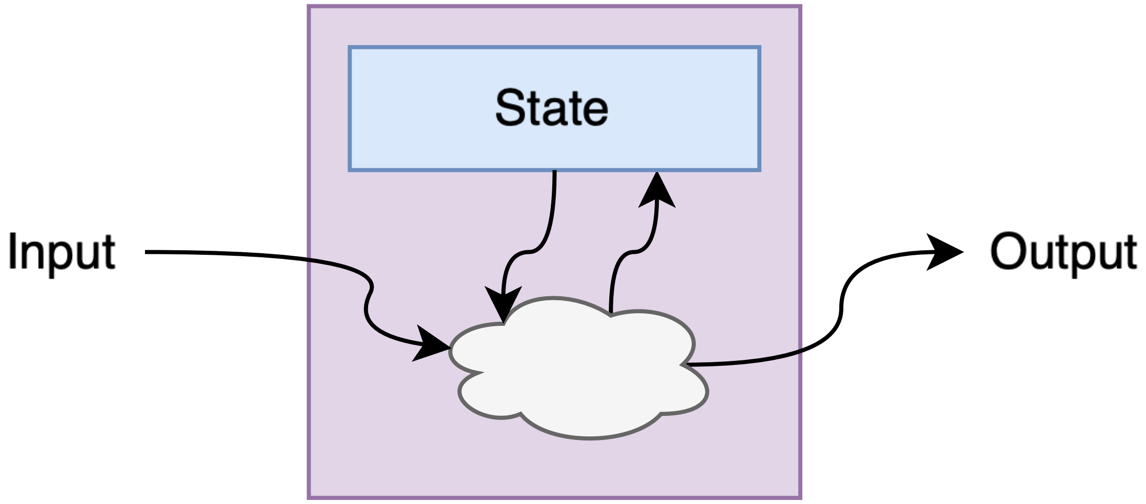
Formal Verification for Processor Cores

Louis-Emile Ploix

$$\textbf{How are processor cores}\ \left\{\begin{array}{l}\textbf{specified}\\ \textbf{organised}\\ \textbf{defined}\\ \textbf{checked}\end{array}\right.\textbf{?}$$

$$\text{How are processor cores} \left\{ \begin{array}{l} \text{specified} \\ \text{organised} \\ \text{defined} \\ \text{checked} \end{array} \right. ?$$

