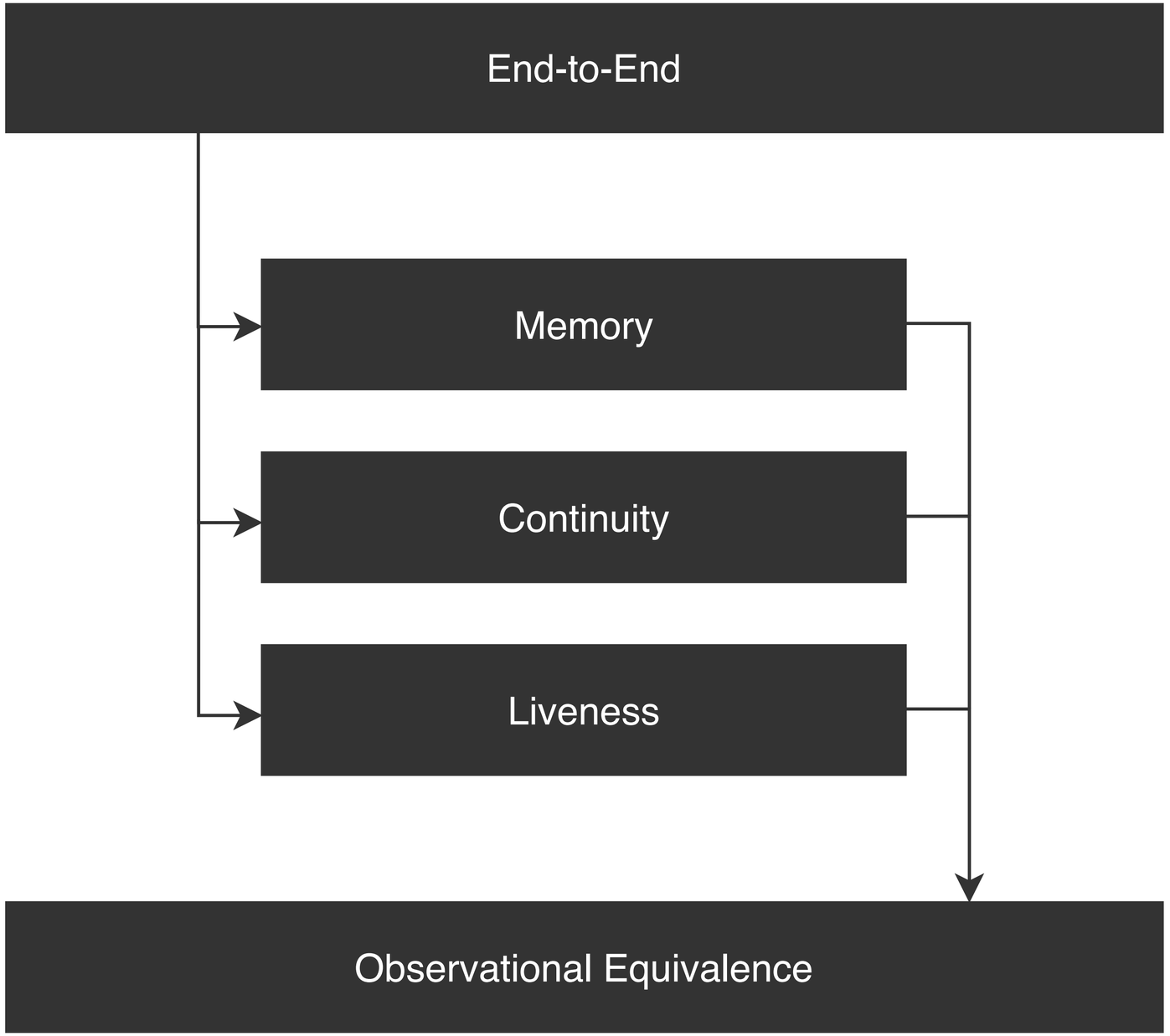
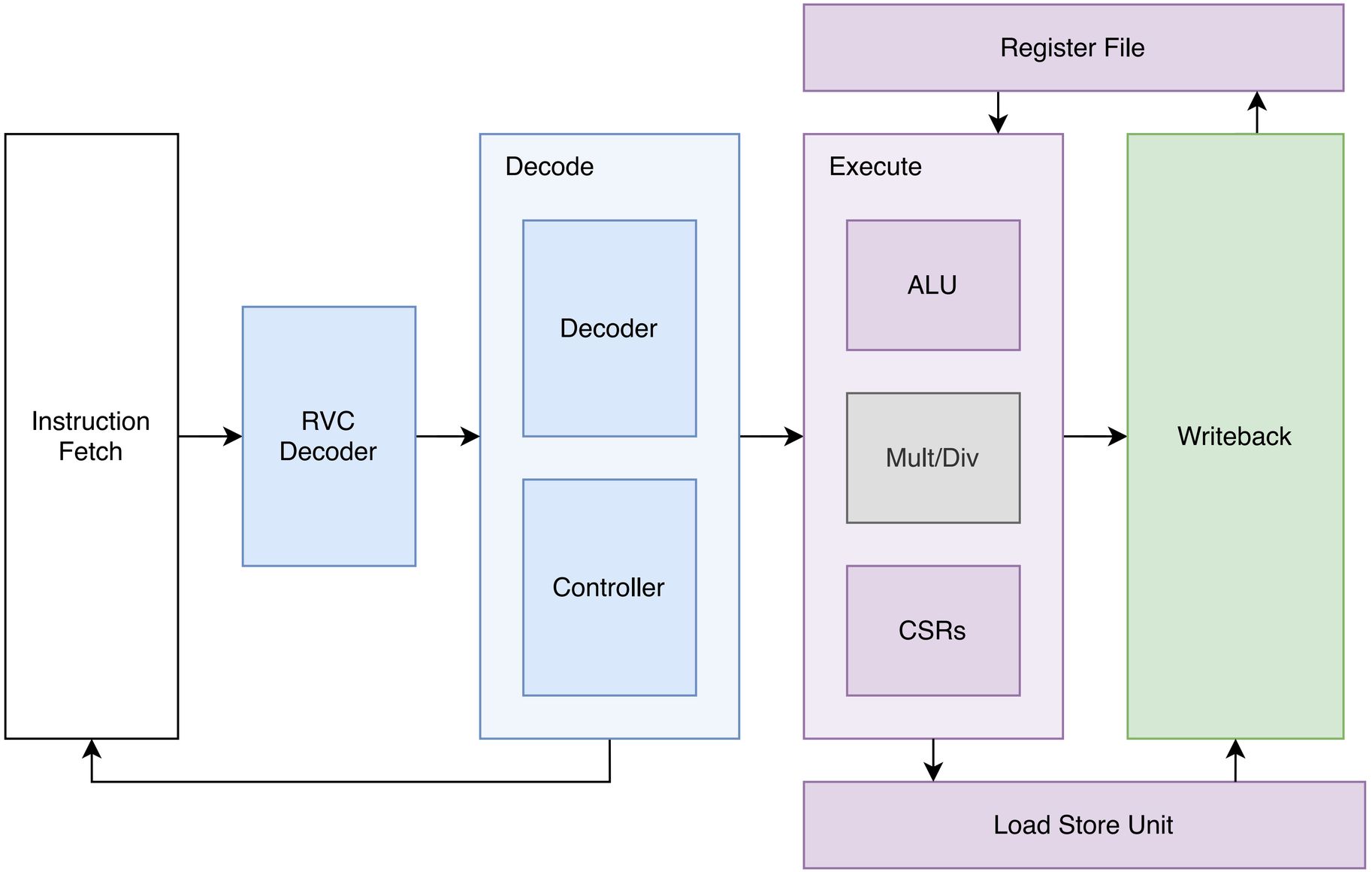
Crossing the Model Checking/ Theorem Proving Gap for Ibex Correctness

