Randomized Testing of RISC-V CPUs using Direct Instruction Injection

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

I. Introduction

TestRIG (Testing with Random Instruction Generation) is a testing framework for RISC-V implementations. The RISC-V community has standardized a formal model of the architecture in the Sail language[1], giving a human-readable specification that can also be used for simulation and verification. The Sail language is designed for this purpose by allowing instruction semantics to be described conveniently (for example, by supporting variable bit-widths). 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. They instead focus on equivalence between combinational logic functions, with verification of a full out-of-order microarchitecture remaining an open problem. 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[3], and to two high-performance emulators: Spike[4], and QEMU[5]. We have also instrumented four RISC-V processor implementations with RVFI-DII, spanning from embedded to superscalar implementations. 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.

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

II. The Dream – Model-based Verification

Architectural extensions are traditionally specified starting with a prose specification, and then four implementations are produced largely independently:

1) Assembler
2) Executable model (simulator)
3) Instruction-level unit-test suite
4) Hardware implementation

While this ensemble of implementation efforts is laborious when done once, its greatest cost lies in discouraging design exploration; design changes require consistency among five independent code bases.

A formal, executable instruction-set architecture (ISA) specification can greatly simplify this workflow. We use the Sail [3] domain-specific language, which features human-readable syntax. Sail excerpts serve as pseudocode in our ISA documents [15]. Sail also produces a simulator (item 2), and will eventually provide verification (item 3) and the assembler (item 1) to be derived from it automatically.

A. Model-based Formal Verification

Formal verification tools for RISC-V have often used the RVFI tracing (see Section IV) interface along with tools like Cadence's JasperGold, to prove that a series of traces from a simple HDL model is equivalent to a series of traces from a pipelined HDL implementation. Unfortunately, these tools can handle only in-order pipelines, and require specialist knowledge. As a result, the formal-verification approach does not yet replace functional testing for entire processors.

B. Model-based Random Testing

While formally proving equivalence for complex microarchitectures has been elusive, pragmatic tools have used other

ways to detect divergence from a model. These approaches cannot *prove* equivalence between a formal model and an implementation, but can refute it with counterexamples.

For example, directed-random test-sequence generation has been used to debug pipeline and memory bugs, as well as to uncover unexpected divergences in implementation behavior [1], [13]. There exist multiple test generators for RISC-V, e.g., RISC-V RTG [14], but RISCV-DV[6] remains the most advanced such sequence generator for RISC-V, and it works well for these use cases, particularly where detailed traces can be compared. RISCV-DV generates assembly programs, ready to be converted to in-memory images for execution. RISCV-DV includes a number of test generators for RV32IMAFDC and RV64IMAFDC – including support for page-table interactions, privileged CSR use, and handling traps/interrupts. These generated test programs are executed on both a golden model and a processor in development. A RISCV-DV test framework would typically detect a divergence by comparing the execution traces.

Although randomly generating tests is a promising approach, it can have several drawbacks:

- Automatically generated counterexamples can be long and convoluted, while hand-written tests can be made short and easy to understand.
- The generator must ensure that useful instructions are found at the target of each randomly generated branch.

Automated reduction of failing test cases has previously been used in software testing. For example, *C-Reduce* [10] can take a program that triggers a bug in a C compiler and reduce it to a minimal example that triggers the bug.

PyH2P [7] applies automated test case reduction randomly generated RISC-V instruction sequences. PyH2P often produces test sequences that contain less than 5 instructions, with every instruction being meaningful for reproducing the error. Nevertheless, PyH2P has three shortcomings:

1) PyH2P does not perform full trace comparison with its internal PyMTL3 model, but only with final memory and register state.
2) PyH2P has difficulty shrinking through branches, as it must produce a valid in-memory program.
3) PyH2P does not use community-standard interfaces that have been proven across a range of implementations.

