Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study

Vladimir Herdt[1] Daniel Große[1,2] Eyck Jentzsch[3] Rolf Drechsler[1,4]
[1]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[2]Chair of Complex Systems, Johannes Kepler University Linz, Austria
[3]MINRES® Technologies GmbH, Munich, Germany
[4]Institute of Computer Science, University of Bremen, Bremen, Germany
Vladimir.Herdt@dfki.de, daniel.grosse@jku.at, eyck@minres.com, drechsle@informatik.uni-bremen.de

***Abstract*—Extensive processor verification at the *Register-Transfer Level* (RTL) is crucial to avoid bugs. Therefore, simulation-based approaches are prevalent but they require efficient test generation methods to achieve a thorough verification.**

In this paper we propose an efficient cross-level testing approach for processor verification targeting the RISC-V *Instruction Set Architecture* (ISA). We generate an endless instruction stream without restrictions on the generated instructions by evolving the instruction stream on-the-fly during simulation. An *Instruction Set Simulator* (ISS) is leveraged as reference model for the RTL core under test in a tightly coupled cross-level co-simulation setting. This enables a very efficient and comprehensive testing process. As a case-study we present results on the verification of the 32 bit pipelined RISC-V core of MINRES *The Good Folk (TGF) Series* Our approach has been very effective in finding several serious bugs.

***Index Terms*—RISC-V, Cross-Level, Processor Verification, Instruction Stream, Co-Simulation**

I. Introduction

RISC-V is an open and royalty-free *Instruction Set Architecture* (ISA) that gained enormous momentum in both academia and industry in recent years. The major goal of the RISC-V ISA is to provide a path to a new era of processor innovation via open standard collaboration. RISC-V features an extremely modular and extensible design that provides enormous flexibility in building application specific solutions that can leverage custom extensions and only include features that are really required. RISC-V became a game changer for embedded systems in several application areas including e.g. IoT and Edge devices. Thus, many emerging designs feature a RISC-V processor, which is at the heart of the design.

Extensive verification of the processor at the *Register-Transfer Level* (RTL) is crucial to avoid bugs, which could lead to longer design cycles and significant follow-up costs. Due to their ease of use and scalability, simulation-based methods are still prevalent in the verification domain and form the back-bone of the verification effort. However, they require efficient test generation methods to achieve a thorough verification.

Several approaches have been proposed for the purpose of instruction stream generation for processor verification. In particular model-based approaches, which separate the test generator from the architecture description, have a long history. Prominent examples using constraint solving techniques are [1], [2]. An optimized test generation framework has been presented in [3]. It propagates constraints among multiple instructions in an effective manner. The test program generator of [4] includes a coverage model that holds constraints describing execution paths of individual instructions. Alternative approaches integrate coverage-guided test generation based on bayesian networks [5] and other machine learning techniques [6] as well as fuzzing [7]. However, these approaches are either not designed for RTL verification or impose restrictions on the generated instruction streams. In addition, they do not target the RISC-V ISA.

In this paper we propose an efficient cross-level testing approach for processor verification at RTL targeting the RISC-V ISA. Our approach generates an endless instruction stream without restrictions on the generated instructions by evolving the instruction stream on-the-fly during simulation. An *Instruction Set Simulator* (ISS) is leveraged as reference model for the RTL core under test in a tightly coupled cross-level co-simulation setting. This enables a very efficient and comprehensive testing process. Our solution provides a testbench that feeds the generated instruction stream to the ISS and RTL core and compares the results after each executed instruction in order to detect errors in the RTL core immediately when they occur. As a case-study we present results on the verification of the 32 bit pipelined RISC-V core of MINRES *The Good Folk (TGF) Series*. Our approach has been very effective in finding several serious bugs in the industrial core. Moreover, our approach is very efficient with more than 200 million processed instructions per hour on a standard laptop.[1]

II. Related Work

We already mentioned related work on general methods to generate processor-level stimuli in the introduction. Here we focus on RISC-V specific solutions which have started to emerge recently.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and no. 16ME0135, and within the BMBF project VerSys under contract no. 01IW19001, and by the German Research Foundation (DFG) as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (http://www.ease-crc.org/) in subproject P04.

[1]Visit http://www.systemc-verification.org/risc-v for our most recent RISC-V related approaches.

First, the officially provided test-suites [8], [9] need to be mentioned. They are hand-written and aim to cover basic sanity checks and several corner-case scenarios with support for different RISC-V instruction set extensions. However, by being hand-written, their overall coverage is obviously very limited and they are not suitable for continuous testing.

