Towards Specification and Testing of RISC-V ISA Compliance*

Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany

[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

{vherdt,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Compliance testing for RISC-V is very important. Therefore, an official hand-written compliance test-suite is being actively developed. However, this requires significant manual effort in particular to achieve a high test coverage.**

In this paper we propose a test-suite specification mechanism in combination with a first set of instruction constraints and coverage requirements for the base RISC-V ISA. In addition, we present an automated method to generate a test-suite that satisfies the specification. Our evaluation demonstrates the effectiveness and potential of our method.

I. Introduction

RISC-V [1], [2] is an open and free *Instruction Set Architecture* (ISA) that gained enormous momentum in both academia and industry in recent years. 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. However, this enormous flexibility also leads to a significantly increased risk of introducing SW incompatibilities between different RISC-V implementations, thus causing fragmentation of the RISC-V ecosystem.

This very important problem is addressed with compliance testing. In contrast to verification, which attempts to prove that an implementation is correct, compliance testing attempts to show that an implementation meets the standard and thus ensures compatibility with the RISC-V ecosystem. It is not yet fully clear how to solve the RISC-V compliance testing problem and very intensive discussions are taking place on how to proceed in order to develop suitable tools, models, and methodologies [3]. For this reason too, a dedicated *RISC-V Foundation Compliance Task Group* has been founded to address the compliance testing problem. The task group is currently actively developing a hand-written compliance test-suite [4]. Fig. 1 (right side) shows how the test-suite is used to test compliance. The generated test-suite is executed once on a reference simulator to generate reference outputs (called *signatures*) for each test-case (test-cases store instruction results in a predefined memory area which is dumped by the simulator). The combination of test-suite with the generated signatures is then used to test other RISC-V simulators / cores by comparing the output signatures. A separate test-suite is developed for the RISC-V base ISA as well as for each standard ISA extensions. However, it requires significant manual effort to create, adapt and evolve these test-suites. Also, it is very difficult to achieve high coverage results.

Contribution: In this paper we propose a specification mechanism to contribute to the goal of enabling automated generation of high-quality compliance test-suites. Fig. 1 (left side) shows an overview.

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001 and within the project SATiSFy under contract no. 16KIS0821K and within the project CONFIRM under contract no. 16ES0565, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

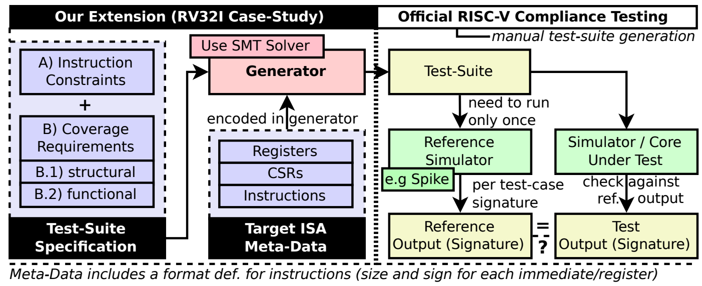

Fig. 1. RISC-V compliance testing (right side) and our contribution (left side)

Starting point is a specification that essentially consists of two parts: A) a set of *instruction constraints* that describe valid instructions, and B) a set of *coverage requirements* that the final test-suite should satisfy. The specification is passed to a generator that leverages an SMT solver to automatically generate a test-suite that satisfies all constraints and coverage requirements. The generation process is supported by target ISA specific meta-data that is encoded in the generator. This essentially includes information about the available registers and instructions as well as the instruction format.

As a case-study, we created a first specification with constraints and coverage requirements tailored for the base RISC-V ISA (RV32I). Our evaluation demonstrates that our method is effective and can further improve the quality of the existing compliance test-suite while significantly reducing the generation effort.[1]

Related Work: Several approaches have been proposed to improve generation of processor-level stimuli for the purpose of verification, which is complementary but related to compliance testing. Model-based test generators use an input format specification to guide the generation process and can integrate constraints processed by CSP/SMT solver [5]–[7]. In [8] an optimized test generation framework is presented which propagates constraints among multiple instructions in an effective way. [9] proposed to mine processor manuals to obtain an input model automatically. Other notable approaches include coverage-guided test generation based on bayesian networks [10] and other machine learning techniques [11] as well as generation [12] and mutation based fuzzing [13].

