DRILLS: Debugging RTL Intelligently with Localization from Long-Simulation

VIGHNESH IYER and DONGGYU KIM, UC Berkeley

With increasing RTL design complexity it is increasingly difficult to debug why a subset of tests fail in chip-level emulation. We propose the use of specification mining to

1 INTRODUCTION

Specification mining is a technique to extract LTL properties from a set of traces of signals. This technique can be applied for several purposes in the domain of digital hardware verification including:

(1) Developing a suite of assertions to be used for design regressions: once a design is mature, most of the interface boundary specifications are well defined and fine-grained assertions derived from specification mining can help catch regressions when design refinements or optimizations are being made.
(2) A starting point for formally specifying a design: once a design can pass random-stimulus based and directed unit tests, specification mining can be used to extract properties that have been consistently observed in the test waveforms. These properties can then be used as assertions to prove formally.
(3) Early anomaly detection and localization on long-running tests on mature RTL: if a specific test fails on an RTL design while many other tests pass, specification mining can reveal *where and when* a failing test begins to produce unusual behavior in the RTL, guiding the designer to the bug location.

In this report, we will focus on applying specification mining to address the last point above.

1.1 Motivation

RTL designs are increasing in complexity and are thus more prone to having subtle bugs that are not caught in regular verification flows. Typical techniques such as randomized-stimulus testing, directed testing, and fuzz testing have a difficult time catching bugs that require the RTL be put into a very specific state.

These subtle bugs are usually caught when performing chip emulation or FPGA prototyping when running realistic workloads on the RTL. Real workloads usually involve traces that are billion of cycles long, and are thus too slow to perform using RTL simulation which provides full design visibility. DESSERT[3] demonstrates a technique to capture full-visibility waveform traces from fast running FPGA simulation. Using DESSERT, an out-of-order RISC-V processor (BOOM[1]) is deterministically emulated on an FPGA with runtime assertion monitors while running the SPEC2017 benchmark suite. During the execution of several tests, synthesized assertions were violated which revealed there exists some subtle bugs in the core causing some benchmarks to fail.

Authors' address: Vighnesh Iyer, vighnesh.iyer@berkeley.edu; Donggyu Kim, dgkim@berkeley.edu, UC Berkeley.

While these assertions are useful for catching errors, they are very high-level and don't direct the designer to where a bug originated. As an example, the "`Pipeline has hung`" assertion (in BOOM) is generated with the following Chisel code:

```
// detect pipeline freezes and throw error
val idle_cycles = freechips.rocketchip.util.WideCounter(32)
when (rob.io.commit.valids.asUInt.orR ||
      csr.io.csr_stall ||
      io.rocc.busy ||
      reset.asBool) {
  idle_cycles := 0.U
}
assert (!(idle_cycles.value(13)), "Pipeline has hung.")
```

In words, this says, "If there is a good reason to stall the pipeline, reset `idle_cycles`, otherwise let it tick up to 13 before declaring something has gone wrong." This assertion does not give any insight as to what bad event happened, or when and where it happened. Since these assertions are thrown after billions of cycles it is possible that some μ-arch state was corrupted early in the simulation and only triggered this assertion much later during execution. DESSERT enables extracting waveforms for a variable number of cycles before the assertion triggers, but even with the waveform dumps in hand, the designer was unable to localize the bug. Our aim in developing this specification mining tool is to hunt out the locations of these trickly bugs in BOOM and fix them.

1.2 Hypothesis

Mature RTL designs (like BOOM) pass almost all tests run on them, including a full set of ISA tests, a boot of Linux, and real applications running on an OS. If a test fails on a mature design, we hypothesize that an *assumption* the designer made about the RTL was violated somewhere and at some time during the failing test execution, that was not violated on any successful test execution. These assumptions can include believing that a certain register cannot hold certain values or higher-level properties such as: "the memory system will respond to my request within 5 cycles".

We believe specification mining can be used to extract designer assumptions about the RTL design by mining fine-grained LTL properties on waveforms of successful test executions. These mined properties can be added to the RTL design as assertions and replaying the failing test should cause a violation of a mined property. These violations can be used to catch a faulty assumption *earlier and with greater locality* than the high-level assertions originally present in the design.

1.3 Problem Definition

Given:

- An RTL design driven with only one global clock
- A large set of VCD (value change dump) files produced when running a full suite of tests in an RTL simulator
- One or just a few failing tests in the suite characterized by a failed high-level assertion, hanging/global timeout, or a bad exit code

Produce:

- A set of mined LTL properties involving signals (combinational nets and registers) of the RTL design that aren't violated on any passing test
- A method of ranking the mined LTL properties
- A program that can check a VCD against the mined properties to find any violations for that execution trace

2 APPROACH

Our plan is to first create the products above with a small RTL design. We use `riscv-mini`[2], a simple 3-stage in-order RISC-V processor as the RTL we will use to test the specification mining engine. `riscv-mini` has real instruction and data caches and implements the RV32UI ISA subset which is capable of running the entire `riscv-tests`[5] test suite. The test suite contains a comprehensive set of ISA tests and benchmarks which will be used to generate the VCD files for mining.

