Reproducing a Proof of Specification Compliance for Ibex with Open Source Tools

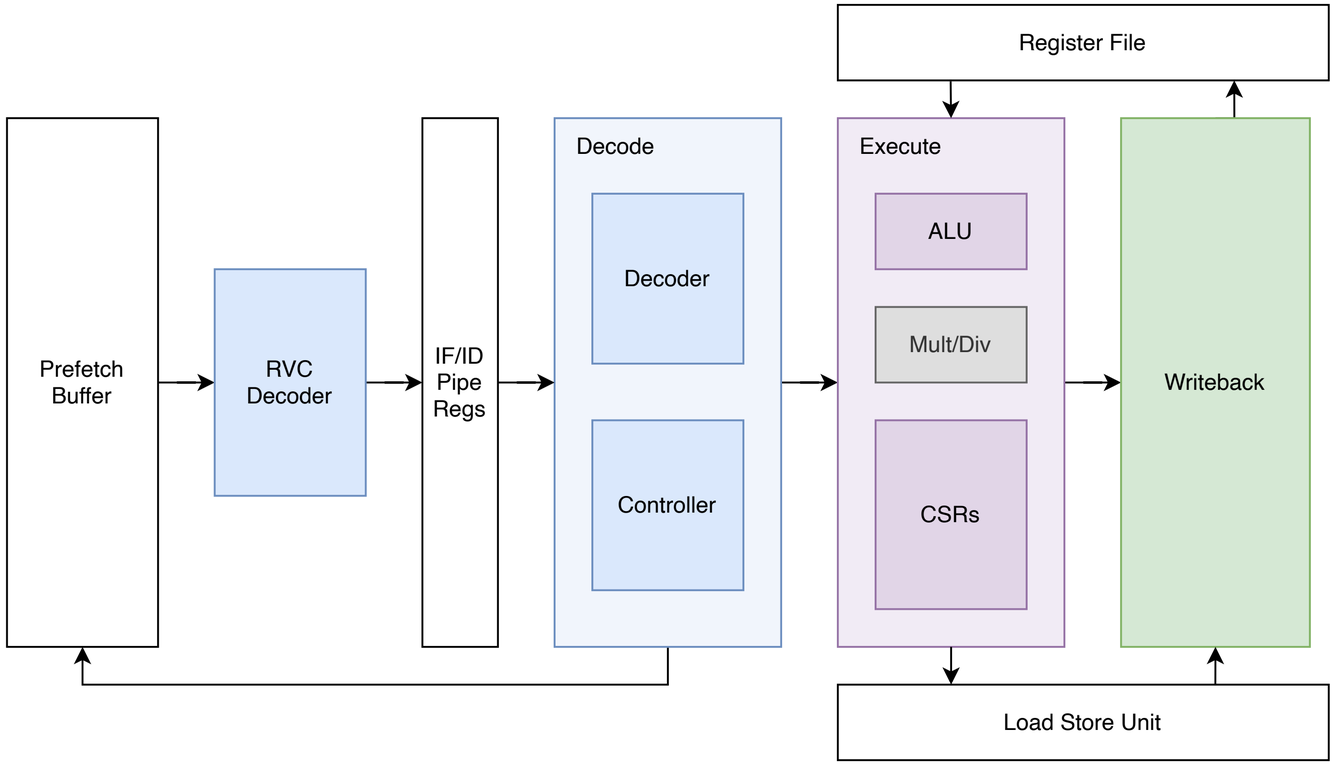
DVClub World — October 21st 2025

louis-emile.ploix@stcatz.ox.ac.uk

Louis-Emile Ploix — lowRISC

Ibex

- Fully open source, owned by lowRISC
- Vanilla RISC-V
- Simple 3-stage pipeline: fetch, decode / execute, write back
- RVC / PMP / SMEPMP
- Used in opentitan
- Also going to talk about CHERIoT-Ibex a bit, which is more or less the same, but with CHERI extensions



End-to-End Verification

- Based on ARM ISA-Formal paper

End-to-End Verification of ARM® Processors with ISA-Formal

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi

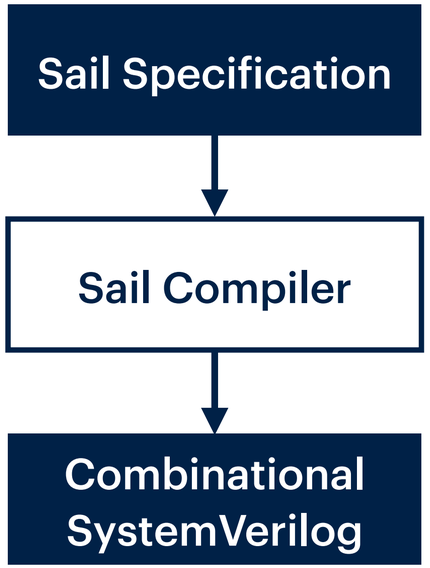
ARM Limited
110 Fulbourn Road
Cambridge, UK
first.last@arm.com

