Strong Formal Verification for RISC-V
From Instruction-Set Manual to RTL

Adam Chlipala
MIT CSAIL
RISC-V Workshop
November 2017

Joint work with: Arvind, Thomas Bourgeat, Joonwon Choi, Ian Clester, Samuel Duchovni, Jamey Hicks, Muralidaran Vijayaraghavan, Andrew Wright
A Cartoon View of Digital Hardware Design

Generator

RTL (e.g., Verilog)

Metaprogramming

CAD tools

Physical Layout

Quite proprietary magic

Silicon
A Cartoon View of Digital Hardware Design

Generator

Metaprogramming

RTL (e.g., Verilog)

CAD tools

Physical Layout

Quite proprietary magic

Silicon
Simplification #1: Prove a Shallow Property
Simplification #1: Prove a Shallow Property
Simplification #1: Prove a Shallow Property

If Foo is in this register, then Bar is in that one. Never Baz here and Qux there at same time.

Common practice: prove some Invariants
Simplification #1: Prove a Shallow Property

If Foo is in this register, then Bar is in that one. Never Baz here and Qux there at same time.

Common practice: prove some Invariants or Boolean equivalence check.
Simplification #1: Prove a Shallow Property

The Kami way:
Behavioral refinement of functional spec

ISA Reference
Simplification #2: Analyze Isolated Components
Simplification #2: Analyze Isolated Components

The Kami way:
Modularly compose proofs of pieces
Simplification #2: Analyze Isolated Components

The Kami way:
Modularly compose proofs of pieces
Simplification #2: Analyze Isolated Components

The Kami way:
Modularly compose proofs of pieces
Simplification #3: Start Over For Each Design

- Memory
- L1 Cache
- CPU
Simplification #3: Start Over For Each Design
Simplification #3: Start Over For Each Design
Simplification #3: Start Over For Each Design
Simplification #3: Start Over For Each Design

The Kami way: Prove once for all parameters

∀ trees. \equiv ISA Reference
A framework to support implementing, specifying, formally verifying, and compiling hardware designs based on the Bluespec high-level hardware design language and the Coq proof assistant.
The Big Ideas (from Bluespec)

Program modules are objects
The Big Ideas (from Bluespec)

Program modules are objects with mutable private state,
Program modules are objects with mutable private state, accessed via methods.
The Big Ideas (from Bluespec)

Program modules are objects with mutable private state, accessed via methods.
Every method call appears to execute atomically.
The Big Ideas

Every method call appears to execute atomically. Any step is summarized by a trace of calls.
Every method call appears to execute atomically. Any step is summarized by a trace of calls. Object refinement is inclusion of possible traces.
The Big Ideas

Recv \( f(1) \),
Send \( h(2) \)

Recv \( g(7) \),
Send \( k(13) \)
The Big Ideas

Composing objects hides internal method calls.
The Big Ideas

Composing objects hides internal method calls.
Some Example Kami Code (simple FIFO)

Definition deq {ty} : ActionT ty dType :=
    Read isEmpty <- ^empty;
    Assert !#isEmpty;
    Read eltT <- ^elt;
    Read enqPT <- ^enqP;
    Read deqPT <- ^deqP;
    Write ^full <- $$false;
    LET next_deqP <- (#deqPT + $1) :: Bit sz;
    Write ^empty <- (#enqPT == #next_deqP);
    Write ^deqP <- #next_deqP;
    Ret #eltT@[#deqPT].
Lemma p4st_refines_p3st: p4st \iff p3st.
Proof.
  kmmodular.
  - kdisj_edms_cms_ex 0.
  - kdisj_ecms_dms_ex 0.
  - apply fetchDecode_refines_fetchNDecode; auto.
  - krefl.
Qed.

Uses standard Coq ASCII syntax for mathematical proofs.
These proofs are checked automatically, just like type checking.
We inherit streamlined IDE support for Coq.
We Are Building:

Design \rightarrow \text{Refines} \rightarrow \text{Spec} \rightarrow \text{Coq tactics to prove refinements}
We Are Building:

- Design
- Spec
- Compiler
- RTL

Coq tactics to prove refinements

Verify semantics preservation of compiler

Refines
Some Useful Refinement Tactics

Monolithic Spec

Sequential Consistency
Some Useful Refinement Tactics

Monolithic Spec

Sequential Consistency

Decompose

Decoupled Spec

Processor

Memory
Some Useful Refinement Tactics

- **Monolithic Spec**
  - Sequential Consistency
  - 1. Decompose

- **Decoupled Spec**
  - Processor
  - Memory

- **Getting Real**
  - Fancy Processor
  - Memory
  - 2. Replace Module
Some Useful Refinement Tactics

Monolithic Spec
Sequential Consistency

1. Decompose

Decoupled Spec
Processor
Memory

Getting Real
Fancy Processor
Memory

2. Replace Module
...
Processor
Some Useful Refinement Tactics

Monolithic Spec
- Sequential Consistency

Decoupled Spec
- Processor
- Memory

Getting Real
- Fancy Processor
- Memory

1. Decompose
2. Replace Module
3. Induction /Simulation

Processor → Processor'
Some Useful Refinement Tactics

Monolithic Spec
- Sequential Consistency

Decoupled Spec
- Processor
- Memory

Getting Real
- Fancy Processor
- Memory

- Replace Module
- Induction / Simulation

1. Decompose
2. Replace Module
3. Induction / Simulation
4. Decompose
Some Useful Refinement Tactics

Monolithic Spec
- Sequential Consistency

Decoupled Spec
- Processor
- Memory

Getting Real
- Fancy Processor
- Memory

- Processor''
- Ring Buffer
- Processor''
- Ideal Queue

1. Decompose
2. Replace Module
3. Induction / Simulation
4. Decompose
5. Replace
Key Ingredient

Formal Semantics for RISC-V ISA(s)

Nikhil just explained the semantics style. We are building a translator for the semantics into the language of Coq/Kami.
An Open Library of Formally Verified Components

- Microcontroller-class RV32I (multicore; U)
- Desktop-class RV64IMA (multicore; U,S,M)
- Cache-coherent memory system

Reuse our proofs when composing our components with your own formally verified accelerators!
The Promise of this Approach

ISA Formal Semantics
The Promise of this Approach

ISA Formal Semantics

Processor

RTL Formal Semantics

(Proved)
The Promise of this Approach

Application Specification

Application Machine Code

ISA Formal Semantics

Processor

RTL Formal Semantics

(Proved)
The Promise of this Approach

Application Specification

Application Machine Code

ISA Formal Semantics

Processor

RTL Formal Semantics

Proved

Proved
The Trusted Computing Base

Where can defects go uncaught?
The Trusted Computing Base

Where can defects go uncaught?

- Coq proof checker (small & general-purpose)
- RTL formal semantics
- Application specification
The Trusted Computing Base

Where can defects go uncaught?

Coq proof checker (small & general-purpose)
RTL formal semantics
Application specification
ISA formal semantics
Hardware design (Bluespec, RTL, …)
Software implementation (C, …)
Shameless plug!

Part of a larger project: The Science of Deep Specification
A National Science Foundation Expedition in Computing

https://deepspec.org/

Join our mailing list for updates on our 2018 summer school: hands-on training with these tools!