PyH2P points in an encouraging direction, and TestRIG matures the approach, proposing a standardized communication interface so that verification engines (VEngines), models, and implementations are interchangeable and can be improved independently. Additionally, instruction injection allows straightforward shrinking of sequences with branches. This has allowed us to completely replace instruction-level unit tests for the sophisticated CHERI extension [16], greatly improving both productivity and assurance, and enabling extension of an array of simulators and processors more efficiently than the CHERI implementations on MIPS or ARM.

Symbolic QED [12] is another approach that generates minimal tests for verification (including post-silicon) using a formal model of the pipeline.

[6]https://github.com/google/riscv-dv

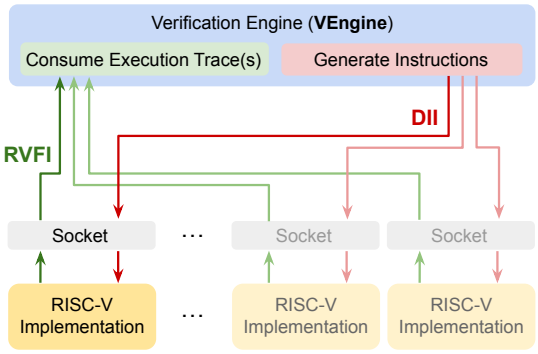

Fig. 1. An illustrative example of the TestRIG ecosystem with a Verification Engine communicating with any two RISC-V implementations over sockets. The Verification Engine injects instruction sequences and compares the execution traces until it finds a divergence.

III. TestRIG

Figure 1 gives an overview of the modular TestRIG ecosystem. In TestRIG, an interactive Verification Engine (VEngine) stimulates RISC-V implementations over RVFI-DII sockets, which are detailed in Section IV. 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, similar to PyH2P, or could drive two independent implementations and compare their RVFI traces, as we have done with QCVEngine, which is presented in Section V. 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 the Sail domain-specific language (either interpreted by an OCaml program or compiled into C). Spike and QEMU are RISC-V simulators written in C and C++. TestRIG also supports hardware implementations like RVBS, Ibex, Piccolo, Flute, and Toooba, which are written in either SystemVerilog or Bluespec. RVBS[7] is a reference implementation, Ibex[8] and Piccolo[9] are simple 32-bit implementations, Flute[10] is a 5-stage in-order pipeline processor implementing RV64, and Toooba[11] is a RISC-V 64-bit superscalar out-of-order processor.

Participants in the TestRIG ecosystem are expected to be identical in every architecturally visible way. Besides a RVFI-DII interface, TestRIG requires 8 MiB of memory accessible at address 0x80000000 (all other addresses returning an access fault), and must support resetting to a known state (zeroed registers, known default values for RISC-V control and status

[7]https://github.com/CTSRD-CHERI/RVBS
[8]https://github.com/CTSRD-CHERI/ibex
[9]https://github.com/CTSRD-CHERI/Piccolo
[10]https://github.com/CTSRD-CHERI/Flute
[11]https://github.com/CTSRD-CHERI/Toooba

registers, zeroed 8 MiB of memory) upon injection of a "reset" DII packet.

IV. RVFI-DII

To participate in the TestRIG verification ecosystem, implementations must be extended with RVFI-DII instrumentation. To ease development, we distribute data structures and libraries in several languages to facilitate RVFI-DII connections over TCP ports.

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. Interactive verification enables automated simplification and shrinking, as discussed in Section V-A. 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.

Not all architectural updates are reported in the RVFI interface, e.g., floating-point registers and extended CHERI capability registers. While this is a limitation, PyH2P relies only on final register and memory state and is still able to usefully detect divergence. We found that occasional instructions that move unexposed values into RVFI-visible state could produce sufficiently succinct counterexamples. This strategy was also used in RVFI formal verification efforts.

An RVFI interface exports internal signals of an RTL design, or internal variables of a simulator or emulator. For more complex RTL designs, such as pipelined or superscalar microarchitectures, extracting the appropriate values may require preserving state for an RVFI report in a commit/write-back stage that did not previously have access to them. Extending the superscalar Toooba core for RVFI-DII required two extra records for each instruction in the Reorder Buffer.As these records are present only when built for simulation with RVFI, this is not a physical overhead for the design.