Abstract. Despite 20+ years of research on processor verification, it remains hard to use formal verification techniques in commercial processor development. There are two significant factors: scaling issues and return on investment. The *scaling issues* include the size of modern processor specifications, the size/complexity of processor designs, the size of design/verification teams and the (non)availability of enough formal verification experts. The *return on investment* issues include the need to start catching bugs early in development, the need to continue catching bugs throughout development, and the need to be able to reuse verification IP, tools and techniques across a wide range of design styles.
This paper describes how ARM has overcome these issues in our Instruction Set Architecture Formal Verification framework "ISA-Formal." This is an end-to-end framework to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors. A key part of making the approach scale is use of a mechanical translation of ARM's Architecture Reference Manuals to Verilog allowing the use of commercial model-checkers. ISA-Formal has proven especially effective at finding micro-architecture specific bugs involving complex sequences of instructions.
An essential feature of our work is that it is able to scale all the way from simple 3-stage microcontrollers, through superscalar in-order processors up to out-of-order processors. We have applied this method to 8 different ARM processors spanning all stages of development up to release. In all processors, this has found bugs that would have been hard for conventional simulation-based verification to find and ISA-Formal is now a key part of ARM's formal verification strategy.
To the best of our knowledge, this is the most broadly applicable formal verification technique for verifying processor pipeline control in mainstream commercial use.

1 Introduction

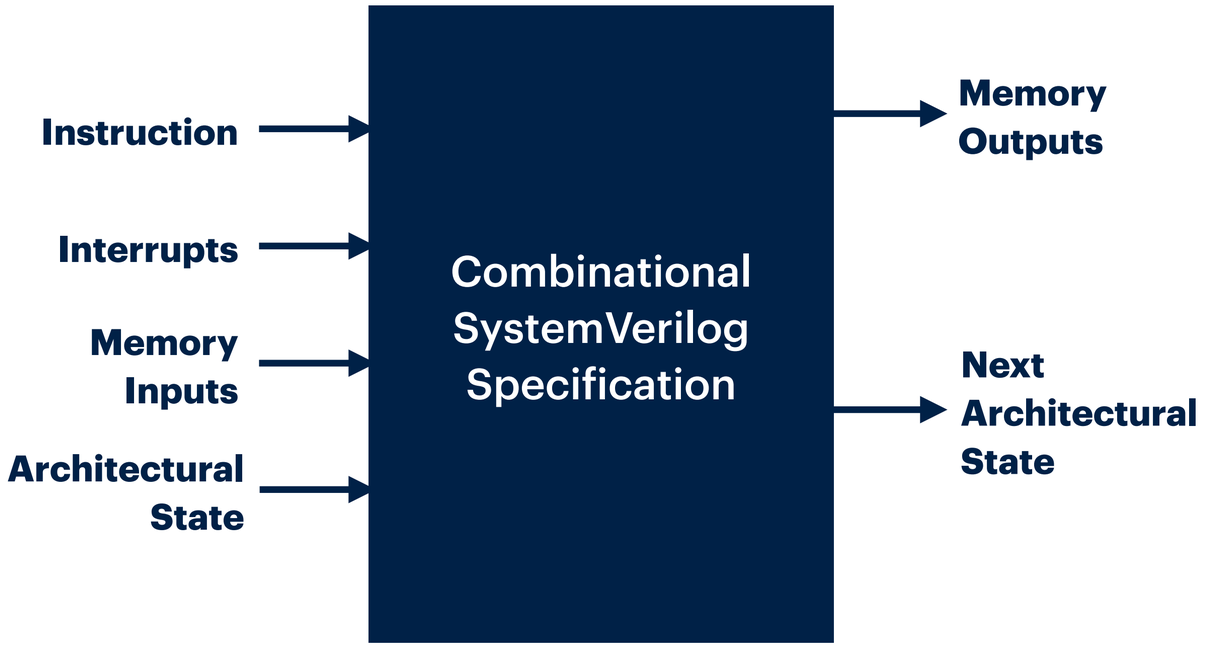
End-to-End Verification

- Based on ARM ISA-Formal paper
- Compile Sail specification to SystemVerilog module
  - (Thank you to Alasdair Armstrong and Peter Sewell at Cambridge)