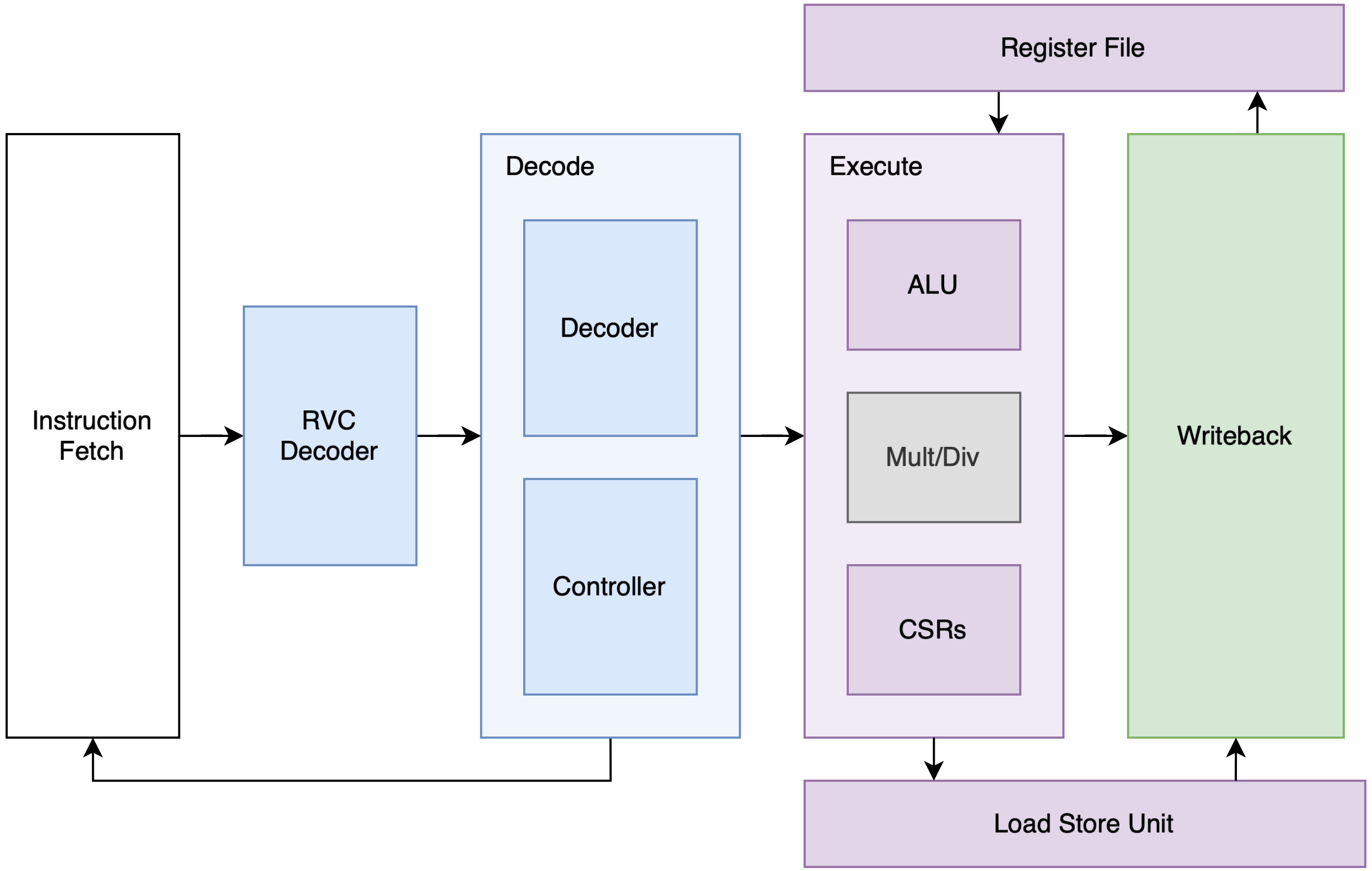
Blocks



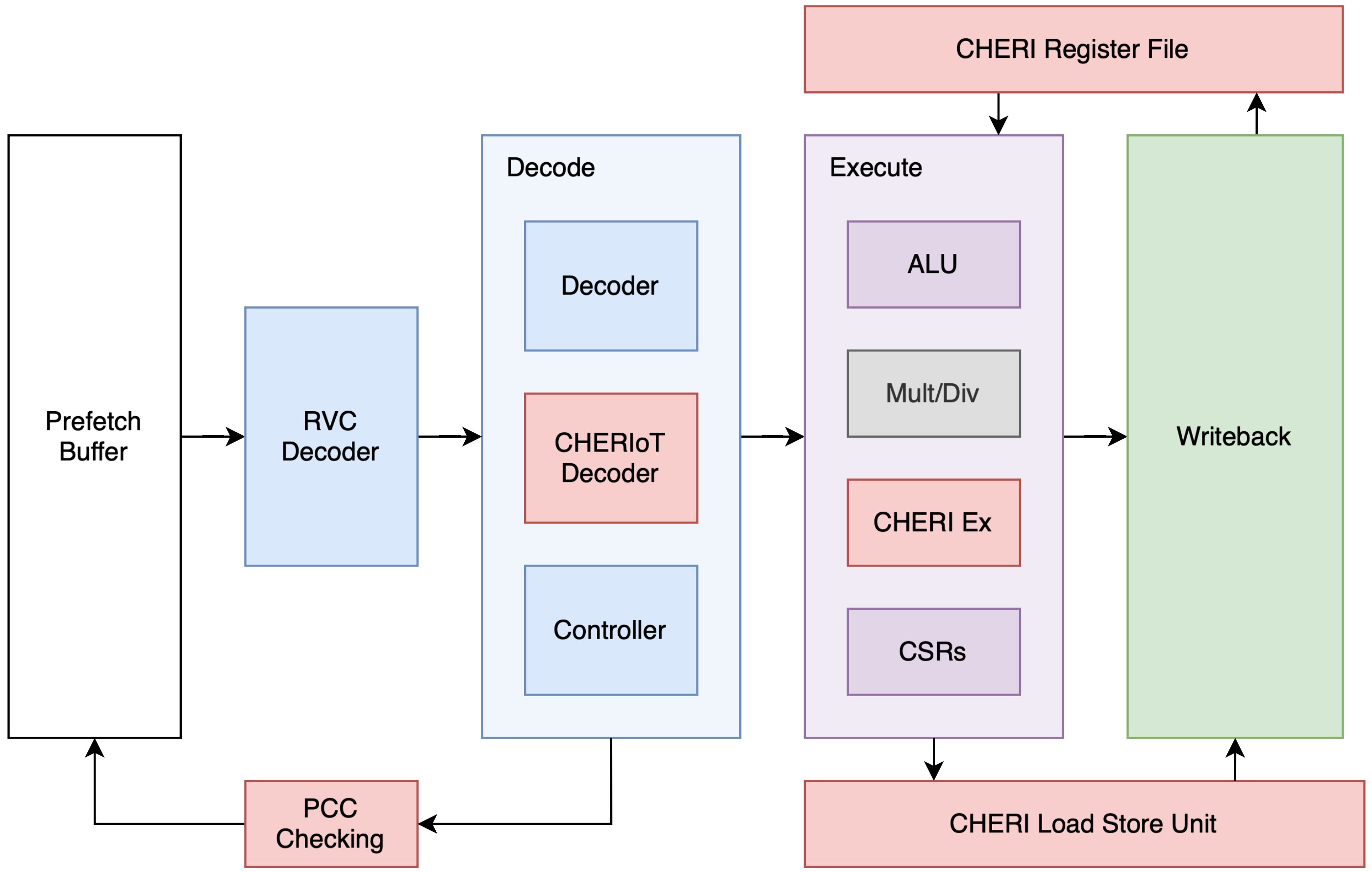
Or, for the theorists, a block is tuple (I, O, S, R, f_O, f_S), where:

- $I, O, S \in \mathbb{N}$ are the input, output and state sizes in bits
- $R \in \{0,1\}^S$ is the reset state
- $f_O : \{0,1\}^I \times \{0,1\}^S \to \{0,1\}^O$ computes the output
- $f_S : \{0,1\}^I \times \{0,1\}^S \to \{0,1\}^S$ computes the next state

