Who tests the TestRIG? Tooling for randomised tandem verification

Peter Rugg, Alexandre Joannou, Jonathan Woodruff, Franz A. Fuchs, Simon W. Moore
University of Cambridge, first.last@cl.cam.ac.uk

Abstract

TestRIG is a framework to test RISC-V implementations first presented at the RISC-V Summit in Zurich in 2019. Since then, the ecosystem has grown, with multiple new implementations integrated and industrial interest. This presentation discusses some improvements to the ecosystem, including mutation-based coverage tooling, features for generating static test suites, and a single-implementation mode that enables more traditional fuzzing. The developments are motivated by testing the Toooba processor, including the CHERI security extensions. This work helps to evolve TestRIG into a tool suite that can increasingly improve assurance in RISC-V designs.

Background

TestRIG is an ecosystem for cross-verifying RISC-V implementations using a standard *RVFI-DII* interface [1]. *Verification Engines* connect to the implementations over this interface: *QuickCheckVEngine* uses Haskell's QuickCheck library to generate tests and automatically shrink any divergences to a minimal reproducer. The RISC-V golden Sail model implements RVFI-DII, allowing implementations to be compared against this (hopefully) correct-by-definition executable simulator [2]. The RISC-V community is in the process of standardising a CHERI extension, adding unforgeable hardware capabilities for memory safety and compartmentalisation [3]. TestRIG is in use to test CHERI in the *Toooba* and *CVA6* processors.

Since initial publication, the TestRIG infrastructure has seen increasing community engagement, including users and contributors from Microsoft Research, lowRISC, and SCI Semiconductor. The repository now links to 10 RVFI-DII-extended implementations, and has several forks from other members of the community. Improved support has been added for RISC-V compressed instructions and other extensions.

Coverage

An important gap in TestRIG so far is a means to measure the effectiveness of Verification Engines. This section introduces a tool to automatically measure this using mutation-based testing.

Coverage measurement is particularly important for directed random verification, as it is otherwise difficult to determine the quality of the distribution of tests produced. Traditional coverage measures properties of the simulator alone, for example whether certain lines of code have been run, or certain state configurations reached. This leaves a gap in assurance: even if code has been run, it may contain bugs that would not be visible in any of the checked outputs of the test. For example, Figure 1 shows a case where a CHERI validity tag gets conditionally cleared: a test may cause this code to be run, but only on already-untagged capabilities, or the capability may be used later in the test, hiding the error.

```
function clause execute(SCMODE(cd, cs1, rs2)) = {
  let cap = C(cs1);
  let mode = execution_mode_encdec(X(rs2)[0 .. 0]);
  let cap = clearTagIf(cap, capIsSealed(cap));
  let hasMode = not(permsMalformed(cap)) & canX(cap);
  let newCap = if hasMode then setCapMode(cap, mode) else cap;
  C(cd) = newCap;
  RETIRE_SUCCESS
}
```

Figure 1: *Auto-generated report showing the results of mutation-based testing of a Sail CHERI function by removing lines of Sail code (just one type of mutation out of several). Blue lines resulted in a model that did not build when deleted, green shows successful detection (the user can click to link to the failing test), and red shows a case where no counterexample was detected. In this case, the user can see that they need to direct random test generation to produce more sealed capabilities. Traditional code coverage would show that this line was run, hiding the blind-spot!*

The Sail model enables us to approach the problem differently: measuring *mutation adequacy* [4] of the TestRIG generators. By introducing artificial bugs into the Sail model, we can assert that the tests definitely do catch those types of bugs. The tests will be run comparing the mutated Sail model against the original, detecting any divergences in outputs. Due to automatic counterexample shrinking, these divergences can typically be made very concise. For example, we can hardcode an `if` condition to `true`. Running this under TestRIG against another unmodified version of the Sail model confirms that the tests relied on behaviour where the `else` was taken. Figure 1 demonstrates the benefits when verifying a CHERI core.

