discusses the evaluation results. Our case study aims to evaluate the applicability of symbolic execution in combination with a co-simulation for cross-level processor verification. The co-simulation for our case-study uses the MicroRV32 processor [30] as *Device Under Test* (DUT) and the ISS from the open-source RISC-V VP [38] as reference implementation. As instruction set we consider the RV32I+CSR ISA that is supported by MicroRV32. The MicroRV32 processor already has been extensively tested using constrained random techniques [39] and was also verified using the RISC-V formal [24] tool, which applies a *Bounded Model Checking* (BMC) approach.
Our case study is structured into two parts. In the first part we report the results we obtained on verifying the MicroRV32 core. In the second part of our case study, we aim to evaluate the symbolic verification performance using a representative set of injected RV32I errors. All experiments of the two parts are conducted on a Linux server with an Intel Xeon Gold 6240 CPU. Next, we will start with the first part of our evaluation.

A. Case Study: Symbolic Verification of MicroRV32

In this section we present the first part of our case study that aims to verify MicroRV32I using the instruction subset RV32I+CSR. In particular we report and discuss all the mismatches and errors that we have found by continously applying our co-simulation based approach. In this process we observed that mismatches and errors can often be detected rather quickly (we will provide further information on runtime results in the next section) but a more comprehensive exploration can take significantly longer runtime. An exemplary execution in this regard had a runtime of 586905 seconds, executed 101434788 instructions, explored 848 paths completely and 408 paths partially, and generated a total of 1256 test cases. Please note that the need for sliced symbolic registers is evidenced by the fact that a non-optimized symbolic execution requires more than 30 days of runtime. The results of our experiment, in finding errors and mismatches, are listed in Table I. The first column of the table states the instruction or the *Control and Status Register* (CSR) that is responsible for the error or mismatch (column: Instruction & CSR). The second column shows an example instruction that triggers the error or mismatch (column: Example). The next column contains a short description of the found error or mismatch (column: Description). Finally, the last column (column: R) classifies the result as being an error in the RTL core (E), an error in the ISS (E*), or an implementation mismatch between RTL core and ISS (M) due to multiple possible valid implementations according to the RISC-V ISA. All the results are reported to and have been directly confirmed by the author of MicroRV32. In the following, we describe the listed errors and mismatches in more detail. The RV32I load and store Instructions LW, LH, LHU, SW, SH, and SHU are used to load and store data from and to the memory. The processor has multiple permissible handling options, according to the RISC-V ISA, if the memory address is misaligned. For example, a core can raise a trap or fully support misaligned loads and stores. The implementations of the ISS and the RTL core have a mismatch in handling this behavior. The RTL core fully supports misaligned loads and stores, and the ISS checks for misaligned addresses and raises traps. The *Wait for Interrupt instruction* (WFI) is defined in the RISC-V privileged architecture and is available in all privileged modes. This instruction aims to hint to the implementation that the current core can stop execution until an interrupt arrives. As the specification states, it is also legal to implement the instruction as a NOP. The RTL core implementation contains the error that the WFI is not implemented at all, and an attempted execution erroneously raises a trap. If a CSR instruction wants to access non-existent CSRs, the implementation must raise an illegal instruction exception. The RTL core is not compliant to the specification, because it does not raise an exception in this case. Each core has the following read-only ID registers: *Machine Architecture ID Register* (marchid), *Machine Vendor ID Register* (mvendorid), and *Hart ID Register* (mhartid). According to the specification, an illegal instruction exception must be raised

TABLE I
Co-Simulation results (R), shows the errors (E) and mismatches (M) in MicroRV32 and the VP (E*)

