Verification Around the Hardware-Software Interface:
Instruction Set, Processors, and Side Channels

Adam Chlipala
Deep Specifications Workshop (colocated with PLDI)
June 2018

Joint work with:
Arvind, Thomas Bourgeat, Joonwon Choi, Ian Clester, Faye Duxovni, Andres Erbsen, Samuel Gruetter, Luke Sciarappa, Benjamin Sherman, Murali Vijayaraghavan, Andrew Wright

Summary: **Results So Far**

Yale | Princeton

CompCert

Yale | Princeton

MIT

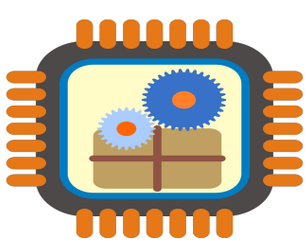
MIT

Verified OS kernel
Initial steps porting to RISC-V

Enhanced verified compiler (influence & connect 64-bit CompCert)
Proved new pass with *flat memory model*

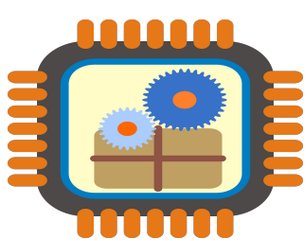
Formal semantics of RISC-V ISA
Committee agrees on our semantics to standardize

Coq framework for coding & verifying hardware
First open-source release & paper [ICFP'17]



MIT

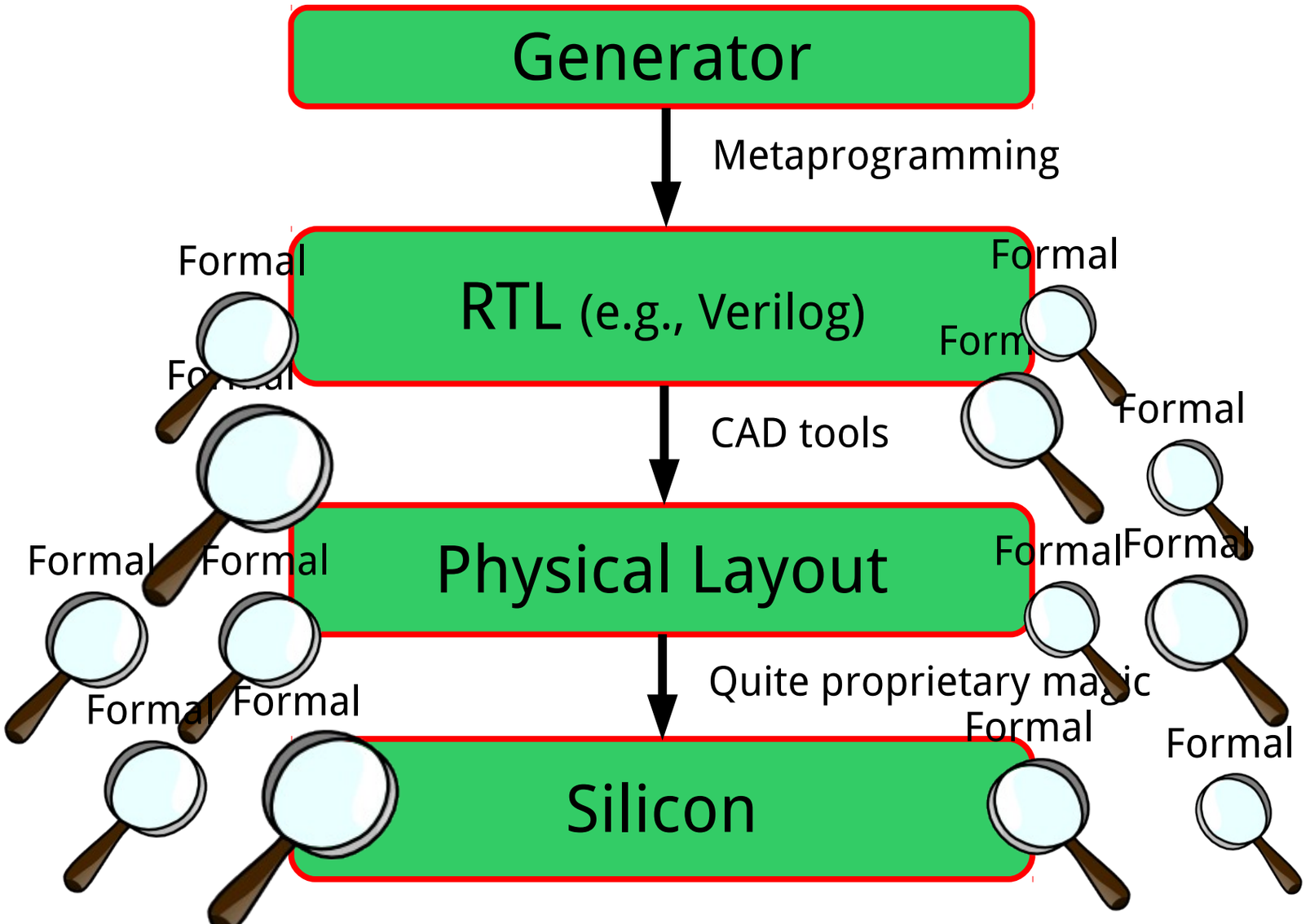
Verified simple processor
Case study from paper