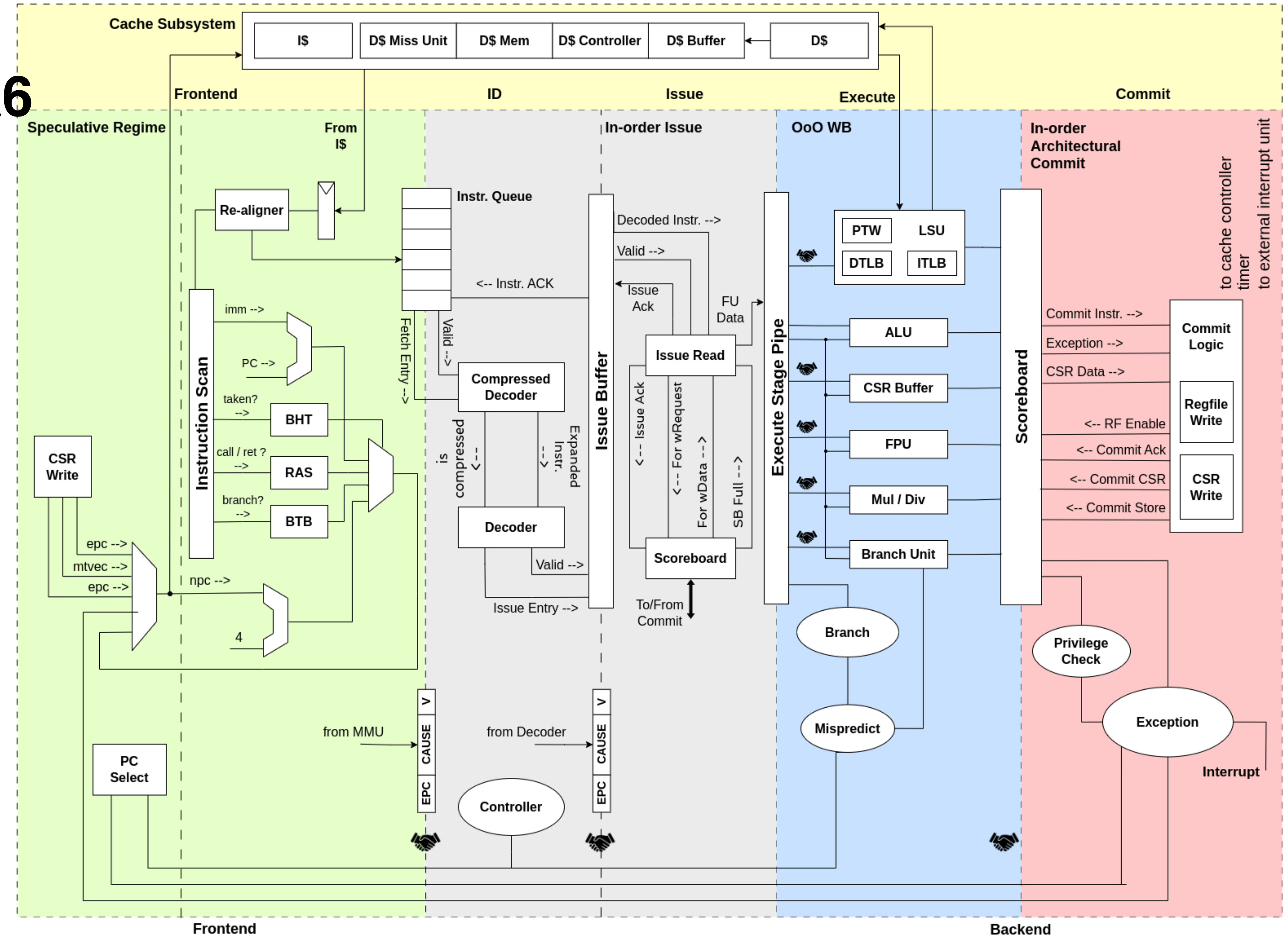
Ibex: Small RISC-V Core



CHERIoT-Ibex: Ibex but with CHERI



CVA6


Cache Subsystem
I$
D$ Miss Unit
D$ Mem
D$ Controller
D$ Buffer
D$
Frontend
ID
Issue
Execute
Commit
Speculative Regime
From I$
Re-aligner
Instr. Queue
In-order Issue
OoO WB
In-order Architectural Commit
to cache controller
timer
to external interrupt unit
Decoded Instr. -->
Valid -->
<-- Instr. ACK
Issue Ack
FU Data
PTW
LSU
DTLB
ITLB
imm -->
PC -->
Fetch Entry -->
Valid -->
Compressed Decoder
Issue Read
Issue Buffer
Execute Stage Pipe
ALU
CSR Buffer
FPU
Mul / Div
Branch Unit
Scoreboard
Commit Instr. -->
Exception -->
CSR Data -->
Commit Logic
Regfile Write
<-- RF Enable
<-- Commit Ack
<-- Commit CSR
<-- Commit Store
CSR Write
Instruction Scan
taken? -->
BHT
call / ret ? -->
RAS
branch? -->
BTB
is compressed <--
Expanded Instr. -->
<-- Issue Ack
<-- For wRequest
For wData -->
SB Full -->
CSR Write
Decoder
epc -->
mtvec -->
epc -->
npc -->
Valid -->
Scoreboard
Issue Entry -->
To/From Commit
Branch
Privilege Check
4
Exception
Mispredict
from MMU
V
CAUSE
EPC
from Decoder
V
CAUSE
EPC
Interrupt
PC Select
Controller
Frontend
Backend

$$\text{How are processor cores} \left\{ \begin{array}{l} \text{specified} \\ \text{organised} \\ \text{defined} \\ \text{checked} \end{array} \right. ?$$

Or equivalently: How do we precisely describe each block?

