Using Kami in the field - experiences integrating Kami into SiFive's Chisel/ Scala-based design flow

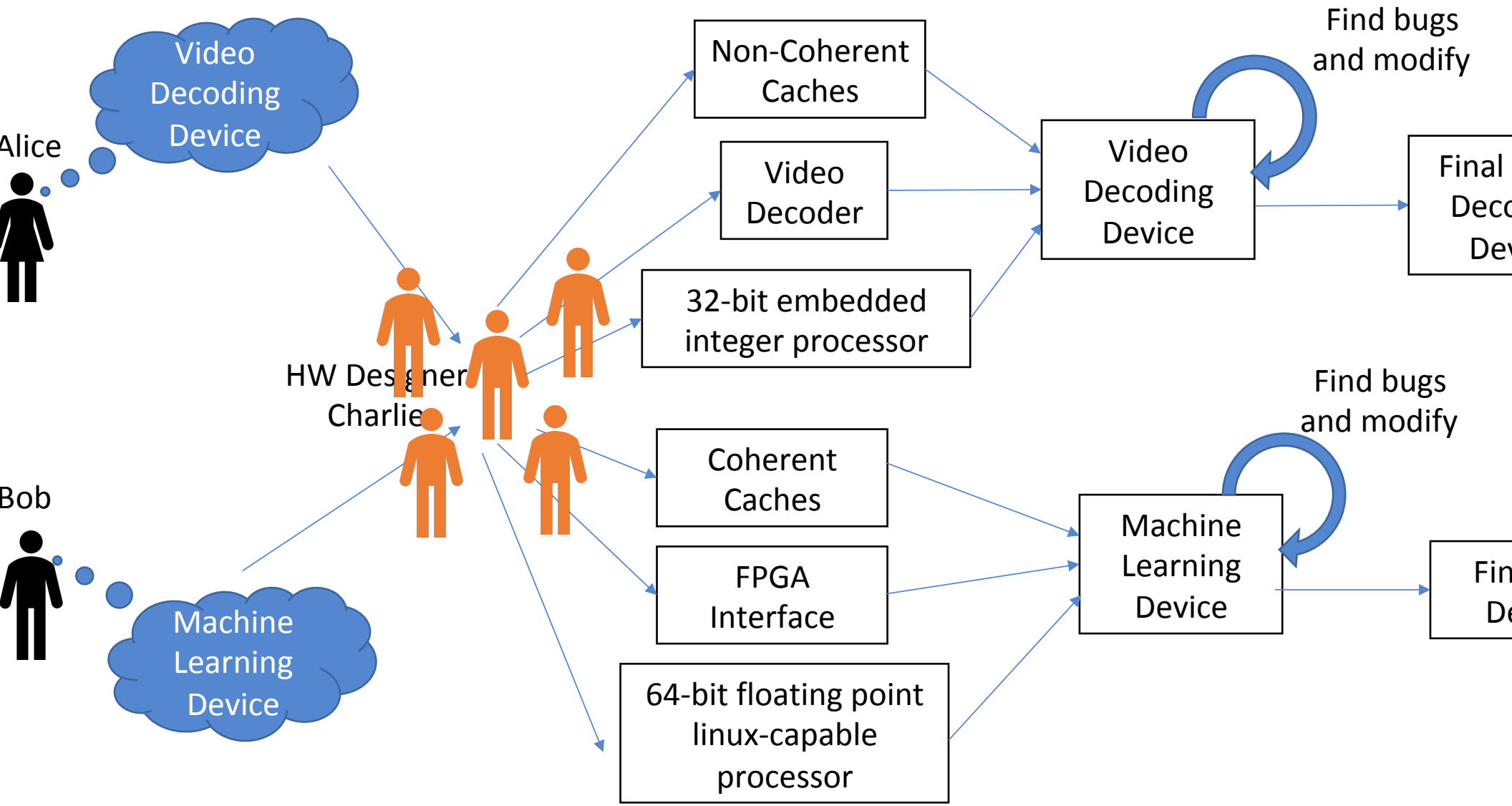
Murali Vijayaraghavan

With Interns: Pedro Abreu, TJ Machado and Kade Phillips

What does SiFive, the company, do

It is basically a hardware design company

State of art in Hardware Design



Contrast with ordering a pizza

Components

Interface

Alice

Bob

re Template
Generator

http://wantpizza.com

Product

Components

Non-Coherent Caches

Video Decoders

Coherent caches

FPGA interface

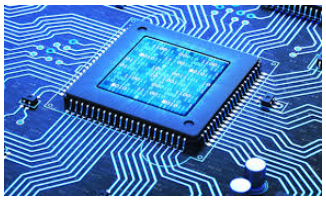
Machine Learn Accelerators

Interface

Tile Link Interconnect

Tile Link Interconnect

Alice



Bob

re Template Generator

RISC-V 32-bit Rocket processor, Integer ALU only, no supervisor mode

RISC-V 64-bit Rocket processor, Integer ALU + FPU, supervisor mode

Product

What are generator templates?

Functions: `Arguments → Hardware Circuit`

- Eg: `(DataWidth x CacheSize) → Processor Core`

Written in a DSL in Scala called *Chisel*

Access to AST allows

- Pretty printing into Hardware Circuits used by backend tools
- Writing Optimization passes directly
- Analyzing the AST to determine compatible parameters
  - TileLink can be configured depending on the number of components - *Diplomacy*

Overall Vision: Verify the generators once and for all

Kami to the rescue

Enables verifying first-order logic based properties

- E.g. ∀ sizes, Cache is Coherent

Security and functional properties can be verified

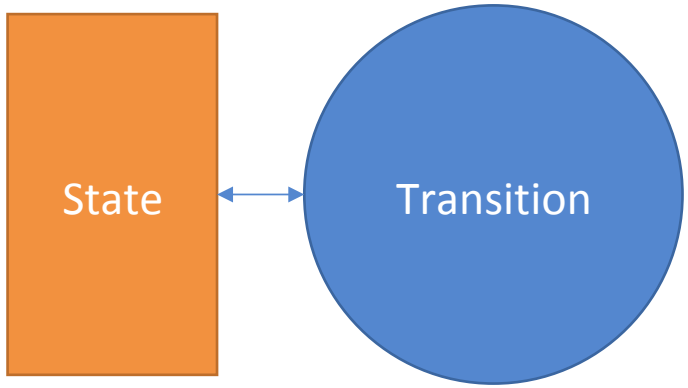
Enables modular specification and verification of each component

- Replace implementation of one module with another without changing any other module

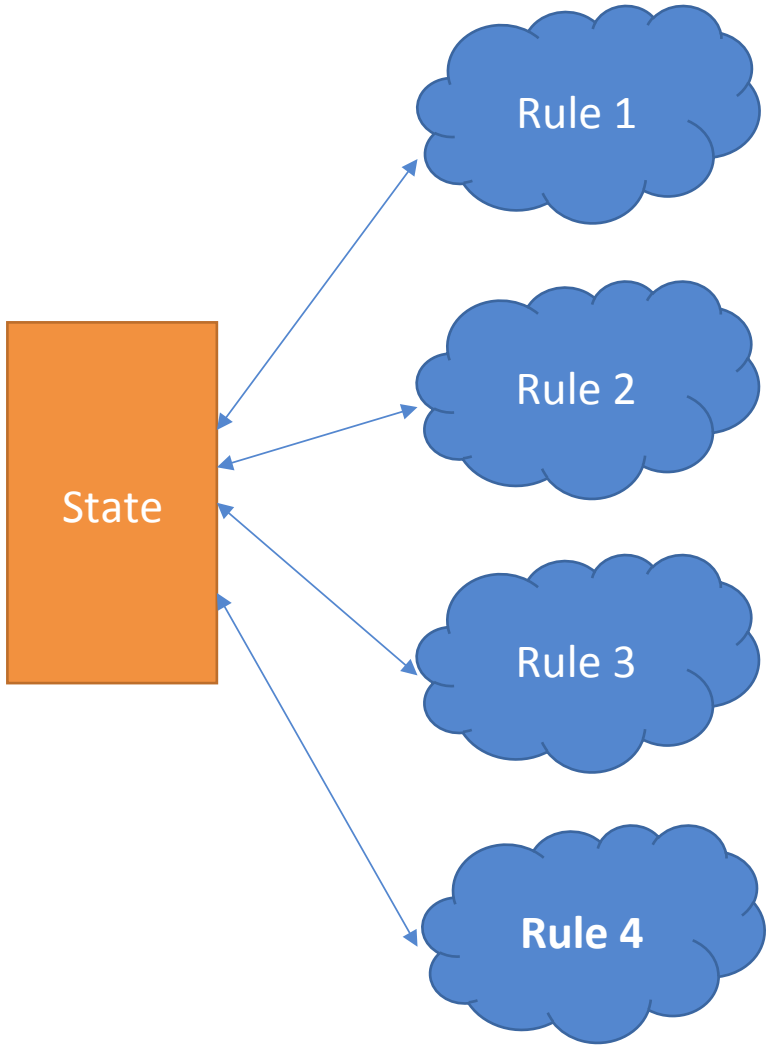
But

- How do we get Kami's assurance about existing Chisel/Scala designs?
- Chisel's semantics are very different from Kami's …

Chisel vs Kami Semantics



Chisel



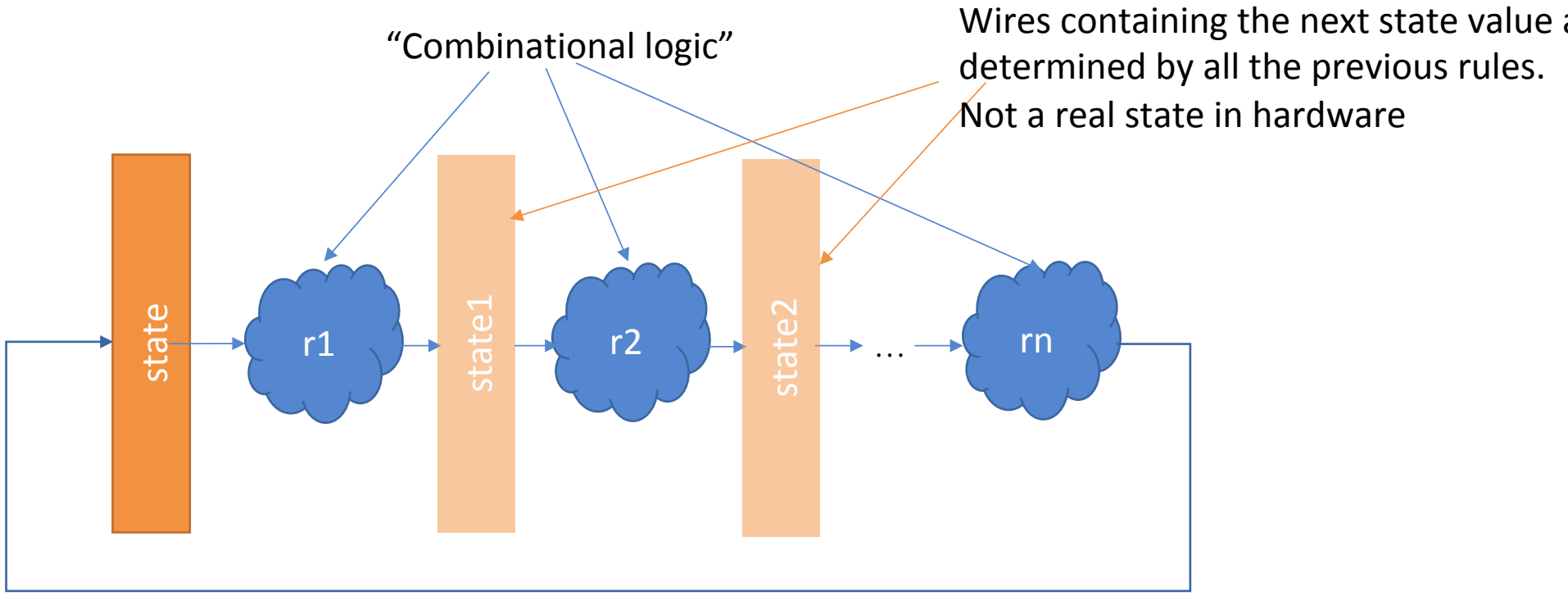
Kami

But after compilation into hardware
circuits, Kami programs are transformed
into state transitions

