Automating Generation and Maintenance of a High-Quality Architectural Test Suite for RISC-V

S Pawan Kumar
pawan.kumar@incoresemi.com
InCore Semiconductors

Shrreya Singh
singh.shrreya@iitgn.ac.in
IIT Gandhinagar

Neel Gala
neelgala@incoresemi.com
InCore Semiconductors

Allen Baum
allen.baum@esperantotech.com
Esperanto Technologies

ABSTRACT

Considering the modularity of RISC-V, proving the compatibility of implementations against the ISA has become critical to prevent fragmentation and ensure its success. The test suite needed to achieve this should cater to all legal enumerations of the ISA and determine with high confidence, the ISA compatibility of the target. This paper proposes open-source tools(RISCV-ISAC and RISCV-CTG) which significantly automate building and maintenance of a high quality and scalable test suite. RISCV-ISAC collects coverage from execution logs and checks for data propagation to signature. RISCV-CTG is a coverage driven test generator, which generates minimal tests for maximum coverage with RISCV-ISAC.

KEYWORDS

Compliance Testing, RISC-V, Test Generators, Coverage

1 INTRODUCTION

RISC-V [7, 8], an ISA (Instruction Set Architecture) maintained by a non-profit organisation, has gained rapid interest and adoption globally by both academia and industry due to its open-source and royalty-free nature. But, it faces the threat of fragmentation – the existence of numerous significantly varying implementations of the same ISA.

The success of RISC-V, therefore, lies in the fact that each of the varied implementations are indeed compatible with the ISA. The testing for compatibility of an implementation against an ISA specification is called *Compliance Testing* or as referred to in the RISC-V community, *Architectural Testing* (AT). One should not confuse AT with the classical definition of design verification (DV). AT deals with verification of the details specified in the ISA specification; any implementation specific details are ignored in this context. AT is merely a tiny subset of DV and aims at validating that the designer has correctly interpreted and implemented the ISA specification. AT does not indicate that the implementation is bug free; DV is required to make such a claim. Henceforth, for simplicity, we shall use the term *target* to refer to an implementation or the design under test (DUT) for which AT needs to be performed.

The RISC-V specification is incredibly permissive i.e it allows the use of encodings from the reserved space, op-code space of unimplemented extensions and any unreserved encodings to define custom instructions. Due to this nature, AT should only test/verify the parts of the specification which have been implemented (positive testing) as opposed to testing all possible behaviors. This is one of the major differences between AT and DV.

Considering the numerous possible implementation choices of the RISC-V spec, the task of building an architectural test suite is imaginably daunting. The test suite should include tests to check all possible legal enumerations and yet declare with high confidence whether the target is ISA compatible or not. Some of the major requirements and challenges associated with designing such a test suite are:

1. The tests must follow a standard format to maintain uniformity and enable maintenance as the ISA specification grows and the scope of testing increases.
2. An ISA coverage specification format, which provides a standard way of indicating the quality of the tests present in the suite and identifying possible holes or gaps in the tests.
3. A coverage extraction tool is required to capture the actual coverage of the tests based on the above mentioned specification.
4. As the spec and the testing scope grow, the task of writing these tests manually will no longer be a viable choice. Thus, an efficient, directed and accurate test generator is required to generate the test cases.

Recently, the RISC-V Architectural Test SIG has released a *Test Format Specification* [3] which provides a standard scheme and macros for writing architectural tests, thereby addressing the first requirement above. This specification introduces convenient assembly macros like *RVTEST_CASE* and *RVTEST_ISA* which provide the necessary infrastructure for selecting and filtering tests based on the target specification. The spec further mandates that the tests be signature based - i.e. each test must maintain a specific region of memory called *signature* where various computational results and other characteristic outputs of the test are stored. This signature generated by the execution of a test on a target must match the signature generated using an approved reference model, to declare the test as *PASS* on the target. This approach is preferred over other prevalent approaches in DV (like Formal or Lock-Step verification) because, AT doesn't verify the entire architectural state at all points in the test. It is enough to test the changes to architectural state as seen by the running software. This approach also lowers the effort required to perform AT and is generic enough to cater to any target, irrespective of implementation details and technologies used. The spec also defines certain target specific macros (like boot-macros, exit macros, interrupt-macros, etc) which are defined by the target and seamlessly integrated with the tests. This ensures that the tests can be run on any target, accounting for the non-ISA variability that is allowed and the implementation specific behaviours, in the context of AT.