A model-based test generation approach is pursued by RISC-V *Torture Test* [10]. It is a Scala-based framework that generates tests based on randomized instruction sequence templates and supports several RISC-V ISA extensions. [11] is another model-based approach that leverages a constraint-based specification for test generation. However, both approaches leverage pre-defined building blocks for instruction sequences which limits their coverage and they do not support illegal instructions or exceptions.

Another research direction considers coverage-guided fuzzing tailored for verification at the ISS level [12], [13]. They loosen some of the instruction stream generation restrictions but still have problems with branches and jumps to avoid non-terminating test-cases and problems with platform dependent CSR and memory access operations. In addition, the fuzzing result is a test-suite with a comparatively small number of test-cases (a few thousand).

RISC-V DV [14] by Google is another test generation approach that leverages SystemVerilog in combination with UVM (*Universal Verification Methodology*) to continuously generate RISC-V instruction streams based on constrained-random descriptions. Each instruction stream represents a test-case and *RISC-V DV* provides a high-level co-simulation interface to compare the results between different simulators via execution log files. *RISC-V DV* supports a large set of features including several RISC-V instruction set extensions and CSR testing capabilities. However, it has two major disadvantages: First, the generated instruction streams are restricted to avoid problems with infinite loops and platform dependent memory access operations. Second, *RISC-V DV* has a significant performance overhead because it is a generic framework that aims to support a large range of simulators (and RTL cores perspectively) and thus fully decoupled the test generation and co-simulation process. This makes the verification process significantly less efficient, because each test-case needs to be compiled and then loaded and executed on the reference simulator and test simulator (and then execution logs are compared to decide a mismatch). In addition, it is much more difficult to pass execution feedback to the test generation engine.

In contrast, our approach generates one endless instruction stream without any of these restrictions on the generated instructions by evolving the instruction stream on-the-fly during simulation. In addition, the tight integration of instruction stream generation and co-simulation environment can make our approach much more efficient performance-wise. It processes more than 200 million instructions per hour on a standard laptop (which typically cannot be achieved with a generate, compile, execute and compare loop).

Beside test generation methods, there are also a few formal verification approaches for RISC-V. Notable approaches that leverage model checking are *riscv-formal* [15] and the *OneSpin 360 DV* RISC-V verification app [16]. Though, *riscv-formal* has only limited support for the RISC-V privileged ISA and *OneSpin 360 DV* is only commercially available. Another direction is to formalize the RISC-V ISA semantics that is pursued by e.g. [17], [18]. Based on these formalization theorem prover can be utilized to reason about the RISC-V ISA semantics and generate simulation backends. While formal methods can provide correctness guarantees, they are significantly more difficult to apply than simulation-based methods and, due to their complexity and potential scalability issues, should be complemented by simulation-based methods.

III. Preliminaries

This section presents relevant background information on the RISC-V ISA as well as SystemC and TLM (*Transaction Level Modeling*) [19], [20]. Our co-simulation testbench is implemented in SystemC and uses TLM.

A. RISC-V

The RISC-V ISA consists of a mandatory base integer instruction set, denoted RV32I, RV64I or RV128I with corresponding register widths, and optional extensions denoted as single letters, e.g. M (integer multiplication and division), C (compressed instructions) etc. Thus, RV32I denotes a 32 bit core without any extensions. It has 32 general purpose registers x0 to x31 where x0 is hardwired to zero. Each register is 32 bit wide. Instructions are grouped into different classes (e.g. computational, load/store, branch/jump). They access registers (source: RS1 and RS2, destination: RD) and immediates to perform their operation. Immediates are available in different sizes and signed/unsigned interpretation. RV32I has five immediate types: I-, S-, B-, U- and J-type. For example, I-type is a signed 12 bit immediate, thus has a value range of [-2048...2047]. Format and semantics (for the base ISA and extensions) are defined in the unprivileged ISA specification [21].

In addition, the privileged (architecture) specification [22] covers further important functionality that is required for environment interaction, operating system execution and trap handling. It includes different execution modes (in particular the mandatory Machine mode) with corresponding *Control and Status Register* (CSR) descriptions. CSRs are registers serving a special purpose, that form the backbone of the privileged architecture description. Example CSRs are:

- MISA provides the supported instruction set.
- MTVEC stores the trap handler address and access configuration.
- MTVAL provides exception specific information in case of a trap.
- MEPC stores the return address from a trap for the MRET instruction.
- MINSTRET counts the number of retired instructions.
- MHARTID provides the read-only core id.
- MSTATUS is the main control and status register for the core.