Louis-Emile Ploix - 04/12/2025 - DVClub Cambridge

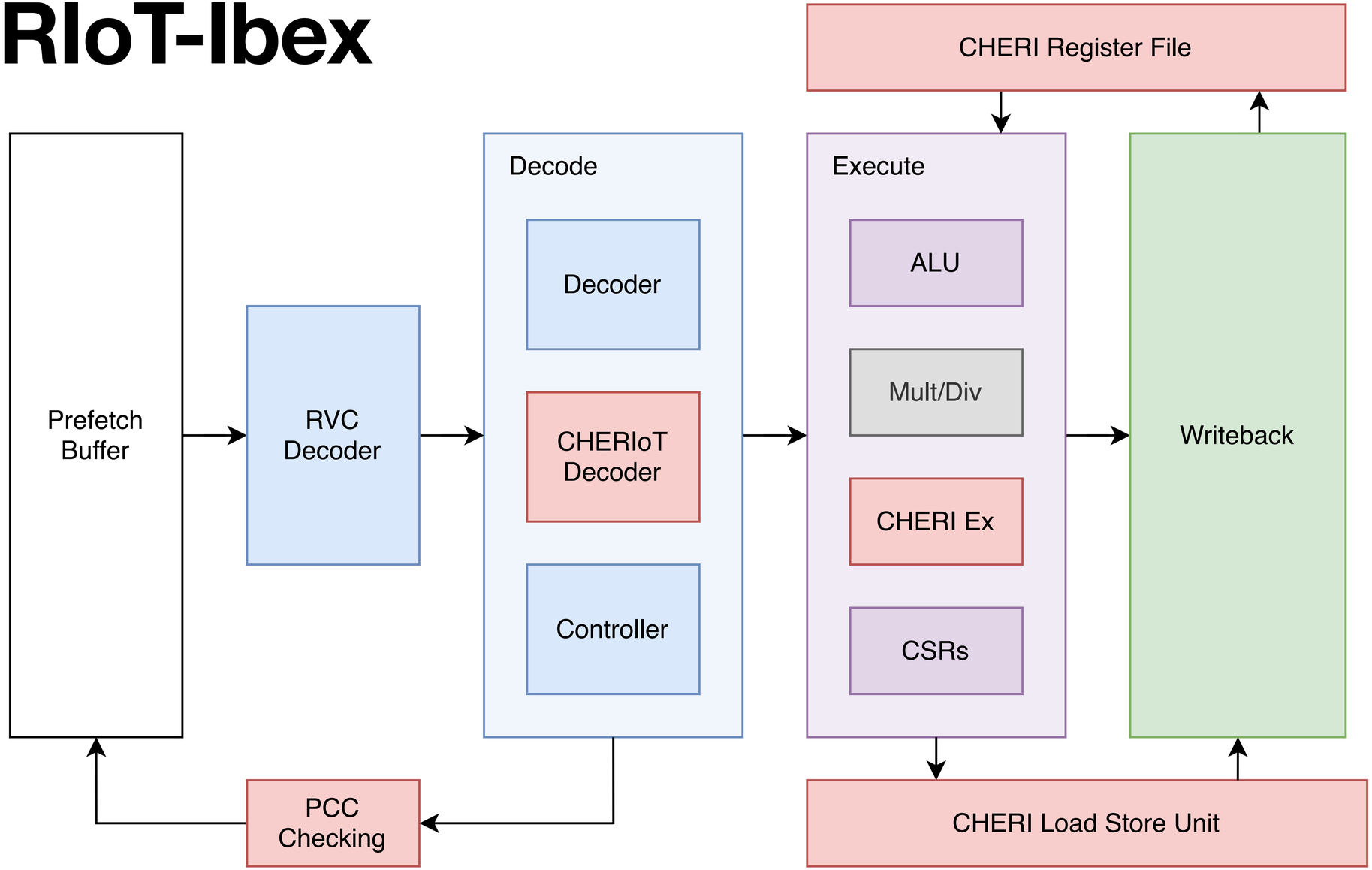
lowRISC


End-to-End
Memory
Continuity
Liveness
Observational Equivalence

Ibex


Register File
Instruction Fetch
RVC Decoder
Decode
Decoder
Controller
Execute
ALU
Mult/Div
CSRs
Writeback
Load Store Unit

CHERIoT-Ibex

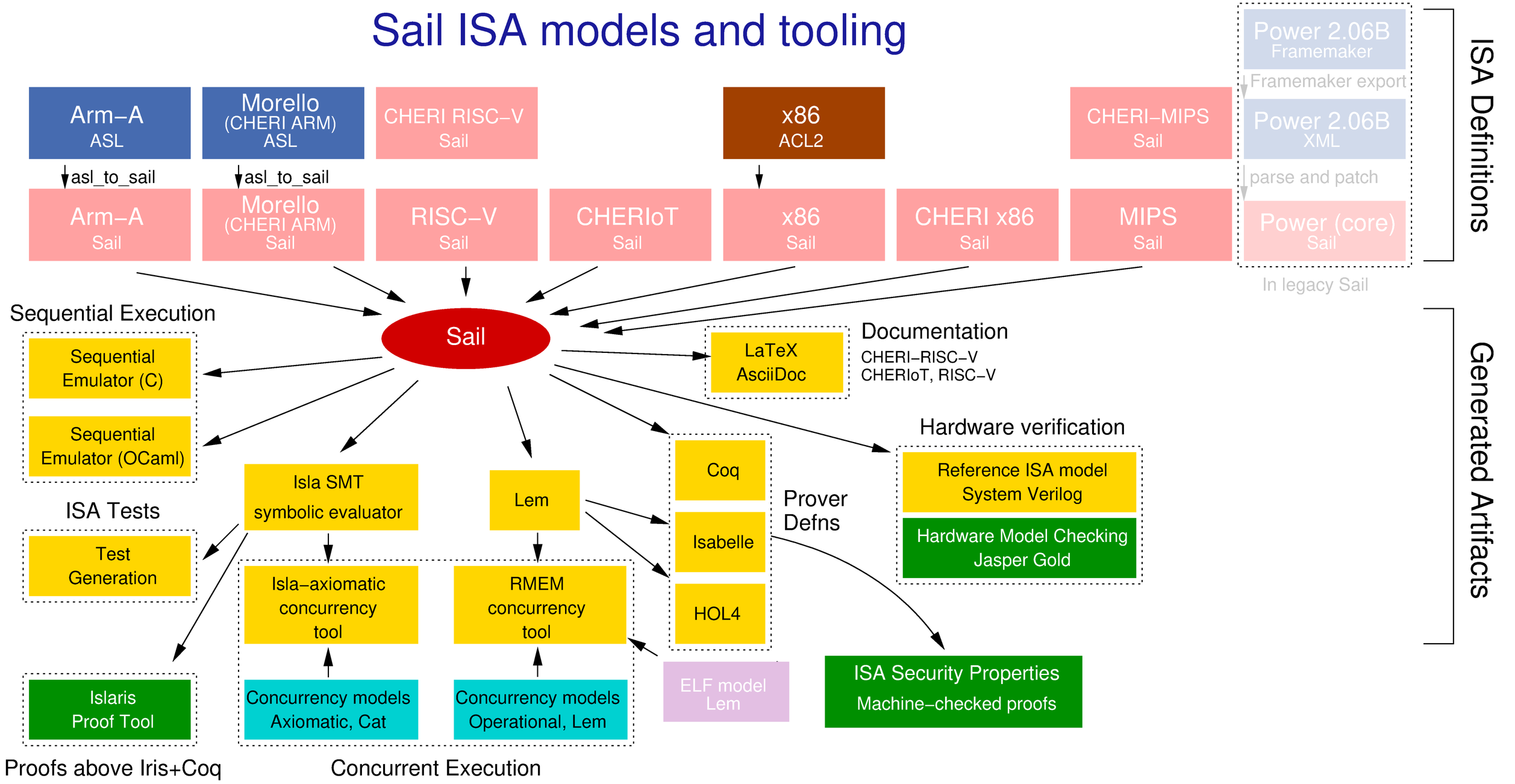


Sail

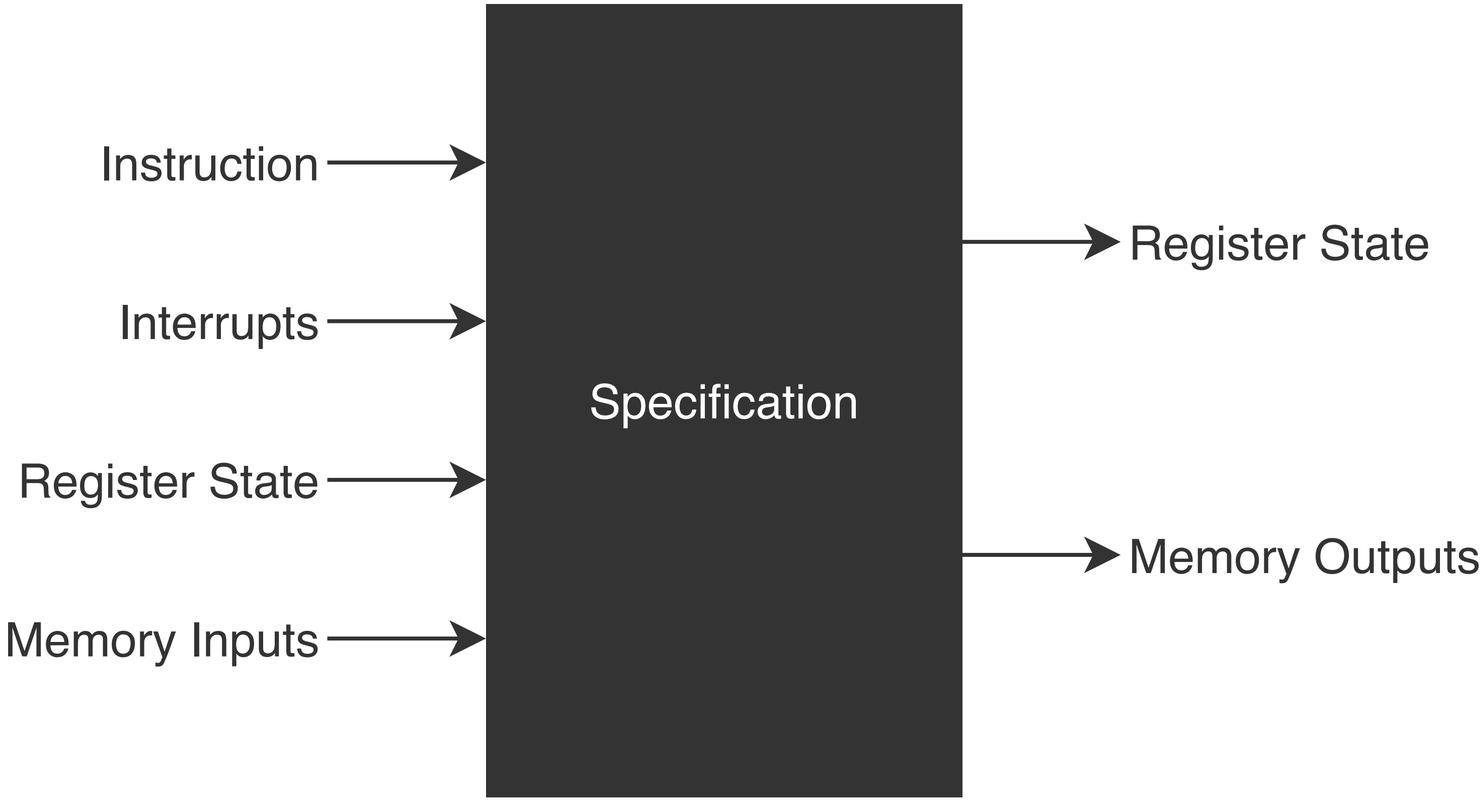
```
mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ encdec_reg(rs1) @ encdec_iop(op) @ encdec_reg(rd) @ 0b0010011

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
}
```