In this paper, we introduce two different open-source tools which address challenges 2 3, and 4 mentioned above:

- **RISCV-ISAC**: enables expressing ISA-level coverpoints in a very intuitive format. It also provides a mechanism to check if a test execution hits the necessary coverpoints without depending on any licensed tools or modifications to the models. The tool also dumps a data-propagation report which indicates if the test was faithfully executed as per expectations.
- **RISCV-CTG**: is an automated, CSP (Constraint Satisfaction Problem) based test generator, with a focus on generating minimal tests which can achieve maximum ISA coverage.

2 RISCV-ISAC

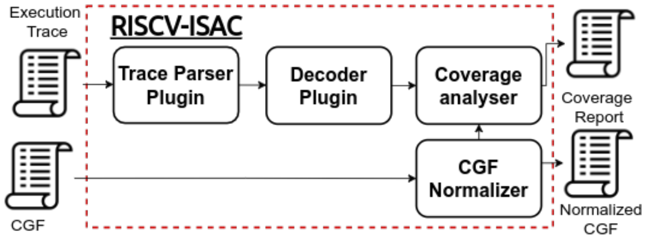

Figure 1: Tool flow for RISCV-ISAC

The primary factor to consider while building a test-suite for AT is the quality of the tests. While it is crucial that each test assesses a specific aspect of the ISA specification, it is equally important to ensure that those aspects of the ISA were indeed covered by executing the test on the target. Deriving concepts from the DV world to address the above concern, one would attempt to create a test-plan followed by defining a set of covergroups and coverpoints whose coverage would provide a qualitative measure of the test-suite. Classically, these coverpoints would typically be written in a language like SV-UVM for a given RTL model. This poses two major challenges : using SV-UVM would entail depending on commercial simulators thereby, increasing the barrier of community contributions. Secondly, the RTL model adopted must be configurable to mimic all possible choices of the ISA - which is not feasible to build and/or maintain by the community, especially when the ISA continues to evolve and grow.

To address these issues, we propose RISCV-ISAC (RISC-V ISA Coverage). ISAC is a pure-python tool that allows the user to define ISA level coverpoints in a simple and intuitive format using architecturally known variables like *rs1, rs2, rd, imm, rs1_val, rs2_val, etc.* We refer to this custom format as *Cover-Group-Format* (CGF) henceforth. An example and more details about the CGF are provided in the next subsection.

Once the coverpoints are defined in CGF, we need a mechanism to check which coverpoints are hit upon the execution of a particular test. To achieve this, we exploit the fact that simulators like SAIL [5] and Spike [6] (which are treated as golden/pseudo-golden models of the ISA) have the capability of generating an instruction execution trace which captures all architectural state-changes occurring due to an instruction execution. ISAC parses these logs at an instruction granularity, checks for coverpoint hits in the CGF and thereafter provides a coverage report of the test or test suite as a whole. Figure-1 shows the tool flow adopted by ISAC.

2.1 CGF

The CGF is a dictionary (a collection of key-value pairs) where each node is a collection of coverpoints called *covergroup*. Each covergroup follows a template similar to the example defined in Listing-1.

In Listing-1, *add_cov* is referred to as the covergroup label or name. The *config* node represents a list of ISA configurations under which the particular covergroup is applicable. The conditions expressed in this list are usually checked against the target specification yaml expressed in the riscv-config [4] format. If none of the conditions in the list evaluates to *True*, then the coverage node is skipped during extraction by ISAC. This feature automatically disables covergroups (and their reporting) for targets that do not implement a particular option that the covergroup is aimed for. Thus, a single CGF file can be shared across targets, avoiding duplication.

