TestRIG: Equivalence Testing for RISC-V Models and Implementations

Jonathan Woodruff, Alexandre Joannou, Peter Rugg

Hongyan Xia, James Clarke, Hesham Almatary, Prashanth Mundkur, Robert Norton-Wright, Brian Campbell, Simon Moore, Peter Sewell

University of Cambridge and University of Edinburgh
RISC-V Workshop - 12 June 2019
Unit Tests For Architectural Compliance

• Labor-intensive to compose
• Maintenance to synchronize with specification and implementation
• Not sufficient for microarchitectural verification, or even full architectural compliance
  • Every condition?
  • Every value?
  • Every side-effect?
  • All forwarding paths?
  • All cache interactions?
  • Every exception condition?
Model Equivalence Verification

Test failure → Divergence from model

- Directed-random test generation
  - Generate instruction sequences
  - Compare traces or output with model
- Formal model checking (RVFL – Thanks Clifford Wolf!)
  - Instrument + unroll pipeline
  - Prove equivalence with model

Why not just use model equivalence? Too hard!
What if it was easier than unit test composition?
Direct Instruction Injection (RVFI-DII)

1. Simple instruction generation
2. Deterministic pipeline injection timing
   *helps catch pipeline and cache bugs!*
3. Automated counter-example reduction

```cpp
genArithmetic :: Template
genArithmetic = Random $ do {
  return $ Distribution [
    (8, Single $ encode add  src1 src2 dest),
    (8, Single $ encode slt src1 src2 dest),
    (8, Single $ encode sltu src1 src2 dest),
    (8, Single $ encode and src1 src2 dest),
    (8, Single $ encode orr src1 src2 dest),
    (8, Single $ encode xor src1 src2 dest),
    (8, Single $ encode sll src1 src2 dest),
    (8, Single $ encode srl src1 src2 dest),
    (8, Single $ encode sub src1 src2 dest),
    (8, Single $ encode sra src1 src2 dest),
    (16, Single $ encode add $ imm src2 dest),
    (8, Single $ encode slti imm src2 dest),
    (8, Single $ encode slti imm src2 dest),
    (8, Single $ encode andi imm src2 dest),
    (8, Single $ encode ori imm src2 dest),
    (16, Single $ encode xor src $ imm src2 dest),
    (8, Single $ encode sll $ imm src2 dest),
    (8, Single $ encode srl $ imm src2 dest),
    (8, Single $ encode sra $ imm src2 dest),
    (16, Single $ encode lui $ longImm dest)
  ]
}
TestRIG Framework

Three interchangeable parts:

• Verification Engine, “VEngine”
  Generates interesting sequences
• Model
  Executable specification, or known-good implementation
• Implementation
  (Models and implementations are interchangeable)

Build community around model-based verification!
Verification Engines

Haskell VEngine

- Uses QuickCheck
  - generate sequences
  - automatic test-case reduction
- Uses InstrCodec library for simple instruction definition
- 5 instruction mix generators so far: integer, memory, flow control…
- Templates to reach deep states

OCaml VEngine

- Infers instruction definitions from the specification itself
- Will use analysis of the specification to generate sequences that target interesting states
Sail RISC-V Executable Model

Design Goals:

- **Readable**
  (and writable!)
- **Executable**
- **Provable**

- RISC-V specification in Sail language
- Theorem prover backends (SMT, HOL4, etc)
- C executable model for tracing
RVFI-DII: Instruction Injection with Tracing

Implementation instrumentation (the price to pay for simplified verification)
TestRIG Development: Counter-example Reduction

- Random sequences of variable depth
- Detect divergence in trace
- Shorten sequence, iterate
- Report shortest diverging sequence

```
sub   r0, r3, r2
slli  r2, r3, 14
slli  r4, r1, 21
ori   r5, r4, 2631
lui   r2, 1041374
xor   r4, r0, 4000
slli  r3, r4, 24
add   r5, r5, r3  // divergence detected
xor   r0, r0, r5
```
Conclusion

• Replace test suite construction with model building!
  (And instruction sequence generator design)
  (And unit proofs of model)

• Potential for better verification with one tool:
  • Architectural correctness
    Including extensive value coverage!
  • Pipeline forwarding verification
  • Cache hierarchy verification

https://github.com/CTSRD-CHERI/TestRIG