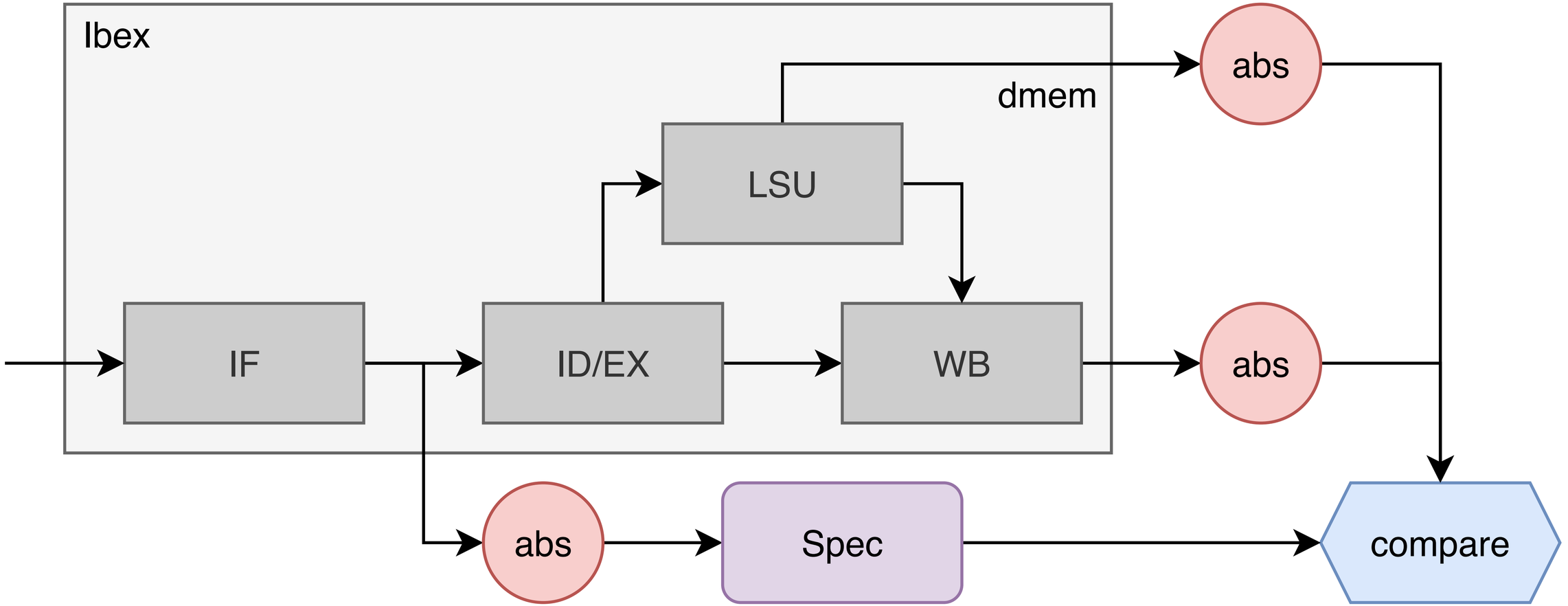
Sail ISA models and tooling



The Specification Module



End-to-end Correctness



Bugs

Illegal CLC load

CLC tag bit leak

CSeal otypes

CJALR alignment checks

CSEQX memory vs. decoded

MTVEC/MEPC legalisation

CSC alignment checks

CSC decoding

CSR clear not flushing, PMP

Store local violation

Memory capability layout

PCC.address vs. PC

CJAL vs. CJALR

Memory bounds check overflow

CLC tag/perms clearing

MSHWM/MSHWMB updates

tvec_addr alignment

Sealed PCC

CSetBounds lower bound check

Illegal instruction MTVAL values

Memory/branch exception priorities

CSpecialRW exception priorities

EBreak MTVAL values

MRet MStatus.MPRV

16 vs. 32 register spec issues

IF granules and overflow

MEPCC set_address

User mode WFI

CSR instruction problems

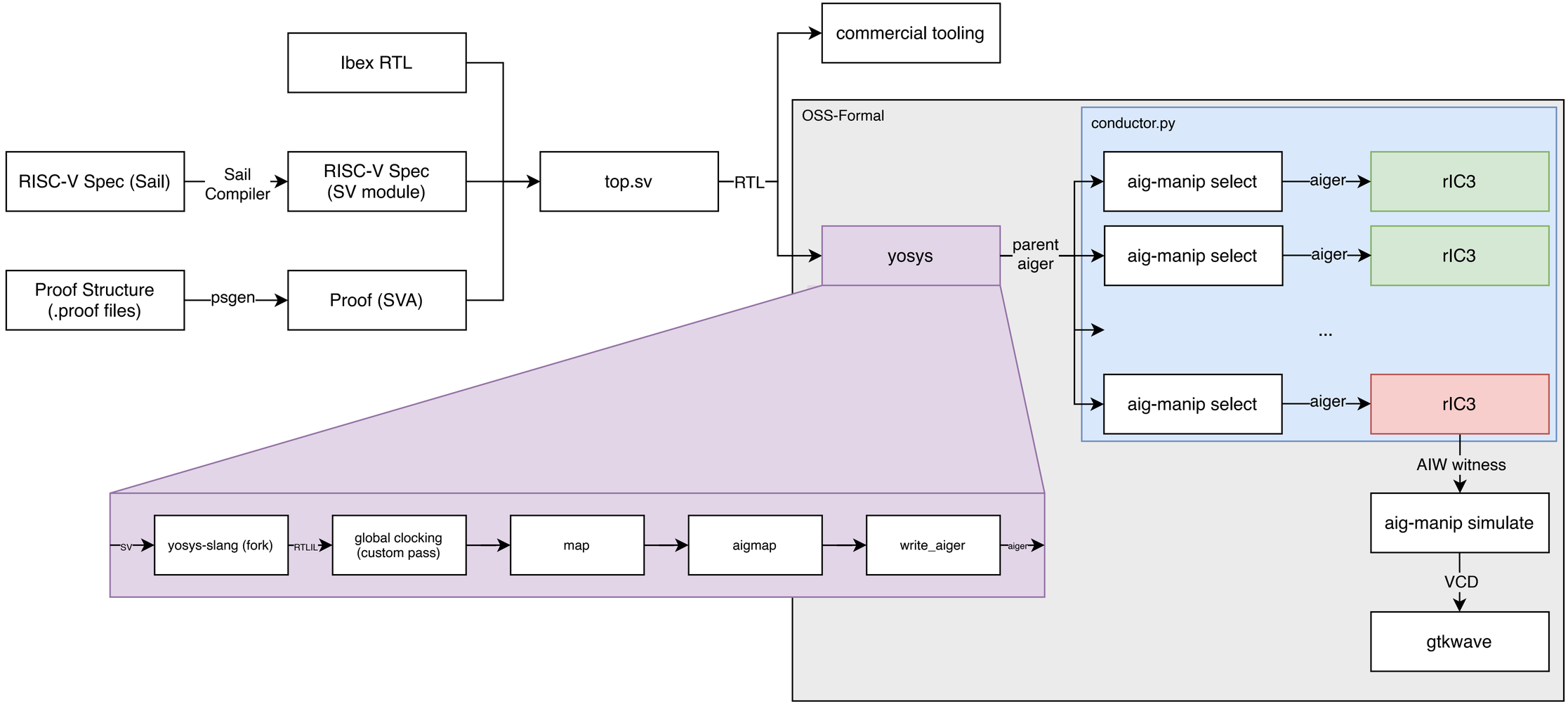
Unspecified CJALR

PMP pipeline flushing on CSR clear

TRVK RF write collision

Stack EPC for CHERI NMIs

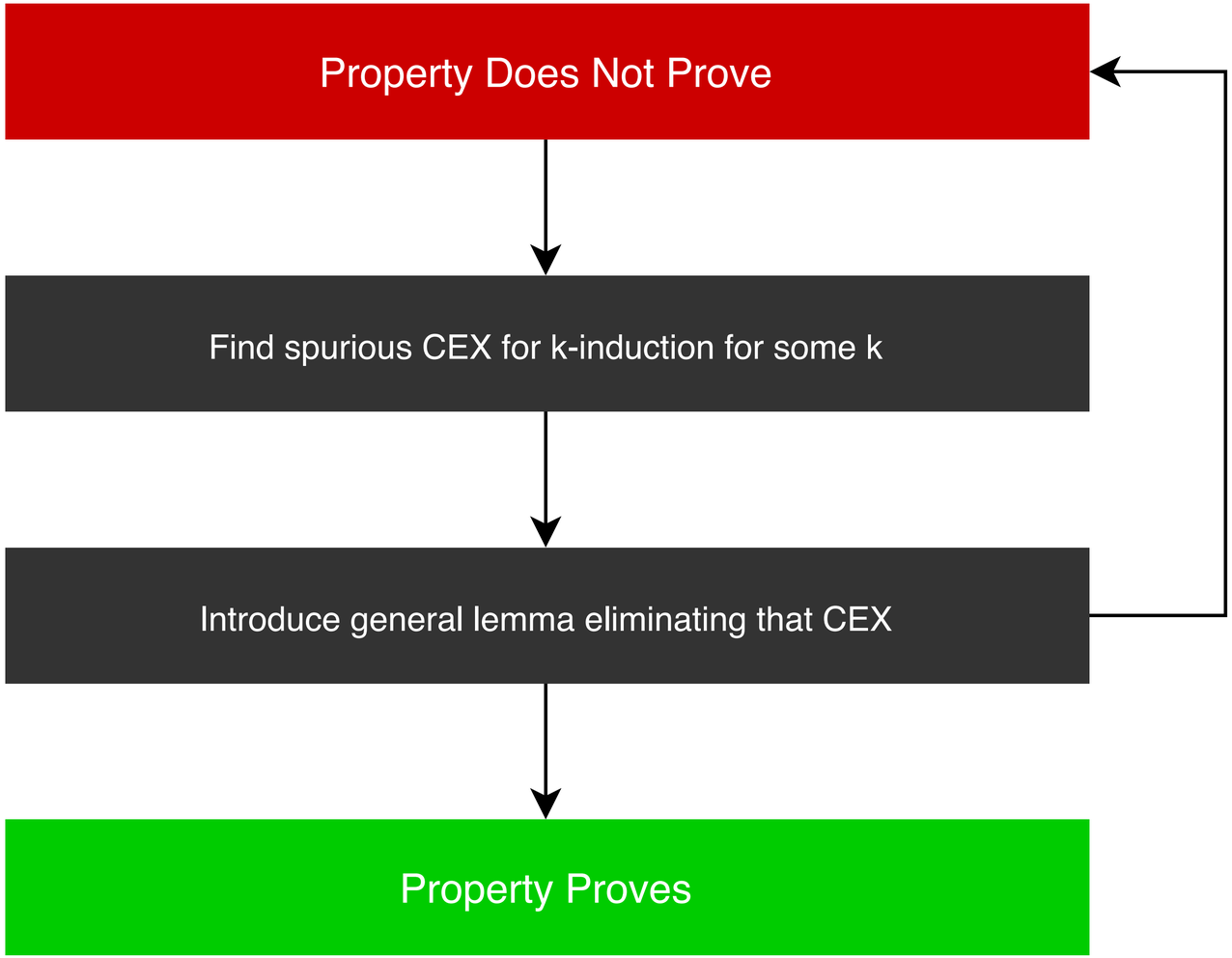
Quick Aside: Open Source



Regression

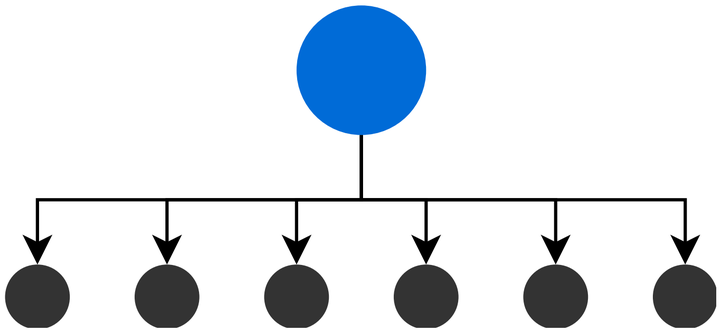
```
4055  [21/08/25 11:43:33.782188] UNSAT: 1 properties in step 9 proven in 53.496s with 13.188GB
4056  [21/08/25 11:43:35.703409] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-277.aig` (55.316s) (13.252GB)
4057  [21/08/25 11:43:35.704089] UNSAT: 4 properties in step 10 proven in 55.316s with 13.252GB
4058  [21/08/25 11:43:39.850663] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-170.aig` (374.093s) (13.262GB)
4059  [21/08/25 11:43:39.851459] UNSAT: 1 properties in step 10 proven in 374.093s with 13.262GB
4060  [21/08/25 11:43:39.952362] Finished `rIC3 build/aiger-290.aig` (22.767s) (6.952GB)
4061  [21/08/25 11:43:39.952722] UNSAT: 1 properties in step 0 proven in 22.767s with 6.952GB
4062  [21/08/25 11:43:45.306982] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-283.aig` (28.428s) (12.885GB)
4063  [21/08/25 11:43:45.307801] UNSAT: 1 properties in step 9 proven in 28.428s with 12.885GB
4064  [21/08/25 11:43:46.115471] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-289.aig` (28.931s) (12.873GB)
