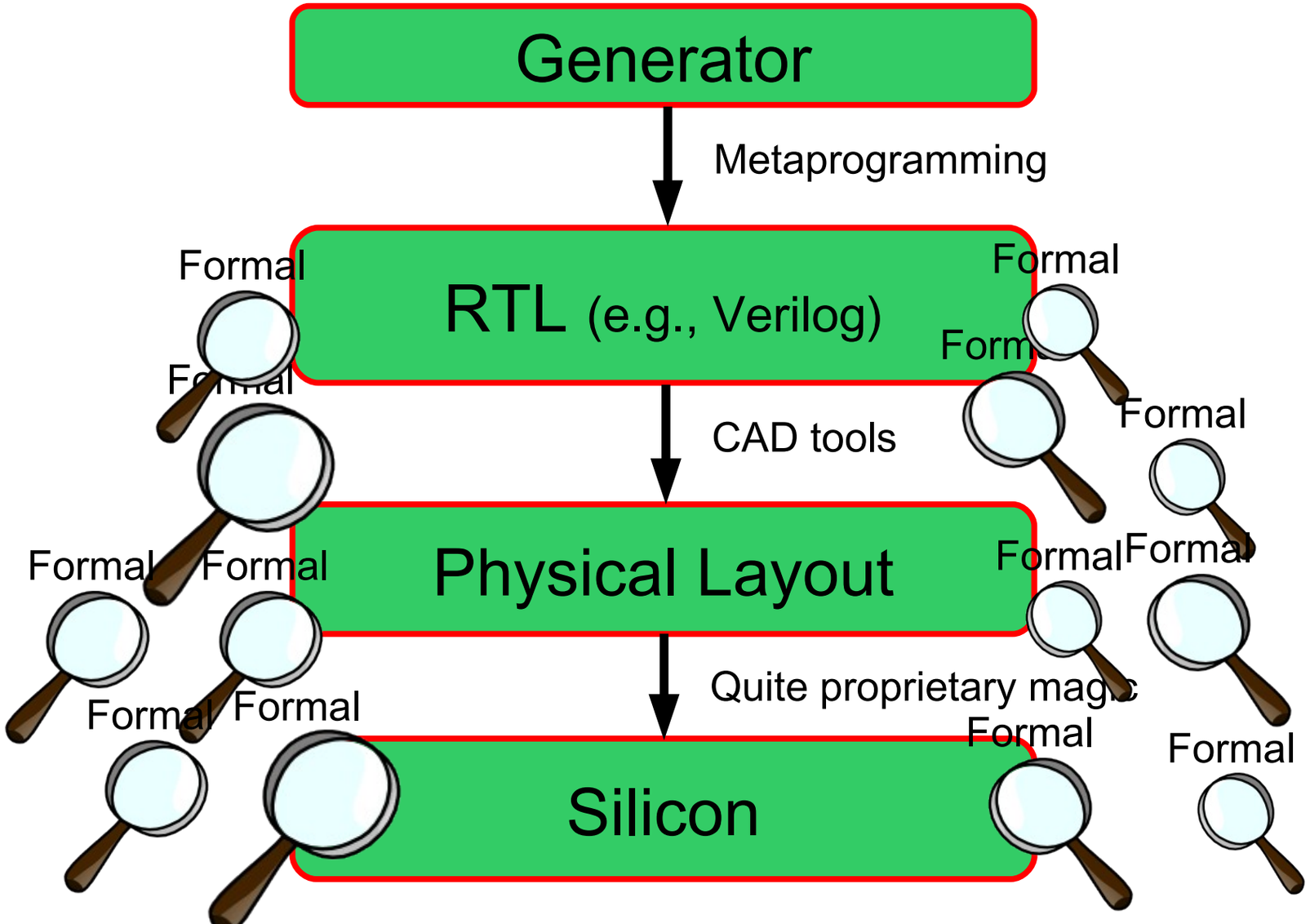
Kami:
Modular Verification of Digital Hardware in Coq