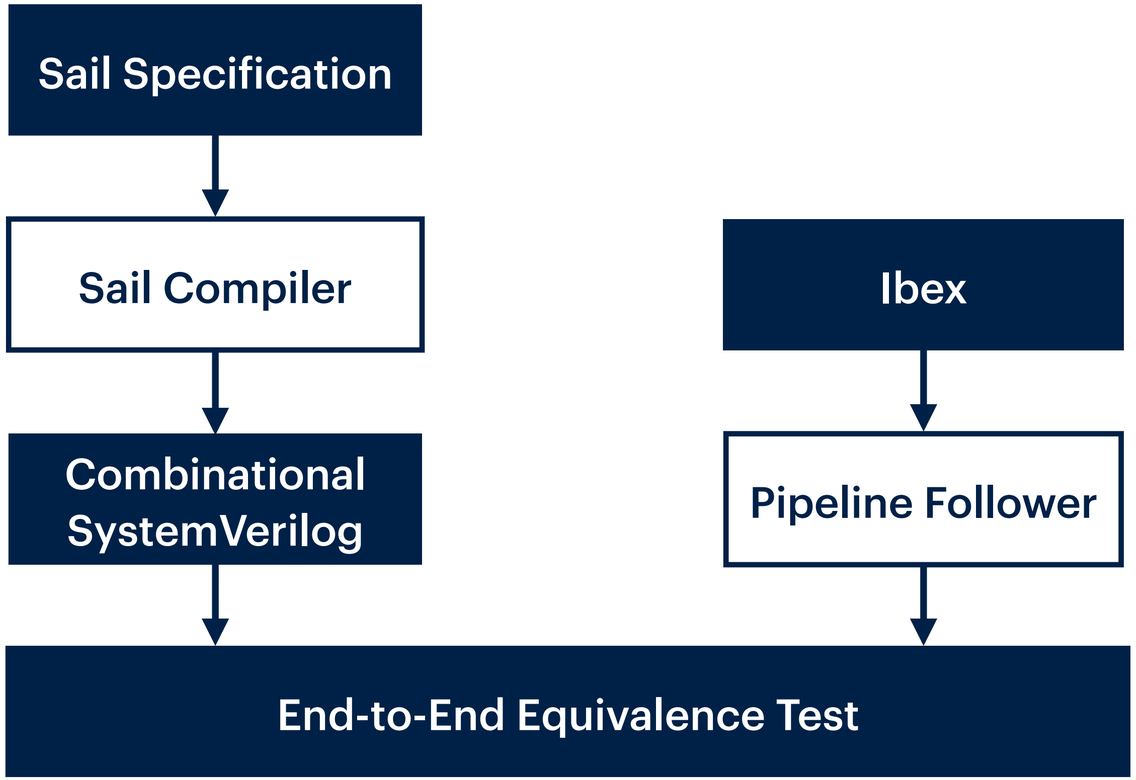
End-to-End Verification

- Based on ARM ISA-Formal paper
- Compile Sail specification to SystemVerilog module
  - Maps architectural state + inputs to next architectural state + outputs



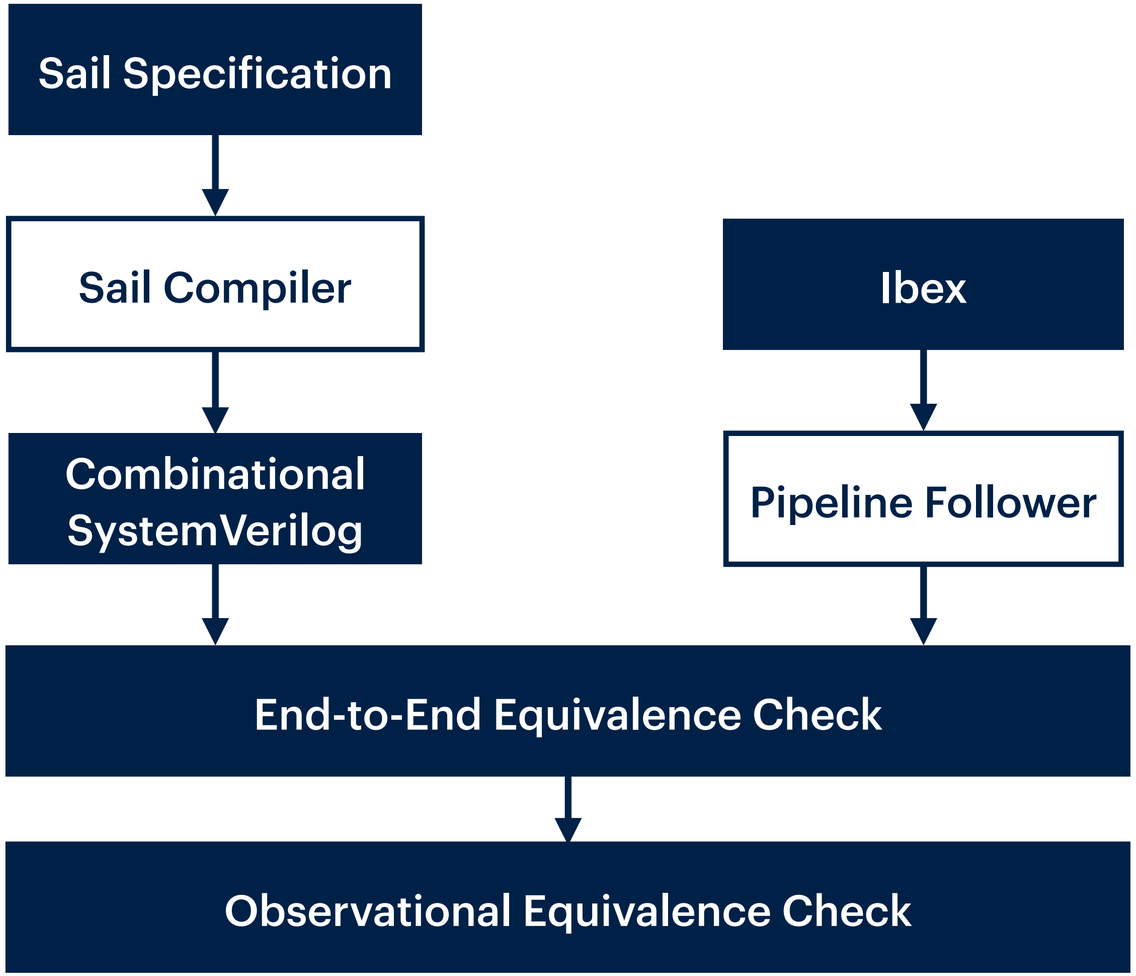
End-to-End Verification

- Based on ARM ISA-Formal paper
- Compile Sail specification to SystemVerilog module
- Monitor start/end of pipeline with pipeline follower
- Check for equivalence
- But depends on internal signals!!!