CSRs can be read-only and consist of different fields. A field is part of the CSR (it has a start position and bitwidth) and has an access specification such as WARL (*Write Any Read*

```
 1 void Memory::operation(tlm_generic_payload &gp){
 2   auto len  = gp.get_data_length();
 3   auto addr = gp.get_address();
 4   uint8_t *ptr = reinterpret_cast<uint8_t*>(
         gp.get_data_ptr());
 5
 6   if (gp.is_read()) { // read access
 7     for (auto i=0; i<len; ++i)
 8       *(ptr+i) = read_byte(addr+i);
 9   } else { // write access
10     assert (gp.is_write());
11     for (auto i=0; i<len; ++i)
12       write_byte(addr+i, *(ptr+i));
13   }
14 }
```

Fig. 1. Example memory access operation using a TLM transaction

Legal). A WARL field can be written with any value but a read access will only return legal values. This allows SW to query the CSRs to obtain more information on the capabilities of the core. In contrast to the instruction set specifications, the CSR behavior is much less rigidly defined and often leaves many legal implementation choices which makes the testing process more challenging.

B. *SystemC and TLM*

SystemC in combination with TLM is an industry-proven modeling standard for building designs at different levels of abstraction. SystemC is not a new language, rather a C++ class library which includes an event-driven simulation kernel [19]. The structure of a SystemC design is described with modules, whereas the behavior is modeled in processes which are triggered by events. The execution of a process is non-preemptive, i.e. the simulation kernel receives the control back if the process has finished its execution or actively suspends itself. Communication can be implemented via signals (commonly used for RTL models) or abstracted using TLM transactions (commonly used for high-level algorithmic models). A transaction object essentially consists of a command (e.g. read/write), the data (payload) to be transmitted and the address.

Fig. 1 shows an example memory interface based on TLM. The memory receives a transaction object called *gp* (*tlm_generic_payload* type). Based on the TLM command either a data read (Line 7-8) or write (Line 11-12) operation is executed. The address, access length and data pointer is obtained from the transaction object (Line 2-4). The *read_byte* and *write_byte* functions read and write a single byte from the memory, respectively.

IV. Cross-Level Testing for Processor Verification

In this section we present our proposed cross-level testing approach for processor verification via on-the-fly endless instruction stream generation. We start in Section IV-A with an overview on the co-simulation testbench design that feeds the instruction stream to the ISS and RTL core. Then, we discuss relevant implementation challenges (Section IV-B) and present our instruction stream generator in more detail (Section IV-C).

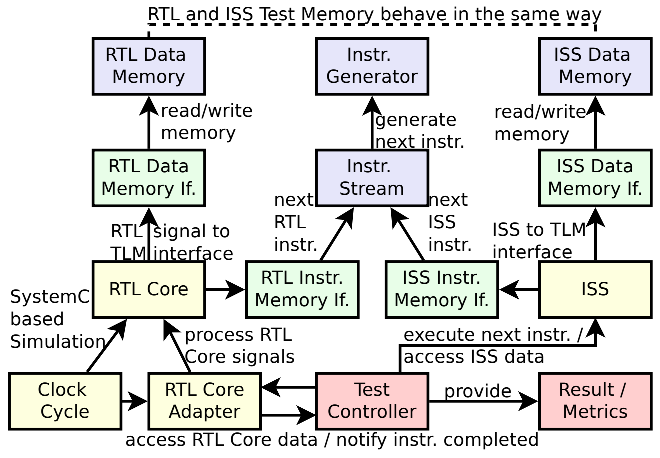

Fig. 2. Overview of our co-simulation testbench for processor verification

A. *Co-Simulation Testbench Overview*

Fig. 2 shows an overview of our co-simulation testbench design. It is implemented in SystemC and enables an efficient co-simulation between the RTL core (left side of Fig. 2) under test and the ISS (right side of Fig. 2) reference model.

The co-simulation is orchestrated by the test controller (bottom center of Fig. 2). Essentially, it repeats the following steps: First, the test controller lets the RTL core execute one instruction. Then, it lets the ISS execute the same instruction. Finally, the RTL core and ISS execution states (the registers in particular) are compared. In case of a mismatch in the execution states, an error is reported. The mismatch has to be analyzed and fixed accordingly. Otherwise (no mismatch), the co-simulation continues until the testing time is exhausted. This basic approach presents considerable challenges that need to be solved, which we discuss in more detail in Section IV-B. In the following we present more details on the co-simulation testbench.