Verilog, SystemVerilog or VHDL

(Usually SystemVerilog these days)

```
module summer( // modules are blocks
  input reg clk_i, input reg rst_i, // clock and reset
  input reg has_new_i, // is a new value being provided in this cycle?
  input reg [31:0] new_i,
  output reg [31:0] sum_o // sum of all values up to this one (exclusive)
);
reg [31:0] sum_q; // registers hold state, and get updated by wires
wire [31:0] sum_d; // wires are functions of registers and wires
assign sum_d = sum_q + new_i; // assign statements define a wire's function
assign sum_o = sum_q;
always @(posedge clk_i) // procedural blocks allow for sequential processing
  if (rst_i) sum_q = 32'h0;
  else if (new_i) sum_q = sum_d;
```

Verilog, SystemVerilog or VHDL

SystemVerilog also has

- Nested blocks
- Custom types
- Parameterised wire widths
- Packages and imports
- Functions
- Preprocessing
- Quite a lot more! It’s a big language.

This can all get quite complex, with a *lot* of logic!

$$\textbf{How are processor cores} \left\{ \begin{array}{l} \textbf{specified} \\ \textbf{organised} \\ \textbf{defined} \\ \textbf{checked} \end{array} \right. \textbf{?}$$

Testing! (Boo!!)

Formal verification! (Yay!!)

Let’s mathematically prove it’s correct.

But to do that, we need to define correct:

$$\textbf{How are processor cores} \left\{ \begin{array}{l} \textbf{specified} \\ \textbf{organised} \\ \textbf{defined} \\ \textbf{checked} \end{array} \right. \textbf{?}$$

Architecture Specifications

Defines *precisely* (hopefully) what a CPU does.

Intel 64 and IA-32 *(x86, basically)*

- Plain text written as prose with some pseudocode.
- 25.9MB PDF, 5,252 pages

Aarch64

- Plain text descriptions primarily, **but also machine readable ASL specification**
- 68MB PDF, 11,952 pages

RISC-V

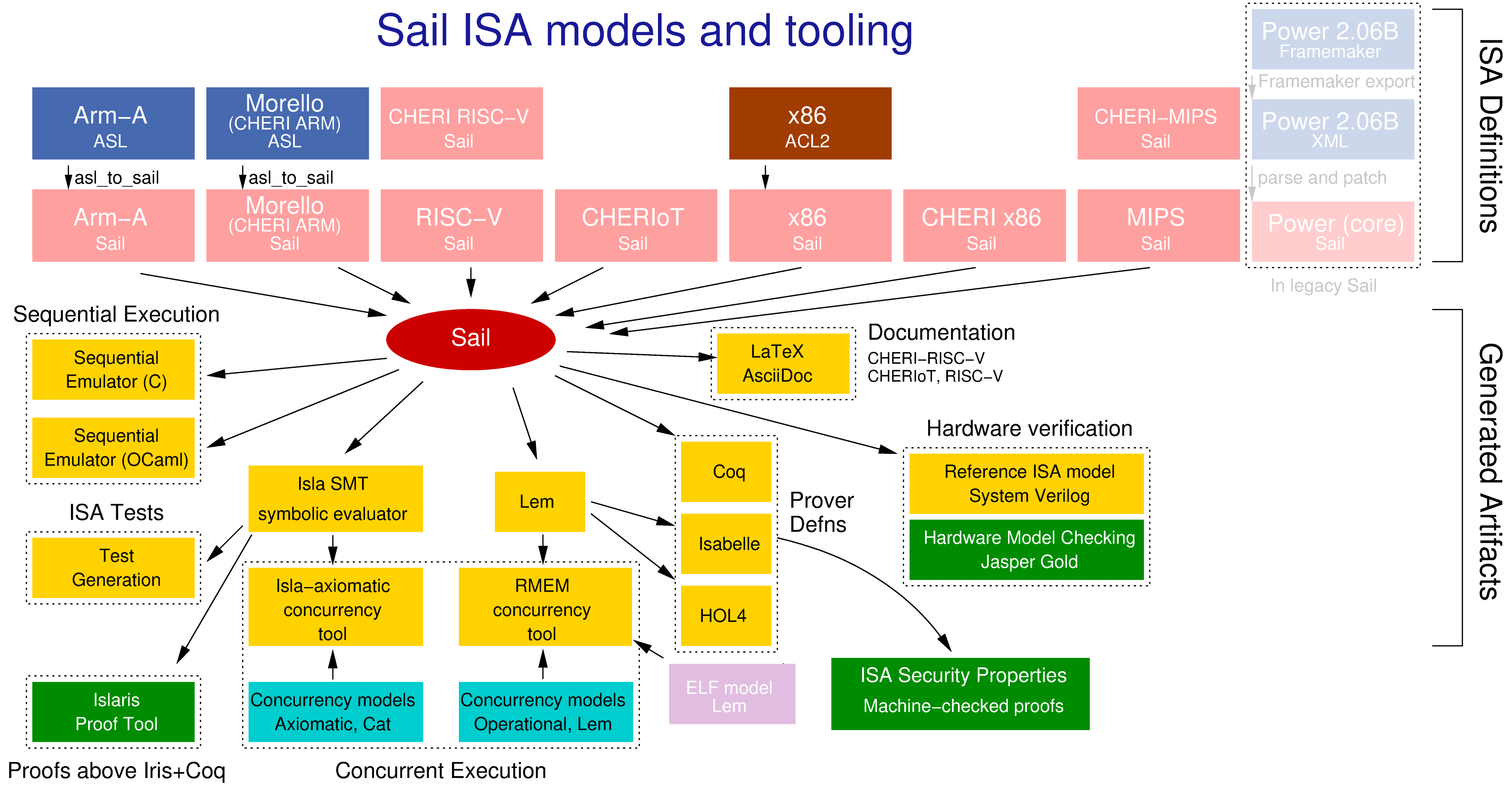
- Plain text descriptions exist, **but also machine readable Sail specification**
- 6.2MB PDF, 958 pages (split across two volumes)