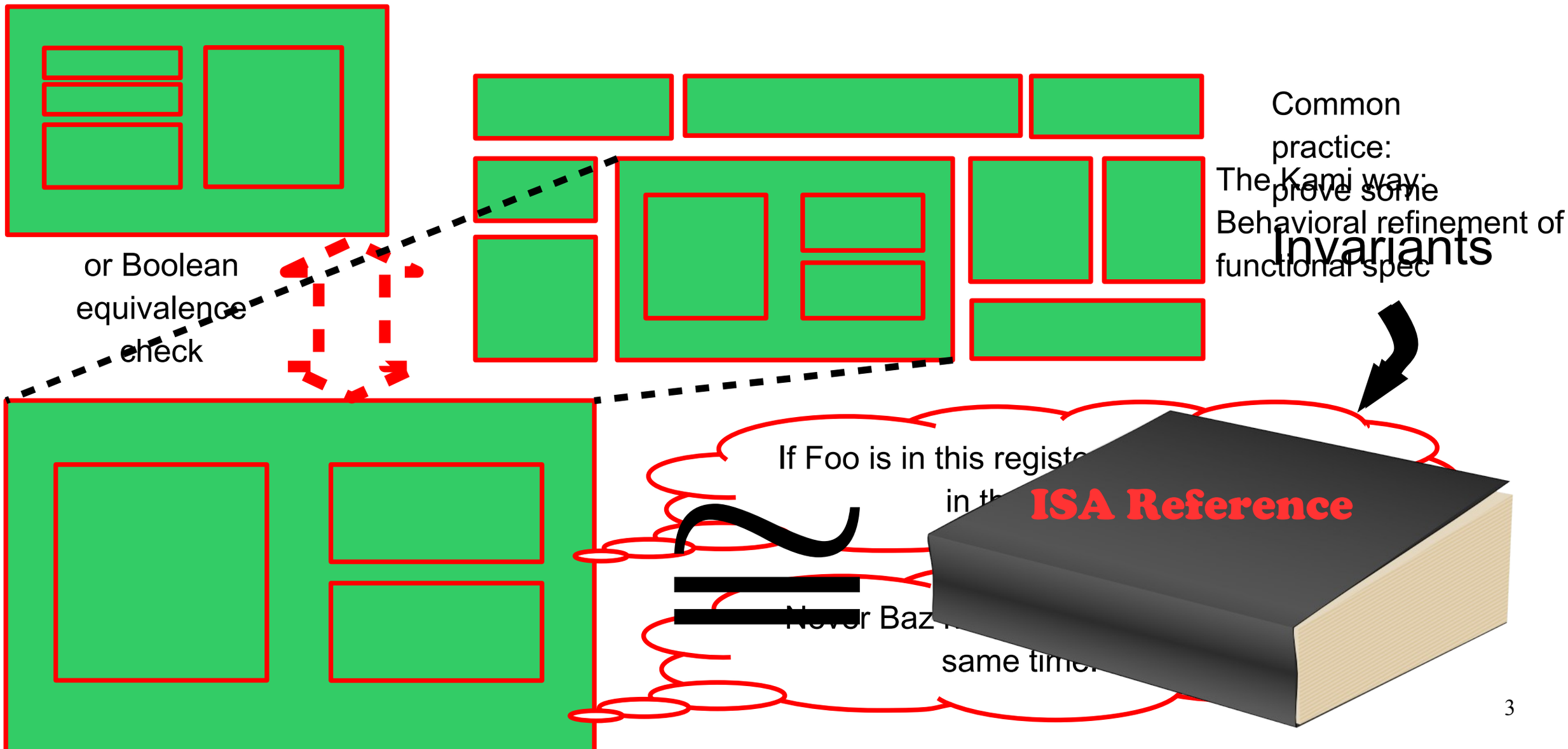
Adam Chlipala
MIT CSAIL
January 2018

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

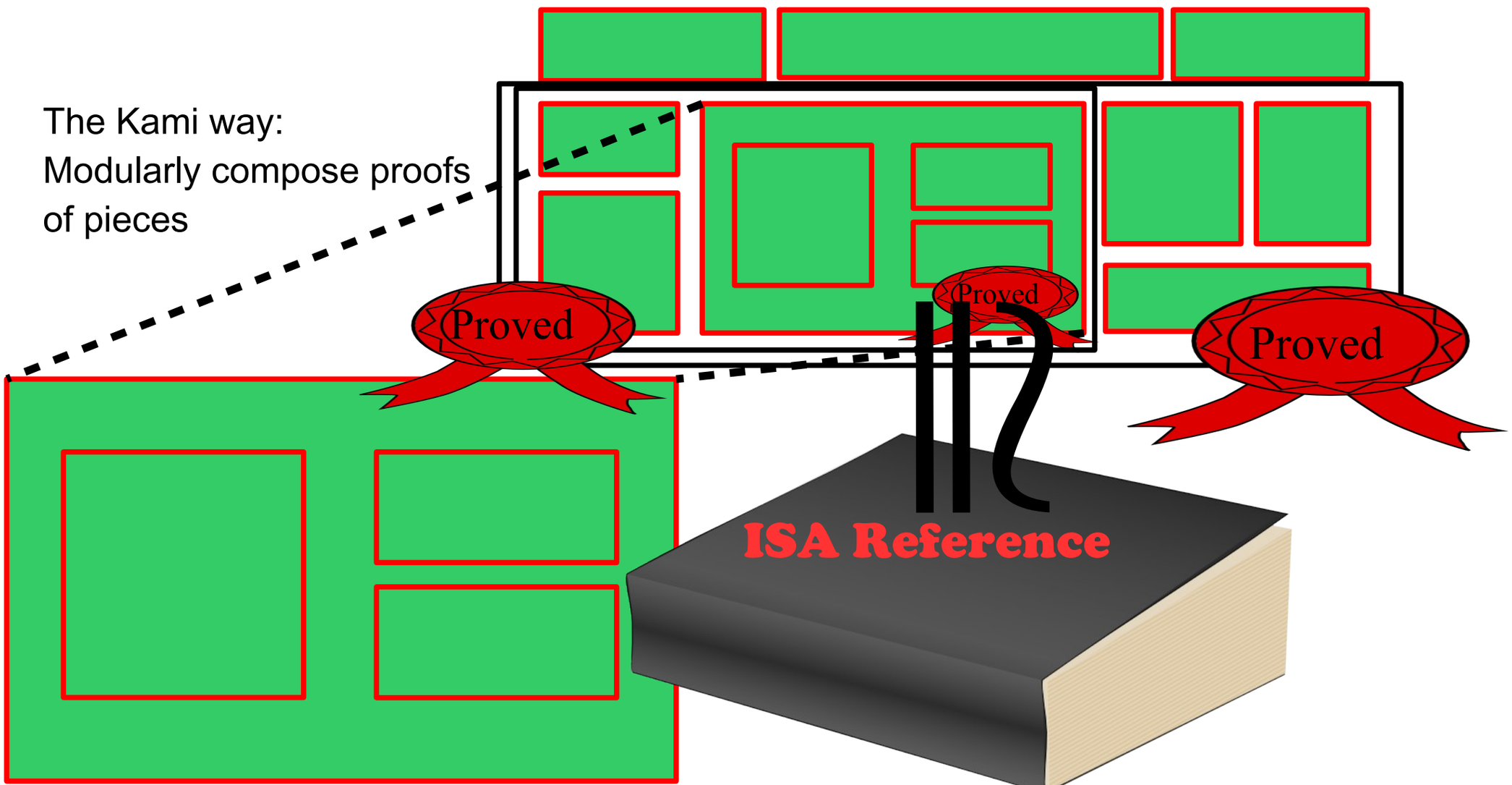


Simplification #1: Prove a Shallow Property

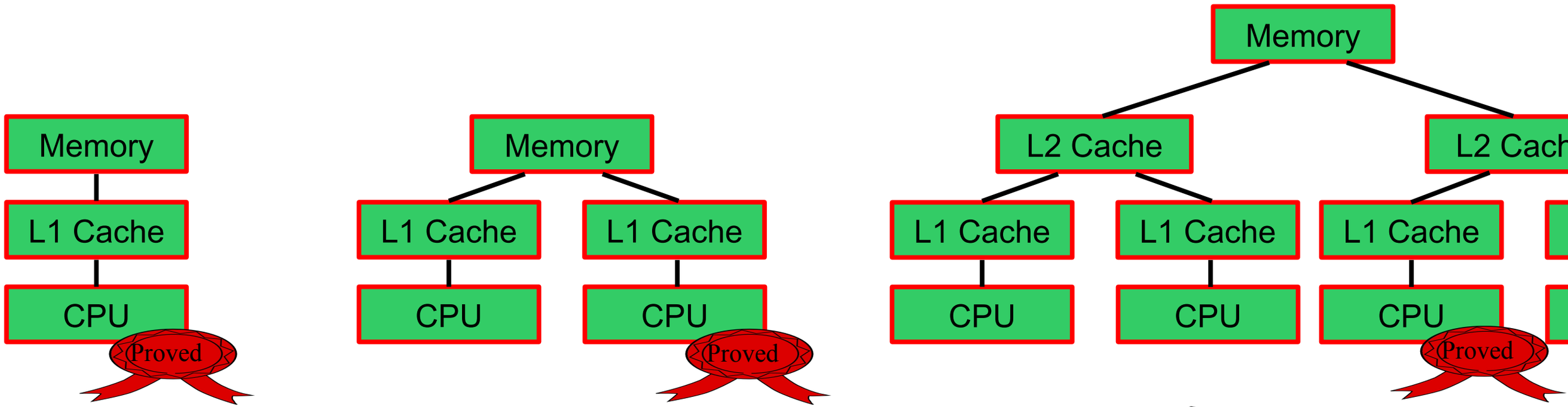


Simplification #2: Analyze Isolated Components

The Kami way:
Modularly compose proofs
of pieces