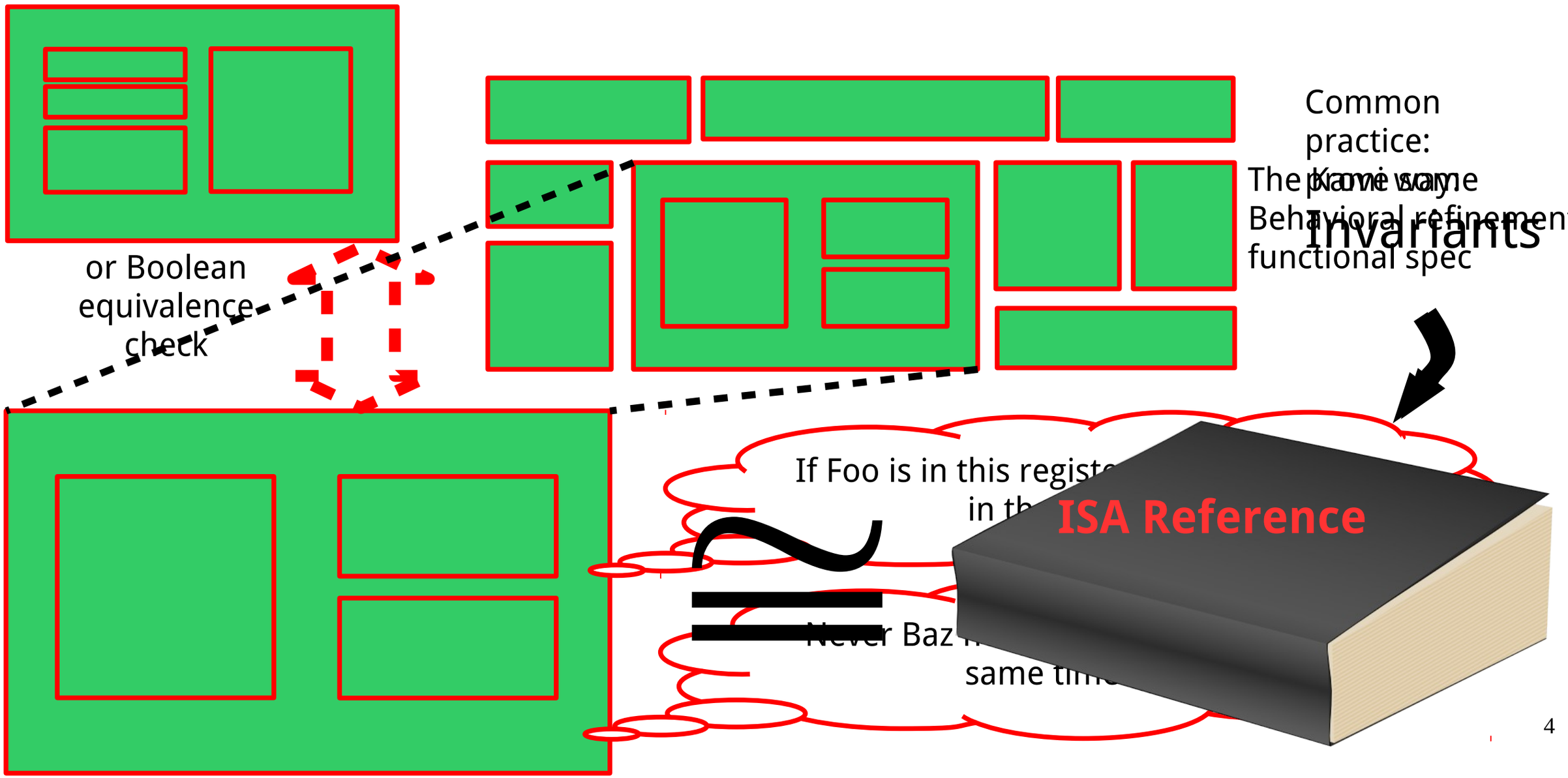
MIT

Start of fancier processor
Virtual memory, etc.
Proved against spec, but spec still being debugged

A Cartoon View of Digital Hardware Design

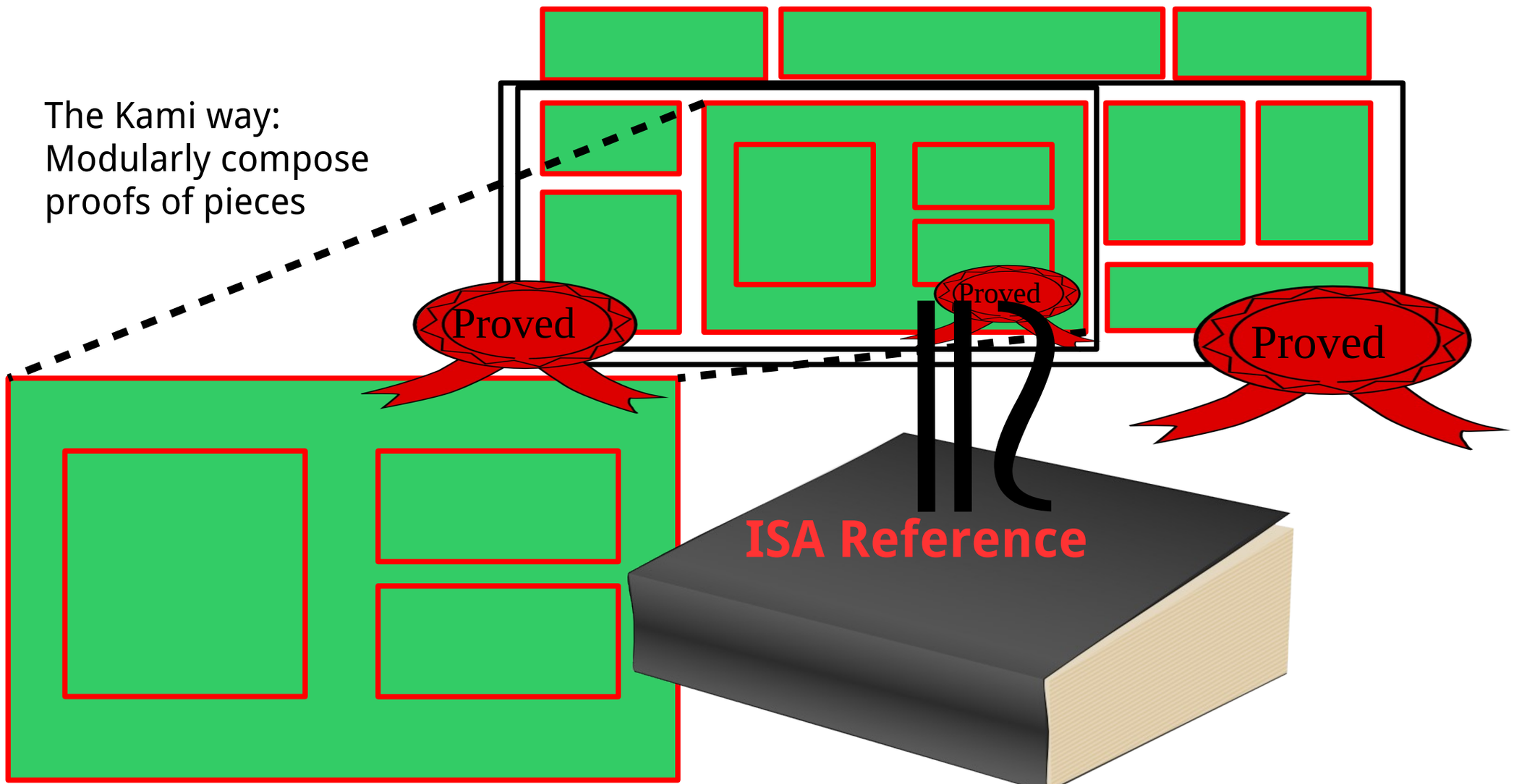


Simplification #1: Prove a Shallow Property

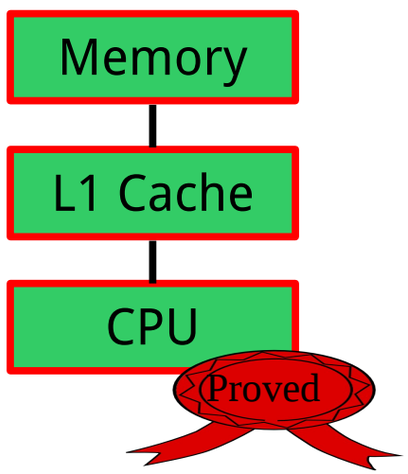


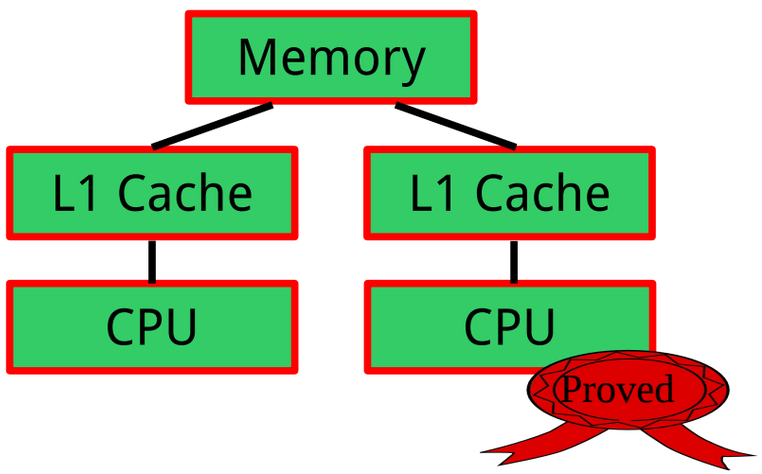
Simplification #2: Analyze Isolated Components