Listing 1: Example for covergroups in CGF

```
add_cov:                                              1
  config: [check ISA:=regex(.*I.*)]                   2
  opcode: {add: 0}                                    3
  rs1: {x1: 0, x3: 0}                                 4
  rs2: {x2: 0, x4: 0}                                 5
  op_comb:{'rs1 == rs2 != rd': 0}                     6
  val_comb:                                           7
    'rs1_val > 0 and rs2_val > 0': 0                  8
    'rs1_val > 0 and rs2_val < 0': 0                  9
    abstract_comb:                                    10
      'walking_ones("rs1_val", 64)': 0                11
      'walking_zeros("rs1_val", 64)': 0               12
  csr_comb:{'mtval == 0xdeadbeef': 0}                 13
  cross_coverage:                                     14
    # <> implies a list in the format                 15
    # <inst-opcode> :: <var-assign> :: <val-rules>    16
    # ? implies don't care                            17
    '[(add):(sw)]::[a=rd:?]::[?:rs2==a or rs1==a]':0  18
```

The *opcode* node indicates a dictionary of opcodes for which the coverpoints in this covergroup are applicable. This allows the user to merge coverpoints across multiple opcodes. The nodes *rs1* and *rs2* contain coverpoints which expect the respective registers to be observed in the *rs1* and *rs2* fields of the instruction at least once. The same format can be used for defining coverpoints for the *rd* field and other register fields of the instruction. Note the "*: 0*" at the end of each coverpoint. This indicates the number of times the coverpoint was hit during test execution. The initial CGF should have these values set to 0.

The *op_comb* node defines coverpoints which indicate register combinations of interest. Line-6 above shows the coverpoint where the source operands are read from the same register while the destination register is different.

Similarly, the coverpoints under *val_comb* indicate the register value and immediate value based combinations. The coverpoints expressed under these nodes are evaluated as python expressions in ISAC. Each coverpoint must evaluate to a boolean value and can use a number of different operators (boolean, arithmetic, etc) to express the coverpoint accurately. For example, to check if the value in *rs1* register is an even positive integer we can use the following expression: '$rs1\_val > 0$ *and* $rs1\_val\%2 == 0$' . Coverpoints related to the outcome of an instruction are currently handled by projecting them onto the inputs or by expressing the mathematical equation of the operation. For example, the coverpoint '$result == 3$' for the *add* instruction can be expressed as '$rs1\_val == 1$ *and* $rs2\_val == 2$' or '$rs1\_val + rs2\_val == 3$'. Having the constraint directly on the result of an operation would necessitate modelling it in software and then establishing that it is also compliant with the ISA specification. This creates a circular dependency in the methodology. Under the *val_comb* is the *abstract_comb* node, which is used to express coverpoints in an abstract fashion using python functions. These abstract functions are used to generate the low level coverpoints defined earlier, during the *normalization* phase at the start of coverage collection. These abstract functions are defined internally within ISAC's library. For example, line-11 of Listing-1 describes a *walking_ones* function which would unroll to 64 coverpoints for *rs1_val* assigning all patterns of a 64-bit walking-one. This allows users to define the CGF succinctly and also leverage python features which can automate coverpoint generation to a vast extent.

The *csr_comb* node is used to describe the coverpoints for Control and Status Registers(CSRs) defined as per the privileged specification [8]. These coverpoints can be written similar to the ones in *val_comb* node by using the respective names of CSRs to refer to their values. CSRs, the architectural state of each hardware thread, are sometimes omitted from the instruction trace. The accuracy of coverage reporting for these types of coverpoints depends on the presence and correctness of CSR value changes in the instruction trace. In case a covergroup contains both : non-empty *opcode* and *csr_comb* nodes , the coverpoints in the latter are checked for a hit only if the instruction opcode has been executed, otherwise they are updated for every instruction in the execution stream processed by ISAC.

The *cross_coverage* node is used to define *cross-coverpoints* across instructions in an arbitrarily large instruction window. Lines 15-17 in Listing-1 depict the general format. Lists are indicated by *:* separated elements. The length of all the lists in a coverpoint should be equal to the size of the instruction window for the coverpoint. For each element in the *inst-opcode* and *val-rules* lists, the corresponding instruction in the window must match the opcode and satisfy the condition in *val-rules* to be considered a hit. The statements in the *var-assign* list are used to pass the values of instruction fields downstream i.e the assignment is performed after checking and on a match, using custom variables(like *a* in line 18). Line 18 depicts a coverpoint corresponding to a Read-After-Write hazard between an *add* and a subsequent *sw* instruction.