Instruction & CSR	Example	Description	R
LW	LW x0, x0, 0x1	Missing alignment check	M
LH	LH x0, x0, 0x1	Missing alignment check	M
LHU	LHU x0, x0, 0x1	Missing alignment check	M
SW	SW x0, x0, 0x1	Missing alignment check	M
SH	SH x0, x0, 0x1	Missing alignment check	M
SHU	SHU x0, x0, 0x1	Missing alignment check	M
WFI	WFI	Missing WFI instruction	E
unimpl. CSRs	csrrwi x0, 0, 0x400	Missing trap at access	E
marchid	csrrci x1, 1, marchid	Missing trap at write	E
mvendorid	csrrw x0, x0, mvendorid	Missing trap at write	E
mhartid	csrrs x1, x1, mhartid	Missing trap at write	E
mideleg	csrrw x1, x0, mideleg	VP traps at mideleg read	E*
medeleg	csrrwi x1, 0, medeleg	VP traps at medeleg read	E*
mip	csrrw x0, x0, mip	Trap at write access	E
mcycle	csrrw x0, x0, mcycle	Trap at write access	E
mcycle	csrrw x1, x0, mcycle	Cycle Count Mismatch	M
minstret	csrrw x2, x0, minstret	Trap at write access	E
minstret	csrrw x1, x0, minstret	Cycle Count Mismatch	M
mcycleh	csrrw x0, x0, mcycleh	Trap at write access	E
minstreth	csrrw x0, x0, minstreth	Trap at write access	E
cycle	csrrsi x1, 0	unimpl. Unprivileged CSR	M
cycleh	csrrsi x2, 0	unimpl. Unprivileged CSR	M
instret	csrrsi x0, 0, instret	unimpl. Unprivileged CSR	M
instreth	csrrsi x0, 0, instreth	unimpl. Unprivileged CSR	M
time	csrrsi x2, 0, time	unimpl. Unprivileged CSR	M
timeh	csrrsi x2, 0, timeh	unimpl. Unprivileged CSR	M
mhpmcounter3-31	csrrw x0, x0, mhpmcounter16	unimpl. Privileged CSR	M
mhpmcounter3-31h	csrrw x2, x0, mhpmcounter3h	unimpl. Privileged CSR	M
mhpmevent3-31	csrrw x3, x2, mhpmevent16	unimpl. Privileged CSR	M
mscratch	csrrw x1, x2, mscratch	unimpl. Privileged CSR	M
mcounteren	csrrwi x1, 0, mcounteren	unimpl. Privileged CSR	M

if a write attempt to a read-only CSR occurs. The RTL core does not raise these mandatory illegal instruction exceptions. All traps at any privilege level are default handled in machine mode. In order to increase performance, the core implementations can provide individual bits in the read-write CSRs medeleg (exceptions) and mideleg (interrupts) to delegate the handling to a lower privilege level. The ISS implementation is erroneous because it raises a trap at every read attempt of medeleg and mideleg. The *Machine Interrupt Register* mip and the *Machine Counters*: mcycle, minstret, mcycleh, minstreth have one thing in common: they can be written. The RTL core has the error that it raises a trap at every write access to the previously described CSRs. In order to enable performance monitoring, the RISC-V machine mode provides machine cycle counters. Examples of these CSRs are the mcycle CSR that counts the number of executed clock cycles, and the minstret CSR that counts the number of retired instructions. The ISS and the RTL core both implement these CSRs but have a deviating counting logic. This deviating logic is no error but only a mismatch because the detailed behavior is not specified (mismatches in the cycle count are actually expected due to the abstract, non-cycle accurate, timing model in the ISS). Other mismatches are the results of the fact, that the ISS implements much more performance counters than the RTL core. The ISS additionally implements the unprivileged counter: cycle, cycleh, instret, instreth, time, timeh, and the Machine Counter mhpmcounter3-31, mhpmcounter3-31h. Furthermore, the RTL core does not implement the Machine Counter Setup CSR mhpmevent3-31, Machine Scratch CSR mscratch, and the Machine Counter-Enable Register mcounteren. Due to the large degree of different valid implementation choices that the RISC-V ISA offers, it is important to have effective methods available that detect mismatches in order to support the designer in providing an exactly matching configuration of ISS and RTL core. In the following, we describe the second part of our evaluation.