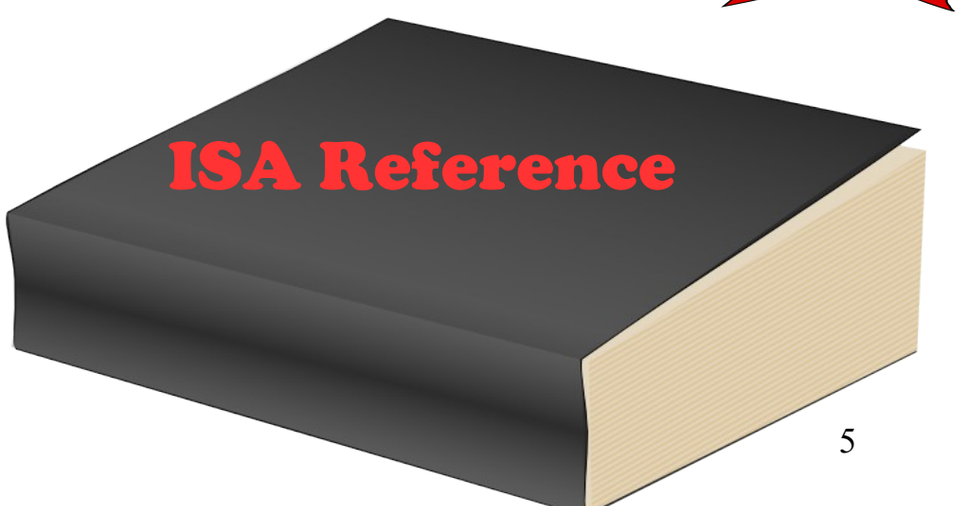


Simplification #3: Start Over For Each Design



The Kami way:
Prove once for all parameters

$\forall$ trees. $\cong$



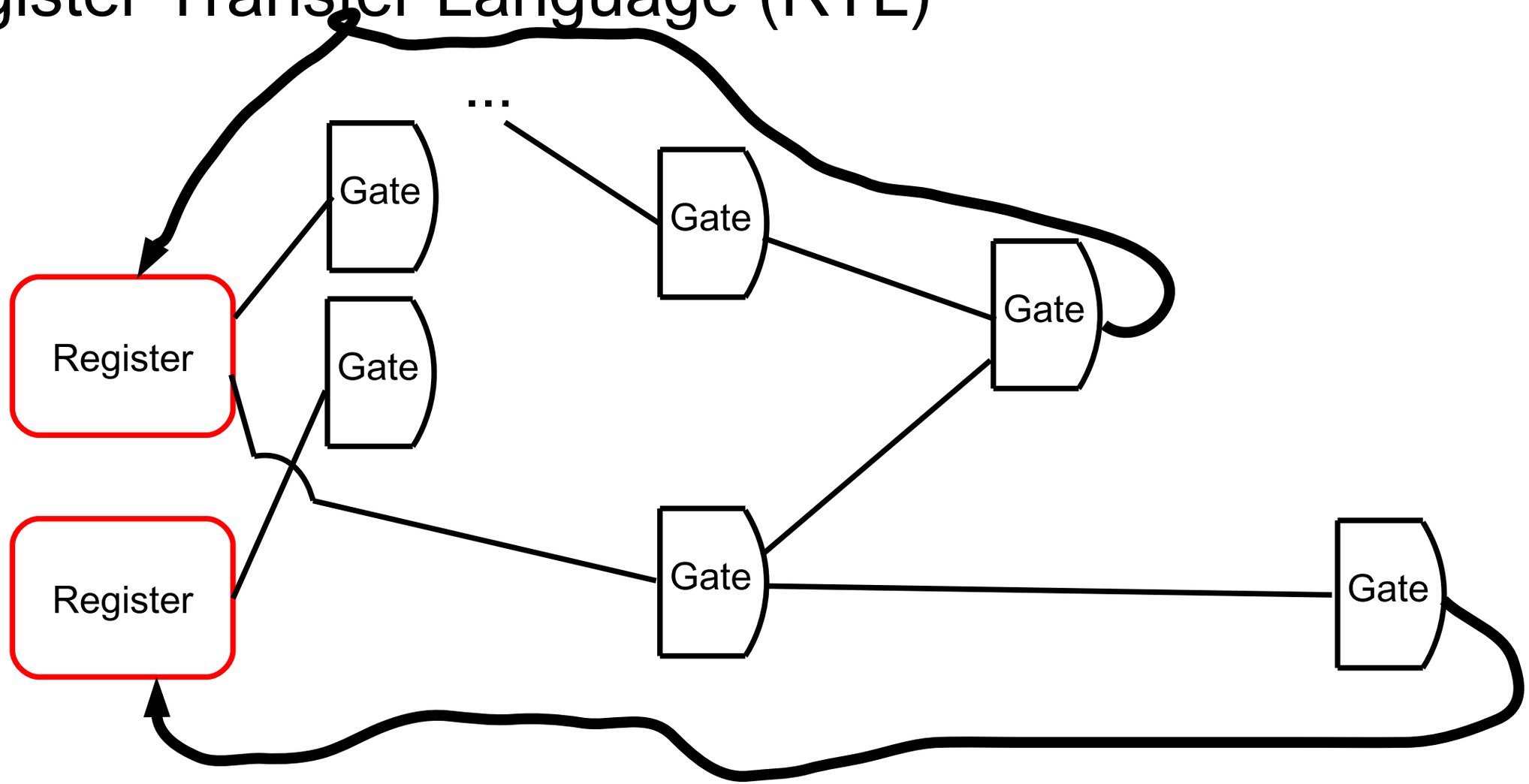
Kami

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

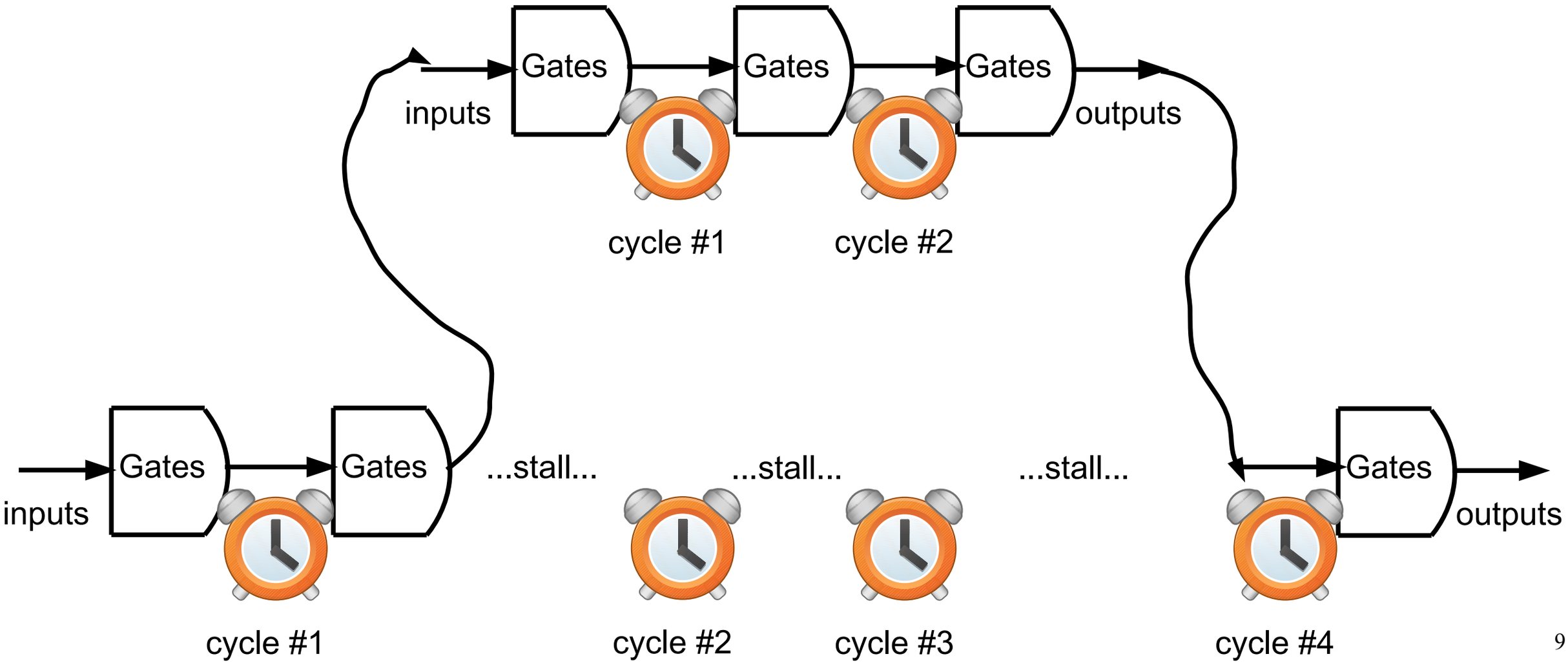
Usual Industry Practice:
Register Transfer Language (RTL)