The coverpoints in CGF are maintained as a python dictionary, hence duplicate coverpoints cannot exist. This is advantageous when there are multiple abstract functions which may generate an overlapping set of coverpoints.

2.2 Coverage Analyser

As mentioned earlier, ISAC extracts ISA coverage from the execution trace files generated by simulators like SAIL and Spike. ISAC uses a plugin based architecture to support instruction execution trace formats of different simulators. The plugin for each simulator should include a file parser which can create a data stream on a per-instruction execution basis which includes the following information: *pc* of the instruction, instruction encoding in hex, and any architectural state changes that may have occurred due to execution of that instruction. This data stream is passed to a decoder plugin which decodes the instruction to obtain all the necessary information about the fields in the instruction. This decoded data stream is used by the ISAC's *coverage analyser* utility to measure the coverage of the test.

The coverage analyser utility maintains a minimal architectural state internally, which includes integer and floating register files and CSRs. These states are updated each time an instruction that causes any architectural change is observed. This architectural state is also used to assign values to the **_val* variables of the *val_comb* coverpoints, thus enabling value-based coverpoint support. ISAC takes care of converting the data stored in the architectural registers to appropriate types (signed, unsigned, float, etc) before evaluating the coverpoints. The instruction stream is also captured in a queue whose size is provided at run-time via the CLI (Command Line Interface). The instruction window for cross-coverpoint evaluations starts at the head of this queue and the length of the window depends on the requirements of the coverpoint. Hence all coverpoints should span over instruction windows less than or equal to the size of the queue. Since the coverpoints are expressed as boolean expressions, ISAC uses the default *eval()* function offered by python, to evaluate each coverpoint. An expression evaluating to *True* is considered a hit for that coverpoint.

2.3 Data Propagation

ISAC also generates a *data propagation report* (DPR) from an execution trace file. This report is particularly useful for signature based tests (as employed by the RISC-V AT), where one would like additional guarantees that the result of certain operations have a direct correlation to the contents in the signature regions. Thus, this report includes the following statistics:

(1) number of instructions that hit unique coverpoints
(2) number of coverpoints hit by multiple instructions
(3) number of signature/memory region overwrites
(4) number of updates to multiple memory regions with the result of a single instruction
(5) number of coverpoint hits without update to signature/memory region

The DPR, ultimately determines if the test meets the expectation and if there are scopes of optimizing it.

3 RISCV-CTG

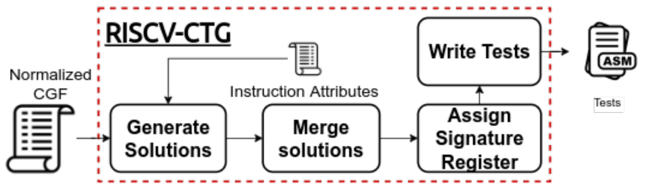

Figure 2: Tool flow for RISCV-CTG

With the help of ISAC's CGF format, one can very easily and accurately define the expectations of each test included in the AT suite. Thus, what is required now are assembly tests which can meet all the coverpoints defined in the CGF. We propose RISCV-CTG (RISC-V Coverage Driven Test Generator) to address this requirement. It is a python based tool which takes the CGF file as input, with an elementary goal of generating minimal assembly tests which can meet as many of the coverpoints as possible, thereby producing a high quality test for AT. Figure- 2 shows the tool flow adopted by CTG.