4065  [21/08/25 11:43:46.116564] UNSAT: 1 properties in step 5 proven in 28.931s with 12.873GB
4066  [21/08/25 11:43:46.619792] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-288.aig` (29.537s) (12.876GB)
4067  [21/08/25 11:43:46.620610] UNSAT: 1 properties in step 5 proven in 29.537s with 12.876GB
4068  [21/08/25 11:43:46.922784] Finished `rIC3 -e kind --no-preproc build/aiger-294.aig` (29.532s) (12.967GB)
4069  [21/08/25 11:43:46.923299] UNSAT: 4 properties in step 14 proven in 29.532s with 12.967GB
4070  [21/08/25 11:43:47.124623] Finished `rIC3 -e kind --no-preproc build/aiger-291.aig` (29.837s) (12.913GB)
4071  [21/08/25 11:43:47.125182] UNSAT: 11 properties in step 3 proven in 29.837s with 12.913GB
4072  [21/08/25 11:43:47.427108] Finished `rIC3 -e kind --no-preproc build/aiger-295.aig` (30.036s) (12.989GB)
4073  [21/08/25 11:43:47.427275] UNSAT: 2 properties in step 15 proven in 30.036s with 12.989GB
4074  [21/08/25 11:43:48.734940] Finished `rIC3 -e kind --no-preproc build/aiger-292.aig` (31.447s) (13.130GB)
4075  [21/08/25 11:43:48.735222] UNSAT: 6 properties in step 16 proven in 31.447s with 13.130GB
4076  [21/08/25 11:43:50.444188] Finished `rIC3 -e kind --no-preproc --start 1 build/aiger-286.aig` (33.463s) (13.108GB)
4077  [21/08/25 11:43:50.444663] UNSAT: 1 properties in step 9 proven in 33.463s with 13.108GB
4078  [21/08/25 11:43:57.679356] Running 2 processes, with 0 pending. Memory used right now: 29.999GB
4079  [21/08/25 11:44:27.722895] Running 2 processes, with 0 pending. Memory used right now: 32.444GB
4080  [21/08/25 11:44:43.712300] Finished `rIC3 -e kind --no-preproc build/aiger-296.aig` (86.219s) (16.236GB)
4081  [21/08/25 11:44:43.719588] UNSAT: 1 properties in step 22 proven in 86.219s with 16.236GB
4082  [21/08/25 11:44:47.228456] Finished `rIC3 -e kind --no-preproc build/aiger-297.aig` (89.734s) (16.208GB)
4083  [21/08/25 11:44:47.228825] UNSAT: 1 properties in step 22 proven in 89.734s with 16.208GB
4084  [21/08/25 11:44:47.245172] Ran strategy in 1908.236s
4085  [21/08/25 11:44:47.245448] 298/298 proof steps were UNSAT
```