Compilation into hardware circuits

Multiple rule firings are sequenced in one hardware clock cycle

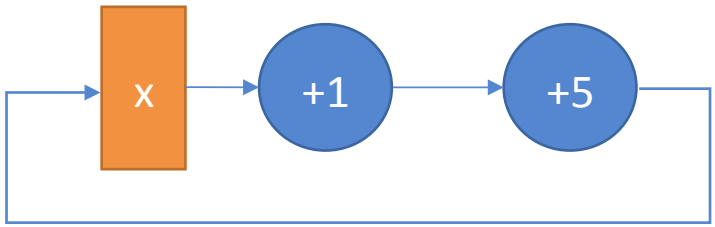
Given an order of firing of rules [r1, r2, …, rn]



Example compilation process

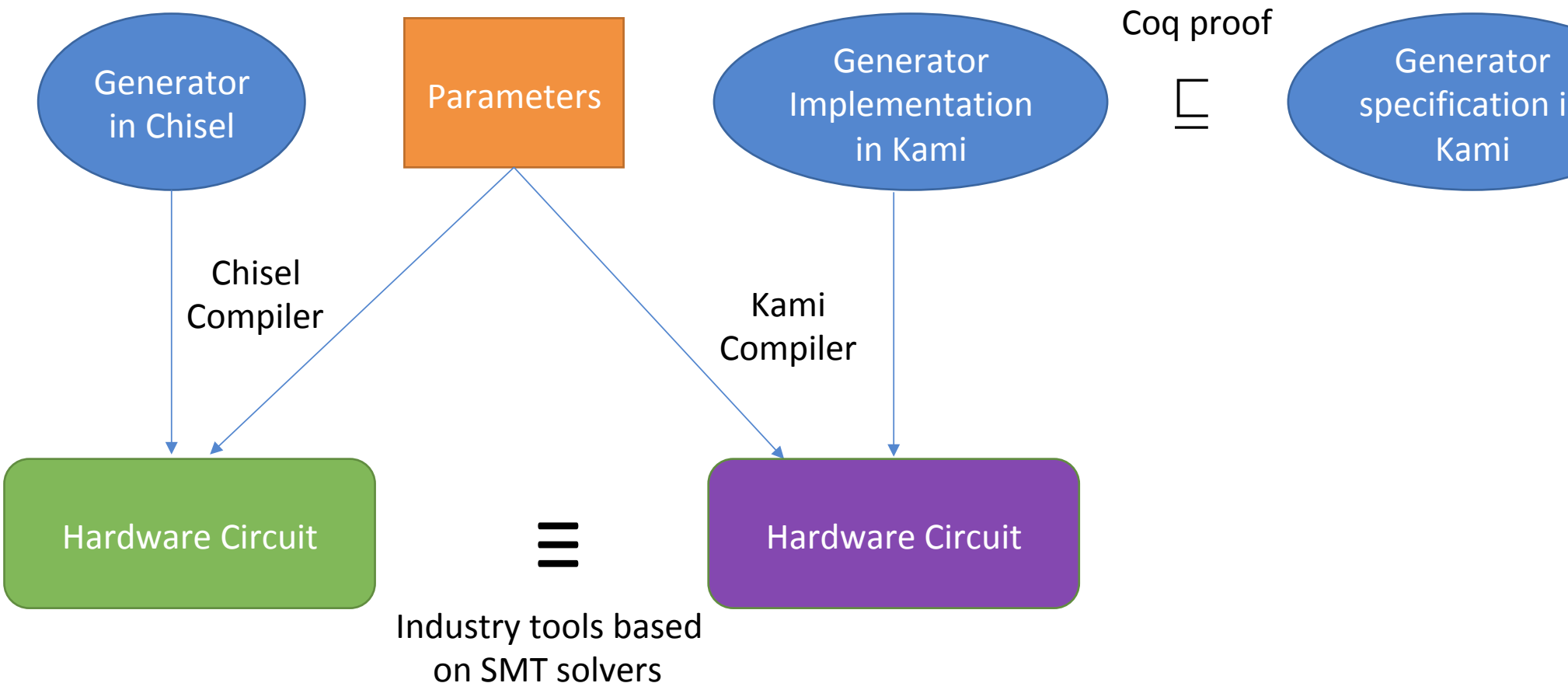
```
Module A:
    Register x
    Rule 1: x := x + 1
    Rule 2: x := x + 5
```

[Rule 1, Rule 2]

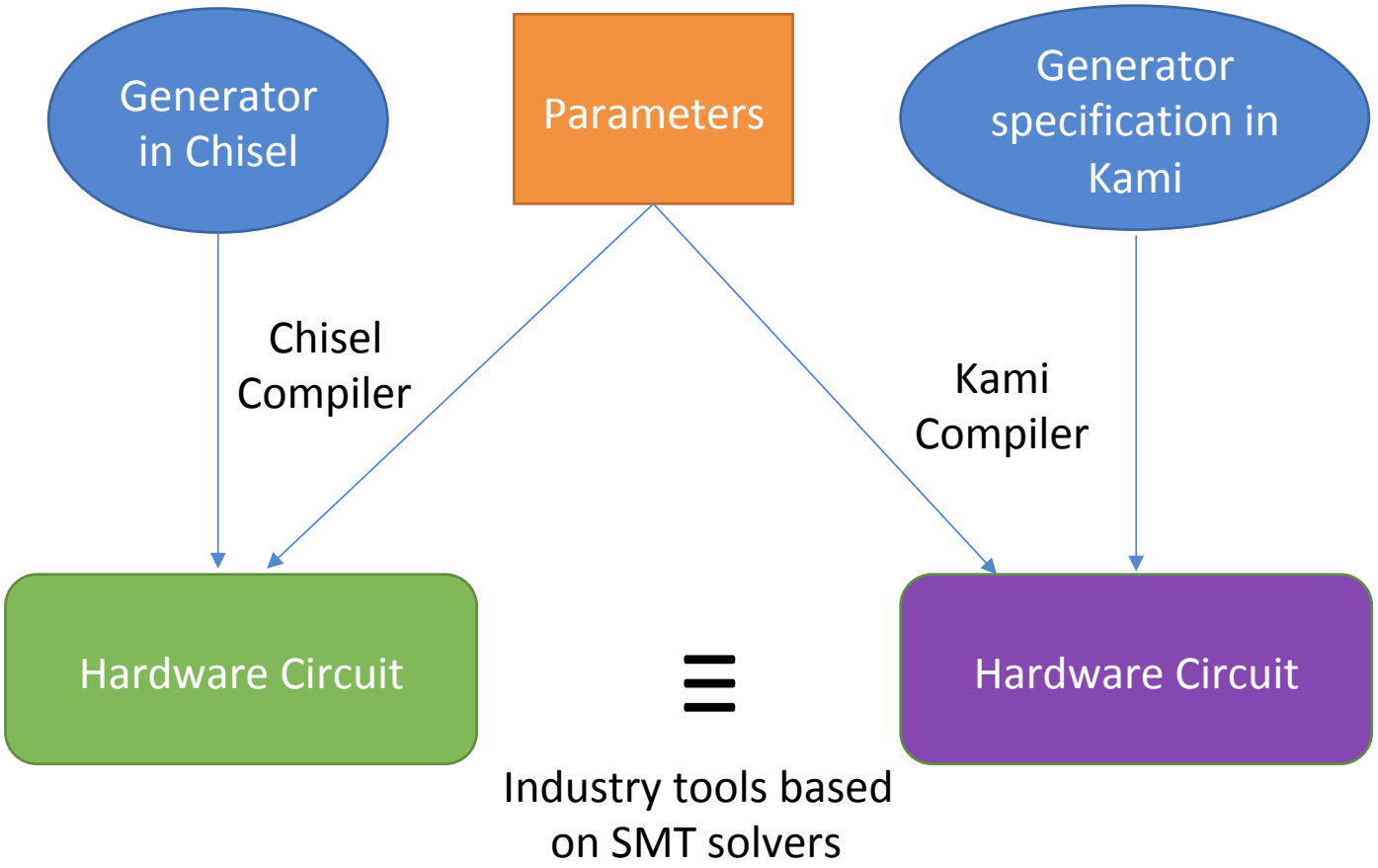


Looks similar to Chisel-based hardware semantics

Short-term plan of reconciling Kami with Chise



Short-term plan of reconciling Kami with Chise



Caveats

For every parameter instance, the SMT-based verification has to be done

- SMT solvers work only if the two hardware circuits are very similar
- But this is push-button

Kami implementation duplicates Chisel implementation effort!

Longer term plan

Design modules (complex processors and others) starting from Kami

Integrate Kami-based designs into Chisel/Scala based design flow

- After generating hardware circuits

Verify Kami compiler

Kami use inside SiFive

Floating Point ALU verification

- Add, Multiply, Division, Square root, Conversions between widths, Conversions to and from Integers
- The specification is very complex
  - Added complexities of understanding open source implementation of FP ALU in Chisel
  - Iteratively comparing specification with Chisel implementation to remove bugs in both

Embedded processor core verification

Other accelerators

Kami development inside SiFive

We are rewriting the Kami internals, including AST, while keeping the surface syntax backwards compatible

- Easy because Coq Notations hide everything
- Goal is to simplify both the proofs of Kami framework theorems and proofs o correctness of hardware designs

Kami version 2!

- https://github.com/sifive/Kami
- Coq 8.8.0

The Kami Intern team at SiFive

ting Point Verification:
ro Abreu, joining Purdue as a PL grad student this Fall

Kami Infrastructure development:
TJ Machado, math grad student at NMSU

Processor Verification:
Kade Phillips, CS Masters Student at MIT

e are hiring for full time positions for Kami-based verification

Backup

Kami version 2 differences

Removed several syntactic restrictions on Kami modules

- Multiple modules can call same method in another module

Original Kami allows two systems to be equivalent if the trace of one is a function of the other, as opposed to being identical

- This complicates proofs of both designs and framework

FMap no longer used to denote sets; instead using lists

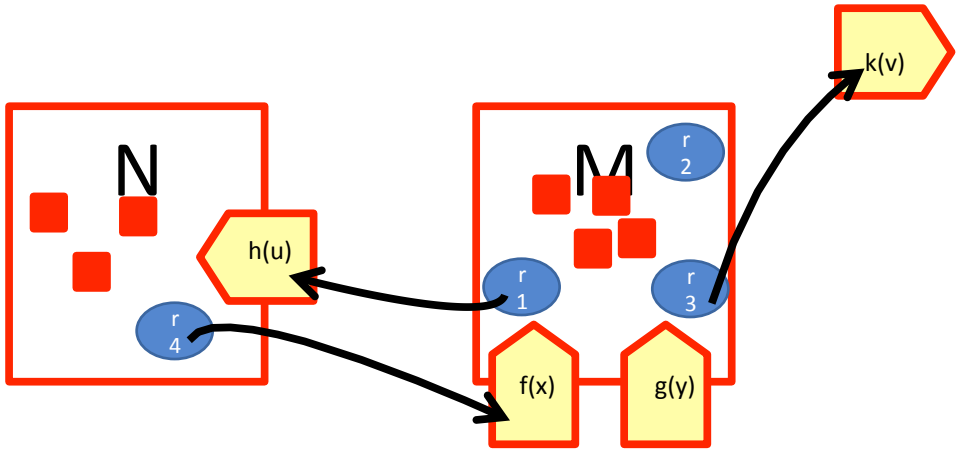
- This makes definition of trace equivalence complicated, but easier to use