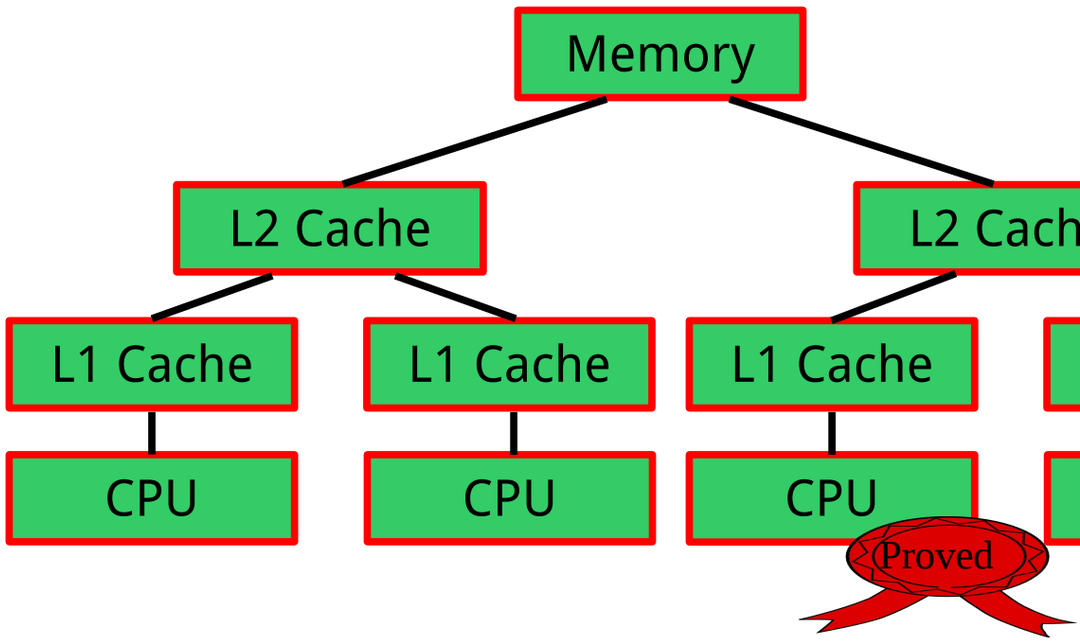
The Kami way:
Modularly compose
proofs of pieces



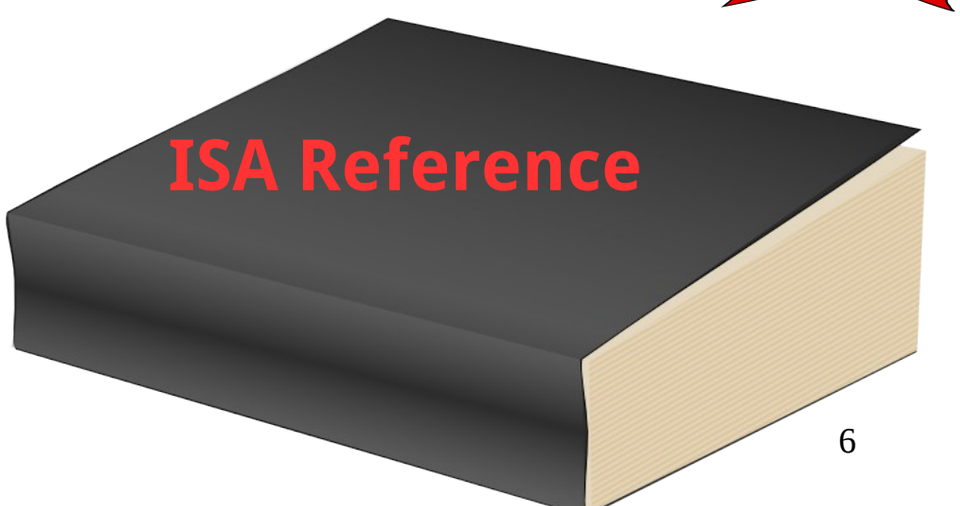
Simplification #3: Start Over For Each Design







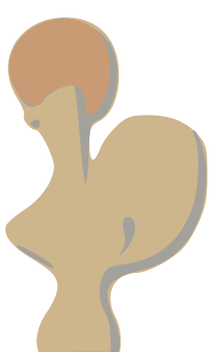
$\forall$ trees. $\cong$



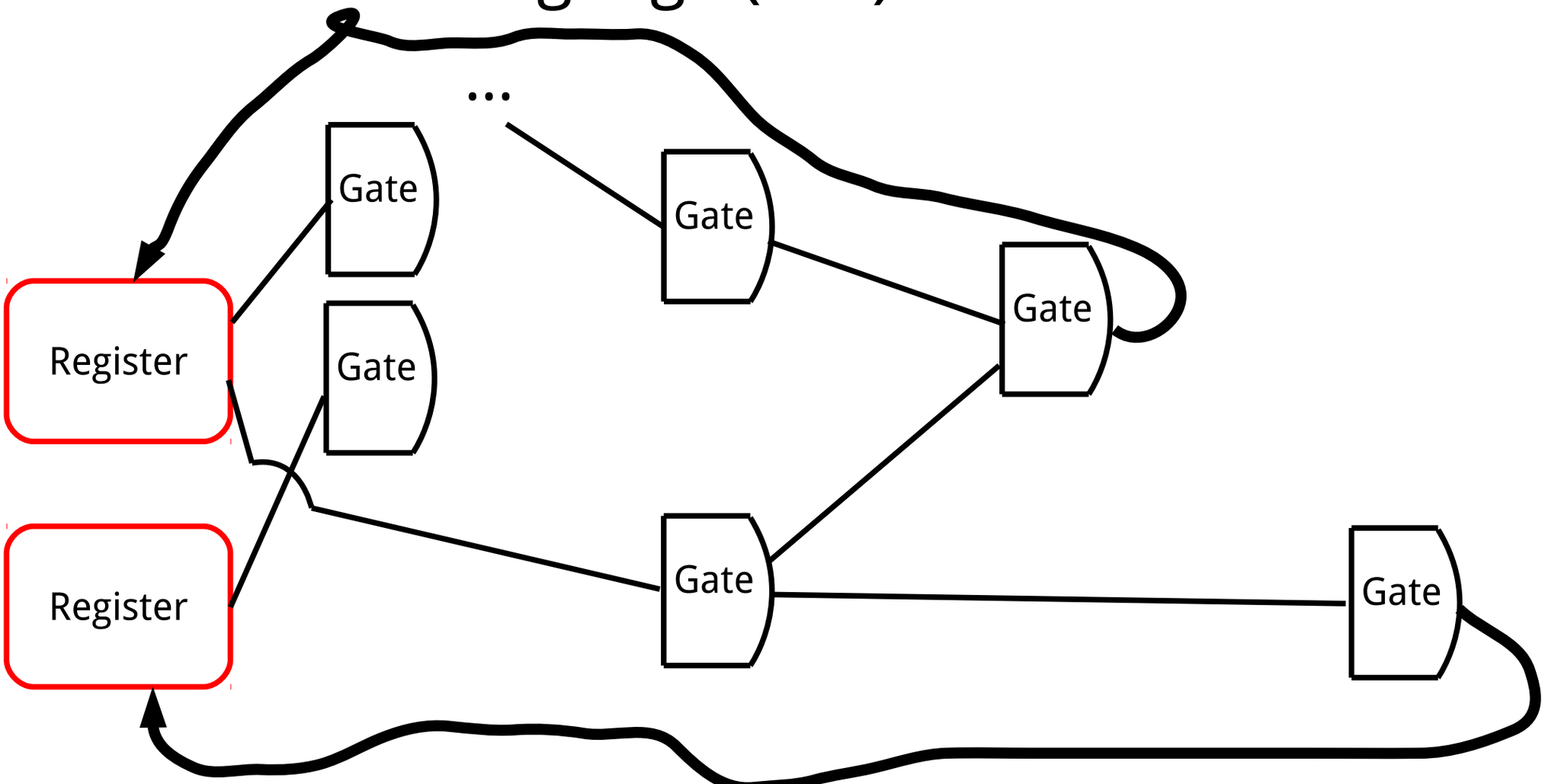
A **framework** to support *implementing*, *specifying*, *formally verifying*, and *compiling* hardware designs

