GRIFT:
A richly-typed, deeply-embedded RISC-V semantics written in Haskell

Ben Selfridge, Galois Inc.
September 13, 2019
Overview

• RISC-V & the RISC-V Formal Specification Task Group

• GRIFT walkthrough and demo

• Questions
RISC-V

- Open source instruction set architecture developed at UC Berkeley
- Attempts to avoid cruft and stagnation of proprietary ISAs
  - Base instruction set (“I”) is very small (~40 instructions)
  - Other instructions available via *extensions* (M, A, F, D, C, …)
- Overall design emphasizes *configurability* across various parameters (register width, available extensions)
  - Example configurations: RV32I, RV64IMAFDC, etc.
- Evolution of RISC-V standards and artifacts is stewarded by the RISC-V Foundation via a number of “task groups”
RISC-V Formal Specification
Task Group

• Goal: Develop a formal specification of the RISC-V instruction set architecture that is:
  
  • Precise/unambiguous
  
  • Readable (as text)
  
  • Executable
  
  • Useful for hardware engineers and formal methods engineers
GRIFT

• “Galois RISC-V Formal Tools” (GRIFT) is our contribution to the RISC-V Formal Spec Group

• Formalizes instruction encoding and semantics in *embedded domain-specific language* (eDSL) within Haskell, as a library

• Includes **command-line tools** for simulation, coverage analysis, and documentation/pretty printing
GRIFT: Design goals

- Express RISC-V configuration in Haskell’s type system (compile-time guarantees)
- Express encoding and semantics in an embedded DSL to allow translation to other languages and environments
- Represent core ISA as data, rather than Haskell functions, so it can be manipulated directly and translated into other environments
instruction data (encoding/semantics)

expressed in

GRIFT DSL

interpreted/translated

- grift-sim (simulation, coverage)
- grift-doc (pretty-printed docs)
- Coq/Verilog backends
- Test generation
Type-level RISC-V Configuration

• RISC-V configurability:
  • 32-bit/64-bit
  • Base ISA + extensions (M, A, F/D, C, …)

• We capture the configuration of a RISC-V system as a type parameter for our core data types

• Instructions in a particular extension can only be used if it is known that the configuration supports that extension
Type-level RISC-V Configuration

```haskell
data RV = RVConfig (BaseArch, Extensions)

data BaseArch = RV32
               | RV64
               | RV128
```
Type-level RISC-V Configuration

\[
data \text{ Opcode} :: \text{ RV} \rightarrow \text{ Format} \rightarrow * \text{ where}
\]

\[
\begin{align*}
\text{Add} & :: \text{ Opcode rv R} \\
\text{Addw} & :: 64 \leftarrow \text{ RVWidth rv} \rightarrow \text{ Opcode rv R} \\
\text{Mul} & :: \text{ MExt} \leftarrow \text{ rv} \rightarrow \text{ Opcode rv R}
\end{align*}
\]
GRIFT semantics DSL

- Instruction semantics represented in an embedded DSL, with AST nodes for:
  - Arithmetic and bitvector operations
  - Register/memory accesses
  - Reading from instruction operands
- Dependently typed, using type-level naturals in GHC to track bitvector widths
- "Shape" of instruction (number and size of operands) captured as a type parameter
GRIFT semantics DSL

```
Mul :: MExt << rv => Opcode rv R

Pair Mul $ instSemantics (Rd :: Rs1 :: Rs2 :: Nil) $ do
  comment "Multiplies x[rs1] by x[rs2] and writes the prod to x[rd]."
  comment "Arithmetic overflow is ignored."

  rd :: rs1 :: rs2 :: Nil <- operandEs

  let x_rs1 = readGPR rs1
  let x_rs2 = readGPR rs2

  assignGPR rd (x_rs1 `mulE` x_rs2)
  incrPC
```
GRIFT’s encoding representation

- Instructions are parameterized by “format”, which determines operand number and width.
- Format determines mapping between operands and their locations in the instruction.
- The fixed bits of a particular instruction must also be defined to perform encoding and decoding.
GRIFT’s encoding representation

Mul :: MExt << rv => Opcode rv R

Pair Mul (OpBits RRepr (0b0110011 :: 0b000 :: 0b0000001 :: Nil))
GRIFT simulator

• ~40,000 instructions per second
• Disassembles ELF binaries compiled by gcc
• Interprets semantics DSL code for each instruction against a concrete machine state
• Dumps output to terminal as directed (register file, section of memory)
GRIFT simulator — coverage analysis