Used Coq 8.8.0's inlined ltac in Notations to make syntax much nicer

Steps in Kami Verification Process (1)

Inlining

- Default Kami semantics reasons about steps in a distributed manner

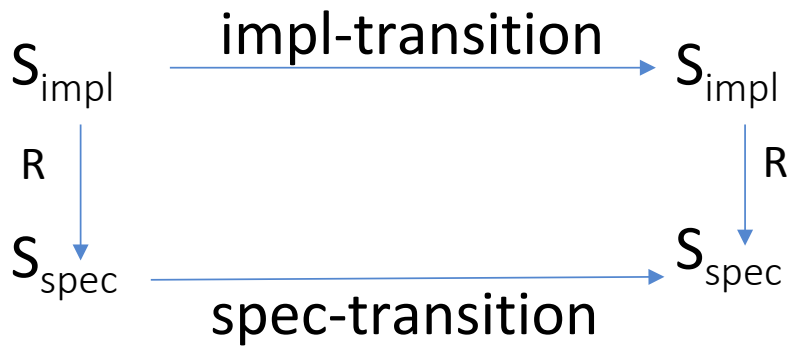


- N produces a trace involving r4, h and f
- M produces a trace involving r1, r2, r3, f, g, h and k
- Composition of M and N produces a trace involving r1, r2, r3, r4, g and k – matching calls and executions cancel out

- We prove that these semantics match the inlined semantics I showed before
  - Thus inlining have to be performed explicitly to use the inlined semantics

Steps in Kami Verification Process (2)

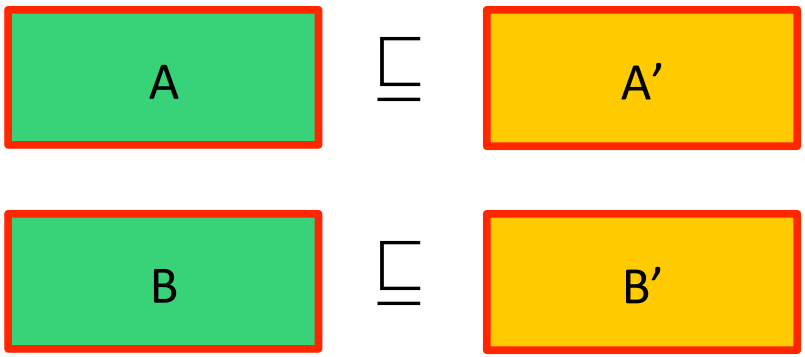
- Simulation Relation

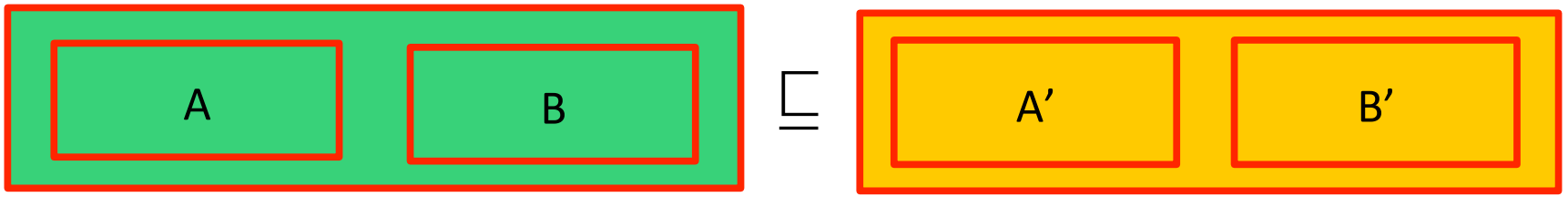


- The transition involves arbitrary combination of defined methods
- For special modules, we have proved simulation relations in the presence of defined meth
- Framework focuses on providing proof automation for systems without defined methods t are externally visible in the final composition

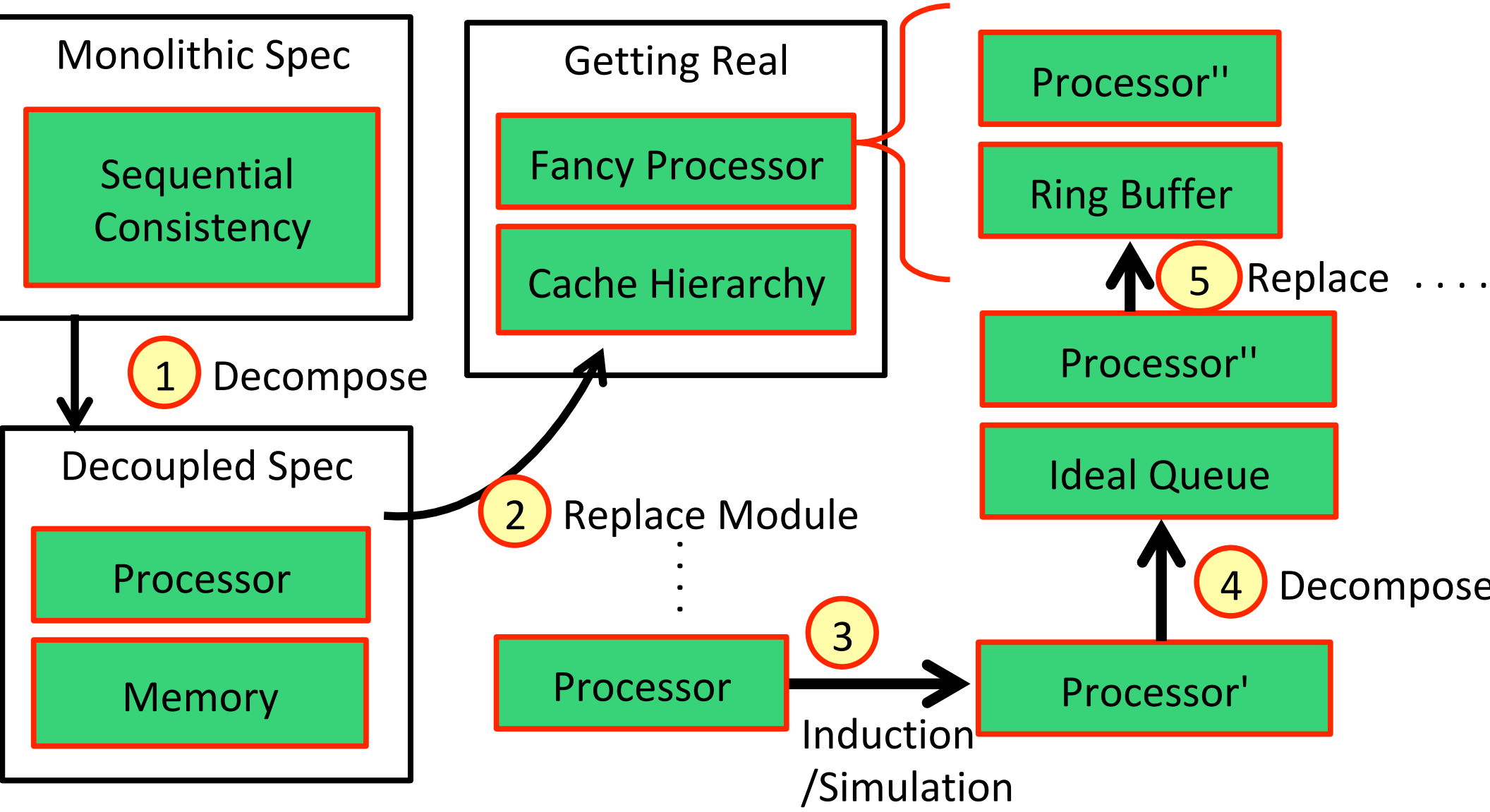
Steps in Kami Verification Process (3)