2.1 Prior Work

Specification mining has been applied to both software and hardware verification.

2.2 High-Level Flow

The long-term plan is to leverage the spec mining engine and the prior work of DESSERT to provide an FPGA-acclererated debugging platform. We plan to take waveforms of successful test executions from RTL simulation and FPGA emulation of BOOM and feed the spec mining engine to infer LTL properties. The buggy RTL is instrumented by taking the LTL properties and converting them to property monitor FSMs which are stiched into the original design. The assertions are synthesized for FPGA emulation using the same methodology described in DESSERT, and the failing tests are executed. It is our hope that the failing tests will trigger failures in the mined properties before the failure becomes visible to the user (via a high-level assertion violation, hanging/timout, or a bad exit code).

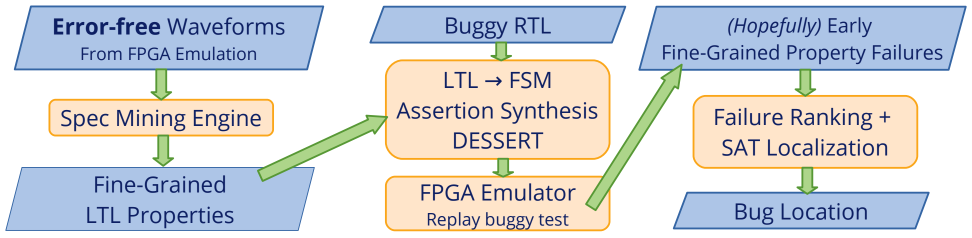

Fig. 1. The proposed tool flow to use specification mining and FPGA-accelerated simulation to pinpoint bug locations in for an RTL design .

The mined properties which were violated can be ranked based on the time of their violation and we can use the SAT localization techniques mentioned above to further pinpoint the bug location.

3 MODEL AND ALGORITHMS

In this section, we will specify the formal model for LTL specification mining, how we adapted LTL formulas for RTL, and the algorithms used in the spec mining engine.

3.1 Hardware Idioms in LTL

They can be used to describe many common idioms present in RTL. A few examples:

- There should eventually be a response (`resp`) after a request (`req`)
  $\mathbf{G}\,(\texttt{req} \to \mathbf{XF}\,\texttt{resp})$
- There should be a response in 2 cycles after a request
  $\mathbf{G}\,(\texttt{req} \to \mathbf{XX}\,\texttt{resp})$
- The ready/valid interface should keep `valid` high once it has been asserted until `ready` goes high
  $\mathbf{G}\,(\texttt{valid} \to \mathbf{X}\,(\texttt{valid}\,\mathbf{U}\,\texttt{ready}))$
- After a ready/valid transaction, the slave should be `ready` again within 2 cycles
  $\mathbf{G}\,((\texttt{valid} \wedge \texttt{ready}) \to (\mathbf{X}\,\texttt{ready} \vee \mathbf{XX}\,\texttt{ready}))$

3.1.1 LTL Templates. The formulas above can be templated by replacing the concrete signals (such as `ready`, `resp`, etc.) with variable placeholders. We consider 4 LTL templates for spec mining derived from Li's prior work[4]:

- Alternating: $a\,\mathbf{A}\,b$
- Until: $\mathbf{G}\,(a \to \mathbf{X}\,(a\,\mathbf{U}\,b))$
- Next: $\mathbf{G}\,(a \to \mathbf{X}\,b)$
- Eventual: $\mathbf{G}\,(a \to \mathbf{XF}\,b)$

where a and b are some boolean expressions derived from the signals in the RTL.

3.2 Adapting LTL Templates for RTL

RTL simulations produce a set of finite-length traces of bitvectors. We assume that these traces are sampled on the rising edge of a given clock signal; from now on we assume all traces contain the values of signals at discrete clock cycles. In contrast, LTL properties are defined over infinite-length traces of atomic propositions. We convert bitvector traces to traces of *delta events* which are treated as atomic propositions, and we employ notions of *falsifiability* and *support* to mine LTL properties on finite length traces.