The RTL core is driven by a clock signal. It has two separate memory interfaces to access the instruction and data memory, respectively. The memory interfaces translate back and forth between RTL core signals and TLM transactions. We leverage TLM transactions to have a unified memory abstraction for the RTL core and the ISS based on a common standard (recall Section III-B). The data memory is implemented to work in a lazy fashion. Initially it is empty. On a write access data is stored in the data memory. On a read access either the existing data is returned or new random data is generated (if no access at this address happened before). To match the RTL core, the ISS is also using two separate memory interfaces. Please note, the ISS and RTL data memories both use the same random seed and thus behave exactly in the same way because RTL core and ISS perform the same data memory access sequences (i.e. in the same order). Finally, we provide a *core adapter* to simplify the access to the RTL core. We provide more details on the core adapter in Section IV-B.

Instruction fetching is handled by the instruction memory interface based on the *Program Counter* (PC). An instruction fetch of the RTL core results in the generation of a new instruction (always, even if this PC has been fetched already), i.e.

on-the-fly during the simulation. An instruction fetch of the ISS receives the corresponding fetched instruction of the RTL core. This matching is handled by the instruction stream (top center of Fig. 2) which is placed between the instruction generator and the respective memory interfaces. We provide more details on instruction matching in Section IV-B. Please note, our approach generates an endless instruction stream without restrictions on the generated instructions. Thus, all memory access instructions (because we wrap the complete address range of the data memory interface) and jump instructions (including self-loops due to our on-the-fly instruction generation) as well as special RISC-V CSR access instructions are supported. This enables a very comprehensive testing. Independent of the generated instructions, ISS and RTL core should behave completely identical on the observable architectural state (i.e. register updates).

B. Implementation Challenges

There are two main challenges that need to be solved in order to implement our proposed approach: 1) it can be difficult to detect when an instruction is completed in the RTL core, and 2) feeding the same instruction stream into the RTL core and ISS requires special attention. We discuss both points and our solutions in the following.

1) Detecting Completed Instructions: The (industrial pipelined) RTL core does not provide a single signal that can be queried to detect that an instruction has been completed. In particular, illegal instruction can bypass several stages of the pipeline (depending where they identified as illegal) and do not trigger any regular register write back notifications. Furthermore, it is not possible to directly consider an illegal instruction completed the moment it is detected in the pipeline, because there may still be legal instructions pending in the pipeline ahead which need to be completed first (to preserve the instruction order). In addition, the pipeline can get flushed (due to jumps and traps) as well as get stuck at different stages (some operations such as shifting can take multiple cycles) and thus cause delays and gaps, which need to be considered as well. Thus, a deep understanding of the pipeline is required to detect when an instruction has been completed.

Therefore, we provide a core adapter to hide the implementation details of the core and provide a clean testing interface. The core adapter observes the internal signal changes of the core (in particular the pipeline) and notifies the test controller each time the RTL core completed one instruction (and also preserves the correct order in case of illegal instructions). In addition, the core adapter provides access to the register values of the RTL core to compare them with the ISS.

2) Instruction Stream Matching: The primary goal of our testing approach is to generate an endless and unrestricted instruction stream. However, this makes it more difficult to feed the same instructions to the RTL core and ISS. The reason is that the RTL core pre-fetches several instructions due to the pipeline. However, those pre-fetched instructions may not be executed in case of a jump or trap. In this case, the ISS will fetch a different sequence of PCs than the RTL core. Furthermore, short jumps (which can also be caused by traps) can cause a new instruction fetch in the RTL core before the ISS had the opportunity to fetch

```
1  function InstrStream::next_RTL_instr(PC)
2      // always generate a new instruction
3      i ← InstrGenerator::next()
4      pending_instrs_queue::push((PC, i))
5      return i // return this new instruction

6  function InstrStream::next_ISS_instr(PC, expected_instr)
7      // search for a matching instruction
8      while not pending_instrs_queue::empty() do
9          (iPC, i) ← pending_instrs_queue::pop()
10         if iPC = PC and i == expected_instr then
11             return i // match found, return it
12     report_mismatch() // no match, something wrong
```

Fig. 3. Instruction fetch matching between the RTL core and ISS

the previous instruction (for the same PC). Thus, a direct matching based on the PC does not work. For example consider a *one instruction backward jump* J from address 8 to address 4. Thus, the RTL core executes J and starts pre-fetching from address 4, before J is fully completed. Therefore, a new instruction is fetched (and thus generated on-the-fly) for address 8 before the ISS would had the opportunity to fetch and execute J.