Modular Substitution Theorem

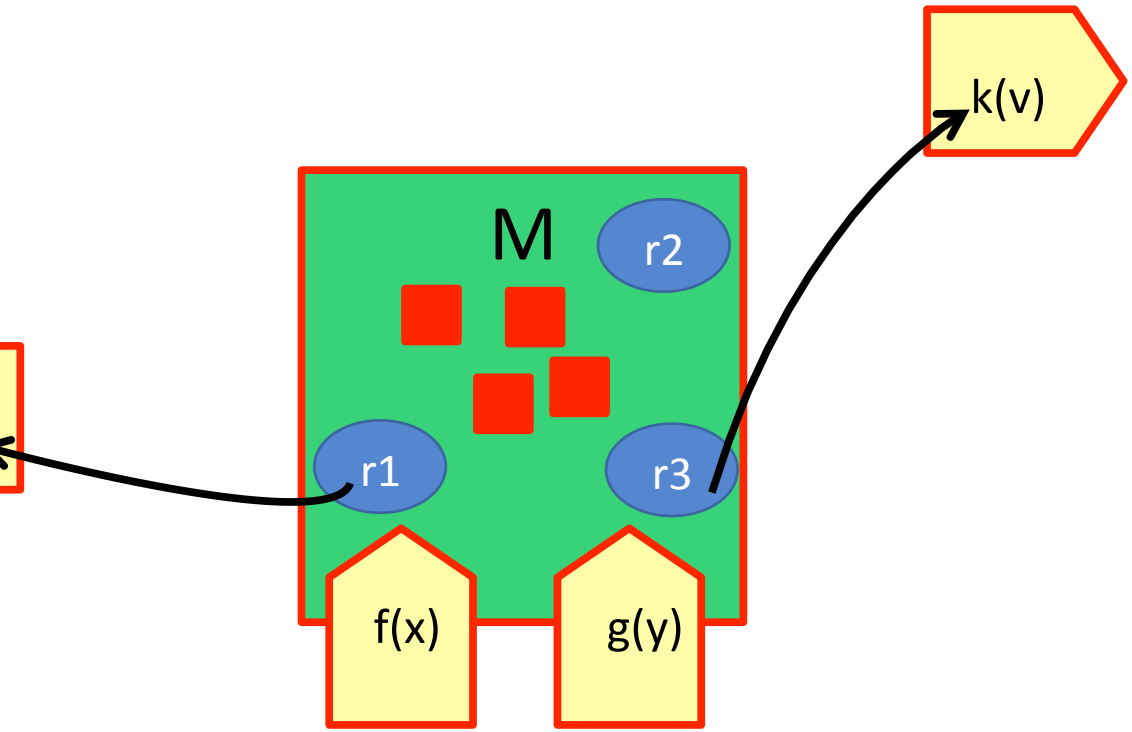




ful Refinement Tactics in Kami



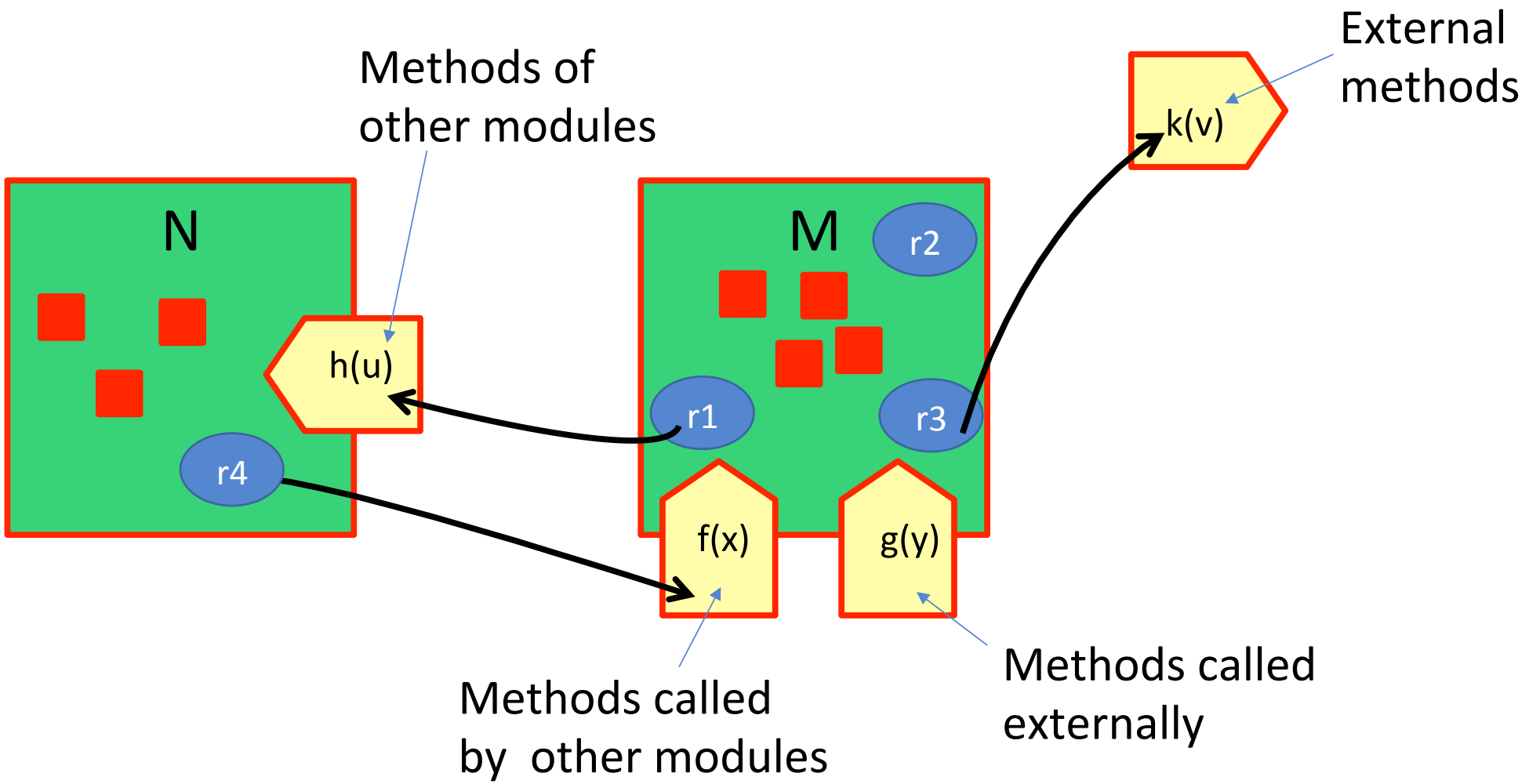
The Big ideas



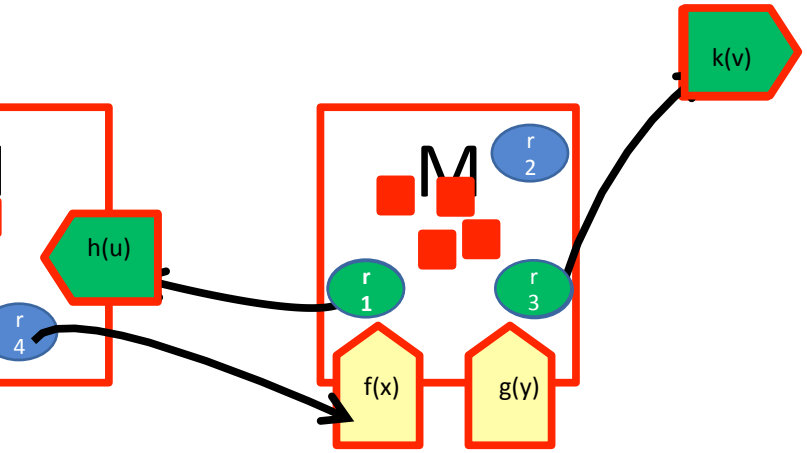
Hardware modules are like objects with

- private state
- methods that other modules can call on this module,
- And atomic transitions or rules that
  - Read and write private stat
  - Call methods

The Big Ideas continued



Modular Semantics of a Kami program



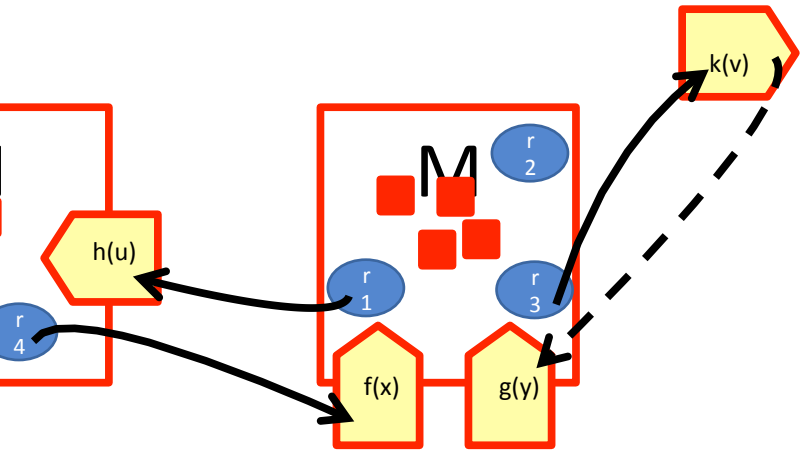
Pick any one rule in any module and "fire" it

- Perform the action corresponding to any meth calls atomically
- If an external method is called, then non-deterministically assume the return value of m
  - The name of the external method, its argument a return value together form the *label* of this step

Repeat the above *step*

A trace of the module is the sequence of labels it produces

What if an external method calls a local method?



- A *step* of a module is more complicated
  - (Several) methods defined by a module ca participate in a step, i.e. get *executed*
    - Incoming Argument is non-deterministically assumed and a return value is calculated using
    - A *label* is a collection of *called* external metho and *executed* local methods of a module
  - A rule need not participate in a *step*
    - External environment may be a module which executes a rule that simply calls local methods module

Summarizing the meaning of a *step* and *trace*

A step executes

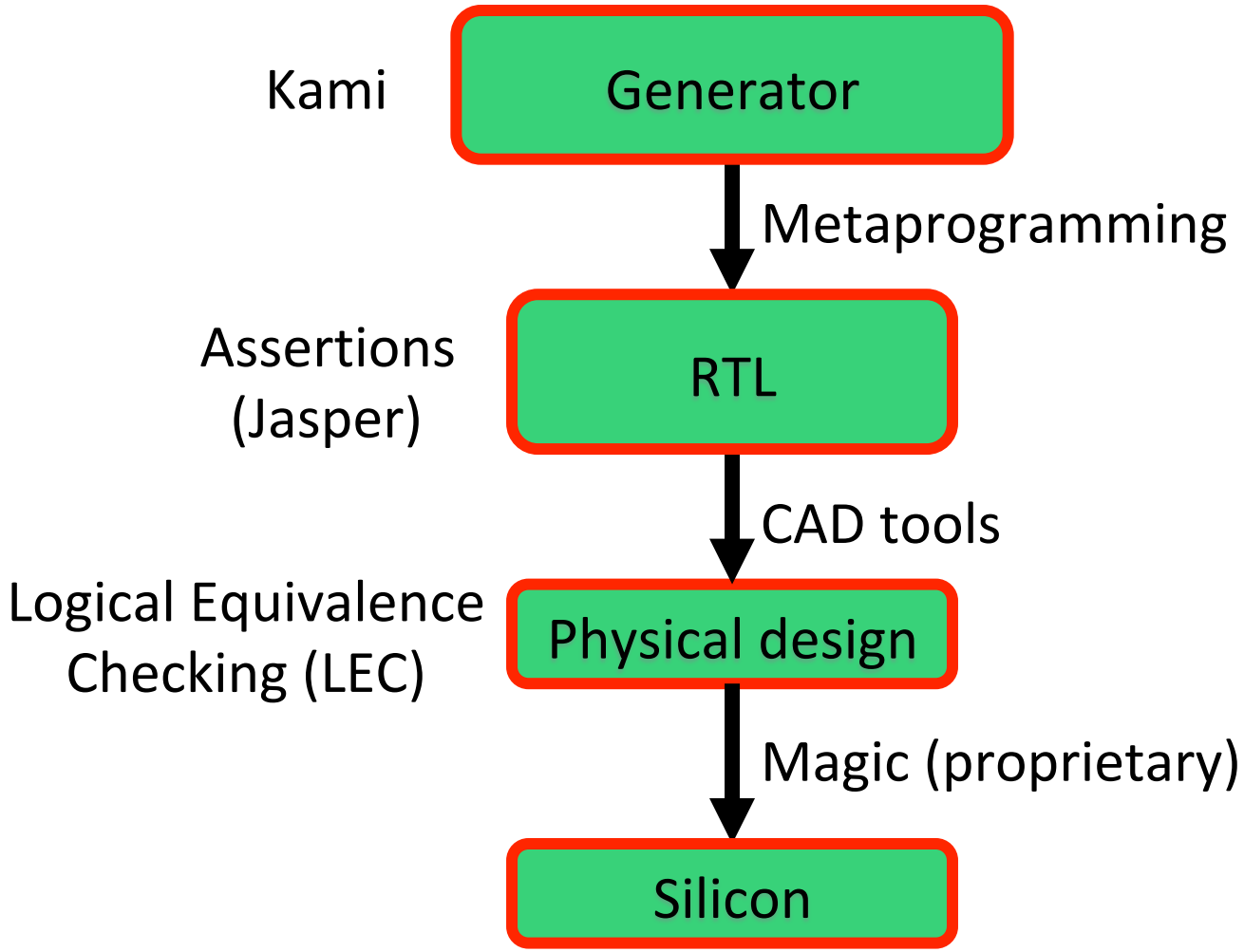
- 0 or 1 rule
- any number of locally defined methods
- Each of these must write disjoint registers

The label produced by a step is the collection of

- called external methods (with arguments and return values)
- executed local methods (with arguments and return values)

A *trace* is a sequence of steps

Hardware Design Flow

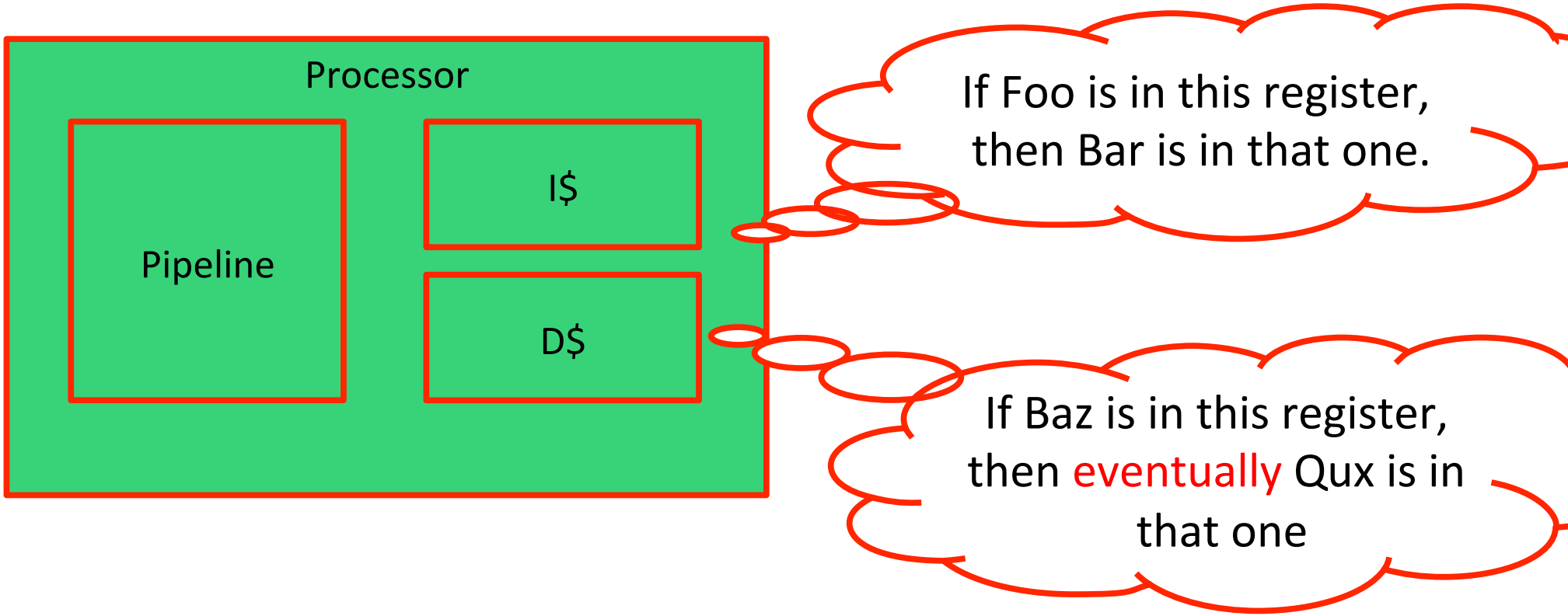


A **framework** to support *implementing*, *specifying*, *formally verifying* parameterized hardware designs at a high abstraction level and *compiling* the designs into RTL