- Bonus feature of simulator (available via command-line options)

- Tracks coverage of individual instructions based on the branching structure of their semantics as expressed in semantics DSL

- Discover coverage holes in RISC-V compliance suites

- Notion of coverage is limited, but could be refined for particular needs
Other RISC-V Formal Specification Efforts

- SAIL RISC-V (Cambridge)
- riscv-semantics (MIT)
- Forvis (Bluespec)
- Kami RISC-V (MIT/Si-Five)
<table>
<thead>
<tr>
<th></th>
<th>Forvis</th>
<th>Grift</th>
<th>Sail</th>
<th>riscv-plv</th>
<th>Kami</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author/Group</td>
<td>Bluespec</td>
<td>Galois</td>
<td>SRI/Cambridge</td>
<td>MIT</td>
<td>SiFive</td>
</tr>
<tr>
<td>Licence</td>
<td>MIT</td>
<td>GPL3</td>
<td>BSD</td>
<td>MIT</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>Metalanguage</td>
<td>Haskell</td>
<td>embedded DSL in Haskell</td>
<td>Sail</td>
<td>Haskell</td>
<td>Kami/Coq</td>
</tr>
<tr>
<td>Functional coverage - Base ISA and extensions</td>
<td>RV32/64IMAFDC</td>
<td>RV32/64GC</td>
<td>RV32/RV64IMAC</td>
<td>RV32/64IMAF</td>
<td>RV32 IMAFC</td>
</tr>
<tr>
<td>Functional coverage - Privilege levels</td>
<td>MUS,Sv32,39,48</td>
<td>M</td>
<td>MUS,Sv32,39,48</td>
<td>Sv39</td>
<td>no</td>
</tr>
<tr>
<td>Specification of assembly syntax and encoding</td>
<td>no</td>
<td>pp</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Concurrency</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Feature</td>
<td>Forvis</td>
<td>Grift</td>
<td>Sail</td>
<td>riscv-plv</td>
<td>Kami</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Floating-point</td>
<td>via Softfloat</td>
<td>via Softfloat</td>
<td>no</td>
<td>via Softfloat</td>
<td>Native implementation of IEEE 754-2008</td>
</tr>
<tr>
<td>Emulation</td>
<td>Haskell</td>
<td>Haskell</td>
<td>generated C or OCaml</td>
<td>Haskell</td>
<td>Verilator</td>
</tr>
<tr>
<td>Emulation speed</td>
<td>??? IPS (40min Linux boot)</td>
<td>40K IPS on Intel Xeon E312</td>
<td>300K IPS on Intel i7-7700 (4min Linux boot)</td>
<td>100K IPS on 6700HQ (Linux boot)</td>
<td>Not measured</td>
</tr>
<tr>
<td>Use as test oracle in tandem verification</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Use for software coverage analysis</td>
<td>???</td>
<td>yes</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td>Theorem-prover definitions</td>
<td>via hs-to-coq?</td>
<td>no</td>
<td>Coq, Isa, HOL4</td>
<td>hs-to-coq</td>
<td>Coq</td>
</tr>
<tr>
<td>Use in documentation</td>
<td>to LaTeX</td>
<td>to text</td>
<td>to LaTeX in RISC-V ISA</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Use in test generation</td>
<td>(at UPenn?)</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Feature</td>
<td>Forvis</td>
<td>Grift</td>
<td>Sail</td>
<td>riscv-plv</td>
<td>Kami</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>Use for concurrency-model litmus test evaluation</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Test coverage - riscv-tests suite</td>
<td>???</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Test coverage - RISC-V compliance tests</td>
<td>all</td>
<td>almost all</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Test coverage - OS boots</td>
<td>Linux, FreeRTOS</td>
<td>no</td>
<td>Linux, FreeBSD, seL4</td>
<td>Linux</td>
<td>no</td>
</tr>
<tr>
<td>Test coverage - Concurrency litmus tests</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Conclusion

• GRIFT: A Haskell library comprising a formal RISC-V specification
  • RISC-V configuration expressed via Haskell types
  • Instruction encoding/semantics expressed in an embedded DSL
• Future work:
  • Applications: binary analysis, hardware/software verification
  • Other backends (Coq, ACL2, Verilog, PDF manuals)
  • Automated test generation
  • Concurrency?