Example definition of RISC-V ITYPE

```
mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ encdec_reg(rs1) @ encdec_iop(op) @ encdec_reg(rd) @ 0b0010011

function clause execute ITYPE(imm, rs1, rd, op) = {
  let immext : xlenbits = sign_extend(imm);
  X(rd) = match op {
    ADDI  => X(rs1) + immext,
    SLTI  => zero_extend(bool_to_bits(X(rs1) <_s immext)),
    SLTIU => zero_extend(bool_to_bits(X(rs1) <_u immext)),
    ANDI  => X(rs1) & immext,
    ORI   => X(rs1) | immext,
    XORI  => X(rs1) ^ immext
  };
  RETIRE_SUCCESS
}
```

Sail ISA models and tooling



Finally, the general idea:

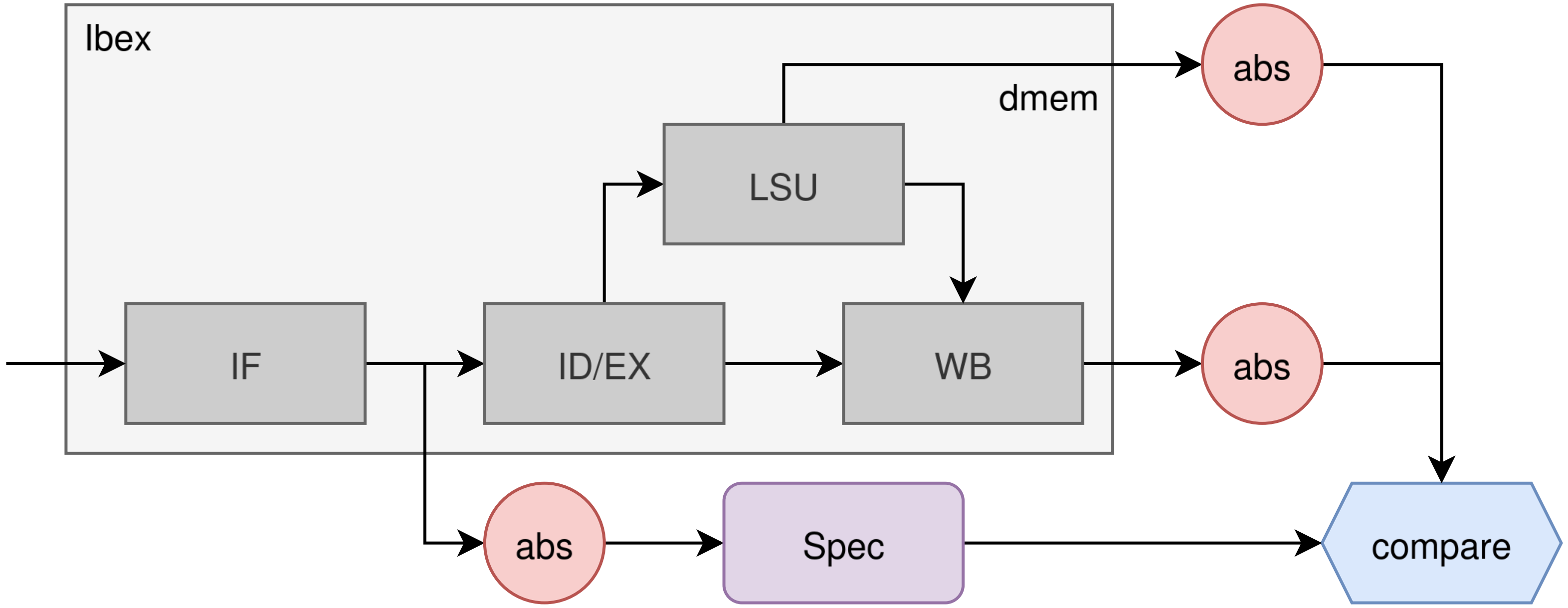
The goal is to prove that the SystemVerilog implementation ‘matches’ the compiled specification.

Here’s how I did it, for Ibex and CHERIoT-Ibex.

As far as I know, I am the first.