Based on the **Bluespec** high-level hardware design language

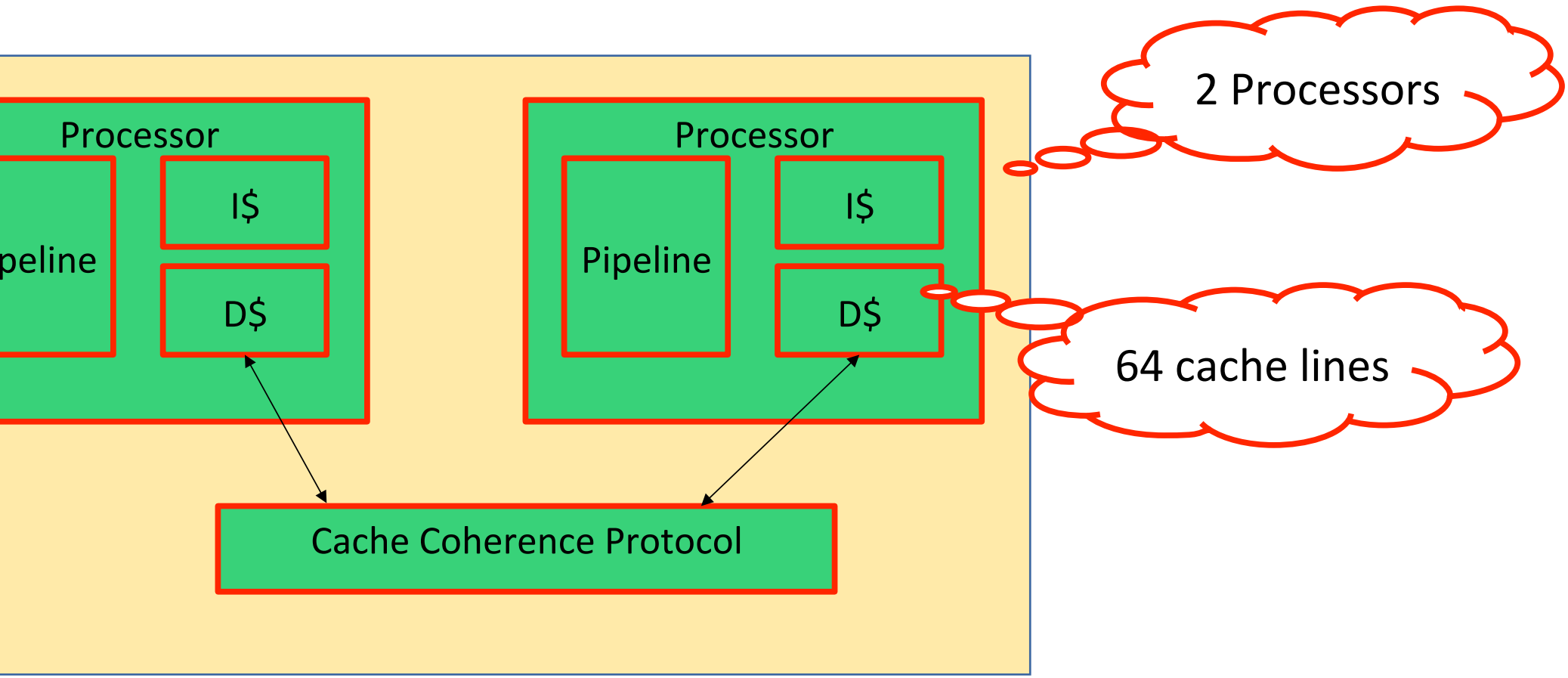
and implemented in the **Coq** proof assistant

Existing Formal Technique: Assertion Verification for shallow properties



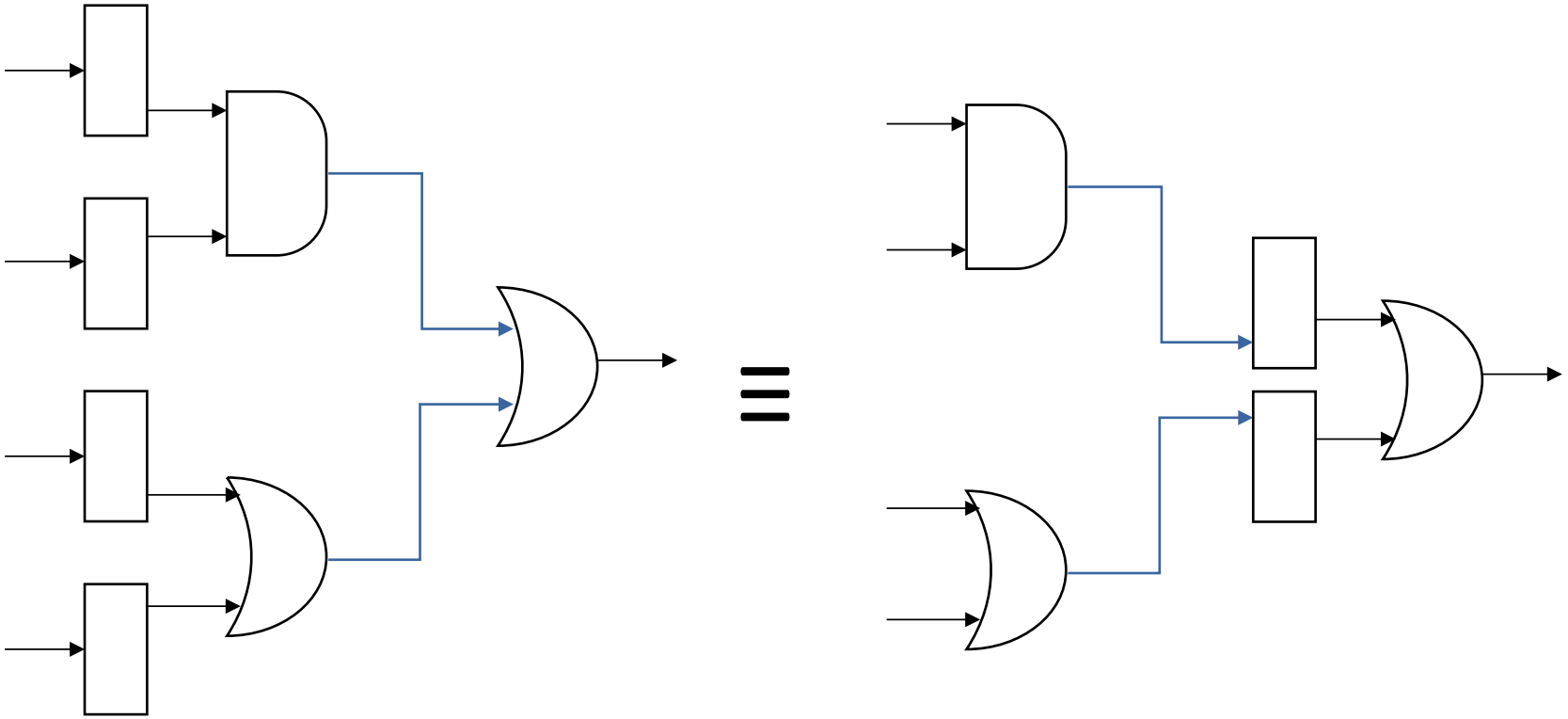
Why does worrying about Foo or Baz increase our confidence in the overall system?

isting Formal Techniques: Specific and Small syste stances



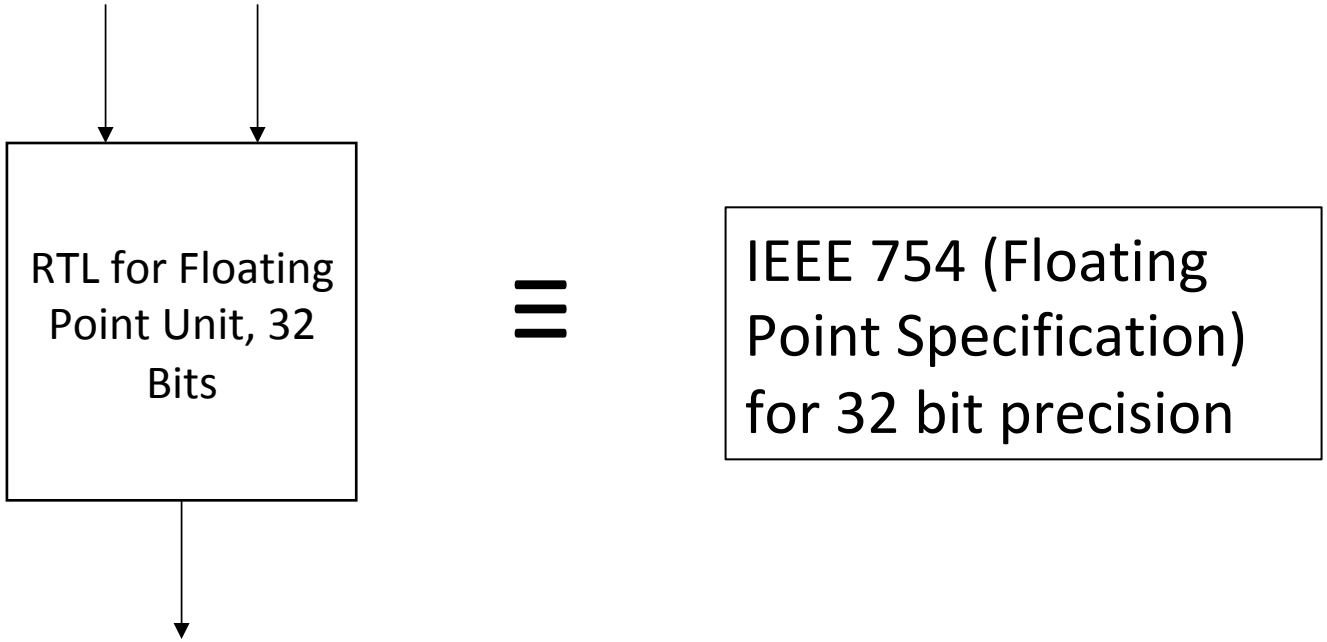
Even a simple chance such as increasing the cache size requires full reverification!