3.1 Generating Solutions for Coverpoints

Once the CTG receives a CGF file, it iterates over each coverpoint to find an appropriate solution. Each coverpoint is effectively a set of values assigned to the opcode and its associated variables (*rs1, rs1_val, rd, etc.*). Therefore, CTG maintains a finite sized domain (finite list of possible values) for the associated variables. Note, a test developer can further constraint or expand the size of this domain as per requirements of the test. The objective, however, is to obtain a solution from the associated domains of the variables which leads to the evaluation of the coverpoint to be true (recall from the previous section, that coverpoints are defined as boolean expressions in CGF). CTG models this as a classical Constraint Satisfaction Problem (CSP) over finite domains, and uses the constraint solver from [1] to obtain the appropriate values for the variables. Once a solution to a single coverpoint is found, CTG checks if the same solution satisfies any of other pending coverpoints. All such coverpoints are disregarded in further iterations. This methodology allows CTG to generate a smaller subset of solutions which can satisfy as many coverpoints as possible, thereby reducing the final test size. This process is repeated for the next unsatisfied coverpoint in the CGF. The solutions for the register names and the values of the operands are solved for independent of each other i.e the register operand nodes($rs1$, $rs2$ etc) and the *op_comb* node are solved for independent of the *val_comb* node. The output of this stage is two lists of tuples, where each tuple includes the values assigned to the variables(register name or value).

The speed of the constraint solver depends on the size of the domains. Larger domains lead to longer times for obtaining a solution. Thus in CTG, it is necessary to restrict the domain size of the variables to a small subset to have realistic execution times. To achieve this, we ensure that the domains of the variables include a range of general values and other specific values which represent patterns of interest. To further improve execution time, CTG also provides a choice in the algorithm to use for solving the CSP - *Min-Conflict or Backtracking.* The Min-Conflict algorithm picks a random set of values (from the domains) as the starting point and heuristically finds a solution. This results in an increase in execution time as compared to backtracking and a variation in the size of the tests generated. Experimental results show that the difference is very low and can be ignored for all practical purposes.

It is possible that the CTG encounters an illegal or faulty coverpoint, such as *'rs1_val <0 and rs1_val >0'*. In such a case, the python-constraint solver is not able to find a solution and CTG simply skips the coverpoint with a warning indicating that a solution could not be found for this coverpoint.

Under special circumstances, a coverpoint may express a complete solution in itself. For eg. *'rs1_val == 2 and rs2_val == 5'*. In such cases the coverpoint is expected to be post-fixed with a special string *"#nosat"* which is recognised by CTG and the solution is extracted from the coverpoint itself rather than employing the constraint solver. This reduces the effort and time spent by the solver. This feature is particularly useful when the domain size of the variables are quite large (like floating point operands) or when the values of all the variables is fixed and known.

3.2 Test Generation

Once all the solutions are generated, multiple solution tuples from the two lists are merged together to provide a single test instance object which contains all the necessary fields. Any missing fields are assigned default values which are guaranteed to not interfere with the intent of the test. This is used to minimise the size of a test without compromising test quality. The assembly tests generated by CTG are always compliant with the Test Format Spec [3]. As per the spec, the output of each test-case must propagate to a unique region of memory denoted as the *signature.* Thus, once CTG completes evaluation of all coverpoints within a covergroup, it also assigns a signature pointer register (which is not being used by the other register fields) to each object.

In order to generate the assembly tests, CTG maintains a database of assembly macros. These macros are responsible for resource initialisation (using inline assembly pseudo instruction *li* or load values from memory) and other necessary elements to test the particular opcode. Hence CTG is impervious to any optimisations in this regard. For eg, to test an *add* operation, the macro would first initialize the 2 registers with the required values, perform the *add* operation and then store the result in the region using the signature pointer. These macros have parameterized arguments which match the values in the cover object. For each test instance object generated, CTG will simply replace the arguments of the macro with the relevant values from the solution and create an instance of the macro in the output assembly file.

Listing-2 shows the pseudo-code of a macro for an instruction with 2 register operands. This macro uses an in-lined pseudo-instruction *li* to initialize the source registers with their respective values. An instance of the macro for the *add* instruction is shown on line 7.

