Kami:
Modular Verification of Digital Hardware in Coq

Adam Chlipala
MIT CSAIL
January 2018

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

Metaprogramming

RTL (e.g., Verilog)

CAD tools

Physical Layout

Quite proprietary magic

Silicon
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 the same time.

Common practice:
prove some Invariants

The Kami way:
Behavioral refinement of functional spec

ISA Reference

or Boolean equivalence check
Simplification #2: Analyze Isolated Components

The Kami way:
Modularly compose proofs of pieces

ISA Reference

Proved

Proved

Proved
Simplification #3: Start Over For Each Design

∀ trees. ⇔

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.
Usual Industry Practice:
Register Transfer Language (RTL)
Differences from Conventional Software

• All state elements must be **finite**.
• Instead of loops & recursion, single clock cycles.
• Almost unlimited opportunity for **parallelism** within one clock cycle!
• However, one long dataflow dependency chain in one part of a design can slow down the clock for everyone.
• So we often break operations into multiple cycles.
The Great Annoyance of Timing Dependency

inputs → Gates → Gates → Gates → outputs

inputs → Gates → Gates → stall → Gates → outputs

inputs → Gates → Gates → stall → Gates → outputs

inputs → Gates → Gates → stall → Gates → outputs

Gates

Gates

Gates

Gates

cycle #1

cycle #2

cycle #3

cycle #4
The Big Ideas (from Bluespec)

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

Every method call appears to execute **atomically**. Any step is summarized by a **trace** of calls. Object **refinement** is inclusion of possible traces.
Composing objects hides internal method calls.
Actually, objects also include *rules*, atomic state transitions that fire on their own. They wind up looking sort of like operational semantics rules.

Bluespec gives programmers the illusion that we repeatedly pick a rule (nondeterministically) and run it atomically.

Parallelism is essential for performance. ...so compiler extracts it automatically, via static analysis.
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].
An Example Kami Proof (pipelined processor)

Lemma p4st_refines_p3st: p4st \leq\leq p3st.
Proof.
  kmodular.
  - kdisj_edms_cms_ex 0.
  - kdisj_ecms_dms_ex 0.
  - apply fetchDecode_refines_fetchNDDecode; 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
- Spec
- RTL

Coq tactics to prove refinements
Verify semantics preservation of compiler
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

4 Decompose

5 Replace

Processor''
Ring Buffer
Processor''
Ideal Queue
Processor'
Processor'
Decompose ↔ Merge & Inline

\[ M \quad f(x) \quad g(y) \quad N \quad h(x) \]

\[ M + N \quad f(x) \quad g(y) \quad h(x) \]

\[ M + N \quad f(x) \quad h(x) \quad g(y) \]

sort of smells like Coq simpl
Replace Module ↔ Congruence

\[
\begin{align*}
M \leq M' & \quad N \leq N' \\
\hline
M+N \leq M'+N' \\
\end{align*}
\]

Joins other classic theorems of process calculus:

\[
\begin{align*}
M \leq M' & \quad M' \leq M'' \\
\hline
M \leq M'' \\
\end{align*}
\]
Direct Simulation

Choose this relation per proof.

\[ \forall \approx \exists \approx \]

Wrinkle: cycles in module call graph make it not so trivial to enumerate all possible steps. We prove that certain syntactic conditions guarantee soundness.
Code walk-through: simple producer-consumer system
Twist #1: Mixing Specs & Impls

- **Processor''**: Must write as Bluespec-style HW design
- **Ring Buffer**: Replace
- **Processor''**: Must write as Bluespec-style HW design
- **Ideal Queue**: Want to write as a normal functional program
- **Processor'**: Decompose
Inductive ty :=
| Bitvector (n : nat)
| Tuple (ts : list ty).

Inductive exp : ty → Set :=
| Bits : forall n, bitvector n → exp (Bitvector n)
| Let : forall t1 t2,
   exp t1 → (var t1 → exp t2)
   → exp t2
| Var : forall t, var t → exp t
| ...

End var.

Definition Exp t := ∀ var, exp var t.
Review: Parametric Higher-Order Abs. Syntax (PHOAS)

Fixpoint tyD (t : ty) : Set := match t with
  | Bitvector n => bitvector n
  | Tuple ts => tuple (map tyD ts)
end.