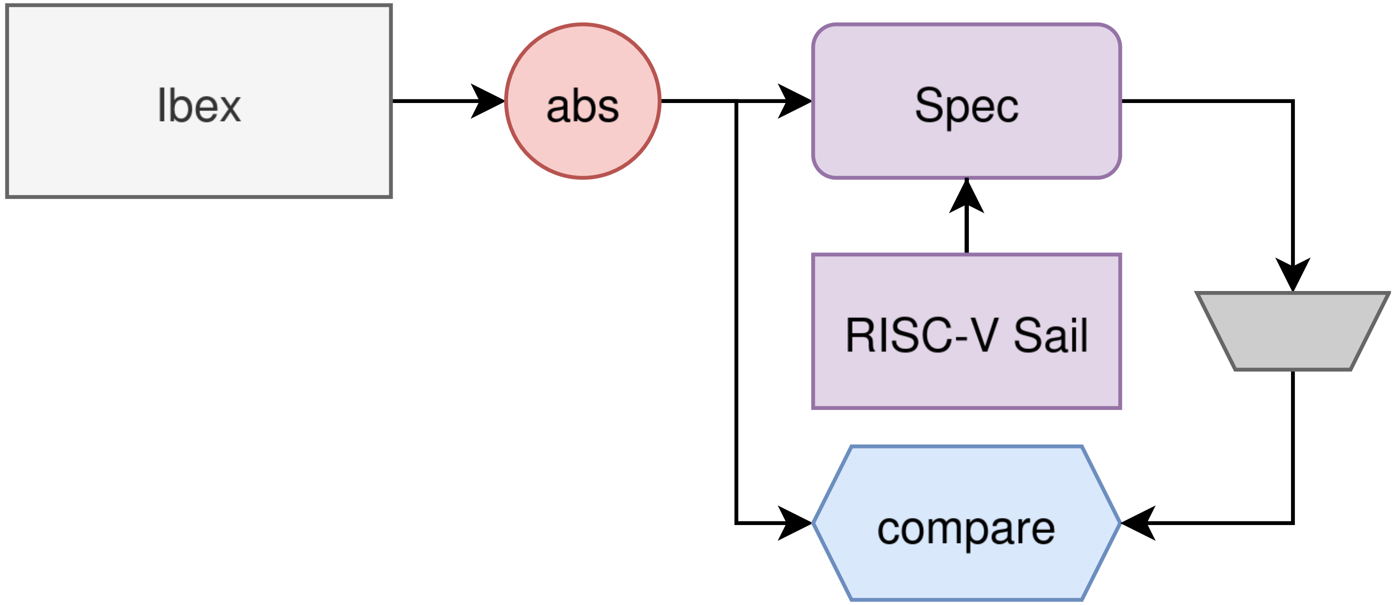
1. End-to-end correctness

Interpret internal state. This is a lemma for future steps, but it's also where all the bug finding power is.



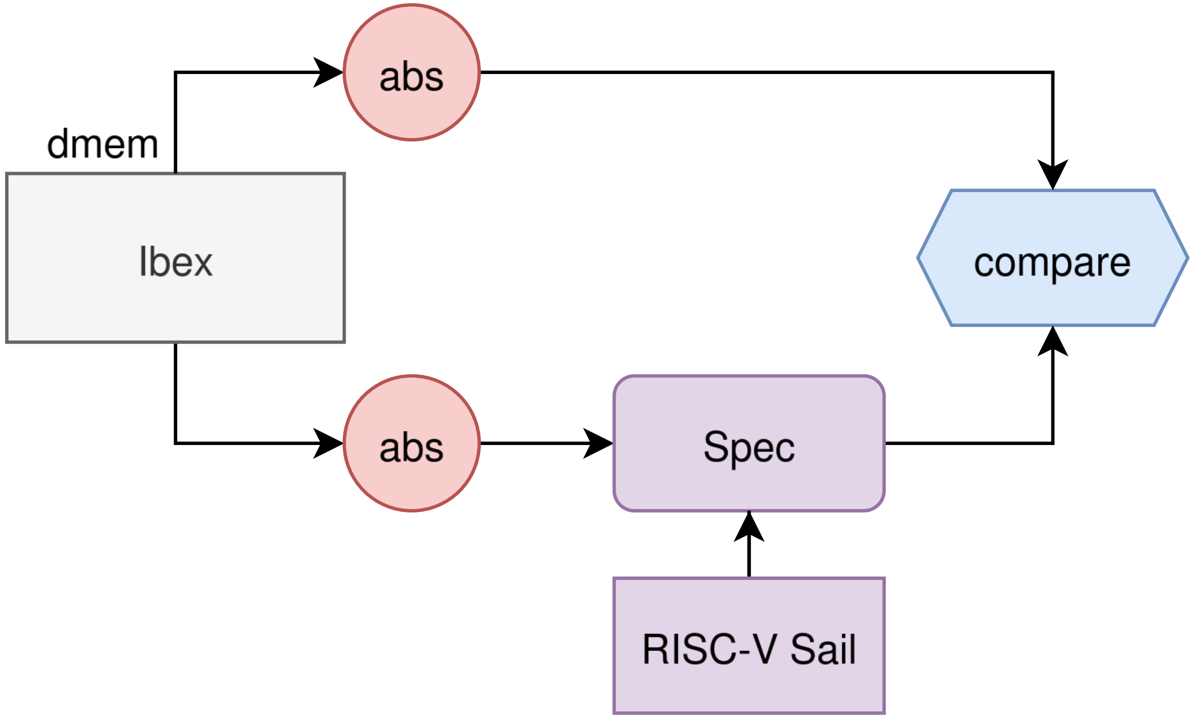
2. Continuity on the specification

Every time the specified is ‘queried’, the new inputs match the previous outputs. This prevents state divergence so long as querying continues.