The K-Induction Game

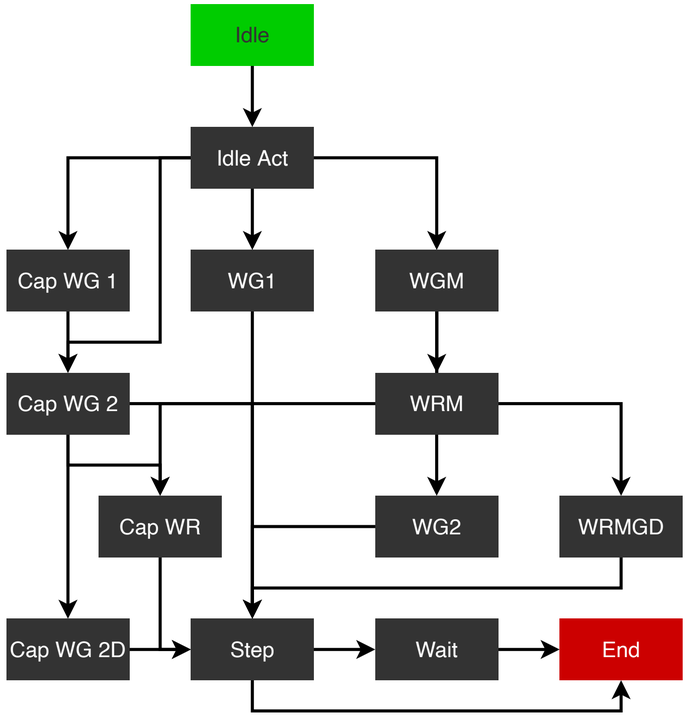


psgen

Case Splitting



Automata



936

properties

243

hand written properties

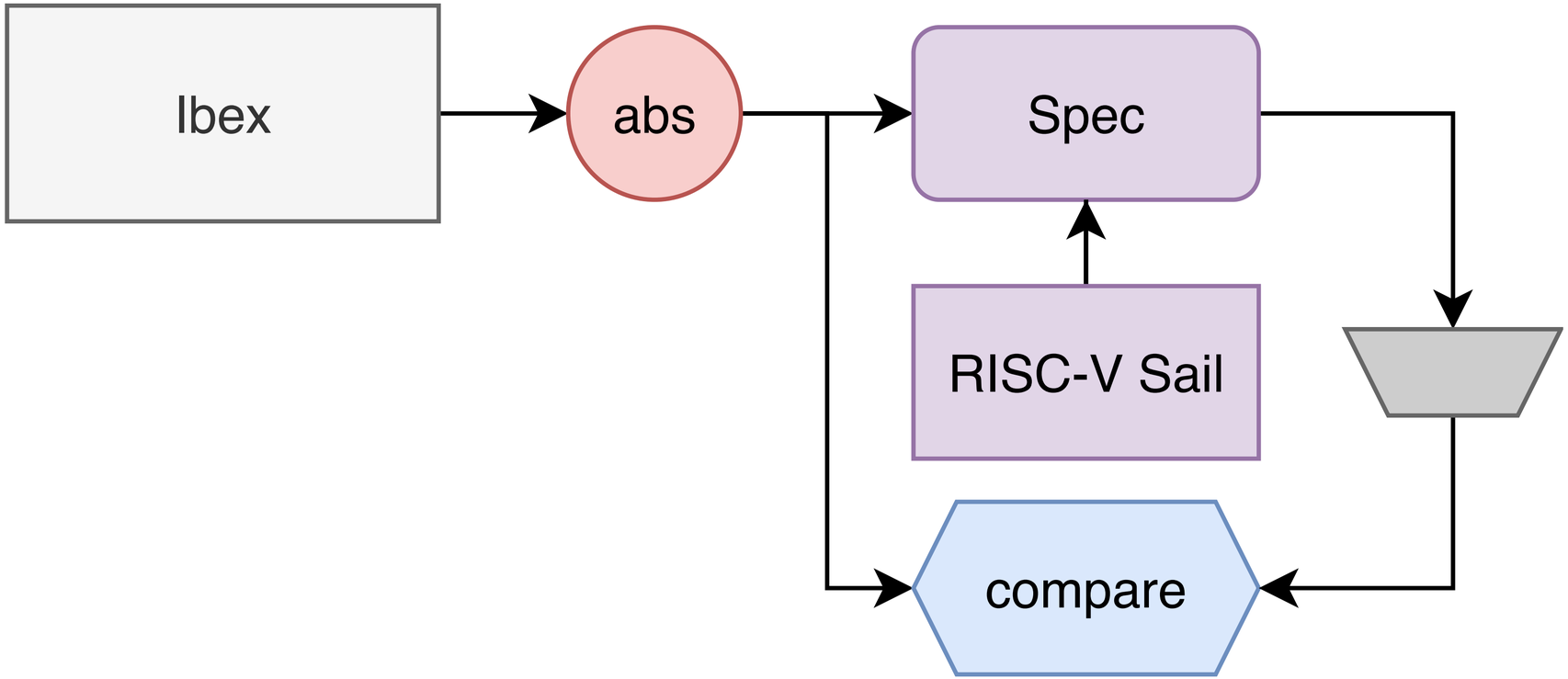
22

assume guarantee steps

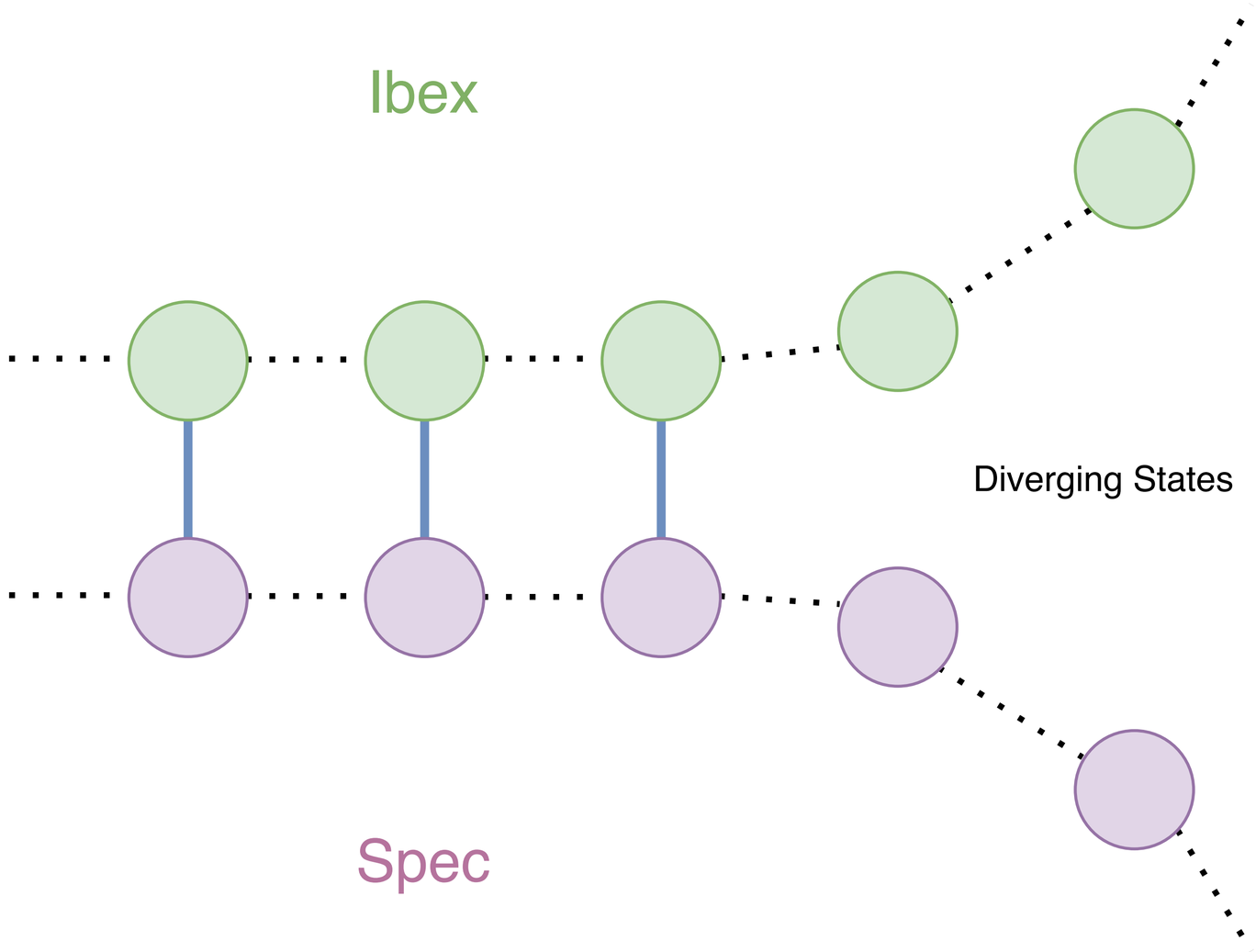
Primary Limitations

- State interpretation
- Internal signals drive verification
- Missing continuity
- Instruction Fetch

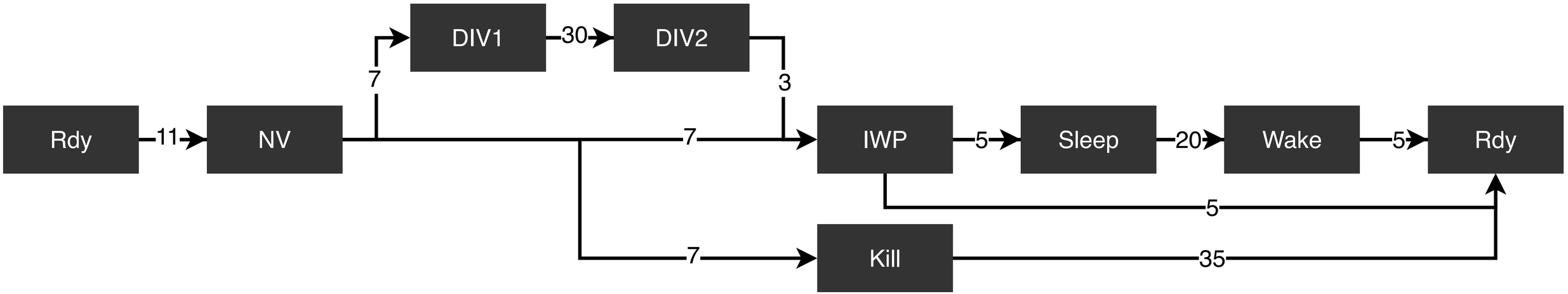
Continuity



Liveness



Liveness



Pen and Paper Proof

validate the constraint as a legitimate assumption about capabilities read from memory. Doing this also has the added benefit of strengthening the DTI, which usually increases both the speed of its own proofs and of dependent proofs.

VIII. Top-Level Observational Correctness

As outlined in Sec. III, the top-level correctness statement for CHERIoT-Ibex is that it produces the same stream of memory interactions as the iterated Sail specification, when these are started in the same initial state. In this section, we explain how this property is established, given end-to-end instruction correctness (Sec. VI) and memory correctness (Sec. VII). We explain the proof bottom-up.