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

and the **Coq** proof assistant



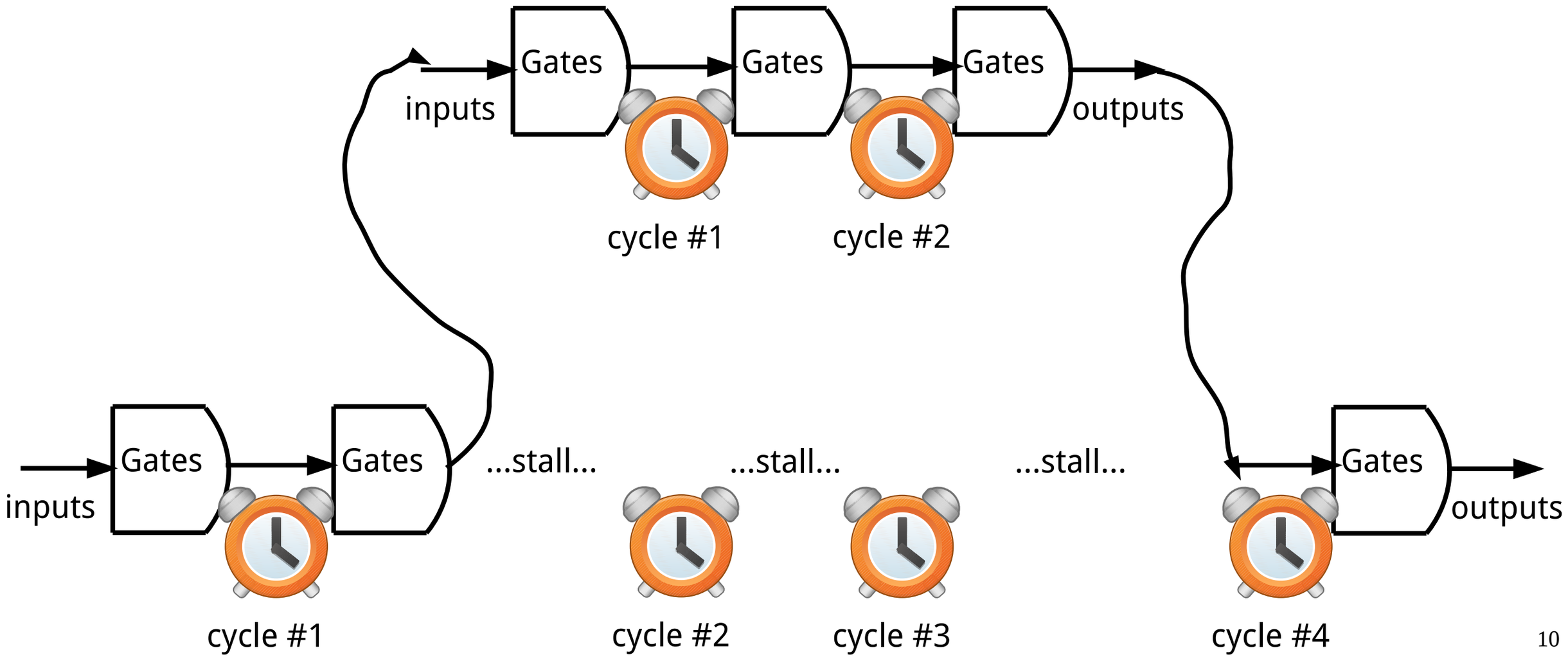
Usual Industry Practice:
Register Transfer Language (RTL)