B. Performance Evaluation with Injected Errors

In order to evaluate the performance of our verification approach we use a error-injection methodology. In particular, we have defined a set of 10 error (E0 to E9) that cover a broad range of different functionality in the RTL core and represent common errors that may occur during the implementation.

E0: The first injected error is a fault in the decoding behavior of the *logical left shift* RV32I instruction SLLI. The highest bits of the instruction encoding are 7-bits with the value 0. SLLIW is a RV64I instruction that has the same encoding as the RV32I instruction RV32I. The RV64I specification also contains a reserved instruction that has nearly the same encoding with only the difference, that the 7th highest bit has the value 1. As **E0** we injected a don't care bit in the decoding table of the instruction SLLI at the 7th highest bit. Thus, the processor decodes the reserved RV64I erroneously to SLLI.

E1: Similar to **E0**, we marked the same encoding bit as don't care in the *logical right shift* instruction RV32I SRLI.

E2: Also in the *logical right shift* instruction RV32I SRLI, we marked the same bit as don't care.

E3: The next injected error is a STUCK-at-0 fault at the lowest result bit in the arithmetic *add immediate* instruction ADDI.

E4: In the subtraction instruction *SUB* we injected a STUCK-at-0 fault at the highest result bit.

E5: The next fault is injected into the unconditional jump instruction *Jump And Link* (JAL). The injected error **F5** prevents that JAL does change the PC.

E6: RV32I provides multiple conditional branch operations. The injected error E6 changes the behavior of *Branch Not Equal* (BNE) into the behavior of *Branch Equal* (BEQ).

E7: The next fault flips the endianness of the *Load Byte Unsigned* (LBU) memory access instruction.

E8: The injected error F8 removes the sign extension from 8-bits to 32-bits in the memory instruction *Load Byte* (LB).

E9: The memory instruction *Load Word* (LW) loads a 32-bit value. With Fault E9, the LW instruction only loads the lower 16-bits from memory.

In our experiments, we use a runtime limit of 24 hours. The processor co-simulation is configured to support RV32I. In this experiments, we only used assumptions that block the generation of CSR instructions (which are not a part of RV32I) to filter the known implementation mismatches that were found in the first part of the case study.

Table II shows the results of our experiments in finding E0 to E9. The table allows the comparison between the symbolic execution runs for the respective injected errors and also allows the comparison of the respective runs with the instruction limit of one and the respective runs with the instruction limit of two. The first column of the table states the injected errors. The next five columns contain the results with an instruction limit of 1. The column result describes whether the injected error was found. The next column with the name *executed instruction* contains the counter of the executed instructions until the error was found. The column time contains the *time* in seconds until the error was found. The column with the name *partial paths* contains the number of partially completed explored symbolic execution paths by KLEE. This number contains the paths that KLEE could not complete. This can be the case because time or memory constraints were reached, or an assertion was triggered inside the voter that quit the test generation. The *paths* column contains the number of complete explored symbolic execution paths by KLEE. The following five columns contain the results for the same experiment but with an instruction limit of 2. It is striking that our symbolic cross-processor verification approach can find all injected errors very quickly, independent of the configuration. Please note, all times are given in seconds (s). The first configuration with the instruction limit of 1, executed between 2239208 and 12367140 instructions and needed between 54s and 3237s until it found the injected error. In median, 4237517 instructions were executed within a period of 543s and in total, this configuration executed 53283172 instructions within 9685s. The second configuration with the instruction limit of 2, executed between 2182018 and 33959660 instructions and needed between 65s and 21994s until it found the injected error. In median, 4560215 instruction were executed within a period of 790s and in total, this configuration executed 88712782 instructions within 37201s. As can be easily seen, the first configuration (instruction limit 1) was faster than the second configuration (instruction limit 2). Although both configurations found the errors very efficiently, it is likely that the instruction limit should be set as low as possible and only increased incrementally for processor verification.