Fig. 3 shows our algorithm to solve the above instruction matching problem. In the instruction stream, we keep a queue of pending instructions (in fetch order) that have been fetched by the RTL core but not yet picked up by the ISS (Line 4). Please note, beside the generated instruction (Line 3) we also store the PC in the queue in Line 4. In addition, we leverage the core adapter to extract the last completed instruction from the RTL core (by carefully analyzing the pipeline signals). We pass this last completed RTL core instruction alongside the ISS PC to fetch the next ISS instruction (Line 6). Based on this arguments we perform a matching with the queue of pending instructions (Line 10). In case of a match the instruction is returned (Line 11). Otherwise, a mismatch is reported between RTL core and ISS (because the ISS tried fetching an instruction which was not delivered to the RTL core) in Line 12. Please note, we do not directly feed the completed instruction sequence from the core adapter to the ISS, because this would compromise the testing approach since we would then rely that the instruction propagation in the RTL core works correctly (and the RTL core is under test).

C. Instruction Stream Generator

Our carefully designed co-simulation setup enables endless generation of unrestricted instructions. Thus, the baseline generation algorithm simply fully randomizes the generated instructions. It forms the foundation of the testing process. In addition, we consider several modifications to guide the test generation towards interesting cases.

The first modification is to inject a random instruction opcode to create a valid instruction but keep the instruction fields randomized. This modification is very simple but at the same time very generic and effective. It is also extremely important to ensure that a large set of legal instructions is considered

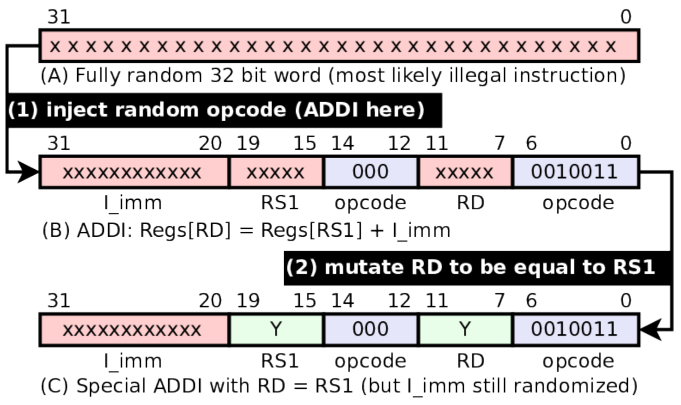
31
0
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
(A) Fully random 32 bit word (most likely illegal instruction)
(1) inject random opcode (ADDI here)
31 20 19 15 14 12 11 7 6 0
xxxxxxxxxxxx
xxxxx
000
xxxxx
0010011
I_imm
RS1
opcode
RD
opcode
(B) ADDI: Regs[RD] = Regs[RS1] + I_imm
(2) mutate RD to be equal to RS1
31 20 19 15 14 12 11 7 6 0
xxxxxxxxxxxx
Y
000
Y
0010011
I_imm
RS1
opcode
RD
opcode
(C) Special ADDI with RD = RS1 (but I_imm still randomized)

Fig. 4. Injection and mutation rule example for illustration

(because pure randomization tends to generate illegal instruction due to the significantly larger state space of illegal instructions). Fig. 4 shows an example. Starting with the fully randomized 32 bit instruction (A), the ADDI opcode is injected resulting in a randomized ADDI instruction (B), by operation (1).

The second modification is to mutate the instruction fields based on a pre-defined rule. We provide a set of rules that reason about the structure and values of the instructions. The rules are derived based on the RISC-V instruction format. We provide rules to inject special values, such as {MIN, -1, 0, 1, MAX}, into the respective immediate field. Other rules reason about the register structure, e.g. mutate RD to zero (since the zero register is hardwired in RISC-V and thus a special case), mutate RD to be equal to RS1 and/or RS2, and mutate RS1 to match RS2. And we provide a rule to mutate the CSR selector field to a supported CSR. Fig. 4 again shows an example. Starting with the randomized ADDI instruction (B) the RD field is mutated to match the RS1 field (C), by mutation (2). Both register fields are still randomized but equal (denoted as Y in Fig. 4).

As a third modification, we consider generation of instruction sequences. A sequence consists of a fixed number of instructions that are designed to perform a specific task and can be randomized. For example, two RISC-V instructions that in combination can load a large immediate value into a RISC-V register (the immediate field of a single instruction is not large enough to load an arbitrary register value). The target register and load value are randomized. Another example is a compute chain, that feeds the result of one instruction into the source register of the next instruction but randomizes the operation (e.g. ADD, SUB, etc) and operand registers. One more useful sequence is a CSR access sequence. It performs a randomized CSR access and then writes the CSR value into a normal register (so it can be compared with the ISS register).

