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

CARRV at ISCA 2022

S Pawan Kumar [1] Shrreya Singh [2] Neel Gala [1] Allen Baum [3]

[1]InCore Semiconductors

[2]IIT Gandhinagar

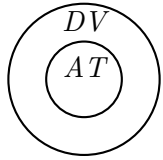
[3]Esperanto Technologies

19 June 2022

Outline

Objectives

- Architectural (Compatibility) Testing involves ensuring that an implementation of the ISA meets all the requirements under all conditions.
- RISC-V is micro-architecture agnostic and Extremely Configurable; Its impossible to create an exhaustive test suite for all implementations.
- However, it is possible to build a suite which test limited areas of the ISA.
- AT is a subset of design verification and is not sufficient by itself to ensure a functionally correct implementation. It only deals with the verification of details specified in the ISA.



Challenges

1. Standardised format for tests which maintains uniformity and enables maintenance.
   - RISC-V Architectural Test SIG has released the *Test Format Specification*[1] which provides standard macros for the tests. This keeps the tests immune to implementation specific choices and behaviour.
   - This spec mandates that the tests be signature based, where the signatures are compared with the signature from a "golden" model.
2. A standard way of indicating the quality of tests and identify possible holes or gaps, an ISA coverage specification format.
3. A tool to measure coverage as per 2.
4. A tool to generate efficient, directed tests as the ISA specification and testing scopes grow.

[1] *RISC-V Architectural Tests*. URL: https://github.com/riscv/riscv-arch-test.

Contributions

1. Low barriers to entry
   - Simple ISA Coverage specification format
   - Flexible & Free: Independent of simulator/implementation
   - Need knowledge about RISC-V and basic python to get started
2. Generated tests are controlled and directed.
   - Generated from coverpoints, ensuring that full coverage is achieved with minimal testing.
   - The tests can be run on any implementation/model
   - No negative testing; Test only for features/behaviour implemented.
3. Data propagation analysis ensures that the signatures generated are influenced by the tests.
4. Open source tools for coverage measurement and test generation

Coverage Definition

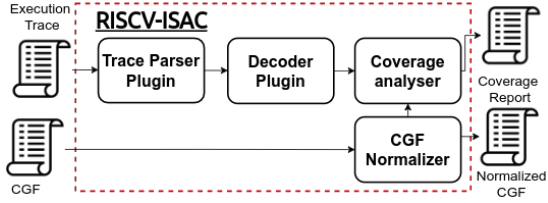
- A coverpoint specifies a boolean expression over the fields of an architectural element(instruction/state) that is required to be covered during execution
- In classical DV: Define coverpoints in SV-UVM for the RTL
  - High entry barrier: Expensive and requires SV knowledge
  - Extremely difficult to implement all possible architectural options in a single RTL
- In ISAC, coverpoints are expressed as pure python expressions.
- The coverpoints are categorised based on the variables/fields which can be tested.
  - Instruction Mnemonics(Opcode)
  - Register Operands
  - Operand Combinations
  - Value Combinations
  - CSR Value Combinations
  - Cross Combinations (To define coverage across multiple instructions)
- Custom functions in python(*abstract_comb*) which are resolved into the standard coverpoints during *normalization*.

CGF Description

```
add_cov:
  config: [check ISA:=regex(.*I.*)]
  opcode: {add: 0}
  rs1: {x1: 0, x3: 0}
  rs2: {x2: 0, x4: 0}
  op_comb:{'rs1 == rs2 != rd': 0}
  val_comb:
    'rs1_val > 0 and rs2_val > 0': 0
    'rs1_val > 0 and rs2_val < 0': 0
    abstract_comb:
      'walking_ones("rs1_val", 64)': 0
      'walking_zeros("rs1_val", 64)': 0
  csr_comb:{'mtval == 0xdeadbeef': 0}
  cross_coverage:
    # <> implies a list, ? implies don't care
    # <inst-opcode> :: <var-assign> :: <val-rules>
    '[(add):(sw)]::[a=rd:?]::[?:rs2==a or rs1==a]':0
```

Tool Flow

- Coverage is computed from the instruction trace(generated by the implementation/model) containing the following information:
  - Instruction Address
  - Instruction encoding
  - Architectural state changes like csr and regfile updates(if any)