Observational Correctness

- Novel extension to end-to-end correctness
- Same sequence of memory operations in same order forever
- ‘Wrap around’ end-to-end checks



The Results

- **~20** assume-guarantee steps
- **~700** properties
- **~30** bugs

Illegal CLC load	Store local violation	Sealed PCC	IF granules and overflow
CLC tag bit leak	Memory capability layout	**CSetBounds lower bound check**	MEPCC set_address
CSeal otypes	PCC.address vs. PC	Illegal instruction MTVAL values	User mode WFI
CJALR alignment checks	CJAL vs. CJALR	Memory/branch exception priorities	CSR instruction problems
CSEQX memory vs. decoded	**Memory bounds check overflow**	CSpecialRW exception priorities	Unspecified CJALR
MTVEC/MEPC legalisation	**CLC tag/perms clearing**	EBreak MTVAL values	PMP pipeline flushing on CSR clear
CSC alignment checks	MSHWM/MSHWMB updates	MRet MStatus.MPRV	TRVK RF write collision
CSC decoding	tvec_addr alignment	16 vs. 32 register spec issues	Stack EPC for CHERI NMIs
CSR clear not flushing, PMP			

Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor

Louis-Emile Ploix*, Alasdair Armstrong†, Tom Melham*, Ray Lin*, Haolong Wang*, and Anastasia Courtney*

*Department of Computer Science, University of Oxford
†Department of Computer Science and Technology, University of Cambridge

***Abstract*—The CHERI architecture equips conventional RISC ISAs with significant architectural extensions that provide a hardware-enforced mechanism for memory protection and software compartmentalisation. Architectural *capabilities* replace conventional integer pointers with memory addresses bound to permissions constraining their use. We present the first comprehensive formal verification of a capability extended RISC-V processor with internally 'compressed' capabilities — a concise encoding of capabilities with some resemblance to floating point number representations.**

The reference model for RTL correctness is a minor variant of the full and definitive ISA description written in the Sail ISA specification language. This is made accessible to formal verification tools by a prototype flow for translation of Sail into SystemVerilog. Our verification demonstrates a methodology for establishing that the processor always produces a stream of interactions with memory that is identical to that specified in Sail, when started in the same initial state. We additionally establish liveness. This abstract, microarchitecture-independent *observational correctness* property provides a comprehensive and clear assurance of functional correctness for the CHERIoT-Ibex processor's observable interactions with memory.

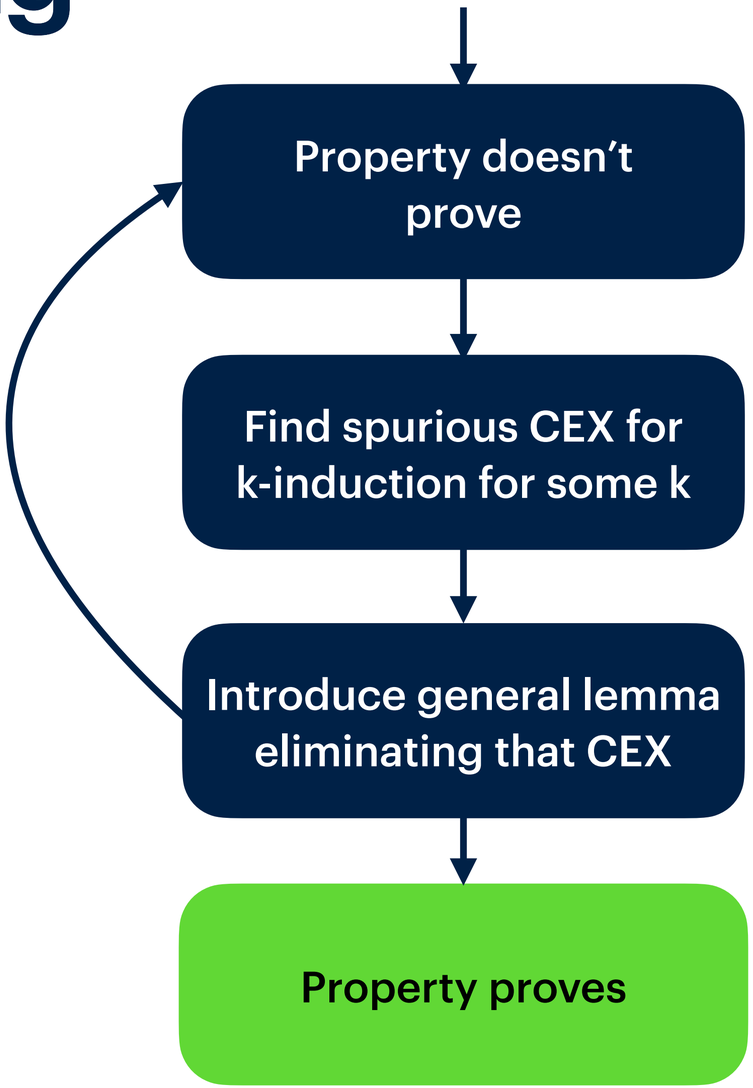
I. Introduction

The CHERI architecture further guarantees that capabilities cannot be created arbitrarily, but only derived from other capabilities. The system starts with certain *root* capabilities from which all other capabilities are derived—whether by a trusted loader, the OS, a capability-aware compiler, or application code. Moreover, a new capability can be derived from an existing one only by narrowing the region of memory it can access or removing permissions. This crucial *non-increasing monotonicity* property is enforced by the hardware and is what lays the solid foundation for strong memory protection and software compartmentalisation.