A. Continuity Properties

In our approach to end-to-end instruction correctness, we step the pipeline follower forward using internal processor signals, rather than some independently-implemented control logic. This methodology is easy to implement and powerful, and it finds many bugs. But it does not fully verify the processor's pipeline control logic, which is often complex and may contain subtle corner case bugs.

In our methodology, we therefore complement the end-to-end proofs with certain continuity properties. These assertions use the end-to-end results as lemmas to prove that, in essence, each time the Sail specification is 'queried', the new inputs to the specification are equivalent to (or at least as strict as, in the case of CHERI) the outputs from the previous time the specification was queried.

To implement this approach, we introduce a small, self-contained piece of SystemVerilog that, when the signal `spec_en` is fired, will verify that the new inputs to the compiled Sail specification—which are abstracted from the microarchitectural state of the RTL—match the old outputs. It also, in the same cycle, stores the new outputs to be checked on the next iteration.

It is relatively easy to get conclusive proofs of the continuity properties when the end-to-end correctness properties are provided as lemmas. With some helper invariant work these proofs can be obtained in a matter of seconds or minutes.

Successful verification of these properties extends the end-to-end properties to show that not only does each instruction run correctly, but that the state we finish in is the state the next instruction begins with.

B. State Matching

The next level of our verification establishes that a certain infinite sequence of periodically sampled states of the processor matches the sequence of states produced by iterating the Sail specification, when they both start in the same initial state. In particular, consider the infinite sequence of microarchitectural states obtained by repeatedly sampling the processor state when `spec_en` is high. Recall that this is the signal that triggers an evaluation of the Sail specification. State matching says that the abstraction of this sequence of microarchitectural states should match the sequence of architectural states produced by iterating the specification.

Logically, it does not matter how `spec_en` is defined, but in our case it is high when either an instruction moves from the execute stage to the writeback stage, or when an interrupt is handled. It does, however, matter that `spec_en` is live. If `spec_en` is not always eventually fired then it may be possible for CHERIoT-Ibex to deviate from the specification without ever being checked for correctness again. Essentially, liveness for `spec_en` ensures that states are always eventually compared.

To prove this in our formal verification, we obtain a weak but finite upper bound on the number of cycles between successive `spec_en` states. This is done under the assumption that a processor sleep following a WFI will always be awoken from in bounded time, and that memory response times are bounded, as explained earlier. We proved these weak bounds by considering the 'path' of microarchitectural states between one `spec_en` state and the next. We then prove small bounds over the edges of each state, which are then summed into longer paths until the full cycle is reached. The initial bounds can take a few minutes to prove, but they compose more or less instantly into proofs for longer paths, including `spec_en` to `spec_en`.

Having established liveness, we now provide an argument that the required state matching property holds, given the formal SVA properties we have established in the formal verification tool environment. We begin with some notation.

a) Specification Notation: First, it is helpful to introduce some mathematical notation to refer to the results produced by iterating the SystemVerilog compilation of the Sail specification. It is important to note that the meaning of this notation is not defined mathematically or formally. It is instead an informal notation for referring—in our argument that state matching holds—to what the compiled specification code in fact does.

We write S_A for the set of all architectural states. These comprise the values of all the registers defined in the SystemVerilog compilation of the Sail specification, which going forward we will refer to simply as 'the Sail specification'. These are the global state of execution of the Sail specification; in the compiled code, they are inputs and outputs of the specification SystemVerilog module.

We now let I denote the set of architectural inputs. These are all the non-state inputs to the Sail specification. They comprise the instruction to be executed, the interrupts, and the responses that will come from memory, in case the instruction input is a memory read — as discussed in Sec. VII.

We can now introduce the following notation to refer to updates to the architectural state that are produced by the Sail specification:

$$\mathrm{Spec} : S_A \times I \to S_A$$

The function Spec maps one architectural state to the next. Essentially, it produces the specified mutation to architectural state that the (compiled) Sail computes. This output is the next

state that the specification would be executed in if it were to be iterated.

b) Specification Strengthening: It happens that CHERIoT-Ibex allow tags to be cleared in some situations when the specification would not. From a security perspective, a limited deviation of this kind is reasonable and acceptable, since the implemented behaviour is only stricter than the specification. It does however mean that we cannot prove direct equivalence with the Sail specification, but instead against a hypothetical, stricter version of the specification.

We first introduce a partial ordering $\sqsubseteq : S_A \times S_A$ on architectural states. We say that for any $s, s' \in S_A$, $s \sqsubseteq s'$, exactly when the tag bit of any capability in a register of s is set only if the tag bit of the capability in the same register of s' is also set, and all other state components are equal. We would then say that s is an architectural state that is 'at least as strict' as s'.

We now introduce a function

$$\mathrm{clear} : S_A \times I \times S_A \to S_A$$

denoting the additional tag clearing that the CHERIoT-Ibex implementation does. Given an architectural state and an architectural input that are to be provided to the Sail specification, along with the next architectural state that the specification would produce for these inputs, clear produces the potentially more strict architectural state that aligns with what the implementation would do:

$$\forall s \in S_A, i \in I.\ \mathrm{clear}(s, i, \mathrm{Spec}(s, i)) \sqsubseteq \mathrm{Spec}(s, i).$$

That is, we will assume that clear will clear tag bits of a state exactly when CHERIoT-Ibex would do the same. It is important to note that clear is never explicitly defined in SystemVerilog in our verification code. We introduce this notation solely to facilitate our exposition of state matching. In practice, specification alignment is implemented in the verification code in essence by implementing $\sqsubseteq : S_A \times S_A$, defined by

$$\mathrm{CSpec}(a, i) = \mathrm{clear}(a, i, \mathrm{Spec}(a, i)).$$

Readers interested in the specifics of tag clearing in CHERIoT-Ibex may refer to the `set_address` function of the CHERIoT-Ibex RTL [17].

c) Microarchitecture and State Mapping: Mirroring the specification notation introduced above, we write S_μ for the set of all microarchitectural states. These are all possible assignments of values to all the hardware registers in CHERIoT-Ibex. Similarly, we write I_μ for the set of all microarchitectural inputs. These are the values present on the input ports of CHERIoT-Ibex. As explained in Sec. III, we verify the instruction fetch part of the processor separately, so for our purposes here the 'input ports' include the instruction input signals delivered by instruction fetch to the compressed instruction decoder.

Following decades of common practice in the processor verification domain [39], we introduce an abstraction function $\mathrm{abs} : S_\mu \to S_A$ that maps microarchitectural state to architectural state.

d) Temporal Alignment: In our verification, the specification is purely combinational while the implemented processor is pipelined, with instruction execution spread over several clock cycles—including a dependence on the timing of memory response. In our verification code, the implementation of the pipeline follower is what achieves a temporal alignment [39] of the two levels. We now introduce further (informal) notation to refer to this.

We write X^+ and X^ω for the sets of finite and infinite sequences of elements of X respectively. If $x \in X^\omega$, then x_i refers to the ith element of x, where x_0 is the first element.

Now define the set T_μ to represent the architectural timing information for any given $i \in I$. An element of T_μ defines how long memory operations will have to wait before receiving responses and for how long the processor will have to stall before a new instruction is received. T_μ is never represented explicitly in SystemVerilog; it instead tracks the decisions made by the model to provide or not provide some response at some time.

We introduce the notation $\mathrm{Realise} : I^\omega \times T_\mu^\omega \to I_\mu^\omega$ to denote the mapping from a sequence of architectural inputs, as presented to the specification, to the corresponding sequence of microarchitectural inputs presented to the CHERIoT-Ibex design over successive clock cycles, using the corresponding timing information.

It is important to note that there isn't a simple mapping from a single architectural input to a contiguous finite sequence of microarchitectural inputs that correspond it. This is because a single invocation of the Sail specification will, in general, mark the start of running CHERIoT-Ibex for several clock cycles. During this period of time, due to the pipelined nature of CHERIoT-Ibex, the microarchitectural inputs for several instructions in flight may be sampled. For example, the memory responses, provided symbolically to the specification as stated earlier, will arrive at the memory input ports of the hardware in the expected clock cycles for the memory read instruction that originated them, even though these occur strictly after the clock cycle in which `spec_en` next holds, i.e. after the `spec_en` state for the corresponding memory read instruction, at which point another instruction may be running in the previous pipeline stage with new instruction bits.

For the purposes of discussing state matching, the mathematical notation 'Realise' we introduce here is an abstract representation of the behaviour of certain SystemVerilog code in our machine-executed verification, including elements of the pipeline follower. This code is hand written and designed by reading the documentation for the CHERIoT-Ibex memory ports. We note that Realise could easily be implemented without access to any internal signals, but doing so achieves little more than duplicating some small section of the implementation to measure the same events. Hence for now we

do not do this. We consider Realise to be part of our top level correctness statement, as it needs human interpretation to implement.

c) State Matching: With this notation introduced, we may now prove the state matching property. We begin by letting $i \in I^\omega$ stand for an arbitrary infinite sequence of architectural inputs, and $t \in T_\mu^\omega$ stand for some arbitrary infinite sequence of micro-architectural stall timings. Note that in the verification code the entirety of i and t are, of course, not constructed in advance. These sequences are generated lazily by the model checker, as their elements are required.