Finally, OneSpin offers verification solutions for RISC-V that enable to perform a complete formal proof of an RTL implementation against an ISA specification and thus can also guarantee compliance to the ISA [14]. However, their property sets (specification) and verification methods are proprietary and thus not freely available.

II. Background on RISC-V

The RISC-V ISA consists of a mandatory base integer instruction, denoted RV32I, RV64I or RV128I with corresponding register widths, and optional extensions denoted as single letters, e.g. M

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

```
1  /* ==[A) general constraints that need
          to be satisfied by all
          instructions]== */
2  @forall(I) { // forall instructions I
3   // 1.1: load/store are aligned
4   I.opcode in [LW] =>
          (@value(I.RS1)+I.I_imm) % 4 == 0
5   I.opcode in [LH,LHU] =>
          (@value(I.RS1)+I.I_imm) % 2 == 0
6   //...SW, SH similar...
7   // 1.2: no self-loop for branch/jump
8   I.opcode in [JALR] =>
          (@value(I.RS1) + I.I_imm) != 0
9   I.opcode in [BEQ, BNE, BLT, BGE, BLTU,
          BGEU] => I.B_imm != 0
10  I.opcode in [JAL] => {
11   I.J_imm >= -1024 // stays in valid
12   I.J_imm <= 1024  // instr. range
13   I.J_imm != 0     // no self-loop
14  }
15  // 1.3: valid address/jump range
16  I.opcode in [JALR,LB,LH,LBU,LHU,LW,
          SB,SH,SW] => {
17   @value(I.RS1) >= -2048
18   @value(I.RS1) <= 2048
19   I.RS1 != x0 // RS1 not hardwired
20  }
21  // 1.4: which CSRs to use
22  I.csr in [MSCRATCH,MHARTID,...,FRM]
23 }
24 /*==[B.1) structural requirements]==*/
25 @foreach(Op, Opcodes) { //every opcode
26  @exists(I) { // exists an instruction
27   I.opcode == Op // with that opcode
28  }
29 }
30 @foreach(Op, [ADD, SUB, SLL, SLT, SLTU,
          XOR, SRL, SRA, OR, AND]) {
31  @foreach(R, Registers) {
32   @exists(I) {//register combinations
33    I.opcode==Op && I.RD==R && I.RS1==R
          && I.RS2==R
34    I.opcode==Op && I.RD==R && I.RS1==R
          && I.RS2!=R
35    I.opcode==Op && I.RD==R && I.RS1!=R
          && I.RS2==R
36    I.opcode==Op && I.RD==R && I.RS1!=R
          && I.RS2!=R
37    I.opcode==Op && I.RD!=R && I.RS1==R
          && I.RS2==R
38    I.opcode==Op && I.RD!=R && I.RS1!=R
          && I.RS2==R
39    I.opcode==Op && I.RD!=R && I.RS1==R
          && I.RS2!=R
40   }
41  }
42 }
43 @foreach(Op, [LB, LBU, LH, LHU, LW]) {
44  // RS1 should not be hardwired zero
45  @foreach(R, Registers - [x0]) {
46   @exists(I) {
47    I.opcode==Op && I.RD==R && I.RS1==R
48    I.opcode==Op && I.RD==R && I.RS1!=R
49   }
50  } // hence the following special case
51  @exists(I) {I.opcode==Op && I.RD==x0}
52 }
53 //... other instructions similar ...
54 /*==[B.2) functional (value)
          requirements]==*/
55 I_imm    :: {MIN, MAX, -1, 0, 1}
56 S_imm    :: I_imm // use I_imm values
57 B_imm    :: {-2048, 2046, -2, 2, -1, 1}
58 U_imm    :: {0, 1,524287,524288,MAX}
59 J_imm    :: {-256, 256, -2, 2}
60 RS1      :: {MIN, MAX, -1, 0, 1}//value
61 RS2      :: RS1 //of register, not index
62 shamt    :: {0, ..., 31}
63 csr_uimm:: {0, ..., 31}
64 csr      :: {MSCRATCH,MHARTID,...,FRM}
65 @cross(RS1, RS2)       // can also cross
66 @cross(RS1, shamt)     // more than two
67 @cross(csr_uimm, csr) // fields
68 JALR {//overwrite toplevel requirements
69  I_imm :: {-2044, 2044, -4, 0, 4}
70  RS1 :: I_imm
71 }
72 // ensure aligned memory access
73 LH, LHU { I_imm :: {-2048, 2046, -2, 0,
          2} }
74 LW { I_imm :: {-2048, 2044, -4, 0, 4} }
75 LB, LBU, LH, LHU, LW { RS1 :: I_imm }
76 // ...store instrs. simular to load...
77 ADDI, SLTI, SLTIU, XORI, ORI, ANDI {
78  RS1 += {0x0000ABCD, 0x12345678,
          0xFEDCBA98, 0x36925814} // magic
          values from compliance suite
79  I_imm += {1024, -1024} // add values
80  @cross(RS1, I_imm)
81 }
82 SLL, SRL, SRA { RS2 += shamt }
83 CSRRW, CSRRS, CSRRC {
84  RS1 += {1<<0, 1<<1, ..., 1<<31}
85  @cross(RS1, csr)
86 }
```