Early designs for CHERI processors had high memory overhead and memory bandwidth consumption because the upper and lower bounds on the accessible memory region were each represented with the same number of bits as the address [4]. This has been replaced by a sophisticated scheme of *compressed* capabilities, greatly improving the practicality of the approach [6]. This optimisation comes at the cost of making certain address and bounds combinations unrepresentable, and a requirement for the microarchitecture

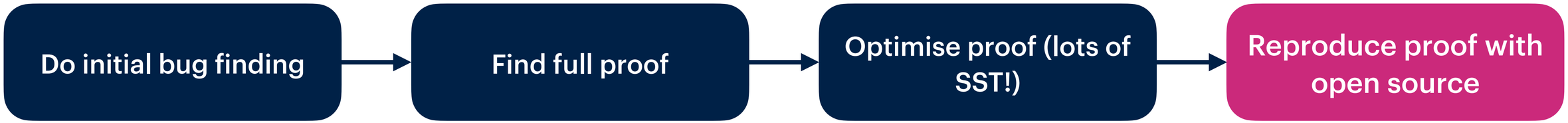
Theorem Proving

- Introduce lemmas to help the model checker
  - Search for spurious k-induction traces
  - Introduces lemmas to eliminate them until none are left
- No big secrets! Just hard thought.



Open Source

- We like open source! Ibex and CHERIoT-Ibex are both open source.
- We want to make the proof reproducible for anyone, at any scale.
- We will use the fact we have well optimised our proof for k-induction, and attempt to reproduce it with an open source model checker.



Open Source Formal Today

- As it stands, there are not many options for open source hardware model checking:
  - EBMC
    - Fails on many SVA properties it considers liveness
  - SBY
    - Mostly a collection of Yosys scripts (the primary open source hardware toolchain)
    - Not actually a model checker, instead invokes others
    - Great for small designs and simple proofs, not well optimised for our scale
- We’re going to use Yosys, as SBY does, since it’s a powerful tool, but we’ll use our own scripts, and build our own pipeline.

Open Source Formal: The Frontend

- First problem: Parsing Ibex + the specification + the verification code requires good SystemVerilog and SystemVerilog Assertion support.
- Options:
  - read_verilog -sv — Native to Yosys, but supports very little SystemVerilog.
  - verific — Requires a license.
  - sv2v — No concurrent assertions.
  - yosys-synlig — Doesn’t parse concurrent assertions.
  - yosys-slang — Parses concurrent assertions (in Slang), but doesn’t yet compile them to RTLIL. That’s pretty close, so let’s do that!

Formally Specifying SVA

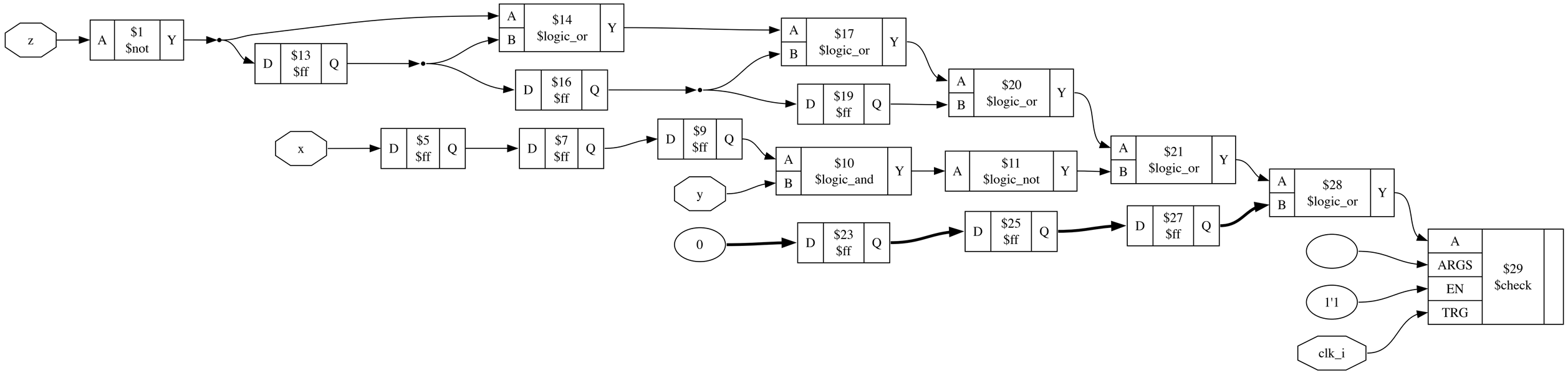
- SVA is a difficult language to define. The specification can be hard to read.
- Therefore, we define the desired semantics of the subset of SVA we care about in Lean 4 (a theorem proving language). We do this in terms of paths.
- With the definitions clear, implementation for this subset becomes straightforward.

```
def SVA.paths : SVA α -> List (SeqPath α)
| state t => [.state t 0]
| seq pre td post => pre.paths.flatMap fun a =>
    td.times.flatMap fun td =>
        post.paths.map fun b =>
            a.and (b.shift (a.end + td))
| or a b => a.paths.append b.paths
| and a b => a.paths.flatMap fun a =>
    b.paths.map fun b =>
        a.and b
| repeats a rc => rc.reps.flatMap (SeqPath.cross a.paths)
| not a => [.not (a.paths.foldl .or .false)]

def SVA.sats (p : SVA α) (pi : Trace α) : Prop :=
    p.paths.any_prop (fun p => p.sats pi)
```

```
inductive SeqPath (α : Type u)
| state : (α -> Prop) -> Nat -> SeqPath α
| or : SeqPath α -> SeqPath α -> SeqPath α
| and : SeqPath α -> SeqPath α -> SeqPath α
| not : SeqPath α -> SeqPath α

def SeqPath.sats (p : SeqPath α) (pi : Trace α)
  : Prop := match p with
| state t n => t (pi.get n)
| or a b => a.sats pi ∨ b.sats pi
| and a b => a.sats pi ∧ b.sats pi
| not a => ¬a.sats pi
```