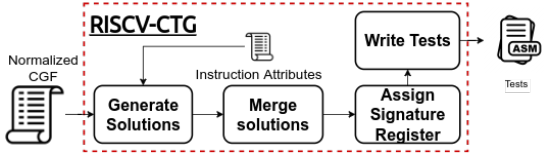
- Trace parser and Decoder are implemented as plugins to support customisation.
- Architectural states are maintained internally.
- Supports filtering instructions which influence coverage.
- Constrain coverage collection to only specific covergroups for a run.
- Custom boundaries for signature region can be specified.
- Generates YAML and HTML reports for coverage statistics and data propagation.

Data Propagation Reporting

- It is essential for signature based tests(as employed by RISC-V AT) to propagate the results of relevant instructions to the signature region.
- ISAC tracks the movement of data into the signature region. The following statistics are reported for each test:
  - number of instructions that hit unique coverpoints
  - number of coverpoints hit by multiple instructions
  - number of signature/memory region overwrites
  - number of updates to multiple memory regions with the result of a single instruction
  - number of coverpoint hits without update to signature/memory region

Generating Solutions for Coverpoints

- Has an internal database with all relevant information for an instruction(like fields and their domains).
- Uses the CGF format defined by ISAC to generate tests from coverpoints.



- The coverpoints are modeled as a classical Constraint Satisfaction Problem.
- A CSP solver[2] is then employed to find solutions. The *op_comb* and *val_comb* coverpoints are solved independently. Supports two types of solvers
  - Min-Conflict
  - Backtracking
- Support for skipping the solver when coverpoints are suffixed with *#nosat*. This is useful when the coverpoint defines all the fields of an instruction definitely. Ex - $rs1\_val == 2 and rs2\_val == 5$

[2] *Constraint Satisfaction Problem resolver for Python*. URL: https://github.com/python-constraint/python-constraint.

Test generation

- Solutions from *op_comb* and *val_comb* are interleaved and any additional fields(such as register for signature pointer) etc are filled in.
- Filter the generated test cases using the coverpoints given to ensure that there are none which does not hit at least 1 unique coverpoint.
- Replace relevant values in the assembly macro template for the instruction and generate assembly files.

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

Experimental results

- *cfg1* - Using Min-Conflict Solver on 8 threads
- *cfg2* - Using Backtracking Solver on 8 threads

Suite (ISA)	Generation Time(s)		Average Generation time per test(s)		Test Cases generated		Coverage Collection Time (s)	
	Cfg1	*Cfg2*	*Cfg1*	*Cfg2*	*Cfg1*	*Cfg2*	*Cfg1*	*Cfg2*
RV32I	293.56	87.87	39.32	10.36	12638	13150	189.07	227.43
RV32M	136.24	35.536	93.96	24.99	5165	5386	59.21	65.90
RV32C	80.13	23.80	21.14	4.36	4633	4822	49.07	99.43
RV64I	864.29	215.88	66.60	19.67	17825	19113	348.82	292.70
RV64M	575.89	128.55	240.10	63.25	9571	10393	121.23	112.92
RV64C	319.96	84.24	71.27	10.58	7843	8409	116.42	136.44

Related Work

- Herdt et. al[3] use a similar approach(SMT solver) to generate tests. But the coverpoints are specified in a complex format and coverage specification is dependent on external tools.
- A Mutation[4] based approach relies on an external simulator to produce tests. The same results can be achieved with CTG in lesser time.
- Negative testing[5] is not feasible for AT due to the permissiveness of the ISA. Any unimplemented functionality can result in unpredictable behaviour of the implementation. Furthermore its difficult to recover from such scenarios or replicate it on the golden model.
- No attempt has been made to track data propagation to the signature region in memory.

[3] Herdt, Große, and Drechsler, "Towards Specification and Testing of RISC-V ISA Compliance*".
[4] Herdt et al., "Mutation-based Compliance Testing for RISC-V".
[5] Herdt et al., "Closing the RISC-V Compliance Gap".

Future Work

- Adding support for new and upcoming extensions like bit manipulation, packed SIMD and floating point.
- Testing of privilege architecture specification
- Improve coverage collection time by considering coverpoints as hit/miss instead of counting number of hits
- Support for testing different hazards.