We have implemented a framework to perform this

work as scripts within the TestRIG repository. Users can implement transformations over the Sail model as Python classes that implement two functions: one to find points to transform (e.g. each `if` clause in the model) and one to perform the transformations (e.g. replacing the condition with a hard-coded `true` or `false`). The framework then manages performing the transformations on copies of the Sail model, building them, and running a Verification Engine. Transformations and results of runs are tracked in an SQLite database and can be displayed in HTML to allow easy visual inspection, as seen in Figure 1. While the scripts currently use text-based transformations written in Python, we would ideally hook in to the Sail compiler to perform transformations on the syntax tree. Inlining function definitions could also make coverage measurements more meaningful.

The framework could also be adapted to allow mutations to the RTL of a particular implementation, testing for microarchitectural coverage rather than just model coverage [5].

Test suite generation

There is a useful side effect of the above coverage approach, combined with QuickCheckVEngine's existing shrinking mechanism. Since only a single difference is introduced to the model at a time, the counterexample produced is very targeted to that line of code. With counterexample shrinking, this produces a minimal test case for that line of the architecture, relying on the minimal features needed to test that behaviour. Repeating this allows us to build up a library of traces that test each line of the model. These tests are typically very short for the types of coverage examined so far: a single-digit number of instructions, as opposed to hundreds required for traditional ISA tests. This makes it much easier to diagnose a failure. We conjecture that such tests could form the basis of a comprehensive architectural compatibility suite.

Debugging lockups

An unrelated TestRIG development was motivated by bugs in the Toooba processor causing it to lockup and not retire instructions. This is one of the worst possible failure modes for a processor, likely requiring a hardware reset to recover. We discovered Toooba could lockup by mis-decoding some illegal instructions.

To ensure we had caught all the relevant cases, we added a mode to TestRIG to allow a single implementation to be run alone, without needing to compare to a model. Simply checking that an RVFI report is received for every DII instruction within a timeout suffices to check for lockup conditions, but we also allow templates to specify asserts that can check for arbitrary properties of the resulting trace. Running as a single implementation is important, as we would like to be able to check for lockup conditions without needing to align the implementation and model on all implementation-defined cases of the specification. This then allows us to inject a string of completely uniform 32-bit instructions into the processor. DII is very useful here, as otherwise managing control-flow in the processor would be difficult. This reproduced the decode issues, and confirmed that the problem was resolved following our fixes to Toooba.

This approach surprisingly also found a subtle and rare branch prediction issue in Toooba that could also cause a lockup. The fetch stage could get stuck in a loop, incorrectly predicting that the first instruction is a compressed jump, then redirecting without correctly retraining the branch predictor due to an associativity issue. This shows that the methodology was able to discover behaviours relying on deep and rare conditions. The framework also identified a fatal assert in a version of the Sail model.

Conclusion

We have shown several new tools that add additional capabilities to the TestRIG framework. This closes key gaps, including validation of the tests generated, support for generating minimal static unit tests, and extra tooling for catching lockup bugs. We hope that processor implementers can see ever-greater benefits from joining the ecosystem. All the work is open-source under permissive licenses: we encourage everyone to use it and contribute suggestions and improvements!

References

[1] Alexandre Joannou et al. "Randomized Testing of RISC-V CPUs Using Direct Instruction Injection". In: *IEEE Design & Test* 41.1 (2024), pp. 40–49. DOI: 10.1109/MDAT.2023.3262741.

[2] Alasdair Armstrong et al. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290384. URL: https://doi.org/10.1145/3290384.

[3] RISC-V CHERI extension TG. *RISC-V Specification for CHERI Extensions*. https://github.com/riscv/riscv-cheri.

[4] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software unit test coverage and adequacy". In: *ACM Comput. Surv.* 29.4 (Dec. 1997), pp. 366–427. ISSN: 0360-0300. DOI: 10.1145/267580.267590. URL: https://doi.org/10.1145/267580.267590.

[5] Y. Serrestou, V. Beroulle, and C. Robach. "Functional Verification of RTL Designs driven by Mutation Testing metrics". In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. 2007, pp. 222–227. DOI: 10.1109/DSD.2007.4341472.