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.

A DII interface receives a reset command followed by a sequence of instructions. A Bluespec implementation of this interface is shown below:

```
typedef struct {
  Bool     rvfi_cmd;  // Instruction or reset command?
  Bit#(10) rvfi_time; // Time to inject instruction
  Bit#(32) rvfi_insn; // Instruction word (32/16 bit)
} RVFI_DII_Instruction
```

For an emulator, this interface simply replaces each fetched instruction with an encoding from the DII queue. For RTL designs, DII support is more complex. An RTL design can remove the instruction cache entirely (but not address translation of the PC, which is architecturally visible) to ensure maximal pipeline packing, or can exercise the instruction cache and replace the bytes of the instruction after they have been fetched. RISC-V compressed instructions present another choice: to substitute picked instructions before decode, or inject 16-bit instruction fragments from DII to exercise the picking logic. The simple single-issue design of Piccolo and Flute enabled us to replace the cache entirely with a DII queue that delivered one instruction every cycle, either compressed or uncompressed. For superscalar Toooba, we began with unmodified instruction-cache access, substituting the vector of picked instructions before decoding. In an effort to debug instruction picking itself, we later moved to bypassing the instruction cache and providing 16-bit instruction fragments to the pipeline, relying on the instruction picker and decode to reconstitute the correct DII instruction sequence.

Canceled instructions present a further challenge to DII. Synchronization is required when instructions are dropped in the pipeline, as RVFI-DII requires a single RVFI trace entry for each DII instruction injected. While adding RVFI-DII to Flute, we arrived at a mature design that attaches a sequence ID to each RVFI instruction and carries it with the PC through the pipeline. Instruction Fetch actively requests each instruction ID from the DII sequence (as with PC requests to the cache), allowing pipeline redirects to work naturally. We adapted this approach to Toooba by adding superscalar fetch and assigning IDs to compressed instruction fragments. This more capable DII unit is available in our RVFI-DII libraries[12], and has been backported to Flute. While DII instrumentation may appear daunting, we have found that beginning with this mature strategy greatly reduces both implementation effort and design disturbance. In retrospect, the few hours invested in this implementation have greatly streamlined the otherwise much longer testing phase.

V. QuickCheck VEngine

Our TestRIG Verification Engine, *QCVEngine*, leverages Haskell's QuickCheck library [5]. Due to the simplicity of Direct Instruction Injection 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 that 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 discovers and uses

[12]https://github.com/CTSRD-CHERI/BSV-RVFI-DII

these generators through the type system and uses them 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. Templates are a common tool for random test generators; for example, IBM's Genesys-Pro [1] is built on templates to intelligently solve for desired deep states.

A. Smart Shrinking

While Direct Instruction Injection allows us to primarily rely on QuickCheck's builtin shrinking strategies, we augmented these with *smart shrinking* functions that not only eliminate instructions, but intelligently transform them to simplify the sequence.

Once a counterexample is found by QCVEngine, QuickCheck uses a builtin list-shrinking function that removes sequences from the list and tests again, hoping to eliminate instructions with no relevance to the errant behavior. Illustratively, here is an initial counterexample found for an artificial hardware bug where the LSB of the `add` instruction's result (but not `addi`'s) is stuck at zero:

```
addi x7, x4, 123 # Generate odd immediate
addi x5, x3, 42  # Generate even immediate
addi x6, x7, 0   # Move x7 to x6
xori x1, x5, 745 # Irrelevant
add  x1, x5, x6  # Perform buggy add
```

The builtin list shrinking results in:

```
addi x7, x4, 123 # Generate odd immediate
addi x6, x7, 0   # Move x7 to x6
add  x1, x5, x6  # Perform buggy add
```

The middle instruction can also be eliminated if the final `add` takes register `x7` as an operand directly. To automate this functionality, we further augment shrinking to intelligently propagate an instruction's output register to future input operands. This allows another pass of list-shrinking to further reduce the counterexample:

```
addi x7, x4, 123 # Generate odd immediate
add  x1, x5, x7  # Perform buggy add
```

We also add a library of simplifications to be used during shrinking. These eliminate esoteric instructions from the trace that perform mundane functions and distract from the root cause of the failure. For example, memory operations often trap; thus, we might attempt to simplify a memory operation to an *ecall*, an instruction that only traps, to make the error more obvious.