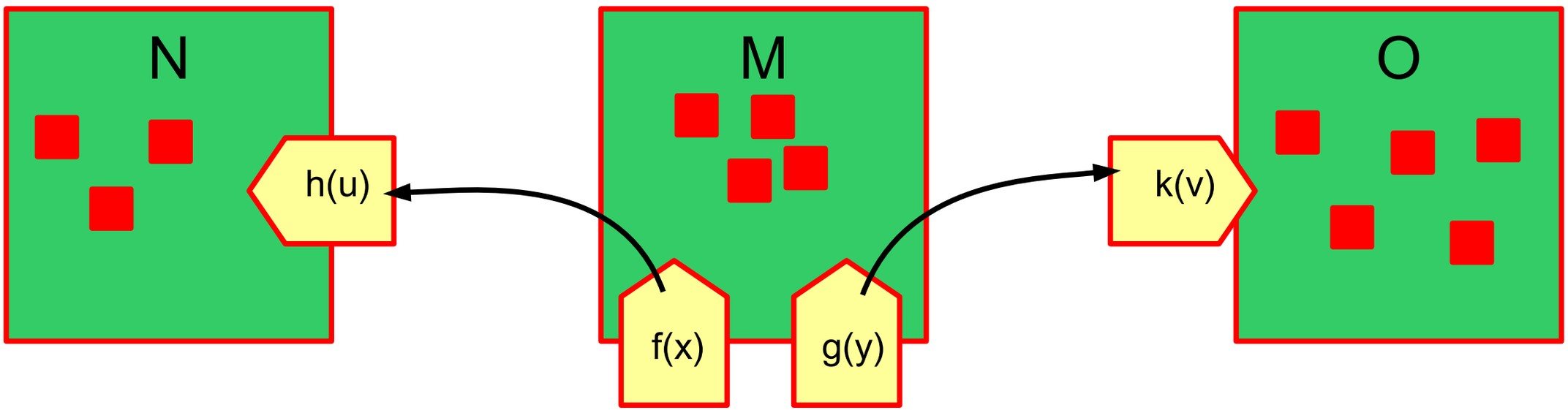
Differences from Conventional Software

- 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

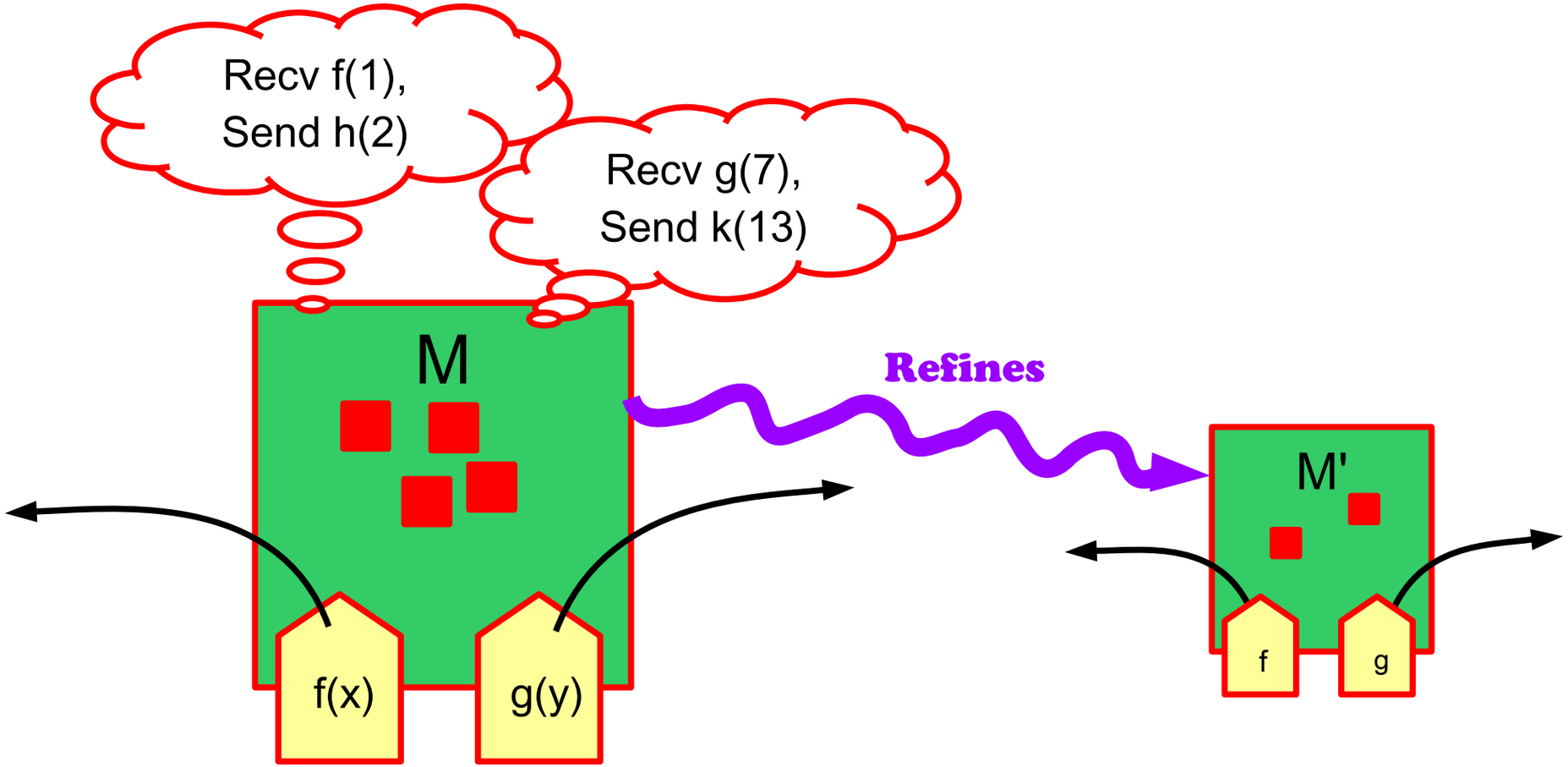


The Big Ideas (from Bluespec)



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

The Big Ideas

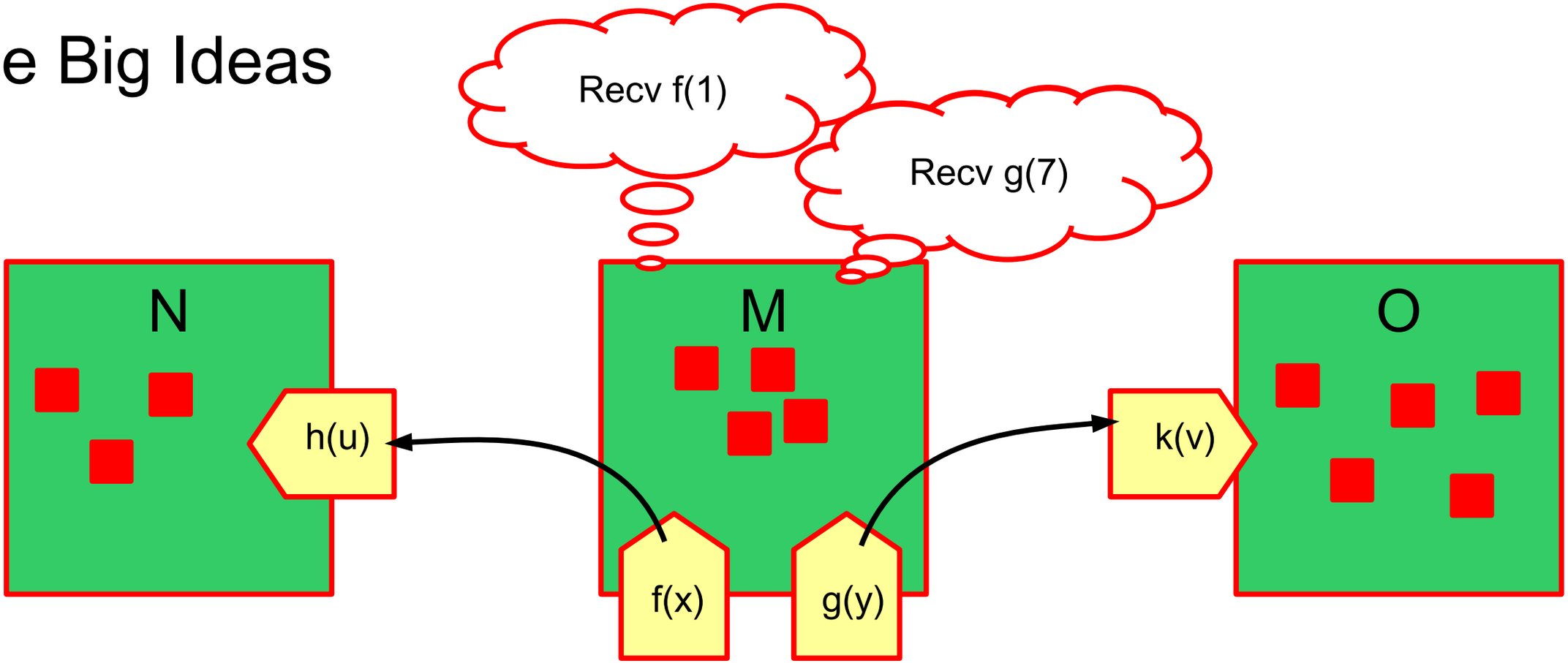


Every method call appears to execute **atomically**.