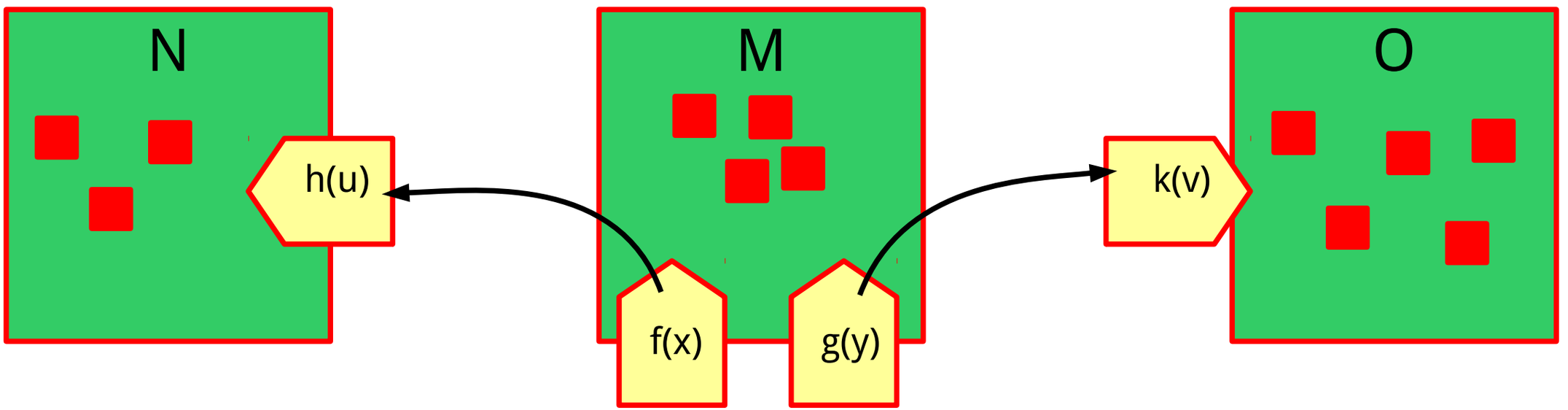
Differences from Conventional Software

- 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

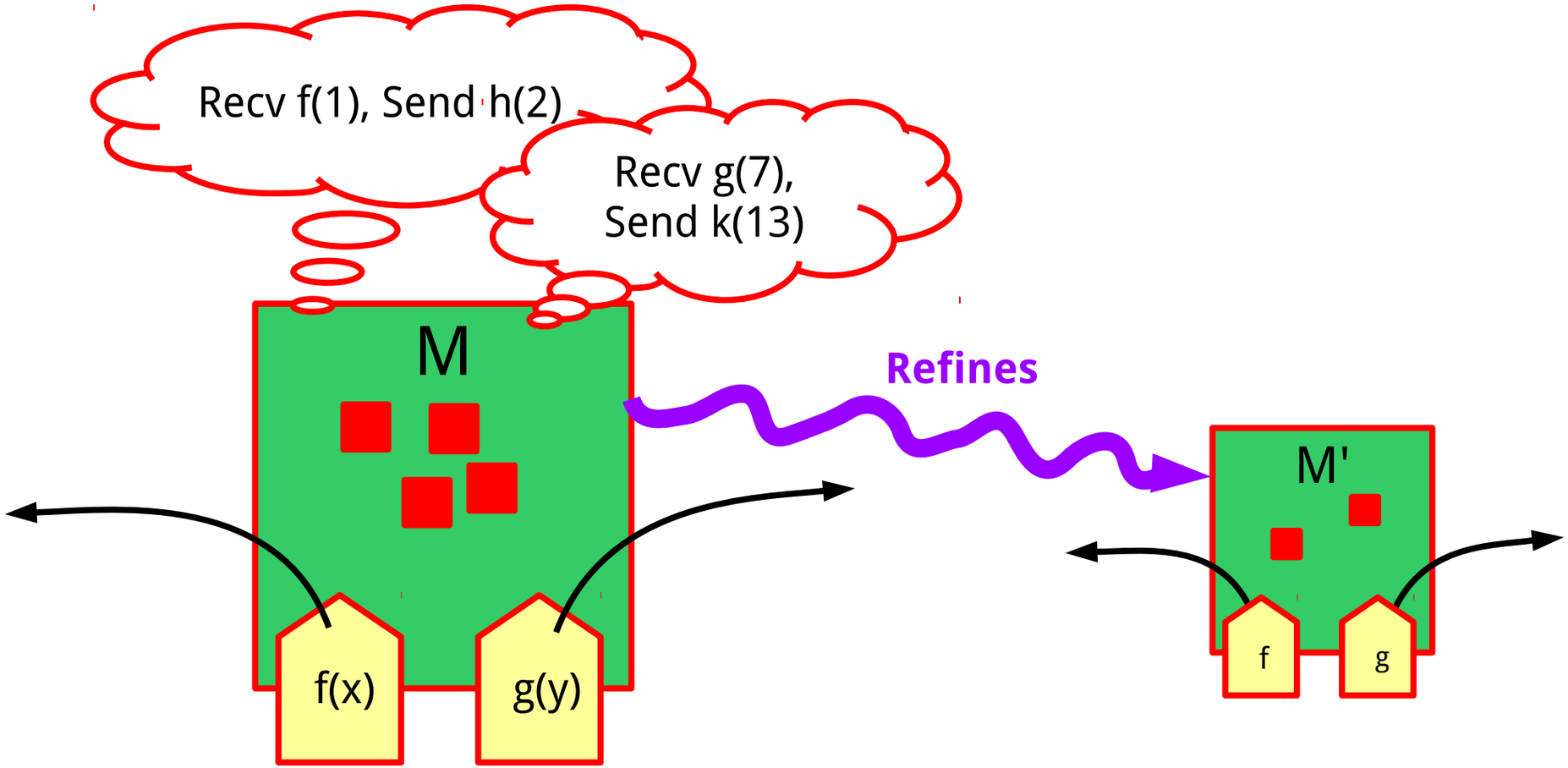


The Big Ideas (from Bluespec)



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

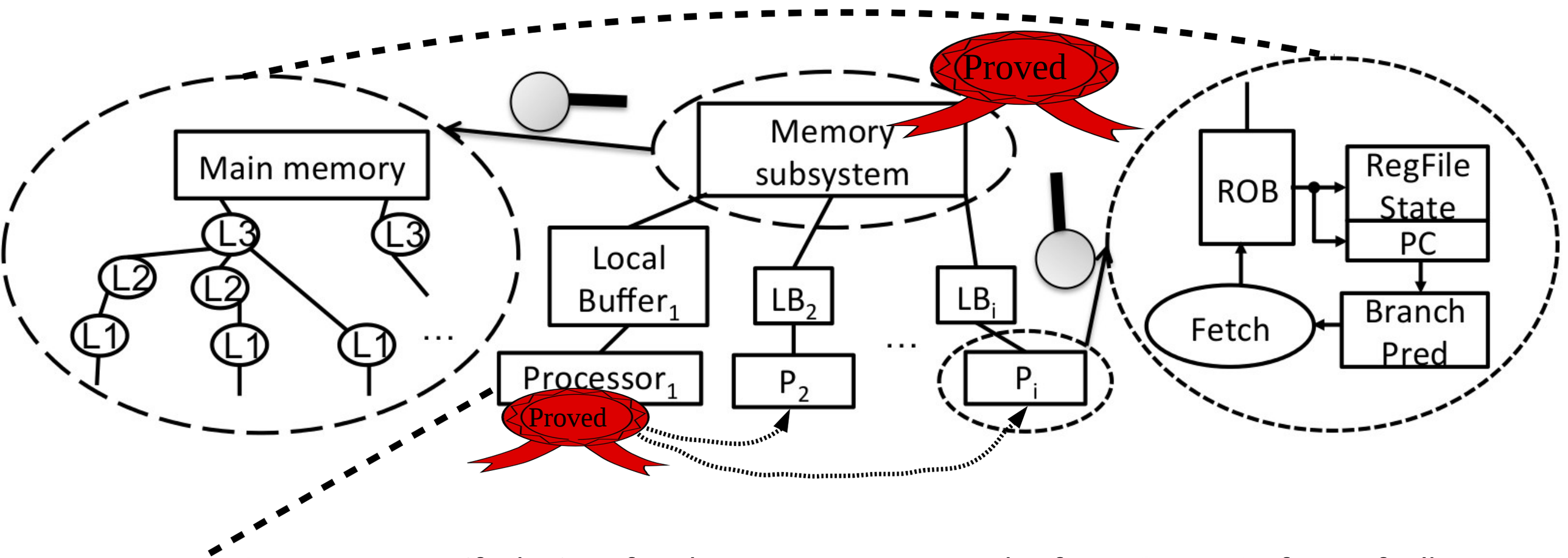
The Big Ideas



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

Decomposing a Processor Proof

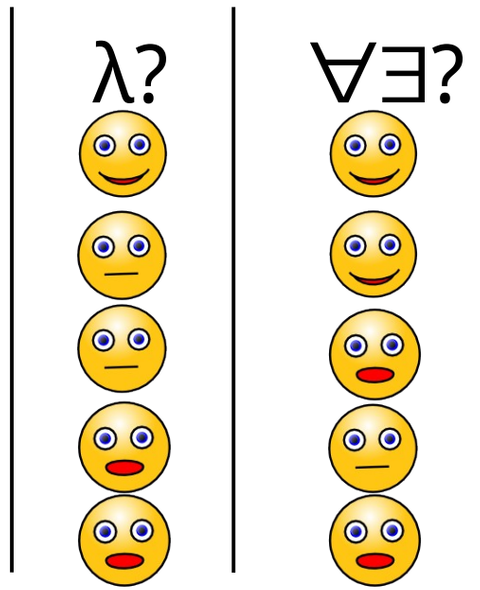
Q: How to avoid surprises from timing side channels? (part 2 of rest of talk)