Any shrinking or simplification is safe to try for model-based testing; any change that still diverges is kept. In rare circumstances, the shrinking may reveal an alternative bug, obscuring the original, but still producing a useful result.

B. Sequence Import/Export

Instruction traces can be converted to (and from) a human-readable format both for terminal reporting, and for reading and writing trace files – individually or in bulk from a directory. This has enabled us to collect a library of regression tests to quickly check all previous counterexamples. Unlike hand-written tests with assertions, these do not require maintenance, as expected behavior updates naturally with the model as the instruction set evolves. We have also used this feature to replay recorded test-suite examples (including *riscv-tests* and *RISCV-DV*), adding full trace-equivalence check with shrinking. This feature has also allowed us to capture traces of an operating system booting on the model implementation, which we could then use to aid bring-up of the same operating system on implementations, with instruction shrinking rapidly highlighting any problems.

C. Non-shrinkable Sequences

Sequences can be annotated as non-shrinkable. This has been used to force initialization to cover divergences in initial state. For example, one implementation did not initialize floating-point registers, which produced trivial counterexamples. A non-shrinkable initialization sequence allowed us to progress to interesting divergences in exception conditions and rounding modes.

D. Assertions

Sequences can include assertions – e.g., that the value written by the previous instruction was non-zero. These make it possible to fail without a divergence. Unusually, sequences with assertions do not require tandem verification to discover a failure, and we have used these to test the limits of implementation-defined behavior.

VI. Evaluation

A. A Coverage Study

Architectural coverage is the first metric for basic verification. We evaluated coverage of the RISC-V architecture using *sailcov*[13], which measured how many branches of the RISC-V Sail model were explored during a run. We compared our TestRIG QCVEngine against the RISC-V test suite (*riscv-tests*[14]) and the RISCV-DV generator.

For our coverage study we conduct two runs of each testing framework (QCVEngine, riscv-tests, and RISCV-DV) for RV32IMC and RV64IMAFDCZicsr. For RV32IMC, we take the Sail RISC-V model coverage of the I, M, and C extension instructions as well as the coverage of the general-purpose registers. For RV64IMAFDCZicsr, we measure the coverage of I, M, A, F, D, C, and CSR instructions as well as the coverage of the general-purpose and floating-point registers. For riscv-tests, we measure the coverage of the Sail RISC-V model running the test binaries. For RISCV-DV, we produce TestRIG traces from the Spike simulator executing the tests and replay them through RVFI-DII while measuring the coverage of the Sail RISC-V model. For QCVEngine, we configure it with the two architecture strings and let it run 500 sequences of each generator. The RV32IMC results are similar across all three testing frameworks, indicating that QCVEngine can support a suitable alternative to unit testing and torture testing, at least with respect to breadth of coverage. The RV64IMAFDCZicsr results more variance, but all three

[13]https://github.com/rems-project/sail/tree/sail2/sailcov
[14]https://github.com/riscv-software-src/riscv-tests

