Randomized Testing of RISC-V CPUs using Direct Instruction Injection

Alexandre Joannou[1], Peter Rugg[1], Jonathan Woodruff[1], Franz A. Fuchs[1], Marno van der Maas[1], Matthew Naylor[1], Michael Roe[1], Robert N. M. Watson[1], Peter G. Neumann[2], Simon W. Moore[1]

1 - University of Cambridge, 2 - SRI International

Abstract

*We present TestRIG, a test framework for RISC-V implementations. To use TestRIG, a **Direct Instruction Injection** interface is added to the implementation under test. Direct Instruction Injection allows the test framework to inject instructions directly into the processor's pipeline (instead of instructions being fetched from program memory). The Direct Instruction Injection approach simplifies randomized testing, particularly when the test programs contain branch instructions.*

We describe some of the main challenges in randomized testing of CPUs, and explain how TestRIG overcomes them. Finally, we give examples of some hardware bugs that were found using TestRIG, including bugs in the floating point library supplied with the BlueSpec compiler and bugs that were detected during development of the CHERI security extension.

Introduction

TestRIG (Testing with Random Instruction Generation) is a testing framework for RISC-V implementations. The RISC-V community has standardized a formal model[1] of the architecture in the Sail language [1], giving a human-readable specification that can also be used for simulation and verification. Ideally, a RISC-V implementor could formally prove equivalence between their implementation and the Sail model, but proof tools are not yet sufficiently automated to be routinely used on the whole-processor level. As a pragmatic compromise, we use TestRIG to check equivalence between the model and an implementation by generating random instruction sequences, executing the same sequences on the model and the implementation under test, and comparing execution traces (tandem execution). This approach does not prove equivalence but can demonstrate divergence, and is usable in all stages of development.

TestRIG uses the RISC-V Formal Interface (RVFI) standard[2] to observe the change in state after each instruction of the implementation under test, and uses a novel technique that we are calling Direct Instruction Injection (DII) for test injection. In normal program execution, the next instruction is fetched from program memory at an address determined by the program counter. With Direct Instruction Injection, the next instruction to be executed is provided by the test harness, regardless of the CPU's program counter.

We are not testing completed, fabricated chips. Rather, we are comparing executable formal models, software ISA simulators and simulated execution of hardware designs. This requires us to instrument the CPU design with an additional interface for Direct Instruction Injection used by the test harness during tandem verification.

We have added the Direct Instruction Injection interface to the Sail RISC-V formal model, and to two high-performance emulators: Spike[3], and QEMU[4]. We have also instrumented four RISC-V processor implementations with RVFI-DII, spanning from embedded to superscalar. We have used TestRIG to test many standard RISC-V extensions, and the experimental CHERI security extension.

We found TestRIG to be easier to use than unit tests, since instructions can be tested as they are implemented without supporting a full testing framework. We also found that TestRIG gave more thorough test coverage due to random generation replacing developer effort to explore possibilities. It is effective at detecting not just issues in instruction semantics, but also in the pipeline and the data caches. As a result, TestRIG has completely replaced our instruction-set level unit testing for development.

TestRIG framework

TestRIG is designed as a modular ecosystem: an interactive Verification Engine (VEngine) stimulates RISC-V implementations over RVFI-DII sockets. An RVFI-DII compatible RISC-V implementation can reset, consume instruction sequences, and report execution traces via its RVFI-DII interface.

A VEngine can drive one or more RVFI-DII compatible implementations; a VEngine might have an internal RISC-V model, or could drive two independent implementations and

[1] https://github.com/riscv/sail-riscv
[2] https://github.com/SymbioticEDA/riscv-formal
[3] https://github.com/riscv-software-src/riscv-isa-sim
[4] https://www.qemu.org

compare their RVFI traces, as we have done with QCVEngine. VEngine instruction sequences could be loaded from disk, generated randomly, or produced with interactive architecture-driven state-space exploration.