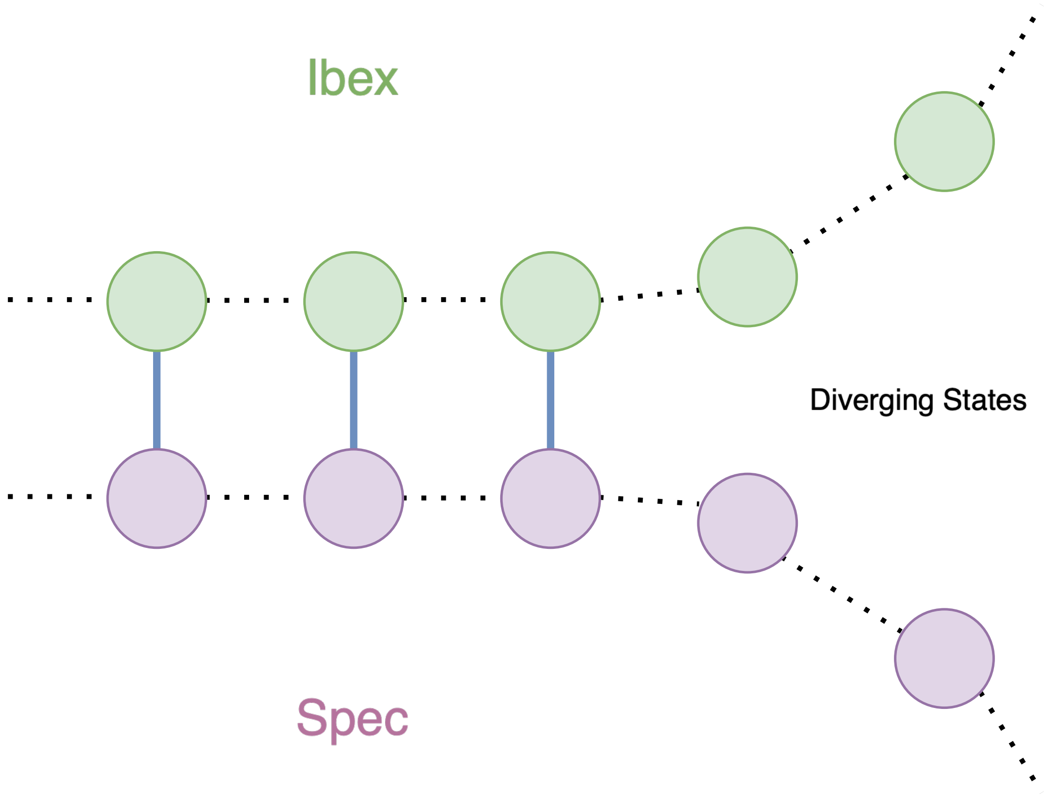
3. Correctness of memory w.r.t. spec

Between the 'query' points, all memory outputs are correct w.r.t the specification, i.e. they are the same, in the same order.