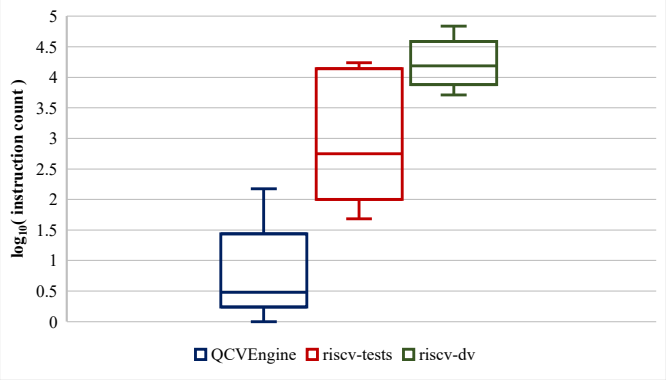

Fig. 2. Counterexample size complexity

remain comparable except in floating point register coverage, in which RISCV-DV excels and CSR instructions in which QCVEngine excels. QCVEngine chooses from a subset of floating-point registers to increase the probability of operand reuse, at a cost to overall coverage. As failures based on register number are rare, we have made a trade-off between increasing the probability of finding violations that require multiple permutations and increasing the overall FD register coverage.

B. Counterexample Complexity

Counterexample complexity is another useful metric. Our archive of QCVEngine traces comprises 3509 shrunken counterexamples discovered during development of our CHERI processor extensions. Figure 2 shows the distribution of our archive of counterexample lengths versus riscv-test and riscv-dv trace lengths, which do not allow shrinking. The median value is 3 instructions, and the third quartile is only 5 instructions. The median for riscv-tests 561 instructions, which is more than 3 times the maximum counterexample found by QCVEngine, and the median riscv-dv sequence is 15339. QCVEngine's small counterexample size is facilitated by Direction Instruction Injection and smart shrinking as described previously. Single digit counterexample length greatly accelerated our discovery of failures and development of fixes compared to even a traditional unit test suite.

VII. Illustrative Cases

TestRIG is useful for a broader range of verification than instruction-level unit tests and improves productivity in all cases. Architectural bugs, which are traditionally targeted by hand-written test suites, are usually discovered quickly with TestRIG. Microarchitectural mistakes, such as register forwarding or pipeline-flush problems, are also discovered quickly and deterministically in TestRIG, but are difficult to anticipate and target in unit-test suites. Memory mistakes, such as cache bugs or memory speculation failures, have also been discovered efficiently with targeted generators, and are notoriously hard to discover using static unit tests. Finally, TestRIG has found *unexpected interactions*, where architectural features interact in unforeseen ways, while unit-test suites are unlikely to test these cases. We have applied TestRIG not only to RISC-V, but also to CHERI-RISC-V (our security extension, noted below).

A. CHERI Introduction

Capability Hardware Enhanced RISC Instructions (CHERI) is a security extension of conventional Instruction Set Architectures that adds *capabilities* – unforgeable and bounded tokens. A capability is a fat pointer [8] containing the address and metadata including permissions and bounds information. Furthermore, validity of capabilities is ensured by a hidden tag. A capability authorizes access to a region of memory, and no data or instruction access is possible without a valid capability. Furthermore, all capability operations are monotonic and therefore cannot increase the privileges a capability grants. As a result, CHERI enforces spatial safety, enables temporal safety, and supports fine-grained software compartmentalization.

B. Architectural Bug

When developing the compressed encoding of CHERI capabilities, we had a bug that unnecessarily cleared the tag of a pointer when setting an address that wrapped the address space. This bug was found with this shrunken counterexample:

```
cSetBoundsImmediate x3, x1, 1106 # Set a short bound
cIncOffsetImmediate x2, x4, -197 # Small negative integer
cSetAddr x4, x3, x2 # Set the integer as the address
cGetTag x1, x4 # XXX Untagged
```

While this case may have been covered in an extensive unit-test suite composed at significant effort, our TestRIG generator required only a list of CHERI instructions to produce a counterexample far more compact than most hand-written tests.