Fig. 2. Specification of instruction constraints (A) and coverage requirements - structural (B.1) and functional (B.2) - for the RV32I ISA

(integer multiplication and division), F (single-precision floating point) etc. Thus, RV32I denotes a 32 bit core without any extensions. It has 32 register (x0 to x31) each with 32 bit witdh, where x0 is hardwired to zero. 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. We refer to them as I_,S_,B_,U_ and J_imm, respectively. For example, I_imm is a signed 12 bit immediate, thus has a value range of [-2048,...,2047]. Finally, RISC-V defines CSRs (*Control and Status Register*) and special instructions to access them. CSRs are registers serving a special purpose, e.g. MTVEC (the interrupt/trap handler address) and MHARTID (the read-only core id). For more details, please refer to the official RISC-V ISA specification volume one [1] (unprivileged ISA which includes RV32I) and two [2] (CSRs and privileged architecture).

III. Test-Suite Specification

We introduce our test-suite specification mechanism in an example driven way. Fig. 2 shows the (almost complete) test-suite specification file that we used in this work for the RV32I ISA. Essentially, it consists of two parts: A) instruction constraints (Line 1-23), and B) coverage requirements Line 24-86. Coverage requirements are further conceptually separated into structural (B.1, Line 24-53) and functional (B.2, Line 54-86) requirements. We present more details in the following.

A. Instruction Constraints

Constraints need to be satisfied by all instructions I. They are specified using the *@forall* construct and are seperated by newlines. It is possible to use the well-known unary and binary operators using C/C++ syntax. In addition, *X in [A, B]* is a short form for *X == A || X == B*. *=>* denotes an implication. *@value(X)* returns the value of register *X*. A semicolon is optional at the end of line. Multiple constraints can be combined at the right hand side of an implication, i.e. *C => { A; B }* equals to *C => (A && B)*.

Essentially, the constraints shown in Fig. 2 ensure that load-/store instruction do not generate invalid memory accesses and branch/jump instructions only perform local relative jumps to stay within a valid instruction address range. The first constraint set specifies that the memory access of load and store instructions is properly aligned (Line 4-6). A word (LW) and half-word (LH,LHU) access (in the general case) requires 4 and 2 byte alignment, respectively. The second constraint set (Line 8-14) ensures that no self-loops are generated (which would cause non-termination of the testing process) by constraining the jump offsets to non-zero. In addition, the jump offset for the JAL instruction is constrained to only perform a short (relative) jump (to not leave the instruction address range). The third constraint set (Line 16-20) ensures that only local addresses are used by load/store operations and the register based jump JALR. The final absolute address is provided by our generator by relocating the value of RS1 (hence RS1 cannot be the hardwired zero register x0). Finally, the fourth constraint set specifies which CSRs can be used (Line 22). We use CSRs that can be written almost freely by the SW (MSCRATCH, MEPC, MCAUSE, MTVAL) to check the effects of read/write operations as well as a read-only CSR (MHARTID) to enable checking that write accesses are correctly rejected and floating point CSRs (FFLAGS, FRM, FCSR) to check that access is not permitted without activating the F ISA extension.