Q: Best way to specify the interface between processor and software? (part 1 of rest of talk)

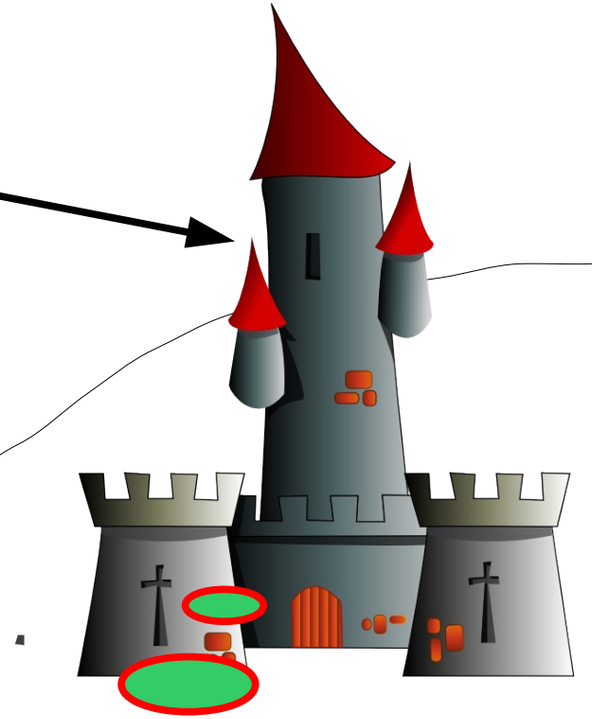
Who is Going to Use a Formal Semantics and How?

- Basic researchers in formal methods for software
- Basic researchers in formal methods for hardware
- Industrial compiler writers
- Industrial processor architects & QA people
- Industrial vendors of system-on-a-chip solutions



Pulls from Two Different Directions

Theory

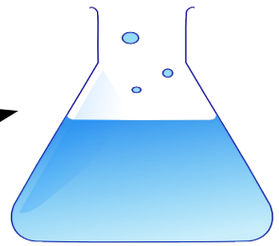


Practice

<u>Huge variation in engineers' backgrounds</u>
Software, digital hardware, analog hardware
Some didn't study computer science.
Some would refuse to install Cabal.

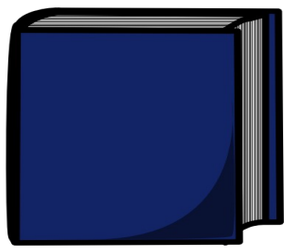
<u>Different formal-methods “camps”</u>
Types?
Higher-order or first-order logic?
Automatic or interactive proofs?
Classical or constructive logic?

Approach to a Shared Formal Semantics



Simulator

Python



ISA Manual
(PDF)

Official RISC-V Binary Formats

Translator

Parser (Haskell)

+

Uses a **minimal subset**

Comments

Our interpreter (Haskell)

"the semantics"

Coq semantics

(uses Penn's hs-to-coq tool, developed in DeepSpec)

Isabelle semantics

ACL2 semantics

https://github.com/mit-plv/riscv-semantics

Adoption

Design & implementation at MIT

committee agrees to adopt our spec

Design & impl. w/ committee

RISC-V Foundation ratification process

Formal spec is official

Formal Spec Committee

conversations about wacky corner cases

ISA Design Committee(s)

Validate against spec

Coq translation

Compliance Committee

Design conformance test suite

RISC-V Foundation ratification process

Official test suite

Processor proofs

Compiler proofs

Demonstrated compliance

Vendor

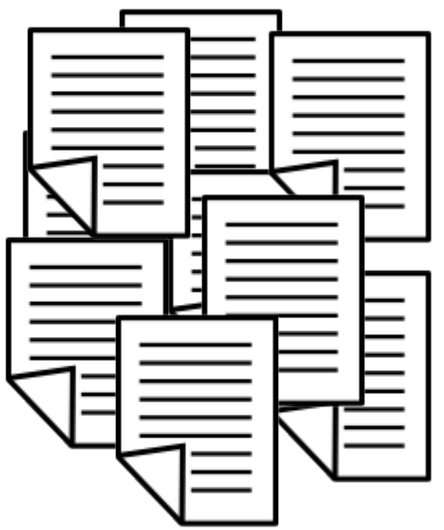
Permission to use “RISC-V” trademark

Big Idea for Palatability to a Wide Audience

Scary Files

Direct use of monads, higher-order functions, type classes, etc.

Nonscary Files



Aspiration: “looks like Python with weird syntax”

Some Example Instructions

```
execute (Andi rd rs1 imm12) = do
  x <- getRegister rs1
  setRegister rd (x .&. (fromImm imm12))
```

Semantics of one bitwise “AND” instruction

```
execute (Lw rd rs1 oimm12) = do
  a <- getRegister rs1
  addr <- translate Load 4 (a + fromImm oimm12)
  x <- loadWord addr
  setRegister rd (int32ToReg x)
```

Semantics of 32-bit memory load

Resolver for virtual-memory addresses, also a nonscary function

```
execute (Jal rd jimm20) = do
  pc <- getPC
  let newPC = pc + (fromImm jimm20)
  if (remu newPC 4 /= 0)
    then raiseException 0 0
    else (do
      setRegister rd (pc + 4)
      setPC newPC)
```

Semantics of
“jump and link”

Some Example Instructions

```
execute Sret = do
  priv <- getPrivMode
  when (priv < Supervisor) (raiseException 0 2)
  tsr <- getCSRField Field.TSR
  when (tsr == 1) (raiseException 0 2)
  spp <- getCSRField Field.SPP
  setCSRField Field.SPP (encodePrivMode User)
  setPrivMode (decodePrivMode spp)
  spie <- getCSRField Field.SPIE
  setCSRField Field.SPIE 1
  setCSRField Field.SIE spie
  sepc <- getCSRField Field.SEPC
  setPC ((fromIntegral:: MachineInt -> t) sepc)
```

Semantics of an instruction for returning from system calls, etc.