Fig. 5 shows the algorithm that we use for instruction generation. If a sequence is active and not yet completed (Line 3), then the next instruction in the sequence is returned (Line 5). With a 1% probability a new sequence is randomly selected and started (Line 6-9). Starting a sequence does randomize it's instructions and returns the first instruction. Otherwise (no active sequence and no new sequence started), a single independent instruction is generated (Line 11-17) and returned (Line 18). We start with a

```
1  function InstrGenerator::next()
2      // sequence is an InstrGenerator class variable
3      if sequence $\neq$ nil and sequence.has_next() then
4          // continue with existing sequence
5          return sequence.next()      // return next instr.
6      if Random::probability(1) then      // enter with 1%
7          // start a new sequence
8          sequence $\leftarrow$ choose_random_sequence_generator()
9          return sequence.start()      // return first instr.
10     // generate a random 32 bit word (instruction)
11     x $\leftarrow$ Random::instruction()
12     if Random::probability(98) then      // enter with 98%
13         // choose any opcode, keep fields random
14         x $\leftarrow$ inject_random_valid_opcode(x)
15     if Random::probability(20) then      // enter with 20%
16         // apply a mutation rule to the fields
17         x $\leftarrow$ apply_random_field_mutation(x)
18     return x      // a single independent instruction
```

Fig. 5. Instruction generation algorithm

fully randomized instruction (Line 11). With a high probability (98%) a random opcode is injected (Line 12-14). In addition, with a smaller probability (20%) a random field mutation is applied (Line 15-17).

V. Experimental Evaluation

We have implemented our proposed cross-level testing approach and applied it for the verification of the pipelined 32 bit industrial RISC-V TGF series core. The core has been implemented in SpinalHDL. It is designed to be highly configurable on the microarchitectural level, such as choosing the shifter implementation and pipeline levels. For this evaluation we use the standard configuration that is available to customers. We obtained the Verilog RTL implementation from SpinalHDL (an option for this use-case is provided) and then applied the *Verilator* tool to obtain the C++ description of the core which we embedded into our SystemC-based co-simulation testbench. As ISS reference model, we use the 32 bit RISC-V ISS of the open source RISC-V VP [23], [24]. We have modified the ISS to exactly match the capabilities of the RTL core (i.e. the supported RISC-V instruction set and CSRs). The RTL core supports the RV32I ISA in combination with the machine mode CSRs. All experiments have been performed on a Linux system with an Intel Core i5-7200U processor.

For the verification process, we iteratively switched between testing and bug fixing until no more bugs were found. In the following we first present and discuss the bugs that we have found and then present performance and execution metrics that we have obtained.

A. Found Bugs

Our testing process revealed that the RTL core already had a very mature implementation of the RISC-V unprivileged ISA.

Only very few special cases have triggered a mismatch with the ISS. Most bugs were related to the RISC-V privileged ISA, in particular the CSR handling. In total we found 10 bugs in the RTL core, which we discuss in the following:

1) Write access to a read-only CSR does not cause an illegal instruction trap. In addition, for specific CSRs and options, a legal write access to a (non read-only) CSR caused an exception.
2) MEPC is not updated correctly on the lower two bits. This allows SW to write an unaligned address into MEPC which can cause an unaligned jump.
3) MISA was not correctly initialized and could be updated by the SW to unsupported values.
4) MTVAL should be set to zero on an ECALL (instead it has been set to the ECALL instruction encoding, which is the default behavior for illegal instructions to help diagnose them).
5) SW can write a reserved value into the MODE field of MTVEC, which should not be allowed since MODE should only be able to hold supported values. This can cause a serious problem with forward compatibility of SW, because (due to the modular and extensible design of RISC-V) SW can query CSRs to obtain their capabilities (and would be misled in this case).
6) EBREAK instruction sets MCAUSE to illegal instruction instead of breakpoint.
7) The FENCE and FENCE_I instructions cause an illegal instruction trap for specific options. The problem has been in the decoder implementation.
8) Writing to the MINSTRET and MCYCLE CSRs erroneously caused an illegal instruction trap (though, according to the specification, this special counter CSRs are allowed to be modified by SW).
9) MINSTRET (which counts the number of retired instructions) is not correctly updated on a write access. In this case it should avoid the increment for the instruction that performs the write access.
10) MRET continues at the wrong instruction for some special instruction sequences that involve multiple MRET and illegal instructions. MRET is a special RISC-V instruction to return from the trap handler. Thus, it is used in a very regular way by SW. In contrast, our approach allows to comprehensively stress test the MRET instruction (and others) and hence is very effective in finding errors.