In summary, we demonstrated that symbolic cross-level processor verification is an efficient approach for bug hunting. During the development process we have been able to find 10 errors in the well-tested RTL-core MicroRV32, 2 errors in the ISS, as well as 19 implementation mismatches between the RTL core and the ISS (as discussed in the first case study part in Section V-A). Our error-injection based performance evaluation in this section confirms the strong bug hunting capabilities of our approach.

TABLE II
INJECTED ERROR RESULTS

	Instruction Limit: 1					Instruction Limit: 2				
Error	Result	# Exec. Instr.	Time [s]	Partial Paths	Paths	Result	# Exec. Instr.	Time [s]	Partial Paths	Paths
E0	✓	2239482	54	63	0	✓	2182018	65	64	0
E1	✓	2239208	93	63	0	✓	2182192	66	63	0
E2	✓	2239208	55	63	0	✓	2182416	66	63	0
E3	✓	3122286	303	77	1	✓	2957146	400	98	0
E4	✓	3178044	311	78	0	✓	2568932	247	68	0
E5	✓	10754876	2960	72	55	✓	22948714	10527	2321	16
E6	✓	5296990	775	98	8	✓	6163284	1179	609	4
E7	✓	5912722	947	92	15	✓	6792846	1325	729	5
E8	✓	5933216	950	92	15	✓	6775574	1332	727	5
E9	✓	12367140	3237	60	72	✓	33959660	21994	2876	27
Sum:	10 ✓	53283172	9685	758	166	10 ✓	88712782	37201	7618	57
Median:		4237517	543	75	5		4560215	790	354	2

VI. Conclusion and Future Work

Our symbolic verification methodology revealed 10 bugs in the well tested RTL-core MicroRV32 and found 2 errors in the reference ISS. Moreover, our error-injection methodology further underlines the effective bug hunting capabilities of our approach. Thus, the results demonstrate the applicability of symbolic execution in full scale processor co-simulation. For future work we plan to focus on hybrid techniques combining symbolic execution with fuzzing to provide a scalable and comprehensive verification methodology. Moreover, we want to investigate the specification of induction-based symbolic constraints to enable a complete processor verification that is not relying on a bounded model checking methodology.

Acknowledgments: This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127, within the project VerSys under contract no. 01IW1900, and within the project ECXL no. 01IW22002.