sting Formal Technique: Logical Equivalence Chec



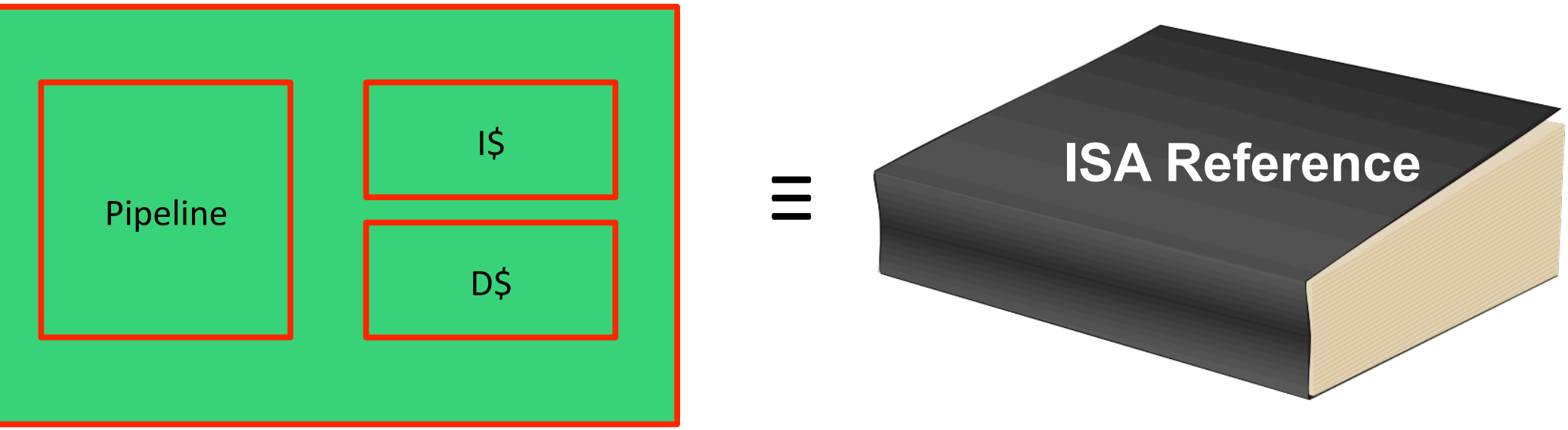
When latency-tolerant components are connected via FIFO buffers, valid refinements no longer preserve Boolean equivalence

ting Formal Techniques: Single component Verifica

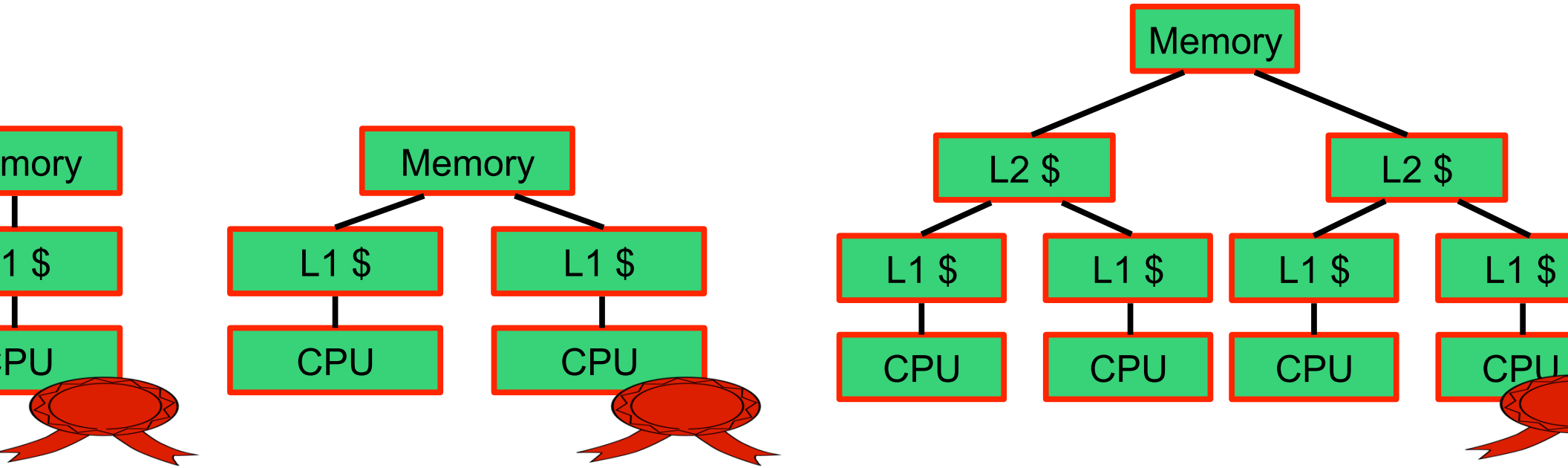


How do we know that this component behaves correctly in the processor's context?

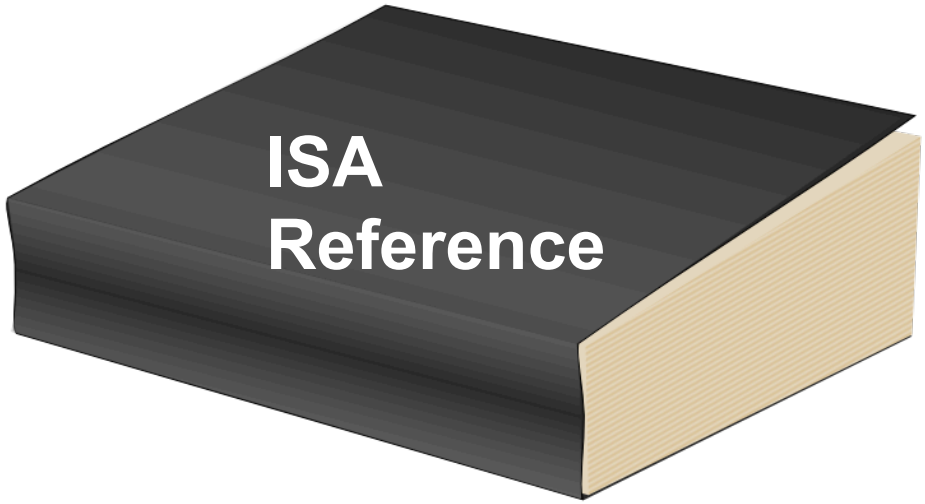
mi: Proving Deep, High-Level Properties



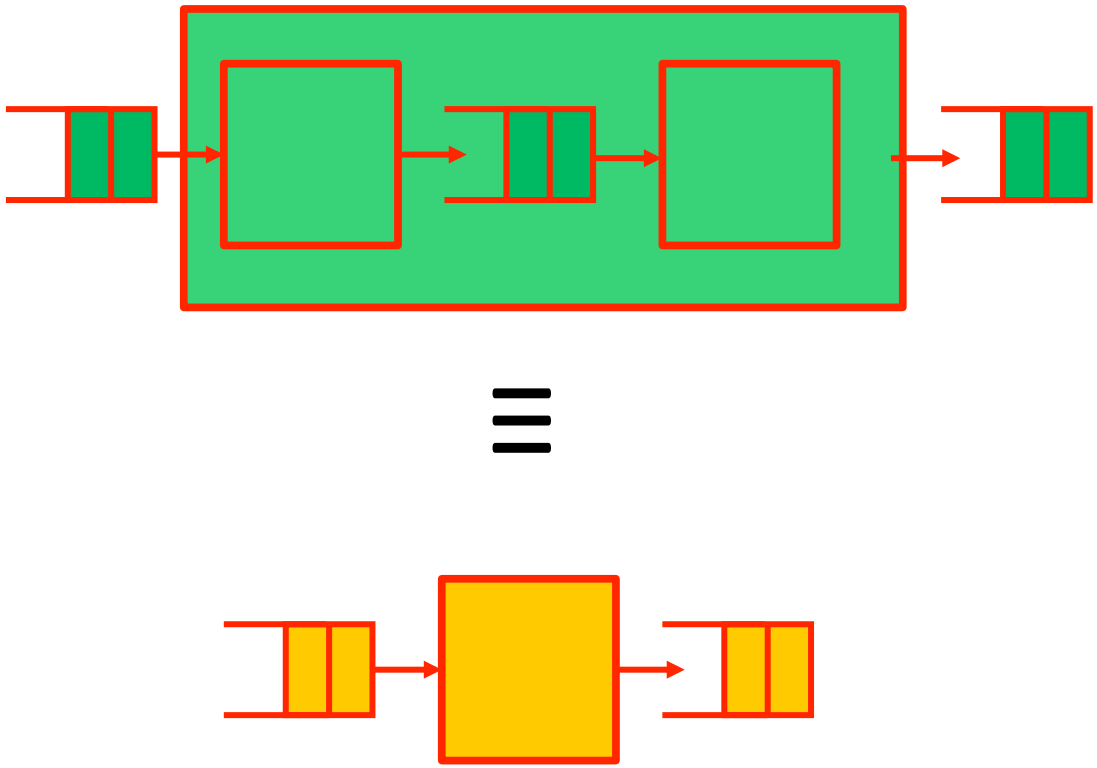
ni: Prove a Family of Instances for Parameterized Designs