B. Coverage Requirements

Coverage requirements are specified using the *@exists* construct. In addition, the *@foreach* construct is used to enumerate multiple *@exists* blocks. We use a short form to specify multiple *@exists* blocks, i.e. *@exists* { *A; B* } is expanded to *@exists* { *A* } and *@exists* { *B* }. Semicolons or newlines are used as delimiter.

We start with description of structural coverage requirements (B.1, Line 24-53). The first requirement in Line 25-29 specifies that every instruction of the ISA is executed at least once. The value of *Opcodes* and *Registers* is resolved in the generator based on the

```
1 //A) register-immediate
2 LI {RS1}, {RS1-value}
3 {OP} {RD},{rs1},{I_imm}
4 LA {RX}, output_begin
5 SW {RD}, 0({RX})
6 //////////////////////
7 //B) load operation
8 LA {RX}, input_middle
9 LI {RS1}, {RS1-value}
10 ADD {RS1}, {RS1}, {RX}
11 {OP} {RD},{I_imm}({RS1})
12 LA {RX}, output_begin
13 SW {RD}, 0({RX})
14 //////////////////////
15 //C) forward jump
16 LI {RX}, 1
17 {OP} {RD}, target
18 #...ADDI {RX},{RX},1...
19 target:
20 LA {RY}, output_begin
21 SW {RX}, 0({RY})
22 SW {RD}, 4({RY})
23 //////////////////////
24 //D) backward branch
25 LI {RX}, 1
26 LI {RS1}, {RS1-value}
27 LI {RS2}, {RS2-value}
28 J branch_begin
29 target:
30 J branch_end
31 #...ADDI {RX},{RX},1...
32 branch_begin:
33 {OP} {RS1},{RS2},target
34 ADDI {RX}, {RX}, 1
35 branch_end:
36 LA {RS1}, output_begin
37 SW {RX}, 0({RS1})
```

Fig. 3. Test-case body generation templates for four different instruction classes. Placeholder are displayed in curly braces and will be replaced with the actual instruction data. {OP} is the actual instruction (opcode) under test (e.g. ADDI,LW,JAL,BEQ).

provided meta-data and selected target ISA. The next requirements specify different structural register access pattern for all instruction classes. Line 30-42 considers computational instructions with two source (RS1, RS2) and one destination register (RD). Line 43-52 considers load instructions. Each load instruction has one source (RS1, memory address) and one destination register (RD, for the loaded data). We introduced a special case to ensure that RS1 is not hardwired to zero (x0), so that our generator can relocate RS1 to the final absolute address. We handle the other instruction classes conceptually similarly (not shown in Fig. 2).

The last part of Fig. 2 shows functional coverage requirements (B.2) that reason about the values of immediates and registers (Line 54-86). The short form X :: $\{v_1, ..., v_N\}$ specifies that for each instruction I that has an immediate or register with name X, I should be executed at least once with $X = v_1$, $X = v_2$, etc. Besides fixed values, it is also possible to use the special values *MIN* and *MAX*, which will be expanded context sensitively (i.e. *MAX*=$2^{31} - 1$ for register and *MAX*=$2^{11} - 1$ for I-type immediates, etc.) based on the encoded meta-data (and selected target ISA) in the generator. Value requirements are evaluated from top to bottom in the specification file, hence assignments are possible as well (see e.g. Line 56 and Line 61). Cross-coverage of two or more fields is specified with the *@cross* construct (e.g. see Line 65-67).

Instruction specific rules can be defined as well (see Line 68-86). Every instruction inherits by default the top-level requirements. It can override and modify existing definitions. For example in Line 73 the immediate value of the load (half-word) instructions LH and LHU is overridden to ensure 2 byte address alignment and in Line 77-81 the existing values for register-immediate computation instructions are extended with additional values and an instruction specific cross-coverage definition is added.

Please note, it is possible to accidentally specify coverage requirements that contradict instruction constraints. In fact, this happened to us a few times during development of the specification (e.g. by not overriding the value requirements for load/store immediates and thus violating the 4 and 2 byte address alignment constraints). However, in this case the SMT solver in the generator returns UNSAT and the source of the contradiction can be quickly identified, since the generator considers coverage requirements one after another. Next, we describe our generator in more detail.