We will say that the inputs provided to the specification for occurrence n of a specification query are just the architectural inputs i_n. Of course, in real execution of the processor on concrete instructions, the value of i_n will be partially constructed in advance. For example, the raw instruction bits will be determined several cycles in advance and stored in the fetch FIFO before being presented to the (RVC) compressed instruction decoder.

We now define $a \in S_\mu^\omega$ to be the infinite sequence of architectural states that arise from i. We suppose that a_0 is the initial architectural state of the specification. The infinite sequence of subsequent architectural states is then defined by

$$a_{n+1} = \mathrm{CSpec}(a_n, i_n) \quad \text{for all } n \geq 0.$$

We now let $i' = \mathrm{Realise}(i, t)$ be the infinite sequence of microarchitectural inputs sent to CHERIoT-Ibex, one for each clock cycle, that corresponds to the sequence i under timings t. We define $s \in S_\mu^\omega$ to be the infinite sequence of microarchitectural states that arise from i', with s_0 being the reset state of the microarchitecture. We argued above that `spec_en` will be true infinitely often, under any sequence of inputs to CHERIoT-Ibex. So we can define $\mathrm{en}(s) \in S_\mu^\omega$ to be the infinite subsequence of s obtained by sampling the sequence s whenever `spec_en` is true, where $\mathrm{en}(s)_0$ is the microarchitectural state when `spec_en` is true for the first time. We use the notation $\mathrm{en}(s)_{..i}$ to refer to the microarchitectural state immediately preceding the reset state.

Finally, the state matching property says that for all $n \geq 0$,

$$\mathrm{abs}(\mathrm{en}(s)_n) = a_n. \tag{1}$$

That is, every time the specification module in our verification code is invoked on an abstraction of the current microarchitectural state—i.e. exactly when `spec_en` is high—the abstracted architectural state produced in our verification code matches the iterated architectural state of the specification. The proof proceeds by induction on n, as follows.

For the base case, when n is 0, then `spec_en` is high for the first time. We take $\mathrm{abs}(\mathrm{en}(s)_0) = a_0$ as an as of yet unproven assumption.

Now suppose that our induction hypothesis (1) holds for an arbitrary $n \geq 0$ and consider microarchitectural state $\mathrm{en}(s)_{n+1}$. At the previous clock cycle when `spec_en` was high (in state $\mathrm{en}(s)_n$), we stored the output of the implemented specification module:

$$\mathrm{CSpec}(\mathrm{abs}(\mathrm{en})_n, i_n) = \mathrm{CSpec}(a_n, i_n) = a_{n+1}$$

By the inductive hypothesis. Our SVA continuity properties directly assert that $\mathrm{abs}(\mathrm{en}(s)_{n+1})$ is equal to this stored state. So we have that

$$\mathrm{abs}(\mathrm{en}(s)_{n+1}) = a_{n+1}$$

as required.

C. Observational Correctness

From state matching (1) we can conclude that under a 'reasonable' definition of abs the sequence of internal states of CHERIoT-Ibex will match those of the specification infinitely often—specifically, whenever `spec_en` is high. But this is not quite abstract enough for our final statement of correctness. This is because the significance for correctness of this property depends on the definition of abs, which is derived from internal signals of CHERIoT-Ibex. Observational correctness improves on this by instead formulating correctness by comparing only outputs. We sketch the argument below.

a) Notation: We will write O for the set of all architectural outputs. These are the outputs of the Sail specification that represent the values involved in memory events. In the case of a memory write, the values are the address, the data sent to memory, and certain byte enable flags. In the case of a memory read, we have only the address and byte enable flags. Finally, there is a value to represent 'no memory event'. We can then introduce the notation

$$\mathrm{SpecOut} : S_A \times I \to O$$

to stand for a mapping from the current architectural state and inputs to the outputs that the specification produces.

For the implementation level, we introduce O_μ for the set of all microarchitectural outputs. These are simply the values on the memory output ports of CHERIoT-Ibex on each clock cycle. These values comprise the memory request, write enable and byte enable signals, and any data that is to be written to memory. The correspondence between values in O_μ and values in O is straightforwardly implemented in our SystemVerilog code.

b) Observational Correctness: Suppose $i \in I^\omega$ is an arbitrary infinite sequence of architectural inputs and $a \in S_A^\omega$ is the infinite sequence of architectural states that arise from i. We can introduce $o \in O^\omega$ defined by

$$o_n = \mathrm{SpecOut}(a_n, i_n) \quad \text{for all } n \geq 0$$

to stand for the corresponding infinite sequence of memory events that would be obtained by hypothetically iterating the specification alone.

We further define $\mathrm{Check} : O \times T_\mu \to O_\mu^+$ to stand for a function that maps each architectural output in the sequence o to the corresponding finite sequence of microarchitectural outputs that would have to be generated by the CHERIoT-Ibex core to correctly enact this architectural output. This depends on the timing information for that architectural cycle, since requests sent to memory will only be granted after some microarchitecturally defined time.

As with Realise, the mathematical notation 'Check' is an abstract representation of some SystemVerilog code in our machine-executed formal verification, and we regard this function as part of our top level statement, needing human involvement to implement.

The top level memory properties verify directly in SVA that for all $n \geq 0$ and any $t \in T_\mu^n$, if $o_n \in O_\mu^+$ is the sequence of memory outputs produced by CHERIoT-Ibex as it progresses from state $\mathrm{en}(s)_{n-1}$ (exclusive) to $\mathrm{en}(s)_n$ (inclusive) under timings t_n, then

$$\mathrm{Check}(\mathrm{SpecOut}(\mathrm{abs}(\mathrm{en}(s)_n), i_n), t_n) = o_n. \tag{2}$$

Given (2), we may now apply state matching (1) to obtain

$$\mathrm{Check}(o_n, t_n) = o_n.$$

Since there are infinite `spec_en` states, we can then conclude that CHERIoT-Ibex and the tightened specification will produce precisely corresponding outputs forever, up to microarchitecturally defined timings and under suitable implementations of Realise and Check. This is the highest level statement one could hope to make in this context, and establishes observational equivalence.

c) Top Level Memory Properties: We now explain how the top level memory properties prove (2). First, we note that our formal verification properties include a check that, for the case of memory instructions, the memory outputs from the compiled specification module in SystemVerilog remain stable during the period from the clock cycle at which CHERIoT-Ibex begins its first request (if any), up to and including the clock cycle in which the next `spec_en` state occurs. It is therefore legitimate to take this stable output result from the compiled specification as the unique reference for correctness of the machine's outputs during this period.

Now, for each clock cycle of the relevant sequence of microarchitectural states, our memory correctness properties check that if a memory request is made, it is correct with respect to the sequence stipulated by $\mathrm{Check}(o_n, t_n)$. Correctness here means that in each cycle in which a memory request is in fact made by the processor, the addresses, byte flag, and (in the case of memory writes) data match what the reference sequence requires. Our memory properties further assert that the same number of memory requests are made by the CHERIoT-Ibex core as $\mathrm{Check}(o_n, t_n)$ requires. Combined, this means that if CHERIoT-Ibex makes a request it does so correctly, and that if it is expected to make a request is makes one.

IX. Design Bugs Revealed

Our work revealed around 30 bugs in the CHERIoT-Ibex design—virtually all to do with the capability extensions added to the original Ibex core. These were reported to the design team, confirmed, and fixed or remained under discussion at the time of writing [40]. The bugs found range from minor inconsistencies to exploitable, monotonicity-breaking vulnerabilities. We focus on the latter here, as these are of primary interest in a processor designed as a foundation for system security.

This bug count is a conservative lower bound. We have, for example, grouped a number of exception priority anomalies into one 'bug', and don't count new, related bugs introduced by trying to fix an already found bug. We also note that the design was at a relatively early state of maturity when we began formal verification. We have no doubt that some of the bugs revealed by formal would have surfaced under intensive design simulation or Burch-Dill though [36] type formal verification. However many of the more interesting corner case bugs are unlikely to have been uncovered without multiple instructions interacting with one another.

It is notable that the new Sail to SystemVerilog flow enabled us to get verification underway fast and to start finding bugs very quickly. Once a pipeline follower is set up bug hunting may begin immediately under bounded verification. While we have gone to great lengths to obtain fast converging proofs for our verification, successful bug hunting, especially in the early stages of development, does not strictly require it.

The following account of some of the bugs we found is somewhat technical and makes significant reference to the specifics of CHERI semantics. For a full understanding of the details, it may be useful to read in conjunction with the main reference for compressed capabilities [6], the CHERIoT-Ibex semantics [9], and the Microsoft CHERIoT technical report [10].

The first vulnerability was discovered by the DTI. It allows for the moving of the address of a capability without doing an associated representability check. Such checks are made whenever the address of a capability is changed. Their purpose is to ensure that the new bounds are identical to the old ones (since those bounds are a function of the address). If the new bounds are different the tag of the offending capability would be cleared. A missing representability check will be caught by the DTI since the `top_cor` and `base_cor` values should also be recomputed when the address is changed.

The `CLoadCapImm` instruction loads a capability from memory by dereferencing another 'authorising' capability. The bug arose when a specific bit of a `CLoadCapImm` instruction was set which should render the instruction illegal. The instruction would decode regardless, and yet it would still be marked as illegal. This caused an exception to be raised, but the load command would nonetheless continue to be dispatched to memory. This is a bug, and the fix was to prevent that load command being sent.