Example Yosys-slang fork output

```
module m_assert_seq_7(input logic clk_i, input logic x, input logic y, input logic z);
    assert property(@(posedge clk_i) disable iff (~z) x |-> ##3 ~y);
endmodule
```



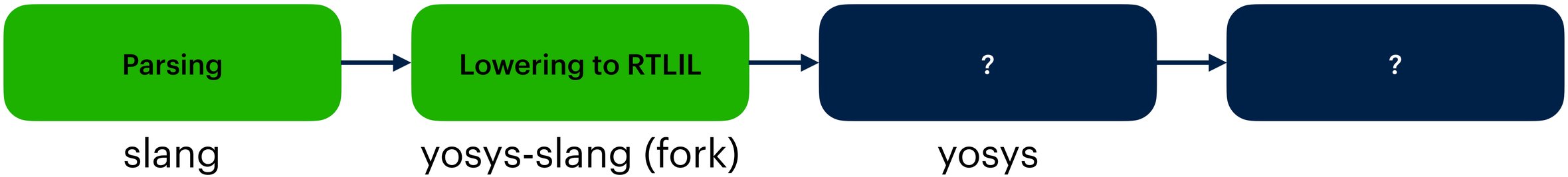
m_assert_seq_7

Notes on property shapes

- Properties with a lot of paths will, in the worst case, blow up in size exponentially.

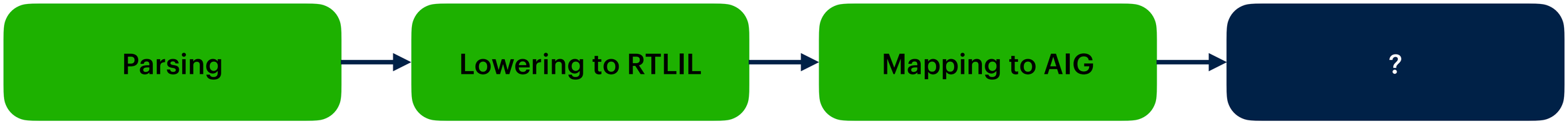
```
assert property(@(posedge clk_i) a ##[0:2] b ##[0:2] c ##[0:2] d ##[0:2] e);
```

- This property takes 3^4 paths, each of which requires 5 nodes.


Parsing
Lowering to RTLIL
?
?
slang
yosys-slang (fork)
yosys

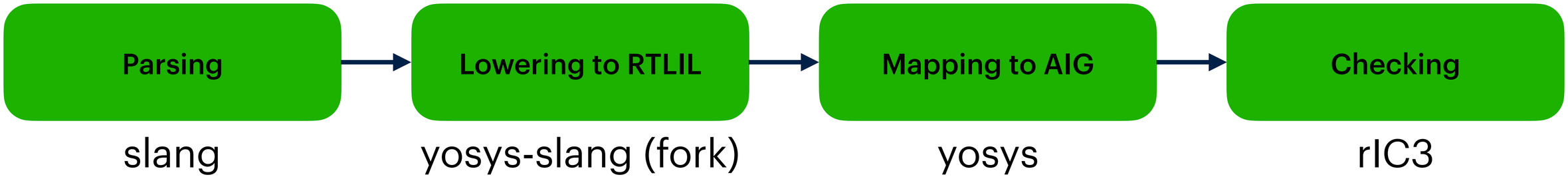
Yosys

- Once we're in yosys, things get relatively simple (but slow).
- We do the bare minimum work to generate an smt2 file and an Aiger file.
  - smt2 files contain the circuit in a form ready for an SMT solver.
  - Aiger files are are topologically sorted sequences of AND and NOT gates. They are incredibly simple and fast to manipulate


Parsing
Lowering to RTLIL
Mapping to AIG
?
slang
yosys-slang (fork)
yosys

Model Checking

- Now we can actually start checking properties, and we have a couple of options:
- yosys-smtbmc
  - The standard solution, happily works with yosys smt2 files to do BMC and K-induction. Essentially just plug in a SAT solver.
  - Not very fast, and many solvers crash due to scale.
- rIC3
  - A Rust implementation of BMC, K-induction and IC3. Runs on Aiger files.
  - Won some competitions and even has a dedicated SAT solver.
- We target rIC3 primarily, but there's little additional cost for us to support yosys-smtbmc too, hence we generate smt2 files anyway.