Listing 2: Assembly Macro pseudo-code

```
#define RR_OP(op,rd,rs1,rs2,v1,v2,ptr,offset) \        1
li rs1,v1; \                                            2
li rs2,v2; \                                            3
op rd,rs1,rs2; \                                        4
sw rd,offset(ptr);                                      5
// Instance of the macro in an add test                 6
RR_OP(add,x2,x1,x2,0x03,0x2,x3,0)                       7
```

CTG, as of today, supports creating a test for all formats of coverpoints defined in the CGF, except for the csr and cross-coverage based coverpoints, both of which are a work-in-progress.

4 EXPERIMENTAL RESULTS

All experiments in this section were performed on a system with an Intel i7-8750H processor and 16GB of memory. The tools used are RISCV-CTG (*v0.5.3*), RISCV-ISAC (*v0.7.3*), RISCV-CONFIG [4] (*v2.10.0*), RISCV-ISA-SIM [6] (*Commit id a31184c*) and SAIL RISCV Model [5] (*Commit id f66d0fb*). All the proposed tools have been

Table 1: Experimental Results

Suite (ISA)	Coverpoints (LOC)	CGF Normalization Time(s)	Generation Time(s)				Average Generation time per test(s)		Test Cases generated		Coverage Collection Time (s)	
			Cfg1	*Cfg2*	*Cfg3*	*Cfg4*	*Cfg1*	*Cfg2*	*Cfg1*	*Cfg2*	*Cfg1*	*Cfg2*
RV32I	16068 (1285)	7.91	293.56	87.87	2005.40	528.52	39.32	10.36	12638	13150	189.07	227.43
RV32M	6302 (524)	3.43	136.24	35.536	751.70	199.94	93.96	24.99	5165	5386	59.21	65.90
RV32C	5431 (833)	2.56	80.13	23.80	549.61	113.37	21.14	4.36	4633	4822	49.07	99.43
RV64I	23052 (1576)	11.65	864.29	215.88	4395.42	1298.43	66.60	19.67	17825	19113	348.82	292.70
RV64M	11884 (619)	7.17	575.89	128.55	3121.25	822.27	240.10	63.25	9571	10393	121.23	112.92
RV64C	9219 (942)	4.61	319.96	84.24	2280.61	338.60	71.27	10.58	7843	8409	116.42	136.44

Table 2: Comparison of Existing Tests with CTG generated tests.

Suite (ISA)	Coverage of Tests		Improvement
	Existing	CTG Generated	
RV32I	3308 [20.59%]	16068 [100%]	79.41%
RV32M	735 [11.66%]	6302 [100%]	88.34%
RV32C	481 [8.86%]	5431 [100%]	91.14%
RV64I	724 [3.24%]	23052 [100%]	96.86%
RV64M	0 [0%]	11884 [100%]	100%
RV64C	0 [0%]	9219 [100%]	100%

implemented in python-3.6 and released as open source software under the BSD 3-clause license.

Within the scope of this paper, coverpoints and tests for the base(I), compressed(C) and multiply(M) extensions for both the 64 and 32 bit versions of RISC-V ISA have been generated. These coverpoints only cover the register combinations, different corner cases depending on the operation and misaligned effective/target address exceptions. Negative testing(testing for illegal instructions) is not considered, in conjunction to AT requirements. The number of coverpoints for each test-suite generated are captured in Column-2 of Table-1. These coverpoints were expressed using the CGF format described in section-2.1. Column-2 of Table-1 also shows the number of lines(LOC) required to specify each CGF using *abstract python functions*, indicating a 10× reduction in size. Column-3 of Table-1 indicates the amount of time taken (in seconds) by ISAC to normalize a CGF containing abstract coverpoints of a suite to a CGF which contains only low level coverpoints.

The coverpoints defined above are fed to CTG to generate the required set of tests. The CTG however, has two different configuration parameters namely:

- The number of parallel processes to spawn during generation. Each process generates a test corresponding to a single node in the CGF.
- Using either the Min-Conflict Algorithm or the Backtracking Algorithm for solving CSPs.