IV. Test-Suite Generation

Test-suite generation, based on the test-suite specification, consists of three subsequent phases: A) pre-processing, B) solving, and C) test-case generation. We describe them in the following.

A. Pre-Processing

In the first step the specification file is pre-processed. Value coverage requirements, such as *RS1 :: {1, -1}*, are expanded into *@foreach* (that enumerates all opcodes that use an RS1 register) and *@exists* (that match the value of RS1 against each value in the set) blocks. Cross-coverage requirements are expanded by using nested @foreach blocks. In the next step all *@foreach* blocks are explicitly unrolled (since they only enumerate concrete values/opcodes/registers) and all multi *@exists* { A; B } blocks are expanded to single *@exists* { A } and *@exists* { B }. Thus, after pre-processing, the specification file only contains one *@forall* block and a list of top-level *@exists* blocks.

B. Solving

Next, (solving phase) each *@exists* is combined with the *@forall* block and is transformed independently into a boolean SMT formula X. Therefore, the generator creates a new symbolic instruction I, i.e. all fields in I are symbolic. I contains symbolic register indices RS1, RS2, RD and values for RS1 and RS2 as well as all immediate fields (I-type, B-type, S-type, J-type, U-type, shamt, csr_uimm, csr) and the opcode. The formula X reasons about I according to the processed *@exists* block. We use an SMT solver to obtain a solution, i.e. concrete values for symbolic fields, for X. Unconstrained symbolic fields (they will not be part of the SMT solution) are assigned random values (from their respective value domain). The result of the solving phase is a list of concrete instruction data.

C. Test-Case Generation

In the final phase, each instruction data is transformed independently into a test-case. Therefore, we provide a template for each instruction class that is filled with the instruction specific data to generate the test-case body. This body is then embedded in the default test-case framework as provided by the official RISC-V compliance testing framework. The default test-case framework provides code to initialize/shutdown the system, handle traps and provides memory regions for input (we initialize each word with an ascending number, starting with 1, at compile time) and output data (which will be dumped as signature).

Next, we briefly discuss the code generation templates for the relevant instruction classes. For illustration, Fig. 3 shows the templates. Placeholder are displayed in curly braces, e.g. {RD}, {RS1-value} and {OP} are replaced with the actual RD register, RS1 value and instruction opcode (e.g. ADDI,LW,JAL,BEQ), respectively, as provided in the instruction data. Please note, RX and RY are registers that do not overlap with the instructions source or destination registers. Hence, RX and RY store temporary values without accidentally affecting the actual instruction execution.

The body for a computational instruction simply initializes the register source value(s), then performs the operation and stores the result (register RD) as signature (Fig. 3 case A). The body for a load instruction (Fig. 3 case B, store handled similarly) initializes the RS1 value (Listing 9) and relocates it to the middle of the input data region (Listing 10). Then it performs the actual load (Listing 11) and stores the loaded word as signature. Please note, the RS1 value relocation ensures that the memory access stays within the input data range. This is important for load/store instructions to avoid undefined behavior due to potential different memory layouts.

We handle branch/jump instructions by generating a new jump label that corresponds with the requested jump/branch offset. We distinguish between forward (positive offset) and backward (negative offset) jumps/branches. A forward jump template is shown as case C in Fig. 3. A list of ADDI instructions, whose number depends on the jump offset, is used to fill the space between the jump and the label (Line 18). We use ADDI instructions instead of NOPs to also increment the result value (stored in register RX, initialized in Line 16) in order to help detecting wrongly implemented jumps. RX and RD (the jump instruction stores the *link* address in RD) form the signature. Case D in Fig. 3 shows a backward branch (e.g. BEQ). Branches compare the RS1 and RS2 register values, hence they are initialized first (Line 26-27). The actual backward branch happens in Line 33. The jump offset is encoded by again adding ADDI instructions and the RX register value is used as primary signature. Another ADDI instruction (Line 34) is added right after the branch to detect a wrong implementation.

The register based jump (JALR) is handled similarly to an offset based jump (JAL) but relocates the RS1 value (jump address) based on the jump target label at runtime. We handle CSR instructions by simply performing the CSR access and then storing the destination register (some CSR instructions store their value into a normal register) and CSR result (by reading the CSR) as signature.