Any step is summarized by a *trace* of calls.

Object *refinement* is inclusion of possible traces.

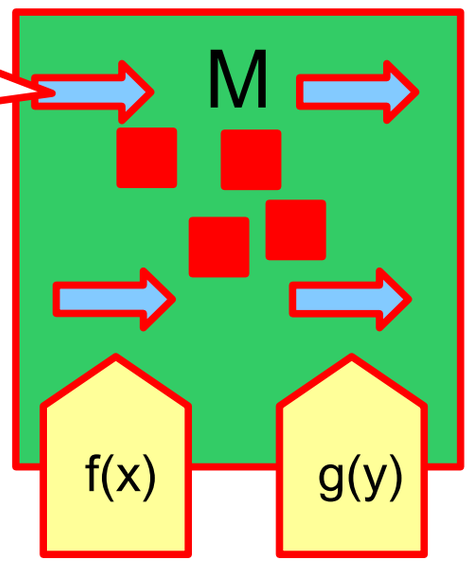
The Big Ideas



Composing objects hides internal method calls.

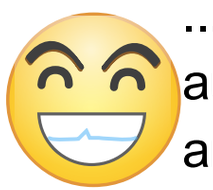
Rules

Example rule:
- Read memory at PC.
- Check that it's an add instr.
- Load from source registers.
- Perform addition.
- Write to destination register.



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.

Actually, objects also include *rules*, atomic state transitions that fire on their own.

They wind up looking sort of like operational semantics rules.

Some Example Kami Code (simple FIFO)

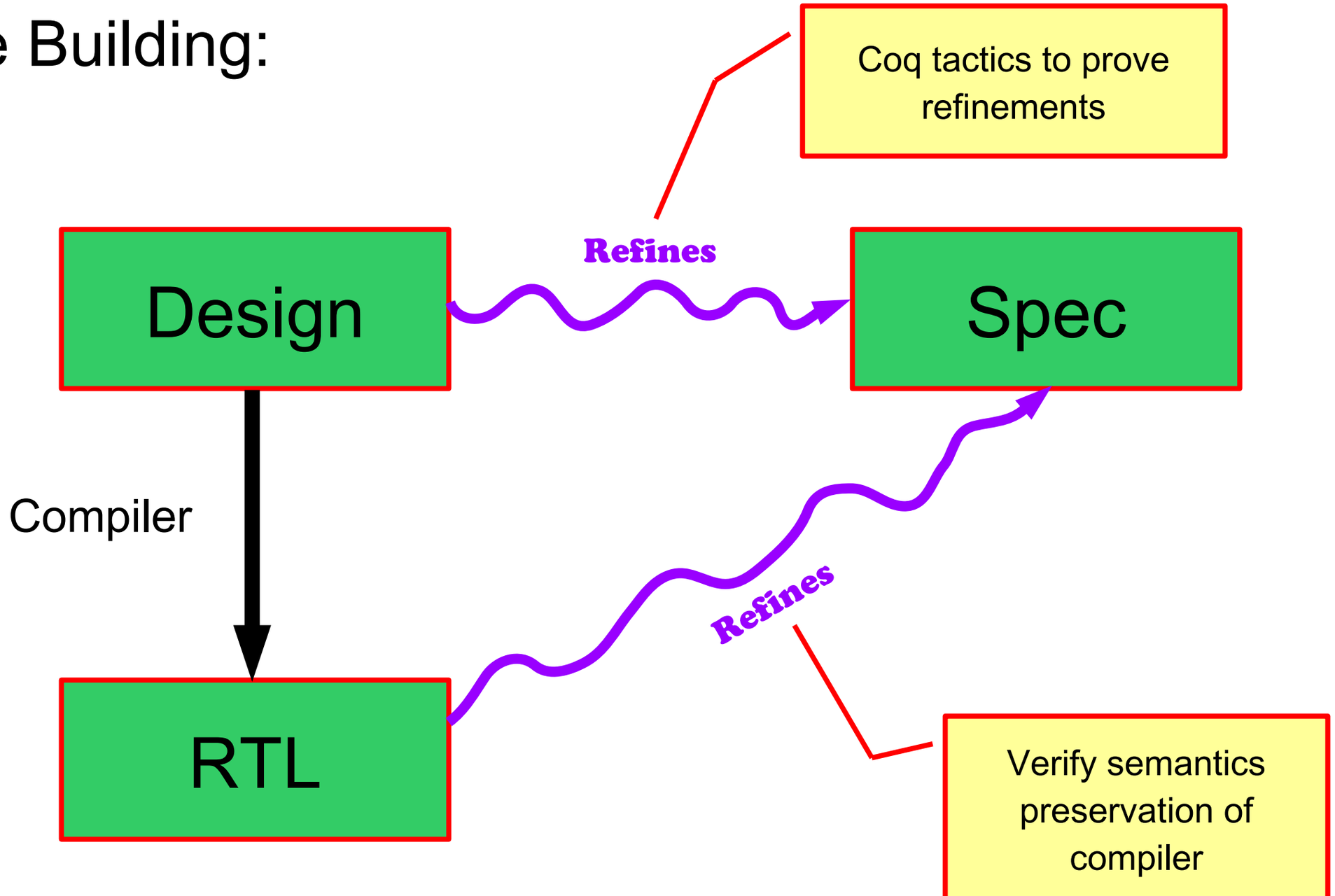
```
Definition deq {ty} : ActionT ty dType :=
  Read isEmpty <- ^empty;
  Assert !#isEmpty;
  Read eltT <- ^elt;
  Read enqPT <- ^enqP;
  Read deqPT <- ^deqP;
  Write ^full <- $$false;
  LET next_deqP <- (#deqPT + $1) :: Bit sz;
  Write ^empty <- (#enqPT == #next_deqP);
  Write ^deqP <- #next_deqP;
  Ret #eltT@[#deqPT].
```

An Example Kami Proof (pipelined processor)