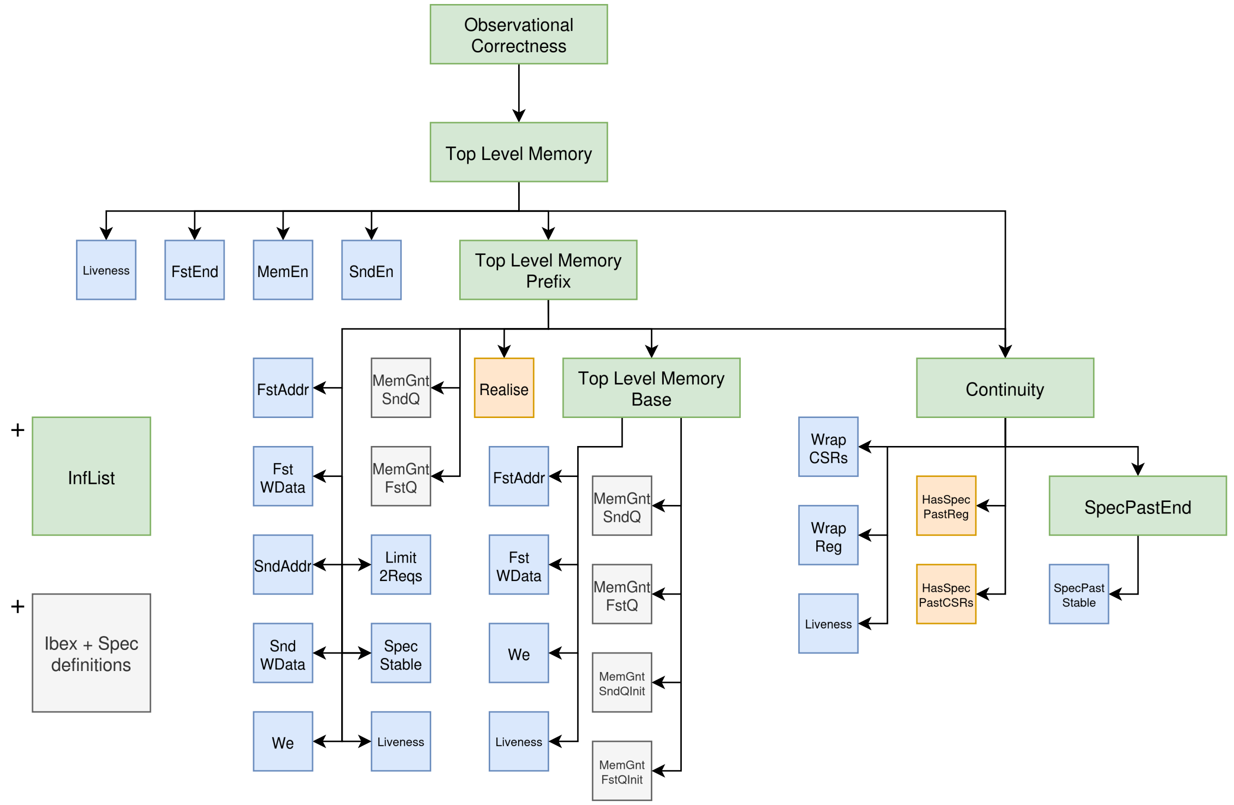
4. Liveness

There are infinite query points, thus preventing divergence.



This composes!

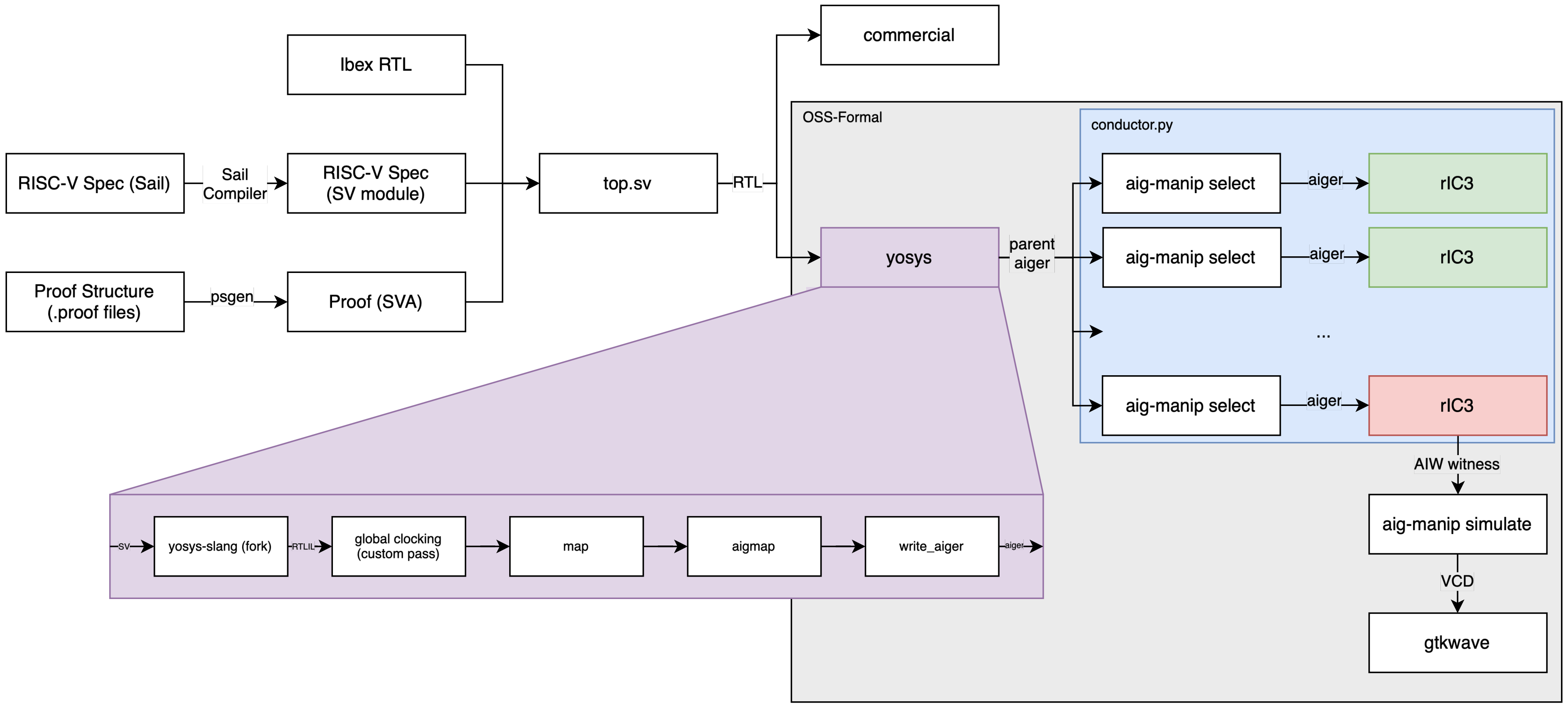
By simple induction over time. We even checked this in Lean 4:



Provided the same sequence of inputs, *Ibex and the Sail-RISC-V specification will produce the same sequence of memory operations in the same order forever*

This is, essentially, trace equivalence.

What's more, it works on open source! *(and commercial tools)*



Thanks!

- Hope you enjoyed!
- This talk covered about 3 years of work in 8-9 minutes! (The last slide was an entire summer!)
- Special thanks to Tom Melham, and lowRISC CIC.
- Questions welcome now or later!