In contrast to the existing testing frameworks for RISC-V, which impose several restrictions on the generated instructions (and thus simply cannot generate specific instruction sequences) our approach avoids these restrictions by evolving the instruction stream on-the-fly during simulation. This is a very important advantage, because many corner-case bugs will only be revealed by very specific instruction sequences with highly unregular control-flow, including tight loops and traps (as for example bug 10 demonstrates).

Beside the 10 bugs in the RTL core, our testing process also revealed 1 bug in the reference ISS, where MTVAL was set incorrectly. The bug is triggered by executing a compressed instruction (which is considered an illegal instruction because the C extension is deactivated) and then reading MTVAL. The reason is that the ISS still expanded the fetched compressed instruction (16 bit) into an uncompressed instruction (32 bit), even though the C extension was deactivated. Thus, instead of the original fetched instruction, the expanded instruction is erroneously written into MTVAL on the illegal instruction trap.

All of the described bugs have been found in less than 5 minutes each. Thus, our approach has been very effective in finding bugs. In the following we present more details on the performance characteristic and other execution metrics.

B. *Performance and Execution Metrics*

The lightweight test-generation process and tight co-simulation between the ISS and RTL core enable our approach to achieve a very high performance. In one hour it generated and co-simulated a total number of 226 million (M) instructions. These total instructions are separated into 12M illegal and 214M legal instructions. From the legal instructions 156M completed normally and 58M caused an exception (i.e. trap). For illustration, Fig. 6 shows how the legal instructions are distributed. It can be observed that they are mostly uniformly distributed, ranging from 6.0M for ADDI and 3.6M for MRET (please note, the y-axis scale starts at 3.0M). The distribution difference are due to the randomness of the generation process and the inclusion of special instruction sequences. For example loading a RISC-V register with an arbitrary number requires two instructions, an ADDI and a LUI which is also reflected in Fig. 6. On average 63K (K = thousand) instructions and 229K (RTL core) cycles are processed per second. This high performance enables a very efficient testing process.

Looking more closely at the instructions we observed between 11M to 22M accesses per register with an average of 12M. Due to the special semantic of the hardwired x0 register in RISC-V, we used generation rules that favor the x0 register (thus it is accessed more often compared to other registers). We observed between 1 (because register x0 is hardwired to zero) and 870K different values per register with an average of 747K. On the immediate fields we observed 5M to 51M accesses with an average of 22M. The amount of observed values in the immediate fields varies largely from 32 to 1M due to the different value ranges of the immediates. In total we observed 99.6% of the possible immediate values (across all instructions in combination). Thus, beside the high performance, our approach also enables a broad coverage. In combination with support for unrestricted instruction sequences (to cover highly irregular control flows) our approach is very suitable for extensive stress testing.

Finally, please note that our on-the-fly instruction stream generation approach is very generic and thus not limited to a specific RISC-V ISA configuration. We expect that only minimal extensions are necessary to provide efficient support for additional RISC-V ISA extensions (covering the privileged as well as unprivileged ISA).

VI. Conclusion and Future Work

We proposed an efficient cross-level testing approach for processor verification targeting the RISC-V ISA. It works by generating and feeding an endless instruction stream into the RTL core under test and a reference ISS in a tightly coupled