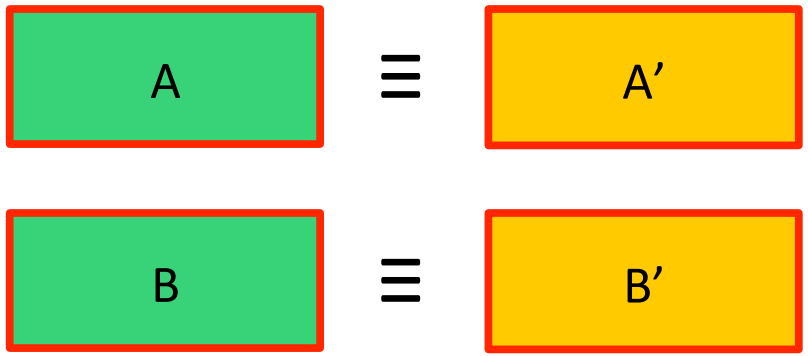
∀ trees. ≡

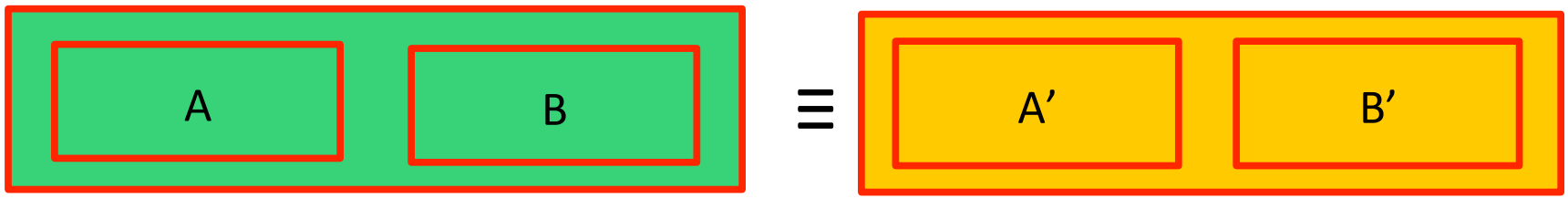


Kami: Latency-tolerant semantics

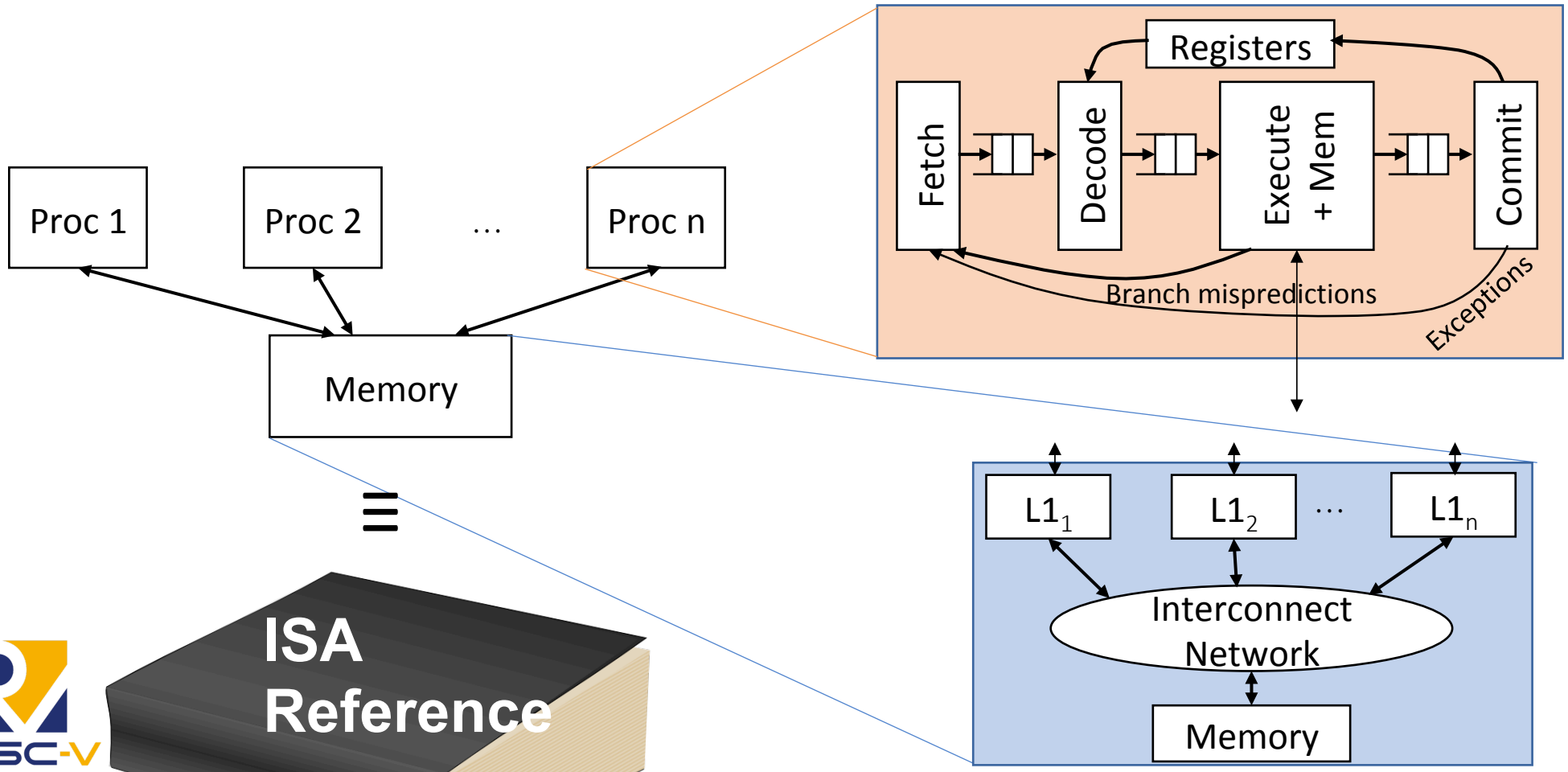


Kami: Modularly composing proofs





Case study: Multiprocessor



≡

(SC)

MSI cache coherence protocol

Runs multithreaded C programs on FPGAs

```
ker's algorithm

enter;
rn;
nter;
ead(int i) {
[i] = 1;
 (enter[1-i] == 1) {
(turn != i) {
nter[0] = 0;
hile (turn != 0){}
nter[i] = 1;


er = counter + 1;
= i;
[i] = 0;
n (counter);
```

Gcc

Compcert

RISC-V binary

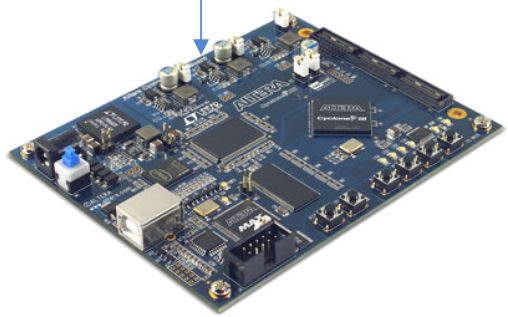
Verified Kami multiprocessor implementation

Kami Compiler

Multiprocessor circuit

Put into Instruction memory

Run



At SiFive

nvestigating its use for verifying certain components

Meaning of Modular Refinement

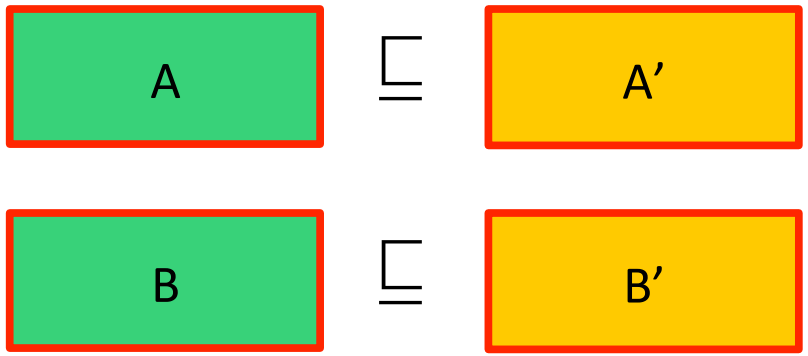
A refines A', denoted A $\sqsubseteq$ A':

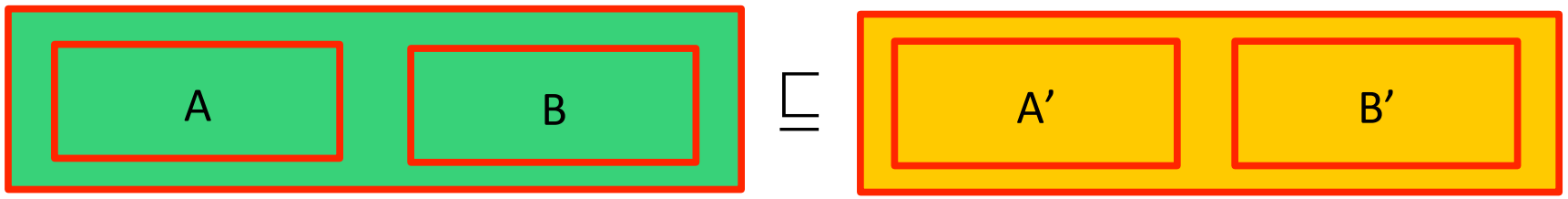
f any trace produced by A can also be produced by A'

- modulo empty-steps, i.e. steps with no methods
- modulo rule names

In any context, A can be replaced by A'

Modular Composition of Kami modules

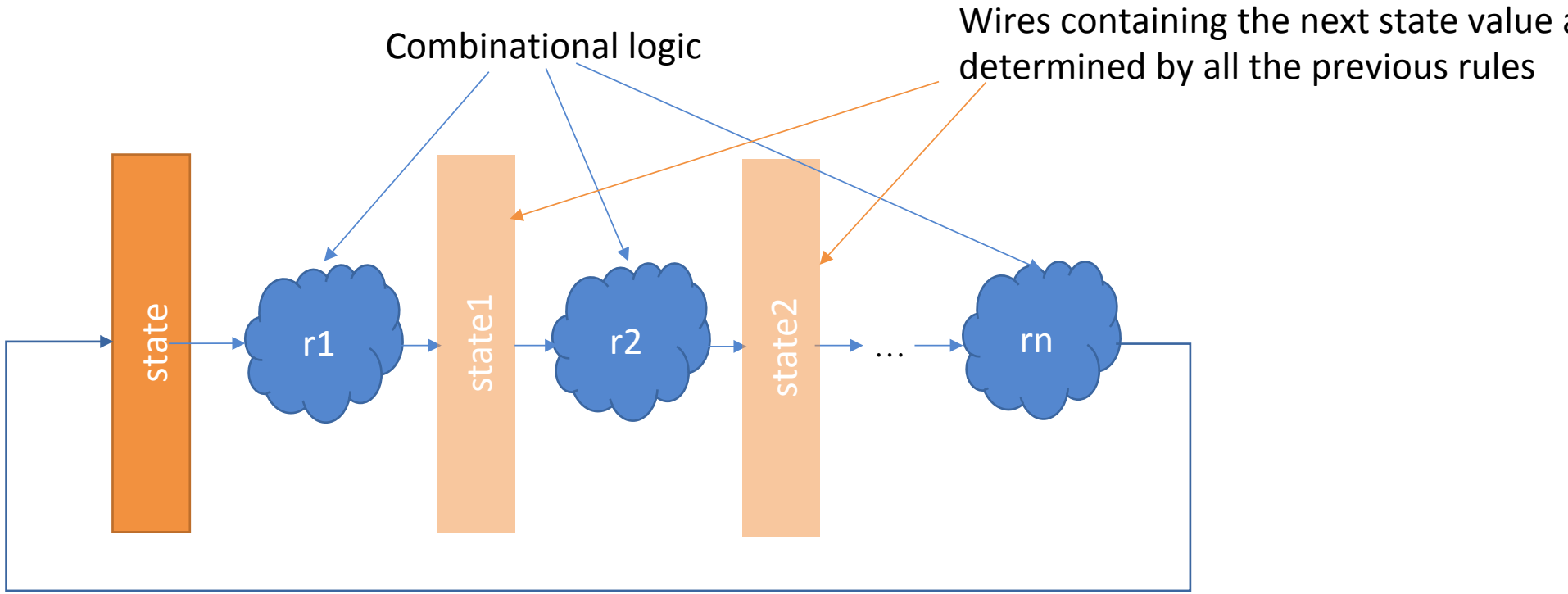




Compilation into RTL

Maps multiple rule firings into one hardware clock cycle

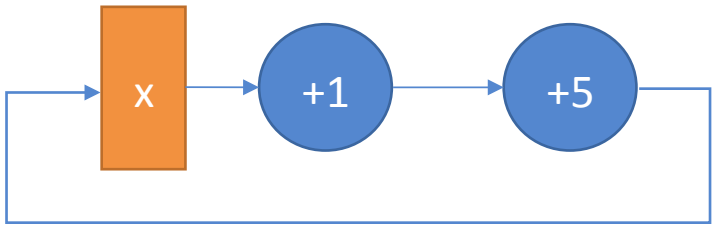
Given an order of firing of rules [r1, r2, …, rn]



Example compilation process

```
Module A:
    Register x
    Rule 1: x := x + 1
    Rule 2: x := x + 5
```

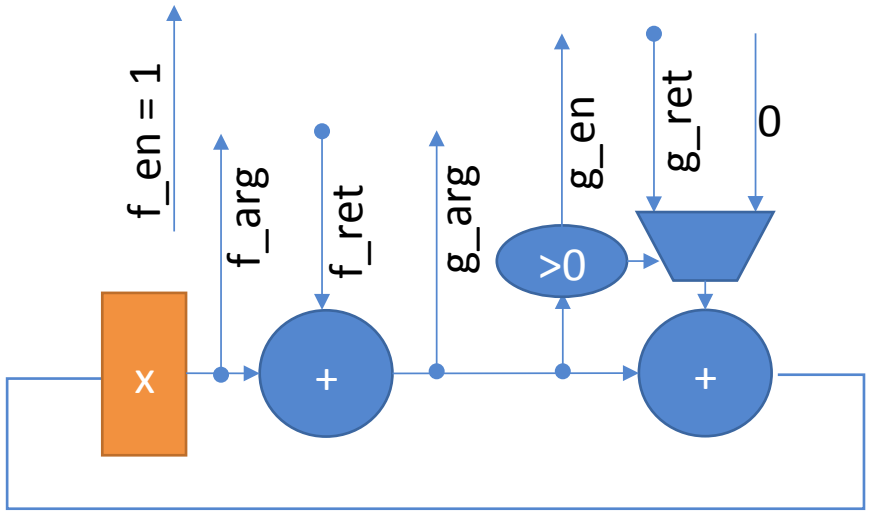
[Rule 1, Rule 2]