Due to space limitations, we provide and compare results for the following configurations of CTG

- *Cfg1*: Eight processes using the Min-Conflict algorithm
- *Cfg2*: Eight processes using the Backtracking algorithm
- *Cfg3*: Single process using the Min-Conflict algorithm
- *Cfg4*: Single process using the Backtracking algorithm

The amount of time to generate tests for each suite for the above combinations is captured in columns 4 to 7 of Table-1. We observe that the generation time is always higher when using the Min-Conflict algorithm due to its heuristic nature(the algorithm terminates only after a fixed number of iterations and not opportunistically). Comparing columns 4,6 and 5,7 of the same table, we see that by using eight parallel processes we get 3× to 4× speed up. Columns 8 and 9 of Table 1 similarly capture the average time taken to generate a single test of each suite. The 'M' extension show higher average time per test because all the instructions in the M-extension are 2 operand instructions with significantly large domains. Though similar instructions exist in the base extension, their impact is offset by the presence of other instructions with smaller domains and fewer operands.

Columns 10 and 11 of Table-1 show that the number of test-cases required to meet all the coverpoints of the respective input CGF is lower when using the Min-Conflict algorithm as compared to the Backtracking algorithm. This outcome is a side-effect of the inherent randomness and heuristic approach adopted in the Min-Conflict algorithm. Using a random starting point provides a solution which is more likely to satisfy multiple unsolved coverpoints as opposed to choosing the first possible solution due to the nature of coverpoints in the covergroups. As the number of test-cases increases, the number of instructions in the execution trace also increases, thereby increasing the time taken by ISAC to collect coverage for a suite. This observation is clear from Columns 12 and 13 of Table-1.

The generated tests were then executed on the SAIL [5] and spike [6] models; the signatures generated were compared against each other to prove compatibility. The coverage for the tests were then measured using ISAC based on the CGFs defined initially and the execution trace generated by SAIL. These tests were used to update the official Architectural Test Suite of RISC-V. Table-2 captures the improvement in coverage achieved by the CTG generated tests as compared to the existing tests on the RISC-V architectural test-suite repository.

Bugs Found using proposed AT: During testing it was found that on the SAIL [5] model compressed hint instructions raised an *illegal instruction exception* due to a decode error, resulting in a signature mismatch. The bug was reported and subsequently fixed in the model.

The behaviour of the *mtval* CSR on an *ebreak* instruction was ambiguous in the ISA specification which resulted in both (Spike and

SAIL) the models implementing different behaviours. The spike [6] model updated the *mtval* register with a value of 0 whereas the SAIL [5] model left the register unmodified i.e. to a previous written value. This ambiguity was reported and resolved to indicate that the address(virtual) of the faulting instruction should be used to update the register in the event of a breakpoint exception. The models were also subsequently fixed to reflect this behaviour.

5 RELATED WORK

Architectural Testing for RISC-V has only recently gained traction with a handful of research papers [11–13] specifically addressing this problem. Authors of [13] use a similar approach to that of CTG, to generate the tests using a SMT solver. However, the format of specification of constraints is complex and the coverage calculation is based on the coverage reported by grift [2]. The coverage metric defined by *grift* is quite rudimentary and insufficient for a qualitative architectural compatibility testing. The CGF coverpoint specification format proposed in this paper allows defining more complex coverpoints which are clearly a super-set of those defined by *grift*. Moreover, the proposed tools in this paper use the coverpoints for coverage as constraints for test-generation, thereby reducing the maintenance and synchronization overheads.

The mutation based approach used in [12] is effective in identifying gaps in a previous version of the test suite for the base ISA hosted at [3] *v1.0(commit id 2636302)*. However, the RISC-V ISA is an evolving specification with myriad extensions. Modifying the instruction set simulator(ISS) for producing tests and expressing mutants in the source code of the ISS is a skill intensive and a time consuming task. Comparatively, the ISAC and CTG offer a very low barrier on skills and use intuitive methodologies to achieve higher quality tests for AT. Furthermore, the nine mutations expressed in [12] can quite conveniently be expressed in the proposed CGF format and the CTG can generate the required tests, such that the mutations are killed without the overheads of long simulation times.

The negative testing approach of [11] aims to complement the compatibility testing by ensuring that no functionality other than the ones defined by the specification are present in the implementation. However, this kind of an approach is infeasible for AT due to the permissive nature of RISC-V. The behaviour of any instruction in the unimplemented encoding space is unknown and cannot be accounted for in the tests.