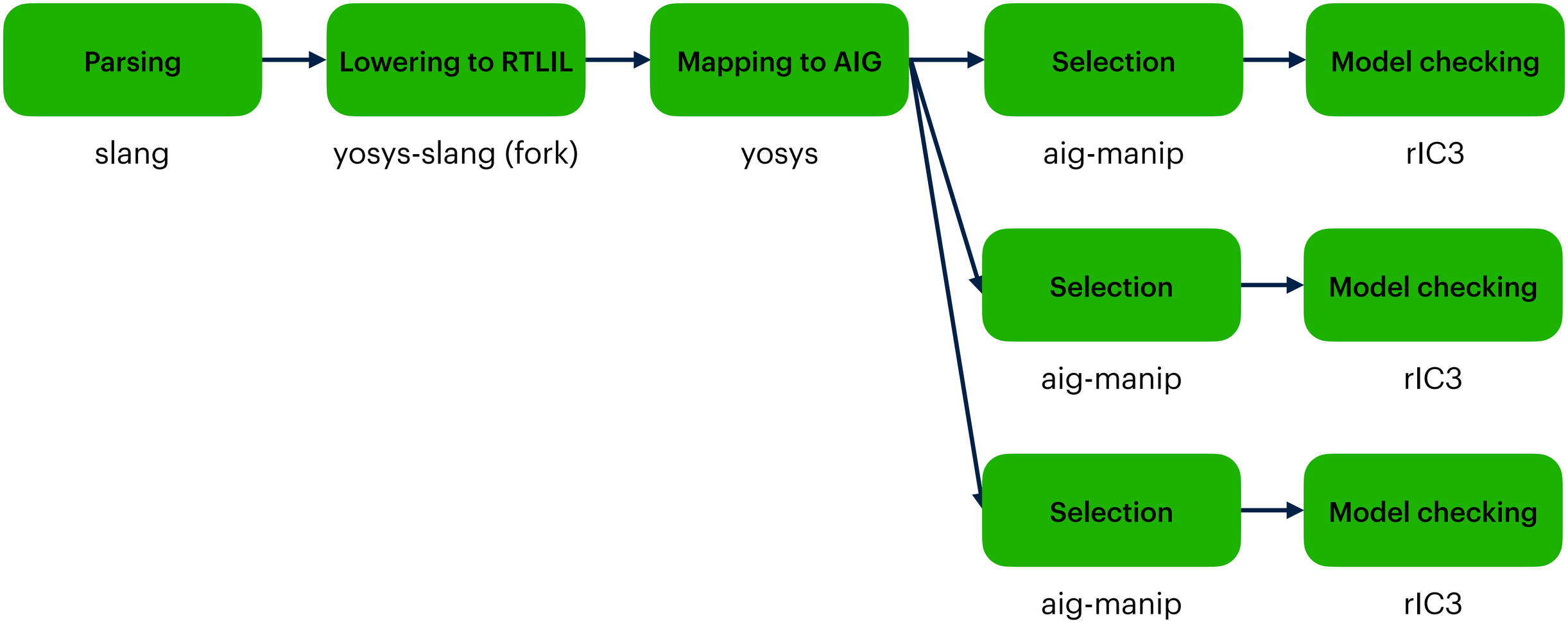

Parsing
Lowering to RTLIL
Mapping to AIG
Checking
slang
yosys-slang (fork)
yosys
rIC3

Yosys is slow

- Some solutions:
  - Make the specification smaller by producing more natural outputs
    - Somehow makes everything worse?
    - Turns out we generate a lot of junk (quadratic in the depth of SV if statements) when doing the proc pass.
    - The opt_muxtree pass (which is mean to clean up that junk) fails because it hits an internal limit.
  - Disable python support in yosys - otherwise it puts all wires in a big global map.

Yosys is slow: Selection

- All the above gets elaboration down to about 10 minutes.
- Hence just elaborate once for all assertions, then edit the output after the fact to remove them, or convert them to assumptions as needed.
- Still far slower than commercial (<1 minute), even for our relatively moderate design, but manageable.


Parsing
Lowering to RTLIL
Mapping to AIG
Selection
Model checking
slang
yosys-slang (fork)
yosys
aig-manip
rIC3
Selection
Model checking
aig-manip
rIC3
Selection
Model checking
aig-manip
rIC3