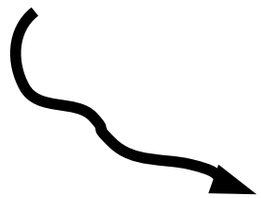
A Daring Expedition into a Scary File: Program Monads

Parameter: a monad in which to run instruction semantics

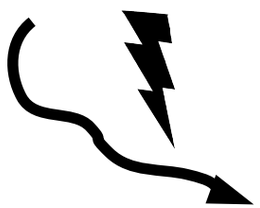
Parameter: a type for representing machine words

```
class (Monad p, MachineWidth t) => RiscvProgram p t | p -> t where
  getRegister :: Register -> p t
  setRegister :: Register -> t -> p ()
  loadByte :: t -> p Int8
  loadHalf :: t -> p Int16
  loadWord :: t -> p Int32
  loadDouble :: t -> p Int64
  storeByte :: t -> Int8 -> p ()
  storeHalf :: t -> Int16 -> p ()
  storeWord :: t -> Int32 -> p ()
  storeDouble :: t -> Int64 -> p ()
  -- and about the same number of others....
```

Many Instances Are Useful

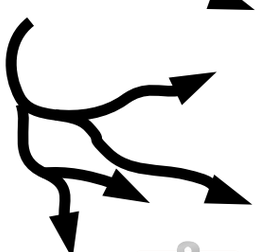


Instance: Simple execution, optimized for readability

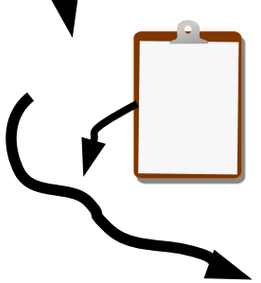


Instance: Fast execution, optimized for simulation

Instance wrapper: Add logging (can be used for runtime verification)



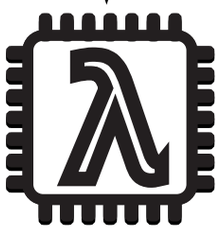
Instance: Resolve nondeterminism in all possible ways, for testing



Instance: Validate a trace that was also checked against an axiomatic memory model

Other Translations, Based More on Introspection

Translated to Python,
to use in an
undergraduate class
where students
implement RISC-V
hardware



Translated to Verilog,
using Cλash Haskell library,
then analyzed with preexisting
tools for hardware verification

Translated to Coq,
using Penn's
hs-to-coq

```
| Decode.Jal rd jimm20 =>
    Bind getPC (fun pc =>
            let newPC := pc + fromImm jimm20 in
            if (remu newPC four /= zero) : bool
            then raiseException zero zero
            else (Bind (setRegister rd (pc + four))
                  (fun _ => setPC newPC)))
```

Status

Simulator passes all of the official RISC-V tests not associated with specific optional modules.

“Epsilon away” from being able to boot Linux in simulation…
…but it will have to wait until students return from internships.

Starting to engage with RISC-V compliance committee,
to use formal semantics as an oracle in creating official test suite.

Surprising processor-design mistakes,
leading to information leaks through **timing**!

Leaky Software

```
def power(x,n)
  result = 1
  while n.nonzero?
    if n[0].nonzero?
      result *= x
      n -= 1
    end
    x *= x
    n /= 2
  end
  return result
end
```

Simple worry:
Running time of procedure depends on input.
More “1” bits implies longer running time!

Subtler worry:
Probably compiled to conditional-jump instruction.
Time depends not just on number of “1” bits, but also on *instruction-cache contents*!
With a shared cache, another process can observe via timing.

Less Leaky Software

```
def power(x,n)
  result = 1
  while n.nonzero?
    b = n[0].nonzero?
    result *= (b ? x : 1)
    n -= (b ? 1 : 0)
    x *= x
    n /= 2
  end
  return result
end
```

Important: compile to conditional-multiply instruction!

Important: compile to conditional-subtract instruction!

Characterizing Timing Security

No matter how we change
the secret inputs
the trace of interactions with the world
includes events at exactly the same times.

Challenge: how do we prove this property
of real designs
while maintaining good *modularity*?

Proving Absence of Timing Side Channels

Execution

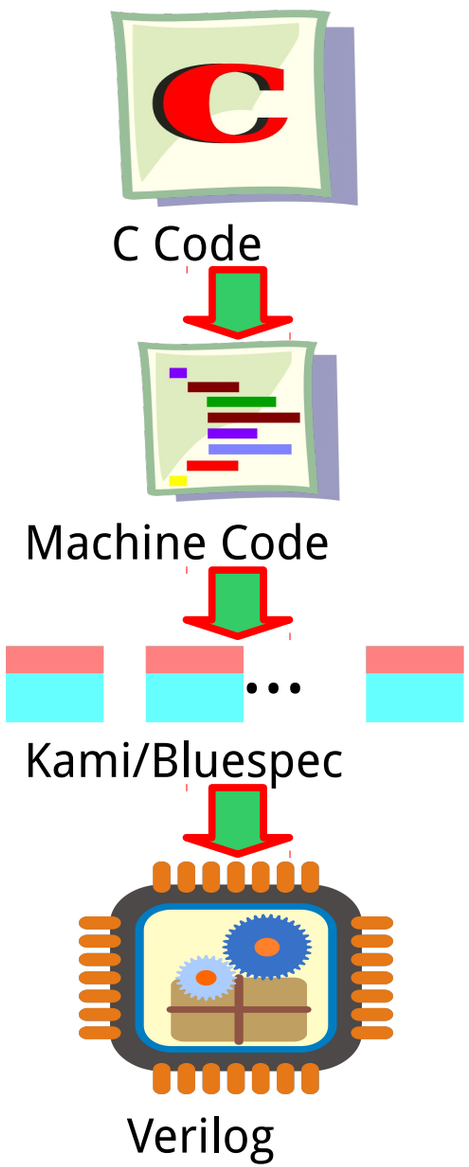
secrets flow in via IO

"Constant time" property:
secrets don't influence choices of memory addresses (code/data).

C compiler preserved the property to its natural machine-lang. equivalent.

Rules implementing instrs. don't introduce new leaks, including via guards.

Number of clock cycles needed for a function call does not vary based on secrets.



Proof Strategy (traces)

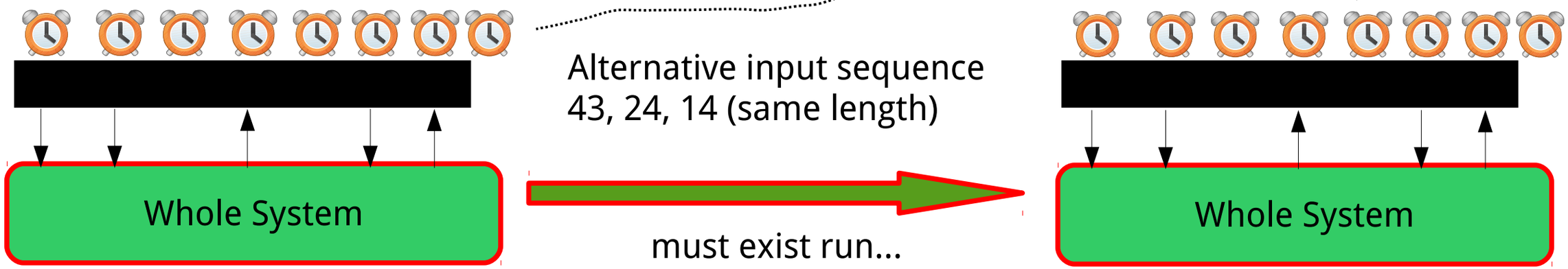
Any traces of program execution have same result of *censorship*, where all IO inputs are erased.