V. Evaluation and Discussion

We have implemented our approach and automatically generated a test-suite with 8900 test-cases in 392 seconds based on our RV32I test-suite specification (186 non-blank lines). The evaluation has been performed on a Linux system with an Intel Core i5-7200U processor with 2.5 GHz. We implemented the generator in Python and used Z3 v4.8.4 as SMT solver.

Table I shows more detailed results grouped by different instruction classes. It shows the number of tests as well as the obtained coverage results for our generated test-suite (column: Our) and the RISC-V RV32I compliance test-suite (column: Official) [4]. The last column shows the maximum possible coverage (column: Max). We use GRIFT (*Galois RISC-V ISA Formal Tools*) [15] to measure coverage. GRIFT is a Haskell-based RISC-V formalization that aims to provide the foundation for several analysis techniques for RISC-V and is currently employed to measure the instruction coverage of the official RISC-V compliance test-suite. In particular, GRIFT measures and reports the *semantic branching structure*, i.e. essentially coverage of cases that influence the instruction execution (like branch not-/taken, RD is-/not x0, etc.). Please note, we approximately counted the number of test-cases in the official compliance test-suite by counting the number of SW operations in the test-body, since signature results are recorded using SW (the compliance test-suite has a test-file per instruction opcode with multiple test-cases).

It can be observed that our method improves the coverage results for every instruction class. The total coverage for the RV32I ISA is increased from 67% to 86%. The remaining coverage gaps are primarily due to jumps/branches (in particular illegal alignment and absolute jump target near the zero address) as well as memory (load/store access near the zero address) and CSR (different CSRs and field combinations) access operations. In particular the case for memory access/jump target to an absolute address close to zero (by using RS1=x0) is problematic to achieve in a general test-suite, since it poses strong assumptions on the architecture.

Based on our test-suite, we generated the reference outputs (signatures) using the official RISC-V reference simulator *Spike* [16].

TABLE I
EVALUATION RESULTS - COMPARING OUR TEST-SUITE (GENERATED IN 392 SECONDS) AGAINST THE OFFICIAL HAND-WRITTEN (RV32I) TEST-SUITE.

Instr. Class	Num. Tests		Coverage (GRIFT)		
	Official	Our	Official	Our	Max
Computational	1020	4677	100	103	106
Load / Store	201	685	24	25	32
Branch / Jump	298	760	113	119	134
CSR / System	117	2778	63	137	174
All Together	1636	8900	300	384	446

For cross-checking, we compared the signatures against the open-source RISC-V VP [17]. This comparison revealed an error in the CSR implementation of the VP. In particular, the instruction *CSRRCI x20, MHARTID, 0* caused an erroneous illegal instruction trap, because *MHARTID* is a read-only CSR and *CSRRCI* is an instruction that in general modifies the CSR. However, for the special case that the immediate argument is zero, the CSR content is only read into the CPU register (x20 in this case) and hence the instruction should not trap in this case. This error was not found by running the official compliance test-suite, i.e. all signatures were equal on *Spike* and VP.

Overall, our evaluation shows that our method is effective and can provide better quality than the hand-written official RISC-V compliance test-suite in combination with a significantly lower specification effort. For future work, we primarily plan to provide specifications for RISC-V ISA extensions and investigate their combinations. We believe that our specification-based approach that enables easy sharing, inheritance and overriding of requirements will be very suitable for this endeavor. In addition, we also plan to consider specification of requirements that reason about instruction sequences, and explore methods to minimize the resulting test-suite and investigate symbolic execution techniques like [18] for complementary test generation.

References

[1] 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, 2017.

[2] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[3] "The challenge of RISC-V compliance," https://semiengineering.com/toward-risc-v-compliance/.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] W. Ma, A. Forin, and J. Liu, "Rapid prototyping and compact testing of CPU emulators," in *RSP*, 2010, pp. 1–7.

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

[11] 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.

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

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

[14] "Formal verification of pulpino and other risc-v socs," https://content.riscv.org/wp-content/uploads/2019/06/13.35-OneSpin-RISC-V-Zurich-2019-SoC-Talk.pdf.

[15] "GRIFT - galois RISC-V ISA formal tools," https://github.com/GaloisInc/grift.

[16] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.

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

[18] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019.