The RVFI-DII bytestream interface allows models and implementations written in various languages to communicate through widely supported networking sockets. QCVEngine is written in Haskell, and the Sail RISC-V model is written in Sail (offering OCaml and C backends). Spike and QEMU are RISC-V simulators written in C and C++. Hardware implementations that support RVFI-DII, including RVBS[5], Ibex[6], Piccolo[7], Flute[8], and RiscyOO[9] are written in either SystemVerilog or Bluespec, although this is not required for TestRIG.

RVFI-DII

To participate in the TestRIG verification ecosystem, implementations must be extended with RVFI-DII instrumentation. The RISC-V Formal Interface (RVFI), specified by Claire Wolf, is an existing trace format for formal verification using symbolic instructions. RVFI exposes select architecturally significant signals such as the instruction encoding and any memory address or value, as well as the indices and values of the operand and writeback registers.

TestRIG extends RVFI with Direct Instruction Injection (DII). DII is for instruction input, RVFI is for trace output, and RVFI-DII supports full interactive verification. DII directly specifies the instruction sequence expected in the output trace, and does not associate instructions with memory addresses. This requires custom pipeline instrumentation, but enables greatly simplified sequence generation and shrinking, as the program counter does not affect the instruction stream. Existing RISC-V cores that implement RVFI can be augmented to participate in the TestRIG ecosystem by implementing DII, and conversely RVFI-DII designs may benefit from RVFI formal verification tooling.

QuickCheck VEngine

Our TestRIG Verification Engine, QCVEngine, leverages Haskell's QuickCheck library [2]. Due to the simplicity of DII execution, which decouples the instruction stream from control flow, QCVEngine can use unmodified QuickCheck utilities to generate, compare, and shrink instruction sequences.

QuickCheck receives a function with a pass/fail return value, and generates inputs in search of a failure. To facilitate this, we construct a function that receives a list of instructions, sends these over two DII sockets, collects RVFI traces back from these sockets, asserts they match, and returns the result.

We then provide a set of generators of arbitrary instruction sequences that are used by QuickCheck to produce inputs to this function. We use convenience functions to define instructions in a syntax closely resembling the RISC-V ISA manual, and provide tailored generators for each instruction field to promote register reuse. QuickCheck automatically uses these generators to construct arbitrary instruction sequences. We also provide targeted generators for simple subsets of the instruction set, as well as generators that leverage templates of varying complexity to reach deeper states, including virtual memory mappings and cache conflicts.

We also develop a mechanism to allow semantic shrinking of counterexample traces, beyond QuickCheck's default of deleting instructions.

Evaluation

We have measured functional coverage of TestRIG over the Sail model compared to the RISC-V test suite[10] and RISCV-DV[11], finding the coverage broadly comparable. Further work is needed to develop templates that cover the architecture more thoroughly. Due to trace shrinking, the counterexamples produced are orders of magnitude shorter than those produced by the other methods, significantly speeding up debugging cycles.

Several significant bugs have been discovered using QCVEngine and TestRIG, spanning architectural and microarchitectural errors in codebases of varying maturities. We will also discuss its application to debugging the CHERI security extension [3] and for measuring transient execution vulnerabilities.

References

1. Alasdair Armstrong et al. Isa semantics for armv8-a, risc-v, and cheri-mips. Proc. ACM Program. Lang., 3(POPL), Jan 2019.
2. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. Acm sigplan notices, 46(4), 2011.
3. Robert N. M. Watson et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, Oct 2020.

[5] https://github.com/CTSRD-CHERI/RVBS
[6] https://github.com/lowRISC/ibex
[7] https://github.com/bluespec/Piccolo
[8] https://github.com/bluespec/Flute
[9] https://github.com/csail-csg/riscy-OOO
[10] https://github.com/riscv-software-src/riscv-tests
[11] https://github.com/google/riscv-dv