The methodologies proposed in [11, 12] also rely on an ISS to drive the test generation and measure coverage. However, both the works have used a custom ISS which is not a community approved/adopted golden model. This leads to overheads of proving compatibility of the custom ISS to the RISC-V ISA. More importantly, certain golden simulators are written in domain specific languages (like SAIL) which may not be amenable to support mutation like syntax. Additionally, in-lining the mutation code within the ISS will impair the readability of the model, thereby further affecting its adoption and maintenance. The work proposed in this paper does not expect any modifications to existing simulators and neither adds a new simulator to the mix.

The authors of [9] describe a rather basic coverage metric namely: register and instruction coverage. The metric merely tests whether the test suite contains all instructions specified by the ISA. The authors measure the number of register (GPR) accesses in the entire test suite. This metric in itself is vague and insufficient for architectural testing. The proposed CGF format supports expression of much fine grained coverpoints (including cross products across instructions and dependency hazards) as compared to those defined in [9].

The authors would like to point out that, none of the existing works in literature test whether the test cases actually influence the signature in the intended way. This is necessary in signature based tests like the ones employed in RISC-V AT. Consider a test-case which is intended to find a fault in the target for a certain opcode. If the test does not propagate the appropriate result to the signature region or the target stores a different value in the signature instead, it can lead to a signature match, thereby causing a false positive (where a faulty target is declared ISA compatible). The Data Propagation Reports generated by ISAC help provide confidence in the test by looking for proof of propagation of the results to the signature.

6 CONCLUSION AND FUTURE WORK

In this paper we proposed two tools which help generate and maintain test suites for RISC-V Architectural Testing. We have generated and released tests for the RV64IMC and RV32IMC configurations. The bugs found out in reference models during the process were reported to the maintainers and fixed subsequently. The tools proposed in this paper are being actively adopted by the community to support the upcoming extensions of the ISA like the floating point, bit manipulation and packed SIMD. Future work in these tools includes exploring generation of tests and coverpoints for the privileged architecture and tests for various micro-architectural hazards. The coverage collection time in ISAC can also be reduced by considering a boolean hit/miss for each coverpoint instead of counting the number of hits.

REFERENCES

[1] 2021. Constraint Solving Problem resolver for Python. https://github.com/python-constraint/python-constraint

[2] 2021. Galois RISC-V ISA Formal Tools. https://github.com/GaloisInc/grift

[3] 2021. RISC-V Architectural Tests. https://github.com/riscv/riscv-arch-test

[4] 2021. RISCV-CONFIG: RISC-V Configuration Validator. https://github.com/riscv/riscv-config

[5] 2021. Sail RISC-V model. https://github.com/rems-project/sail-riscv

[6] 2021. Spike, a RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim

[7] 2021. Volume 1, Unprivileged Spec v.20191213. https://github.com/riscv/riscv-isa-manual/

[8] 2021. Volume 2, Privileged Spec v.20190608. https://github.com/riscv/riscv-isa-manual

[9] Peer Adelt et al. 2021. Register and Instruction Coverage Analysis for Different RISC-V ISA Modules. In *MBMV 2021; 24th Workshop*. 1–8.

[10] Vladimir Herdt et al. 2020. Adaptive Simulation with Virtual Prototypes for RISC-V: Switching Between Fast and Accurate at Runtime. In *2020 IEEE 38th ICCD*. IEEE, Hartford, CT, USA. https://doi.org/10.1109/ICCD50377.2020.00059

[11] Vladimir Herdt et al. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side *. In *2020 57th ACM/IEEE DAC*. https://doi.org/10.1109/DAC18072.2020.9218629

[12] Vladimir Herdt et al. 2021. Mutation-based Compliance Testing for RISC-V. In *Proceedings of the 26th ASPDAC Conference*. ACM, Tokyo Japan. https://doi.org/10.1145/3394885.3431584

[13] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Towards Specification and Testing of RISC-V ISA Compliance*. In *2020 DATE*. 995–998. https://doi.org/10.23919/DATE48585.2020.9116193