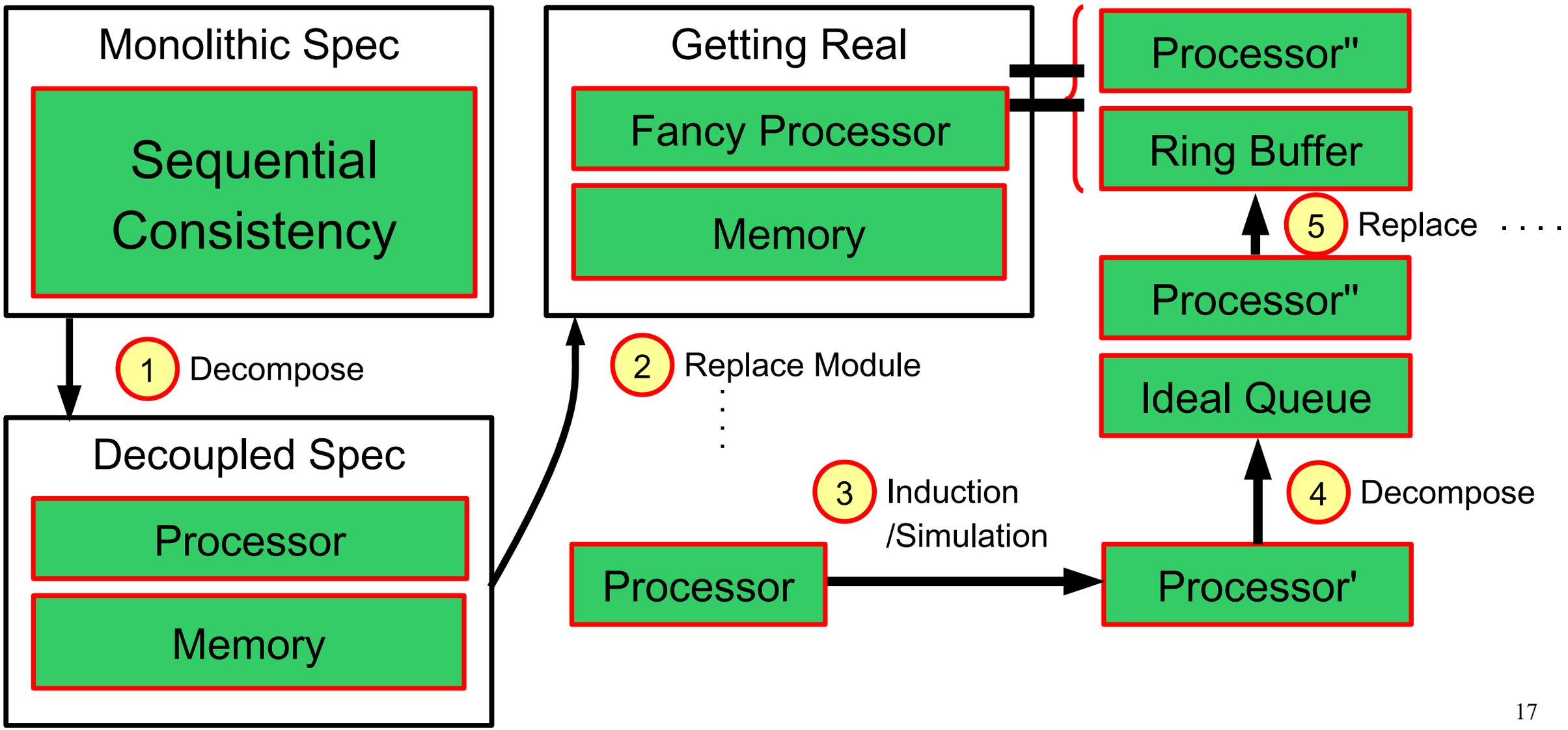
```
Lemma p4st_refines_p3st: p4st <<== p3st.
Proof.
  kmodular.
  - kdisj_edms_cms_ex O.
  - kdisj_ecms_dms_ex O.
  - apply fetchDecode_refines_fetchNDecode; auto.
  - krefl.
Qed.
```

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:

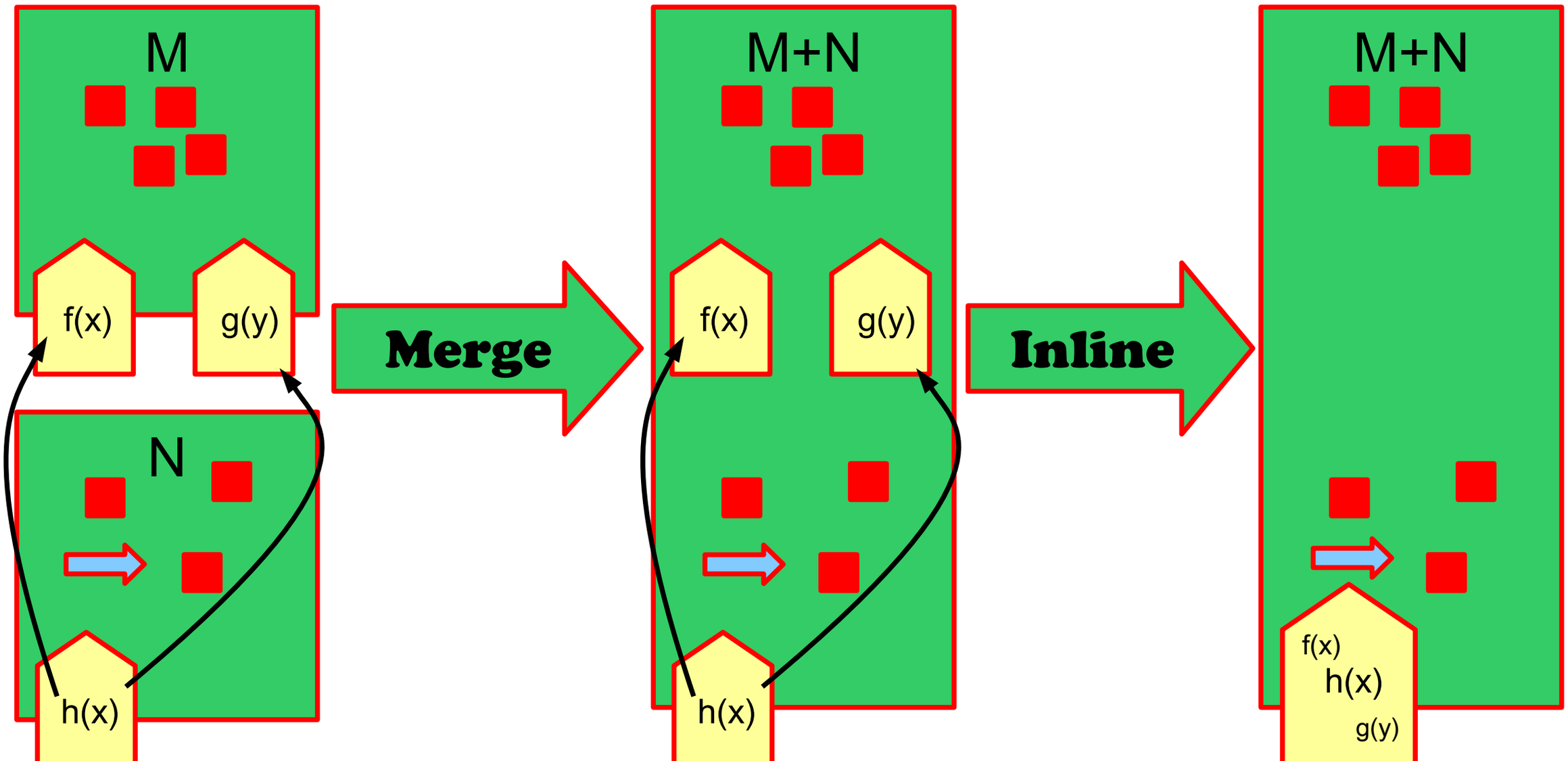


Some Useful Refinement Tactics



Decompose ↔ Merge & Inline

sort of smells like Coq `simpl`



Replace Module ↔ Congruence

sort of smells like Coq `rewrite`

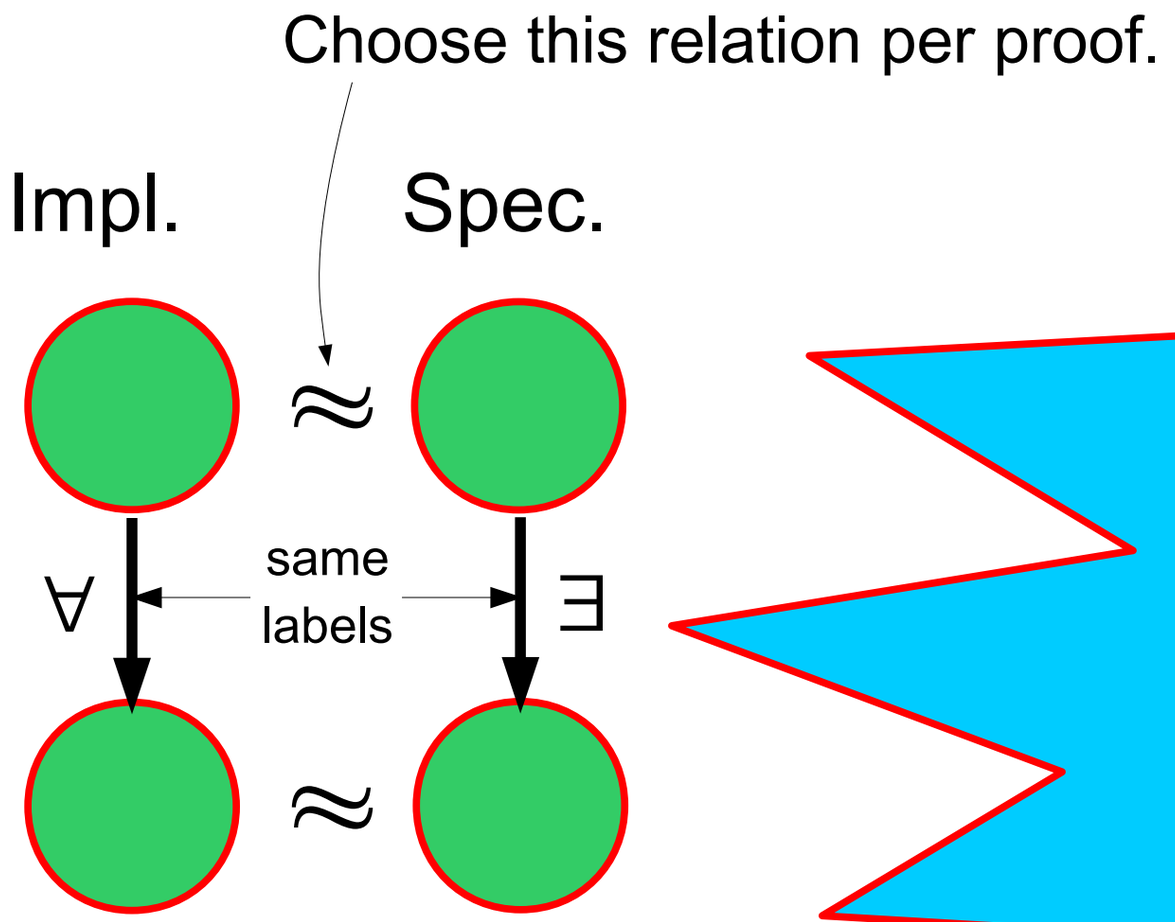
$$\frac{M \leq M' \qquad N \leq N'}{M+N \leq M'+N'}$$