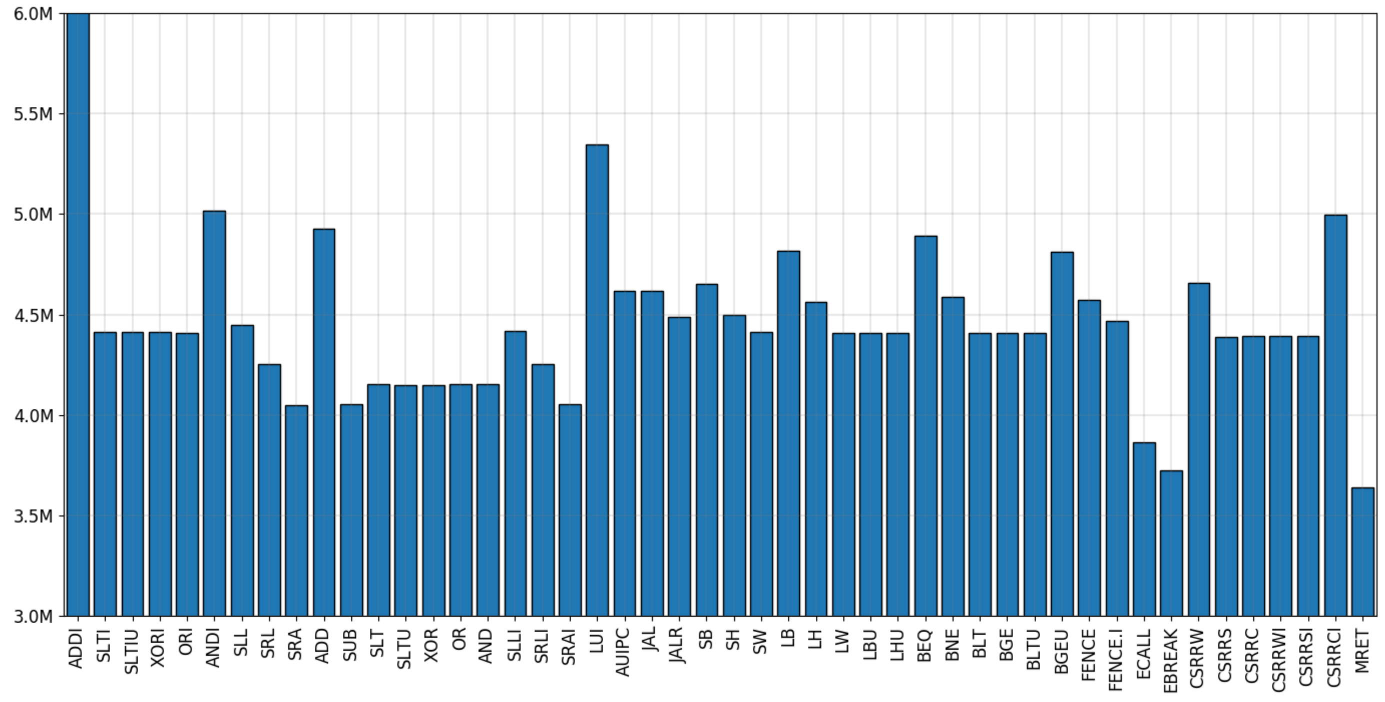

Fig. 6. Distribution on the executed legal instructions for a 1 hour testing process. The X-axis shows the instructions and the Y-axis the count (M = Millions).

co-simulation setting. The instruction stream evolves on-the-fly during simulation and thus avoids restrictions on the generated instructions. Our approach has been very effective in finding several serious bugs in the pipelined industrial RISC-V TGF series core and worked very efficiently with more than 200 million processed instructions per hour. For future work we plan to:

- Investigate parallelized test sessions (using different random seeds) and utilizing FPGAs to further boost the testing process.
- Consider testing the interrupt interface of the RTL core which is quite challenging as it needs to be synchronized with the instruction stream co-simulation (to avoid spurious mismatches between ISS and RTL core).
- Extend and evaluate our approach on additional RISC-V ISA extensions. As already mentioned, we believe that our approach is very well prepared for this task due to the generic on-the-fly instruction stream generation.
- Investigate new coverage metrics that also consider RTL specific coverage and develop execution feedback mechanisms to further guide the test generation process.

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] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *DATE*, 2012, pp. 15–20.

[4] M. Chupilko, A. Kamkin, A. Kotsynyak, and A. Tatarnikov, "MicroTESK: specification-based tool for constructing test program generators," in *HVC*, 2017.

[5] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.

[6] C. Ioannides, G. Barrett, and K. Eder, "Feedback-based coverage directed test generation: An industrial evaluation," in *Hardware and Software: Verification and Testing*, S. Barner, I. Harris, D. Kroening, and O. Raz, Eds., 2011.

[7] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *ISSTA*, 2009, pp. 261–272.

[8] "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[9] "RISC-V compliance task group," https://github.com/riscv/riscv-compliance.

[10] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.

[11] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of RISC-V ISA compliance," in *DATE*, 2020.

[12] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.

[13] V. Herdt, D. Große, and R. Drechsler, "Closing the RISC-V compliance gap: Looking from the negative testing side," in *DAC*, 2020.

[14] "RISCV-DV," https://github.com/google/riscv-dv.

[15] "RISC-V formal verification framework," https://github.com/SymbioticEDA/riscv-formal.

[16] "OneSpin 360 DV RISC-V Verification App," https://www.onespin.com/solutions/risc-v.

[17] "Formal specification of RISC-V ISA in kami," https://github.com/sifive/RiscvSpecFormal.

[18] "Riscv sail model," https://github.com/rems-project/sail-riscv.

[19] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

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

[21] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[22] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[23] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[24] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, 2020.