What about methods?

```
Module A:
    Register x
    Rule 1: x := x + f(x)
    Rule 2: x := x + if (x > 0) then g(x) else 0
```

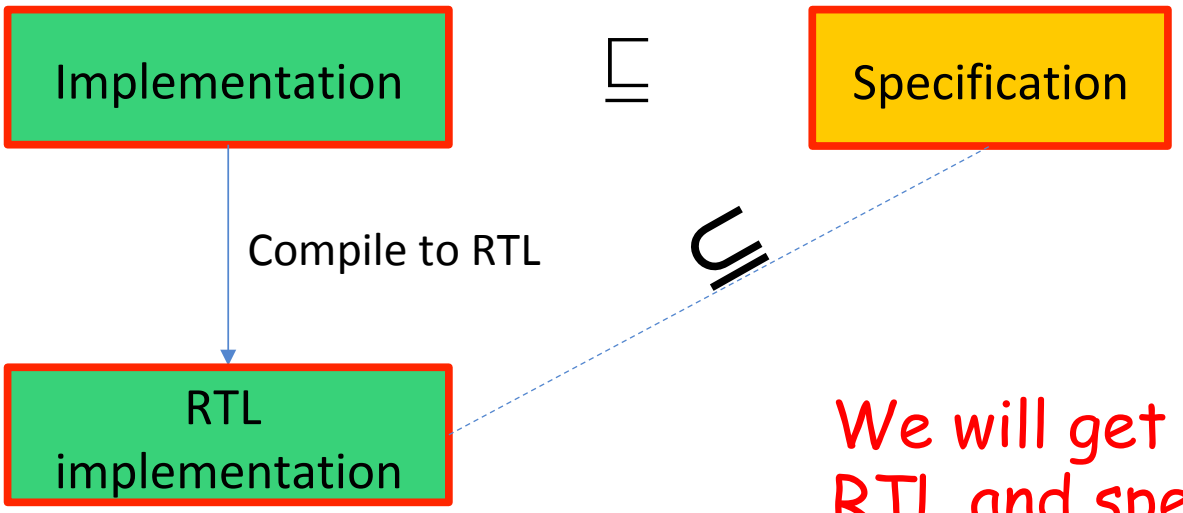
[Rule 1, Rule 2]



Defined methods?

- If exactly one use, then in at call
- Otherwise currently canno synthesize (not seen in practice)

Kami Verification – High Level View



We will get the ⊆ relation betwe
RTL and specification once the
compiler is formally proven

State of Kami

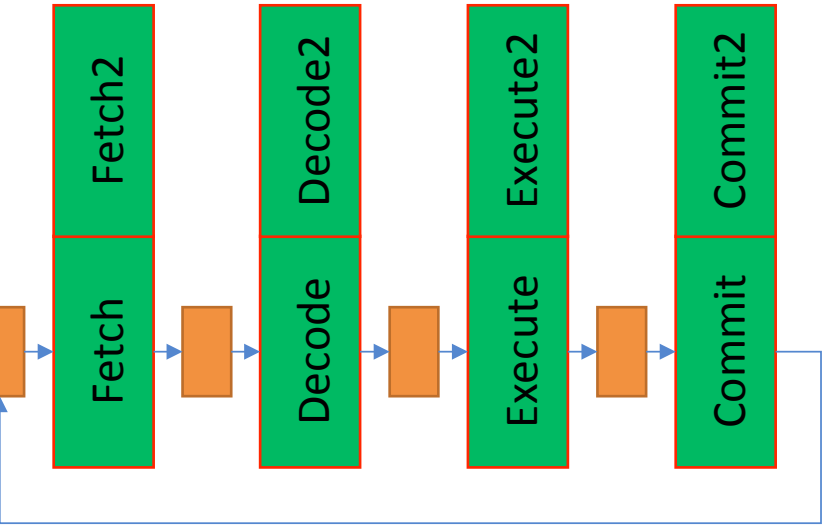
Stable Kami Version 1 developed at MIT

- http://github.com/mit-plv/kami
- http://github.mit.edu/plv/kami-puar
  - The 8-stage pipeline with privileged mode support
  - Work in progress
  - Needs collaboration account with MIT

Kami Version 2 at SiFive

- http://github.com/sifive/Kami
  - Complete rewrite from scratch based on lessons learnt from Kami Version 1
  - Some infrastructure work needs to be completed

Why One-Rule-at-a-time Semantics



$\sqsubseteq$

Atomic Processor

Schedule: [Commit, Execute, Decode, Fetch]

dule: [Commit, Commit2, Execute, Execute2, Decode, Decode2, Fetch, Fetch2]

othing changes in the verification, only schedule for compiler changes!

What about Bluespec Guards

Bluespec has a notion of guards, which affects if a rule is *ready-to-fire* or not

- The guard is like a predicate on *all* of the rule's actions
- Essentially simplifies syntax for specifying state transitions denoted by a rule

But the notion of guard is extended to any action

- If an action's guard is false, then not only is this action disabled, but also any other action that is supposed to atomically execute with this action

```
x := x + 1
if(pred)
then { guard(g); y := y + 1 }
```

If [pred] is true, and [g] is fal
[x := x + 1] wont fire

This affects the circuit generated for methods:

- Any called method creates an input guard wire, which must be true for the calling rule to fire

Backup

Properties of Kami modules

Reflexivity: $A \sqsubseteq A$

Transitivity: $A \sqsubseteq B$, $B \sqsubseteq C \Rightarrow A \sqsubseteq C$

Associativity of composition: $A+(B+C) = (A+B)+C$

Commutativity of composition: $A+B = B+A$

Substitution theorem:

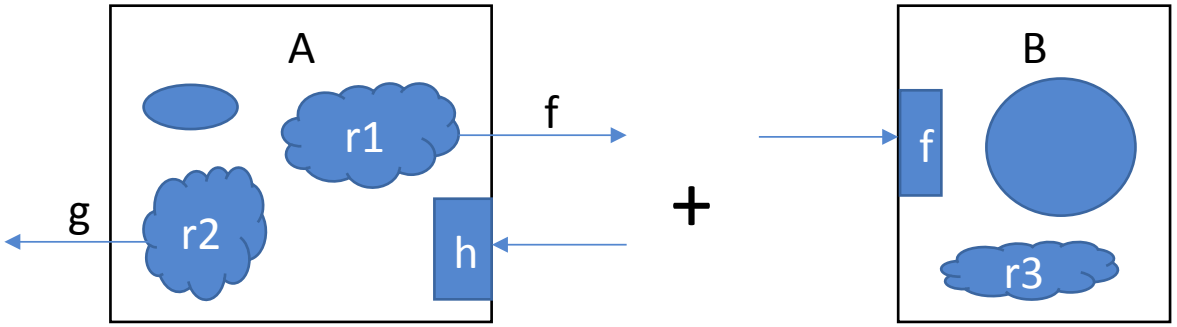
$A \sqsubseteq A' \Rightarrow A+B \sqsubseteq A'+B$

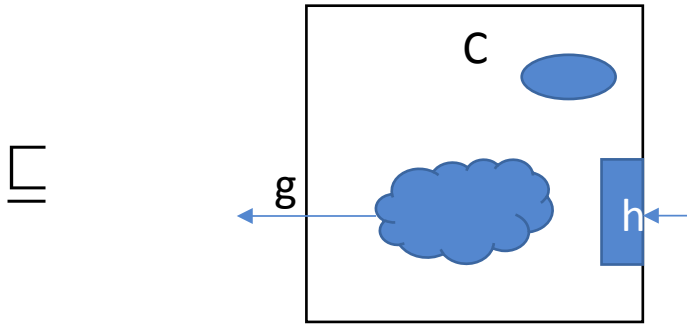
Kami Processor Implementation

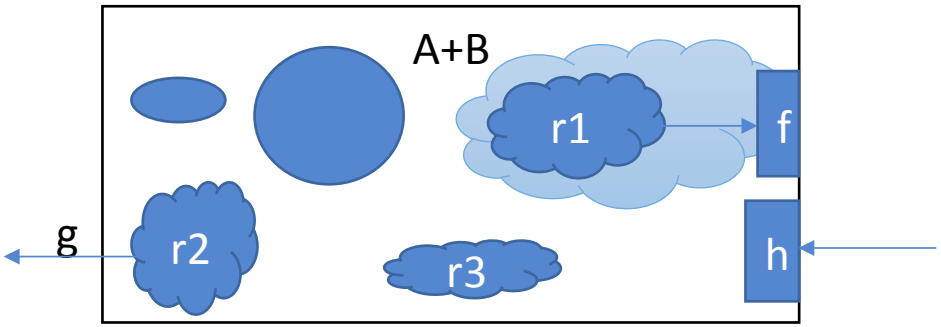
	Kami	Bluespec
$IPC_{average}$	0.21	1.00
Clock Frequency (MHz)	142.86	31.25
#LUTs per core	3,061	2,184
#FFs per core	1,545	3,440

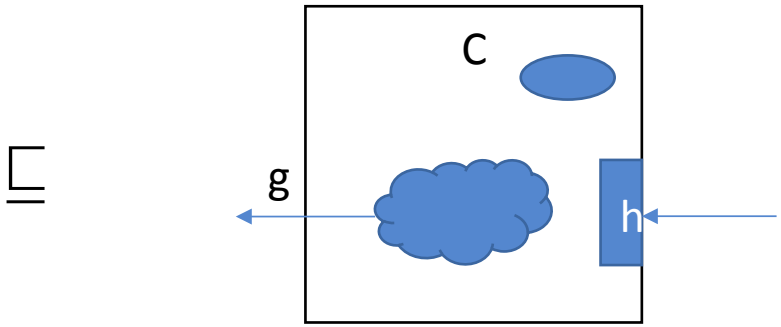
Fig. 15. RISC-V cores in Kami and Bluespec

nlining

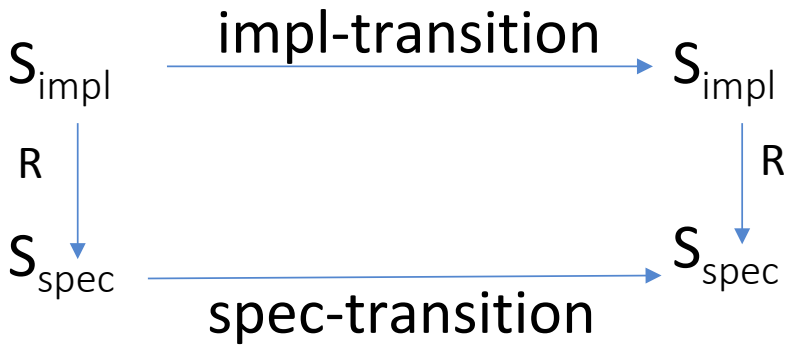








nduction/Simulation proof



- Invariants of the implementation has to be specified to prove a simulation relation on states

Examples

$$\{\texttt{counterReg} \mapsto v\} \xrightarrow{\{\text{enq}(v)=()\}} (\{\texttt{counterReg} \mapsto v+1\}, ())$$

```
Read val <- "counterReg";
Call "enq"(#val);
Write "counterReg" <- #val + $1;
Retv
```

$$t \neq h \Rightarrow \{\texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow{\{\}} (\{\texttt{tail} \mapsto t+1\}, e(t))$$

```
Read elts <- "elts";
Read head <- "head";
Read tail <- "tail";
Assert (#tail != #head);
Write "tail" <- #tail + $1;
Return #elts@[#tail]
```