C. Microarchitectural Bug

We have also used TestRIG for discovering microarchitectural vulnerabilities in transient execution. One such generator produces a sequence of arbitrary instructions, followed by an assertion that no additional cache misses were counted, which would indicate a transient violation of the capability system. The following shrunken counterexample demonstrated a vulnerability in `cSetBoundsImmediate`:

```
.noshrink
... # initialize counters
... # bound x31 to 8 bytes
.shrink
# Illegally increase the bound on a pointer
cSetBoundsImmediate x31, x4, 797
# Load through this illegal capability
lb.cap lb.cap x31, x31[0]
... # Delay for counter to propagate
.noshrink
csrrs x30, hpmcounter3 (0xc03), x0 # Read L1 cache miss
.assert rd_wdata == 0x0 ""
```

Because CHERI only allows bounds to be reduced, the `cSetBoundsImmediate` instruction is illegal and throws an exception due to attempting to enlarge the bounds. Nevertheless, the capability that would have been produced is forwarded in the pipeline during the flush and causes a cache fill that could lead to side-channel attacks.

This sequence uses both `.noshrink` and `.assert`. The former is required to initialize the state of the counters so that the final assertion on the L1 cache miss counter is deterministic.

D. Cache Bug

We received Flute as a working in-order RV64G design, and discovered that the data cache was direct-mapped and 4 KiB, rather than 2-way associative and 8 KiB – as specified. An experiment with parameters confirmed that the 2-way cache could not boot the operating system. This bug had not been found with the unit-test suite, so we used a generator that constructs addresses within the TestRIG memory range (see Section III), as well as random loads and stores. This generator quickly discovered the bug with the following shortened sequence, after 42 tests and 20 rounds of shrinking:

```
lui  x1, 262148
slli x1, x1, 1
lui x20, 262148 # Value used as data
ori  x3, x1, 1  # A page address
lui  x2, 262148
slli x2, x2, 1  # The same page address
lhu  x4, x3[1]  # Load from address
sh  x20, x2[2]  # Store to an overlapping byte
lhu  x2, x3[1]  # Divergence on reloading
```

The final sequence contains only three memory operations: two loads with a single store in between, all to overlapping addresses. This counterexample was found less than 10 seconds into the TestRIG run, and was fixed within the hour. The fix is reproduced below:

```
Bool hit = False;
for (Integer way = 0; way < num_ways; way = way + 1)
  begin
    Bool hit_at_way  = (tags[way].state != EMPTY)
                    && (tags[way].tag == pa_tag);
    hit = hit || hit_at_way;
    if (hit_at_way) // XXX This line was missing!
      way_hit = fromInteger (way);
  end
```

While this bug was trivial to resolve with a TestRIG counterexample, it had escaped the entire development process of the Flute processor. It was not found with the RISC-V unit-test suite and was overwhelmingly difficult to debug from a full software trace.

VIII. Future of TestRIG

Despite having an array of models, simulators, and implementations supporting RVFI-DII, the generators of our initial TestRIG verification engine, QCVEngine, are still rudimentary.

QCVEngine Generators: The Haskell infrastructure in QCVEngine supports rich and complex generators. However, the generators for virtual memory, cache testing, and floating-point operations can be enriched with more intelligent directed-random templates for reaching deeper states.

Memory Concurrency Testing: TestRIG should support memory-model testing. RVFI-DII instruction streams injected with specified timestamps into multiple shared memory cores should allow precise stimulation of concurrency behaviors. These would require a more advanced verification engine that tests RVFI traces not only for equivalence, but also against higher-level memory-model semantics – as in Axe [9].

Pipeline Performance: Similarly, a higher-level model of pipeline scheduling and performance could be used to analyze the timing of instruction traces committed in a pipeline to discover performance bugs and track performance improvements. The high level of control possible with direct instruction injection should enable precise detection of performance anomalies.

Model-derived Engine: TestRIG's modular design enables a variety of engines to drive RVFI-DII compatible RISC-V implementations. With QCVEngine, the test maintenance burden is greatly simplified, but not entirely eliminated. Past experience suggests that even deep-state tests can be automatically generated from a model specification, as with IBM's Genesys [1]. Previous CHERI work used tests generated from a formal model of our CHERI-MIPS ISA (written in the L3 [6] specification language), compiling from L3 to HOL4, and then using constraint solving to automatically generate instruction sequences to reach a desired state without triggering undefined behavior. This approach has also been applied to the CHERI ARM Morello instruction set starting from a Sail model [4], [11]. Brian Campbell, a key contributor to this work, has also begun on a Sail-OCaml VEngine with direct access to the data structures of our Sail RISC-V model. This eliminates independent encodings in the VEngine, and we expect this approach to be taken further to automate generation of templates that target specific deep states in the architectural model using constraint solving.