Joins other classic theorems of process calculus:

$$\frac{}{M \leq M} \qquad \frac{M \leq M' \qquad M' \leq M''}{M \leq M''}$$

Direct Simulation

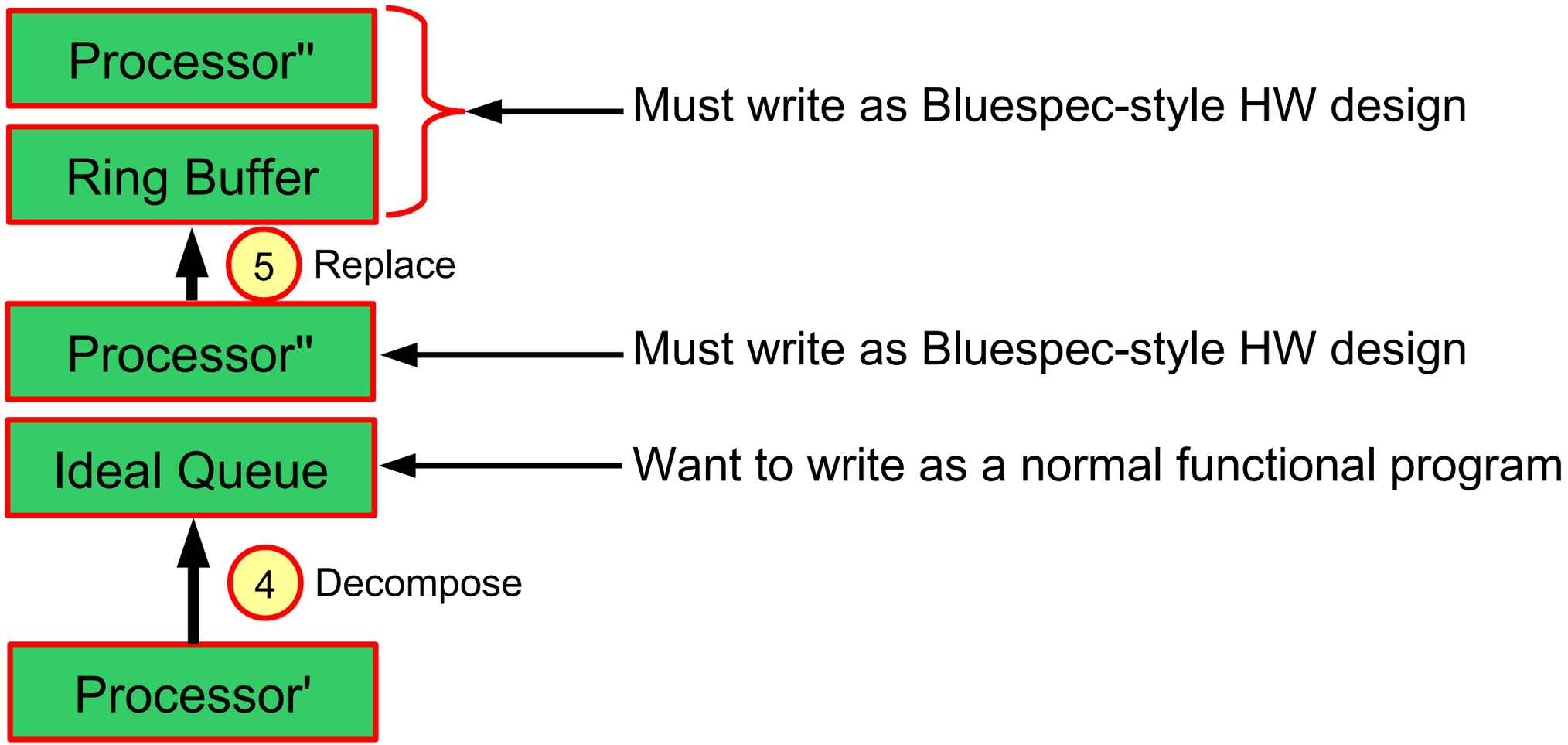
sort of smells like Coq `induction`



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



Review: Parametric Higher-Order Abs. Syntax

(PHOAS)

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

```
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.
```

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

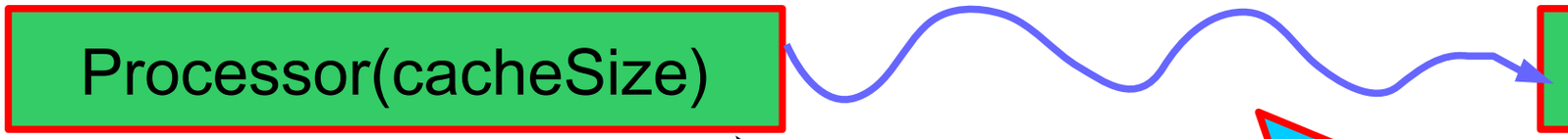
```
Inductive ty :=
    | Bitvector (n : nat)
    | Tuple (ts : list ty).

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.
```

```
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.
```

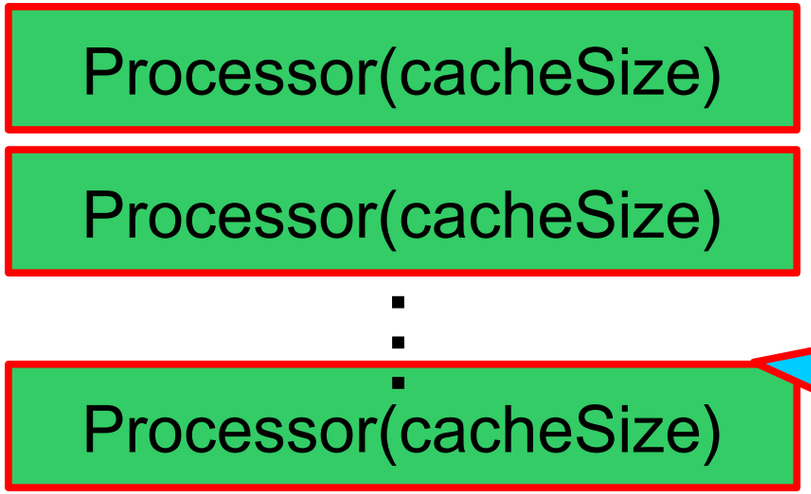
!

!