Fixpoint expD t (e : expr tyD t) : tyD t :=
  match e with
  | Bits bv => bv
  | Let e1 e2 => expD (e2 (expD e1))
  | Var x => x
  | ...
end.

Definition ExpD t (E : Exp t) := expD (E _).

Section var.

Variable var : ty → Set.

Inductive exp : ty → Set :=
  | Bits : forall n, bitvector n → exp (Bitvector n)
  | Let : forall t1 t2, exp t1 → (var t1 → exp t2) → exp t2
  | Var : forall t, var t → exp t
  | ...
End var.

Definition Exp t := ∀ var, exp var t.
Mixing It Up: Allowing Native Coq Code

Section var.

Variable var : ty → Set.

Definition var' (t : ty') : Set := match t with
| Syntactic t => var t
| Semantic T => T
end.

Inductive exp : ty' → Set :=
| Bits : forall n, bitvector n → exp (Syntactic (Bitvector n))
| Let : forall t1 t2, exp t1 → (var' t1 → exp t2) → exp t2
| Var : forall t, var' t → exp t | …

End var.

Inductive ty :=
| Bitvector (n : nat)
| Tuple (ts : list ty).

Inductive ty' :=
| Syntactic (t : ty)
| Semantic (T : Set).

Definition ty'D (t : ty') : Set :=
match t with
| Syntactic t => tyD t
| Semantic T => T
end.
Twist #2: Parametric and Repeated Designs

∀cacheSize. Processor(cacheSize) ≤ ISAspec

∀cacheSize, n. [Processor(cacheSize)]

Kami programs are actually Gallina programs to compute deeply embedded Bluespec-style syntax! Parameterization is just use of Gallina functions, and repetition is a simple recursive function.
Handy Proof Rules

\[ M \leq N \]

\[ M^n \leq N^n \]

Implementation challenges are to make the main Kami tactics work well with metaprograms, doing \textit{just enough} reduction.
RISC-V: An Open Instruction Set

Platinum Members

- Berkeley Architecture Research
  FOUNDING PLATINUM

- Bluespec
  FOUNDING PLATINUM

- C-SKY
  PLATINUM

- Cortus
  FOUNDING PLATINUM

- Google
  FOUNDING PLATINUM

- Micron Technology
  PLATINUM

- Microsemi
  FOUNDING PLATINUM

- NVIDIA
  FOUNDING PLATINUM

- NXP
  PLATINUM

- Qualcomm
  FOUNDING PLATINUM

- Rambus Inc.
  FOUNDING PLATINUM

- Samsung
  PLATINUM

- Sanechips Technology Co.
  PLATINUM

- SiFive
  FOUNDING PLATINUM

- Western Digital
  FOUNDING PLATINUM
Official Formal Semantics for RISC-V

Interpreter in a tasteful subset of Haskell

“the semantics”

Coq
Isabelle/HOL
SMT
ACL2

Reference manual (using prose from specially formatted comments)

Fast emulator

Test cases / test oracle
Sample Code for Semantics WIP

Decoding machine instructions

decode_sub opcode
| opcode==opcode_LOAD, funct3==funct3_LB
  = Lb  {rd=rd, rs1=rs1, oimm12=oimm12}
| opcode==opcode_LOAD, funct3==funct3_LH
  = Lh  {rd=rd, rs1=rs1, oimm12=oimm12}

Executing decoded instructions

execute (Lwu rd rs1 oimm12) = do
  a <- getRegister rs1
  x <- loadWord (a + fromIntegral oimm12)
  setRegister rd (unsigned x)
execute (Addw rd rs1 rs2) = do
  x <- getRegister rs1
  y <- getRegister rs2
  setRegister rd (s32 (x + y))
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

- Application Specification
- Application Machine Code (Proved)
- ISA Formal Semantics
- Processor (Proved)
- RTL Formal Semantics
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!
In Summary...

• With the right tool support, digital-hardware development is just another kind of programming.
• Functional programming & Coq are a great match for this domain.
• The rough edges that still exist are just the kind that the ICFP crowd enjoy smoothing!
• The chance to tinker with the HW layers is freeing – ask me later about getting rid of weak memory models. :}
https://github.com/mit-plv/kami