IX. Counterexample-driven Development

TestRIG's model-based testing leads to counterexample-driven development, an advancement over test-driven development, a widely known technique of software engineering. Typical test-driven development for processor design requires a basic working design before architectural unit tests can be used. Counterexample-driven development using TestRIG can automatically provide reduced stimulus for the most basic features and can carry development all the way to advanced interactions. The CHERI extension to Ibex was a striking example. After extending Ibex with RVFI-DII support, a summer intern was able to independently add full CHERI functionality to Ibex in a month, due to the tight cycle of reduced counterexamples provided by QCVEngine.

X. Conclusion

We have collated all the current TestRIG-compatible implementations and verification engines into the open-source TestRIG repository[15]. This repository includes documentation that has been followed and improved multiple times by new users. TestRIG accelerates development at all stages, providing a tighter debugging loop than we have experienced in any other processor development paradigm. We expect TestRIG to lead the way to a standardized testing framework for RISC-V that leverages instrumentation of open implementations, to greatly simplify verification. Such a framework improves upon traditional instruction-set-level unit testing in every way, and subsumes specialized random test generators into a cohesive community of easy-to-use verification tools.

[15]https://github.com/CTSRD-CHERI/TestRIG

ACKNOWLEDGEMENTS

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA), under contract HR0011-18-C-0016 (“ECATS”) as part of the DARPA SSITH research program. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. This work was partially supported by the Innovate UK, Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694). For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to the accepted version of this manuscript.

REFERENCES

[1] Allon Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2), 2004.

[2] Merav Aharoni, Sigal Asaf, Laurent Fournier, Anotoly Koifman, and Raviv Nagel. FPgen–a test generation framework for datapath floating point verification. In *Eighth IEEE International High-Level Design Validation and Test Workshop*, pages 17–22, 2003.

[3] Alasdair Armstrong et al. Isa semantics for armv8-a, risc-v, and cherimips. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[4] Thomas Bauereiss et al. Verified security for the morello capability-enhanced prototype arm architecture. Technical report, University of Cambridge, Computer Laboratory, 2021.

[5] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4), 2011.

[6] Anthony Fox. Directions in ISA specification. In *ITP*, 2012.

[7] Shunning Jiang et al. PyH2: Using PyMTL3 to create productive and open-source hardware testing methodologies. *IEEE Design & Test*, 38(2), 2020.

[8] Trevor Jim et al. Cyclone: A safe dialect of C. In *ATEC 2002*, Berkeley, CA, USA. USENIX.

[9] Matthew Naylor et al. A consistency checker for memory subsystem traces. In *FMCAD 2016*. IEEE.

[10] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *33rd ACM SIGPLAN Conference om Programming Language Design and Implementation (PLDI 2012)*, 2012.

[11] Peter Sewell. Engineering with full-scale formal architecture: Morello, cheri, armv8-a, and risc-v. In *FMCAD 2021*. IEEE.

[12] Eshan Singh, David Lin, Clark Barrett, and Subhasish Mitra. Logic bug detection and localization using symbolic quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[13] Shajid Thiruvathodi and Deepak Yeggina. A random instruction sequence generator for arm based systems. In *2014 15th International Microprocessor Test and Verification Workshop*. IEEE.

[14] Dai Duong Tran, Thi Giang Truong, Truong Giang Do, and The Duc Do. Risc-v random test generator. In *2021 15th International Conference on Advanced Computing and Applications (ACOMP)*, pages 150–155, November 2021.

[15] 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, October 2020.

[16] Jonathan Woodruff et al. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA 2014, Minneapolis, MN, USA*. IEEE.