Twist #2: Parametric and Repeated Designs



∀cacheSize. Processor(cacheSi



∀cacheSize, n. [Processor(ca

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

$$\frac{M \leq N}{M^n \leq N^n}$$

Implementation challenges are to make the main Kami tactics work well with metaprograms, doing *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

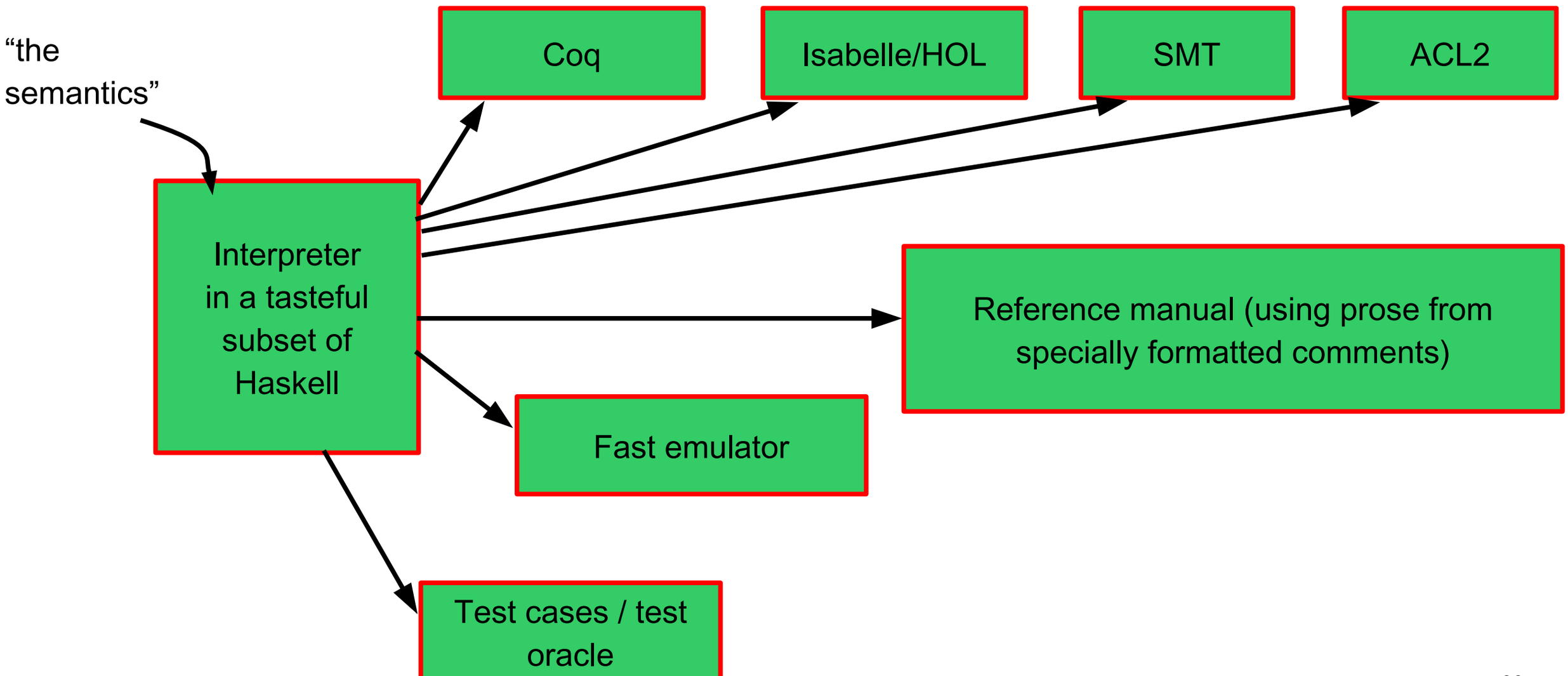
Samsung
PLATINUM

Sanechips Technology Co.
PLATINUM

SiFive
FOUNDING PLATINUM

Western Digital
FOUNDING PLATINUM

Official Formal Semantics for RISC-V



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

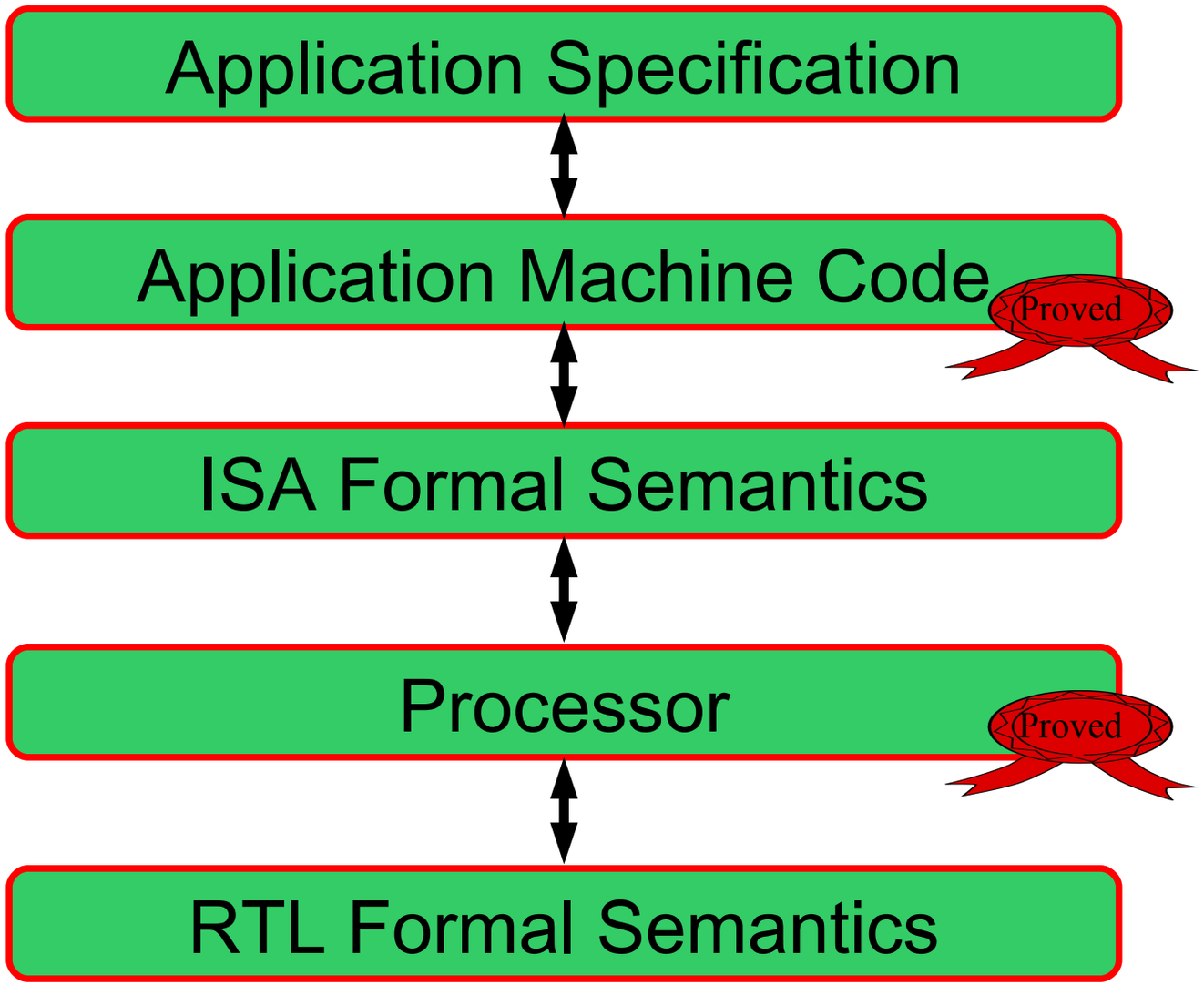
```
execute (Lwu rd rs1 oimm12) = do
  a <- getRegister rs1
  x <- loadWord (a + fromIntegral oimm12)
  setRegister rd (unsigned x)
execute (Addw rd rs1 rs2) = do
  x <- getRegister rs1
  y <- getRegister rs2
  setRegister rd (s32 (x + y))
```

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



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