Final Note: Debugging

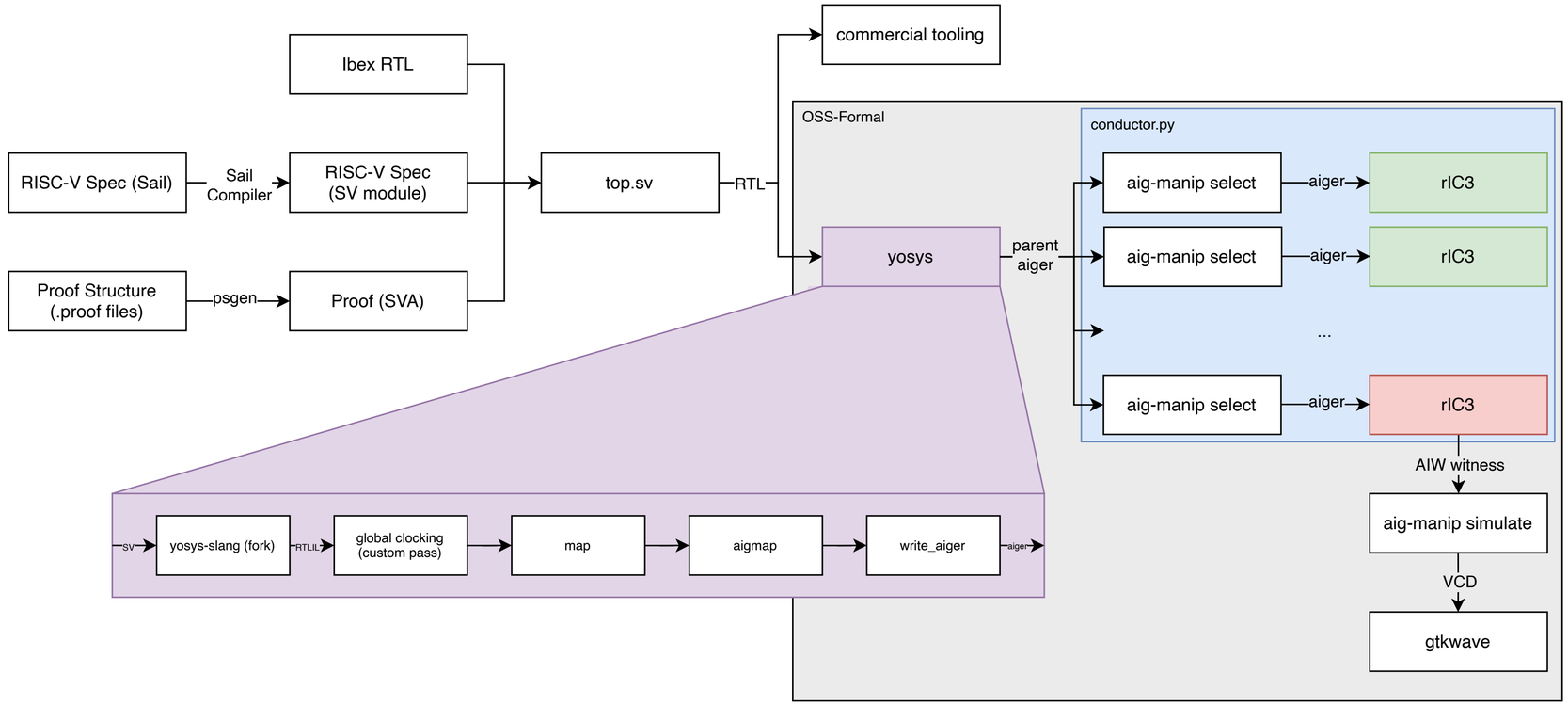
- rIC3 produces witness traces in the AIW format. It's essentially just a sequence of bits representing the initial state and inputs.
- Yosys is meant to be able to transform this into a VCD, and it might be able to do, but it will be incredibly slow.
- Therefore, we have a tool to do that transformation very quickly.
- In general though, debugging in gtkwave is difficult, and is best done with an industry tool.

Automation

- Now we can prove properties at will, we need some way of orchestrating their collective proof.
- We have a relatively straightforward python script (conductor.py) to do that.
  - It cashes proof strategies, proof algorithms, and the k used for k-induction, etc.
  - It attempts to find good combinations of properties to prove as groups instead of individually.

Results

- Full proof for Ibex in ~40 minutes. Running in regression on all PRs. With no cache it takes roughly a day.
- Currently requires an instant memory bound (i.e. all memory requests terminate within 1 cycle).



Conclusion

- Open source formal is a thing, kind of!
- It’s very slow, difficult to work with, and very incomplete, but it’s getting there.
- Not going to replace commercial tools any time soon, but it is nice to be able to offer our proofs in the open, for anyone to run, and even relatively quickly.

Thanks

- Professor Tom Melham, for his supervision and guidance.
- Alasdair Armstrong, for his work on Sail.
- Marno van der Maas, Harry Callahan, John Thompson and all the others at lowRISC for supporting my work there.
- Kunyan Liu for his work on CHERIoT-Ibex.
- Professor Peter Sewell, Alastair Reid, Laurent Arditi and others for their guidance and suggestions.
- Amy, friends and family for their support.

Links

- Me: louis-emile.ploix@stcatz.ox.ac.uk
- Thomas Melham: thomas.melham@balliol.ox.ac.uk
- Paper: https://arxiv.org/abs/2502.04738
- Sail: https://github.com/rems-project/sail
- CHERIoT-Ibex:
  - Formal setup under *dv/formal* in https://github.com/microsoft/cheriot-ibex
  - Sail specification on branch *formal* in https://github.com/lowRISC/cheriot-sail
- Ibex:
  - Formal setup under *dv/formal* in https://github.com/lowRISC/ibex
  - Sail specification on branch *ibex* at https://github.com/lowRISC/sail-riscv