References

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, “Genesys-pro: innovations in test program generation for functional processor verification,” *D&T*, pp. 84–93, 2004.
[2] B. Campbell and I. Stark, “Randomised testing of a microprocessor model using SMT-solver state generation,” in *Formal Methods for Industrial Critical Systems*, F. Lang and F. Flammini, Eds., 2014, pp. 185–199.
[3] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, “X-gen: a random test-case generator for systems and socs,” in *HLDVT*, 2002, pp. 145–150.
[4] Y. Katz, M. Rimon, and A. Ziv, “Generating instruction streams using abstract CSP,” in *DATE*, 2012, pp. 15–20.
[5] “Microsoft security development lifecycle,” https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2018.
[6] “Oss-fuzz - continuous fuzzing for open source software,” https://github.com/google/oss-fuzz, 2018.
[7] “libFuzzer - a library for coverage-guided fuzz testing,” https://llvm.org/docs/LibFuzzer.html, 2018.
[8] “american fuzzy lop,” http://lcamtuf.coredump.cx/afl/, 2018.
[9] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, “Testing CPU emulators,” in *ISSTA*, 2009, pp. 261–272.
[10] N. Bruns, V. Herdt, D. Große, and R. Drechsler, “Efficient cross-level processor verification using coverage-guided fuzzing,” in *GLSVLSI*, 2012.
[11] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, 2018.
[12] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, 2008, pp. 209–224.
[13] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: a platform for in-vivo multi-path analysis of software systems,” in *ASPLOS*, 2011, pp. 265–278.
[14] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, “SOK: (state of) the art of war: Offensive techniques in binary analysis,” in *IEEE S & P*, 2016, pp. 138–157.
[15] Y. Zhang, W. Feng, and M. Huang, “Automatic generation of high-coverage tests for rtl designs using software techniques and tools,” in *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, 2016, pp. 856–861.
[16] A. Ahmed, F. Farahmandi, and P. Mishra, “Directed test generation using concolic testing on rtl models,” in *DATE*, 2018, pp. 1538–1543.
[17] Y. Lyu and P. Mishra, “Scalable concolic testing of rtl models,” *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2021.
[18] L. Liu and S. Vasudevan, “Efficient validation input generation in rtl by hybridized source code analysis,” in *DATE*, 2011, pp. 1–6.
[19] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, “End-to-end automated exploit generation for validating the security of processor designs,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 815–827.
[20] R. Zhang and C. Sturton, “A recursive strategy for symbolic execution to find exploits in hardware designs,” in *Proceedings of the 2018 ACM SIGPLAN International Workshop on Formal Methods and Security*, 2018, p. 1–9.
[21] P. Bavonparadon and P. Chongstitvatana, “Rtl formal verification of embedded processors,” in *2002 IEEE International Conference on Industrial Technology, 2002. IEEE ICIT ’02.*, 2002, pp. 667–672 vol.1.
[22] S. Deng, W. Wu, and J. Bian, “Bounded model checking for rtl circuits based on algorithm abstraction refinement,” in *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*, 2006, pp. 2082–2084.
[23] A. Goel and K. Sakallah, “Model checking of verilog rtl using ic3 with syntax-guided abstraction,” in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds., 2019, pp. 166–185.
[24] “RISC-V formal verification framework,” https://github.com/YosysHQ/riscv-formal, 2020.
[25] “About risc-v,” 2022, accessed: 2022-07-14. [Online]. Available: https://riscv.org/about/
[26] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, 2019.
[27] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, 2019.
[28] “Welcome to spinalhdl’s documentation! — spinalhdl documentation,” 2022, accessed: 2022-07-14. [Online]. Available: https://spinalhdl.github.io/SpinalDoc-RTD/v1.3.1/index.html
[29] “Vexriscv,” 2018, accessed: 2022-07-14. [Online]. Available: https://github.com/SpinalHDL/VexRiscv
[30] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, “Mircorv32: an open source risc-v cross-level platform for education and research,” in *Proceedings of the Workshop on Design Automation for CPS and IoT*, 2021, pp. 30–35.
[31] “verilator,” 1994, accessed: 2022-07-14. [Online]. Available: https://www.veripool.org/verilator/
[32] “Tutorial one: Testing a small function,” 2014, accessed: 2022-07-14. [Online]. Available: https://klee.github.io/tutorials/testing-function/
[33] “Overview of the main klee intrinsic functions,” 2014, accessed: 2022-07-14. [Online]. Available: https://klee.github.io/docs/intrinsics/
[34] “Amba axi protocol specification version c: Write strobes,” 2003, accessed: 2022-07-14. [Online]. Available: https://developer.arm.com/documentation/ihi0022/c/Data-Buses/Write-strobes
[35] A. Cicuttin, “Introduction to the wishbone bus interface,” 2012, accessed: 2022-07-14. [Online]. Available: https://indico.ictp.it/event/a11204/session/35/contribution/22/material/0/0.pdf
[36] “Picorv32 - a size-optimized risc-v cpu,” 2019, accessed: 2022-07-14. [Online]. Available: https://github.com/YosysHQ/picorv32
[37] “RISC-V formal interface (rvfi),” https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md, 2020.
[38] V. Herdt, D. Große, H. M. Le, and R. Drechsler, “Extensible and configurable RISC-V based virtual prototype,” in *FDL*, 2018.
[39] “RISCV-DV,” https://github.com/google/riscv-dv, 2020.