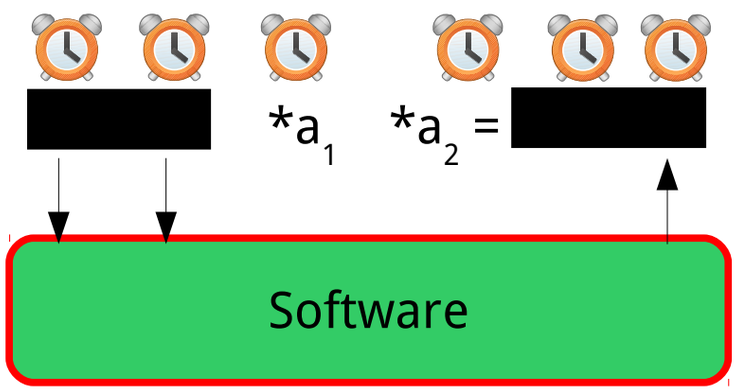
Analogous trace property for machine code

Assume trace property for loaded executable and *prove* analogous property for HW (break proof into lemmas for processor & memory system).

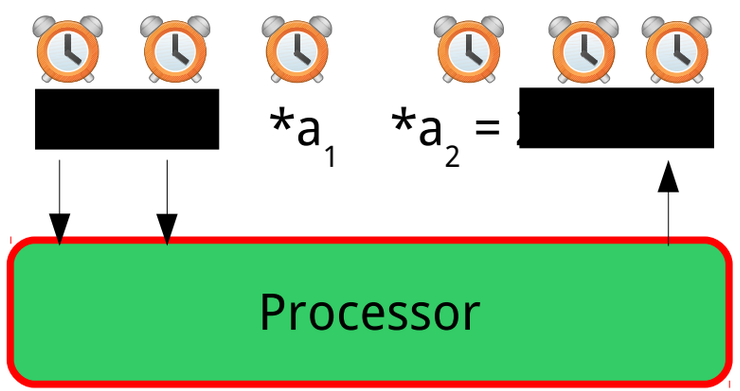
Per-clock-cycle state transfer function is independent of secrets.

Censorship at Multiple Levels

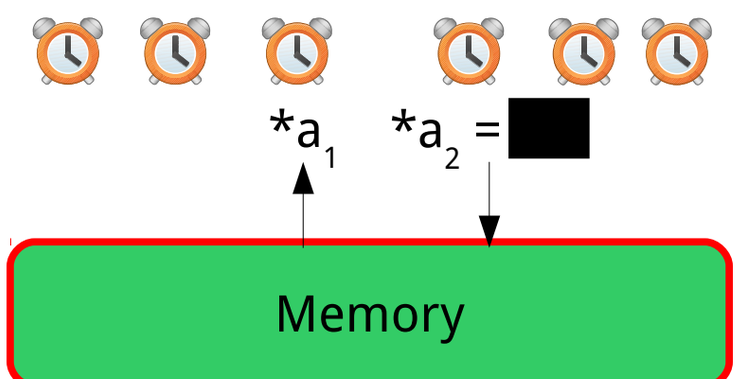




This is the “constant time” property well-known to crypto implementers.



Essentially the same property, but proven *assuming* it for software.



I.e., memory treats data values opaquely.

Proof Effort

Software

Use a verified static analysis (symbolic execution)!
Runs program with special “poison” values substituted for program inputs and flowed through system.
Analysis fails if program ever tries to inspect poison.

Hardware

Significant manual effort by developers.
Come up with invariants relating states of spec & implementation.

Example Software: Salsa20

```
// rotate x to left by n bits, the bits that go over
// the left edge reappear on the right
#define R(x,n) (((x) << (n)) | ((x) >> (32-(n))))

// addition wraps modulo 2^32
// the choice of 7,9,13,18 "doesn't seem very important" (spec)
static void quarter(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d) {
  *b ^= R(*d+*a, 7);
  *c ^= R(*a+*b, 9);
  *d ^= R(*b+*c, 13);
  *a ^= R(*c+*d, 18);
}

void salsa20_words(uint32_t *out, uint32_t in[16]) {
  uint32_t x[4][4];
  int i;
  for (i=0; i<16; ++i) x[i/4][i%4] = in[i];
  for (i=0; i<10; ++i) { // 10 double rounds = 20 rounds
    // column round: quarter round on each column; start at ith element and wrap
    quarter(&x[0][0], &x[1][0],&x[2][0], &x[3][0]);
    quarter(&x[1][1], &x[2][1], &x[3][1], &x[0][1]);
    quarter(&x[2][2], &x[3][2], &x[0][2], &x[1][2]);
    quarter(&x[3][3], &x[0][3], &x[1][3], &x[2][3]);
    // row round: quarter round on each row; start at ith element and wrap around
    quarter(&x[0][0], &x[0][1], &x[0][2], &x[0][3]);
    quarter(&x[1][1], &x[1][2], &x[1][3], &x[1][0]);
    quarter(&x[2][2], &x[2][3], &x[2][0], &x[2][1]);
    quarter(&x[3][3], &x[3][0], &x[3][1], &x[3][2]);
  }
  for (i=0; i<16; ++i) out[i] = x[i/4][i%4] + in[i];
}

// inputting a key, message nonce, keystream index and constants to that transormation
void salsa20_block(uint32_t *out, uint32_t key[8], uint64_t nonce, uint64_t index) {
  static const char c[17] = "expand 32-byte k"; // arbitrary constant
  #define LE(p) ( (p)[0] | ((p)[1]<<8) | ((p)[2]<<16) | ((p)[3]<<24) )
  uint32_t in[16] = {LE(c), key[0], key[1], key[2],
                     key[3], LE(c+4), nonce&0xffffffff, nonce>>32,
                     index&0xffffffff, index>>32, LE(c+8), key[4],
                     key[5], key[6], key[7], LE(c+12)};
  salsa20_words(out, in);
}

// enc/dec: xor a message with transformations of key, a per-message nonce and block index
void salsa20(uint64_t nonce) {
  int i, j;
  uint32_t msgword;
  uint32_t block[16];
    uint32_t key[8];
    for (i = 0; i < 8; i++) {
        key[i] = fromhost();
    }
  for (i=0; ; i++) {
    salsa20_block(block, key, nonce, i);
    for (j = 0; j<16; j++) {
      msgword = fromhost();
      tohost(msgword ^ block[j]);
    }
  }
}
```

Status

Full stack proved for very simple hardware and software.
In progress: adapting proofs to more complex hardware.

The technique so far applies to “information-flow-style” timing properties.
We also want to add “resource-partitioning-style” timing properties.

- Example: **Sanctum** technique developed at MIT
  (by Srini Devadas & students), which partitions cache resources
  across protection domains

Proofs need one more level: timing-safety of Kami compiler to Verilog.