Without the fix however, if the response from memory took long enough to arrive, the load-store unit could return the result to the writeback stage at the same time that another instruction was retiring. By a simple and reasonable design choice of the writeback stage, the addresses of the illegally loaded capability and of the capability coming from the other instruction would be ORed-together before being sent to the register file for storage. This means that the address of the capability changed without any representability check being made, meaning the bounds could move.

Dependent Types

```
new :: (n : Int) -> Vec n
```

A Formal Definition of Ibex

- μ - The set of micro-architectural states
- $R \subseteq \mu$ - The set of reset states
- i_μ - The set of micro-architectural inputs
- Ibex : $\mu \times i_\mu \rightarrow \mu$ - The step function
- *sig* : $\mu \times i_\mu \rightarrow v$ - The extraction function for any signal *sig*

A Formal Definition of The Specification

- A - The set of architectural states
- R_A - The architectural reset state
- i_A - The set of architectural inputs
- Spec : $A \times i_A \rightarrow A$ - The step function
- SpecOut : $A \times i_A \rightarrow O$ - The output function

Lean Spec Definition

- A - The set of architectural states
- R_A - The architectural reset state
- i_A - The set of architectural inputs
- Spec : $A \times i_A \to A$ - The step function
- SpecOut : $A \times i_A \to O$ - The output function

```
structure Spec (v r i : Type u) where
  step : AbsState v r -> i -> AbsState v r
  routputs : RestrictedAbsState v r -> i -> List (AbsMemOp v)
  init : AbsState v r
```

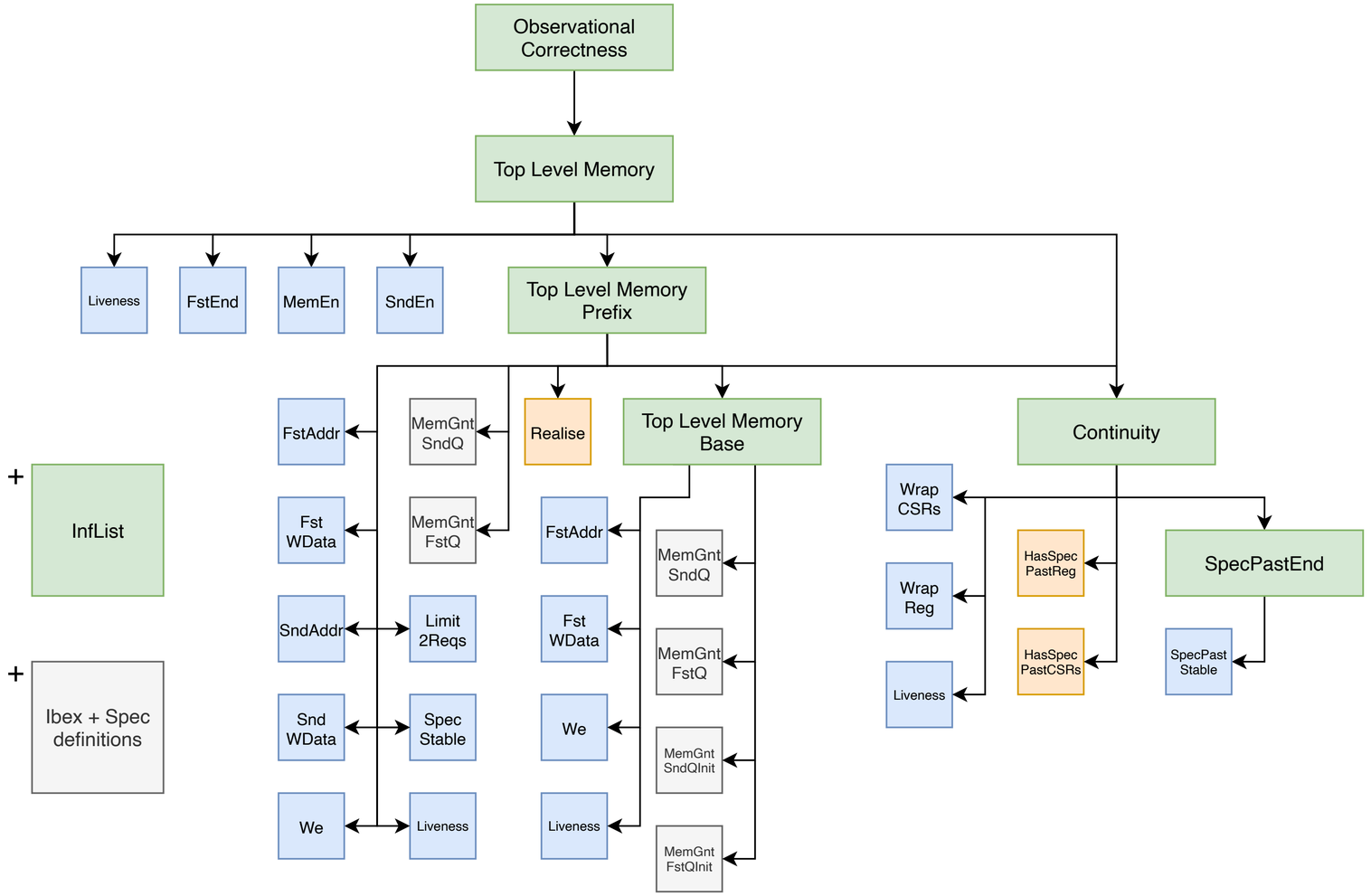
Lean Axioms

```
SndEn: assert property (mem_req_snd_d |-> spec_mem_en_snd)
```

i.e. if Ibex had a second memory operation, then the spec did too

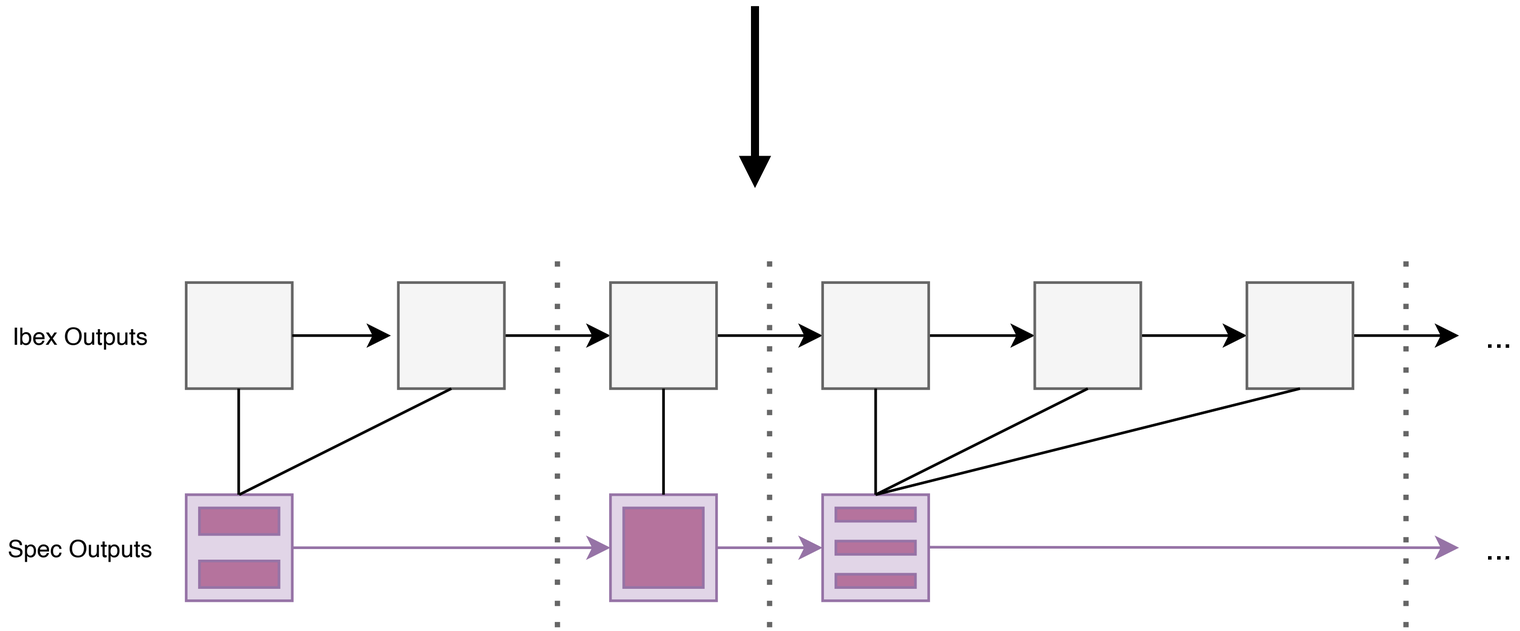
```
axiom SndEn (v : @Verif αi μ va r) (s : CombIn μ va) (reach : v.ibex.reachable s.state) :
  v.mem_req_snd_d s -> v.spec_mem_en_snd s
```

Lean Proof Structure



Lean ObservationalCorrectness Definition

```
theorem ObservationalCorrectness (v : @Verif αi μ v r) (is : InfList αi) :
  let spec_outs := (v.sample_spec is).map fun (s, i) => v.spec.outputs s i
  let ibex_outs := (v.sample is).map fun s => (v.ibex.outputs s).interpret
  ∃ (ln : InfList Nat),
    (spec_outs.zip (ibex_outs.chunk ln).val).map_all fun (x, y) => x = y.flatten
```



Caveats

- Assumption of equivalent inputs
- Instruction fetch
- Some CSRs not missing
- Assumptions about debug mode
- Side channels

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.