Let a signal trace τ_i be a tuple of length T which contains the value of the signal i (a bitvector) at every discrete timestep of an RTL simulation. Let there be N signals in the RTL design: $i \in [0, N)$. We can convert τ_i to its delta trace $\tau_{\Delta i}$ as such:

$$\tau_{\Delta i}(t) = (\tau_i(t-1) \neq \tau_i(t)) \quad \forall t \in [1, T)$$

Note that $\tau_{\Delta i}$ is a tuple of length $T-1$ and is a trace of atomic propositions. Simply put, we convert a bitvector signal trace to a boolean trace which is 1 whenever the signal changes value, and 0 otherwise. We now consider the templates in section 3.1.1 in the context where $a, b = \tau_{\Delta i}, \tau_{\Delta j}, i \neq j$, for some i, j.

Here is a concrete example where k, q are bitvector signals in an RTL design where k is 1 bit wide and q is 10 bits wide:

$$\begin{aligned}
\tau_k &= (0, 1, 1, 1, 0) \\
\tau_q &= (0, 0, 200, 200, 300) \\
\tau_{\Delta k} &= (1, 0, 0, 1) \\
\tau_{\Delta q} &= (0, 1, 0, 1)
\end{aligned}$$

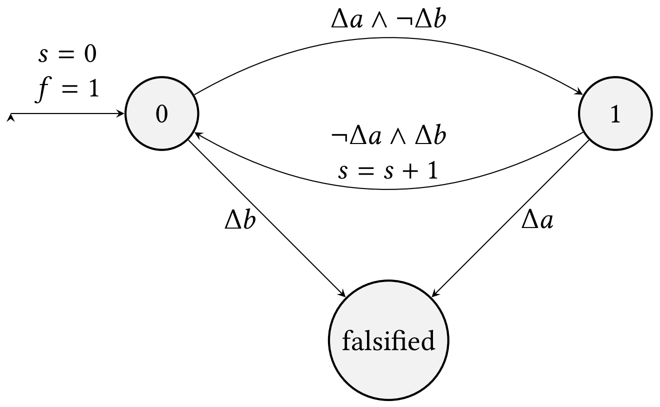
$\Delta a \wedge \neg\Delta b$
$s = 0$
$f = 1$
0
1
$\neg\Delta a \wedge \Delta b$
$s = s + 1$
Δb
Δa
falsified

Fig. 2. Automaton that mines the alternating LTL pattern

Note that even though k is a boolean signal and can already be treated as an atomic proposition, we only mine LTL properties on its *delta trace* and not the signal itself.

In RTL designs, it is usually the case that the density of transitions for a given signal is fairly low. Signal traces can be stored in a compressed format where we only record the timestep in which the signal value changes. Indeed, this is conveniently how traces are stored in a VCD (value change dump) file which is the input to the spec miner. We can extract sparsely represented delta traces from a VCD file as a tuple of timesteps where the signal transitions.

$$\tau_{\Delta k, compressed} = (0, 3)$$
$$\tau_{\Delta q, compressed} = (1, 3)$$

We want to consider the templates in section 3.1.1, so we can zip 2 compressed delta traces together that indicates which of the signals transitioned.

$$\tau_{\Delta k, \Delta q} = ((\Delta k, \neg\Delta q), (\neg\Delta k, \Delta q), (\Delta k, \Delta q))$$

This means that initially, k transitioned and q did not on a given timestep, then on a future timestep q transitioned and k did not, and on another future timestep both k and q transitioned. Note that in this data structure, it is impossible to have a tuple where both k and q *did not* transition. This data structure is perfect to construct automatons that mine LTL properties.

3.3 Mining with Automata

We construct FSMs for each of the LTL property templates that take zipped compressed delta traces as an input.

3.3.1 Falsifiability and Support. A given LTL property template is *falsifiable* over a zipped delta trace if its mining automaton reaches a state from which it can move to the *falsified* state in one step. We loosely define the *support* for an LTL property over a trace as the number of times the pattern "repeats". It will be made explicit in the FSMs for each LTL template as to when a pattern "repetition" occurs; we have yet to formalize this notion. We will abbreviate falsifiable as f and support as s.

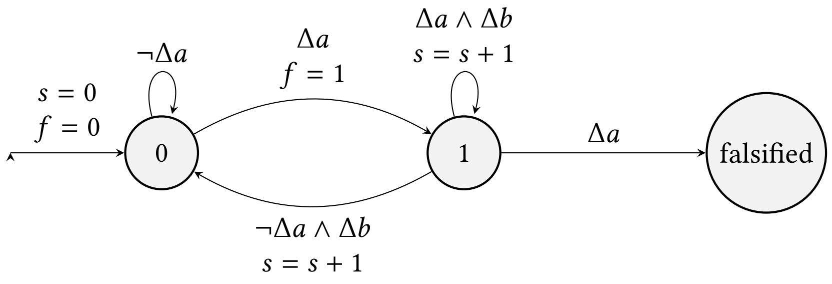

Fig. 3. Automaton that mines the until LTL pattern

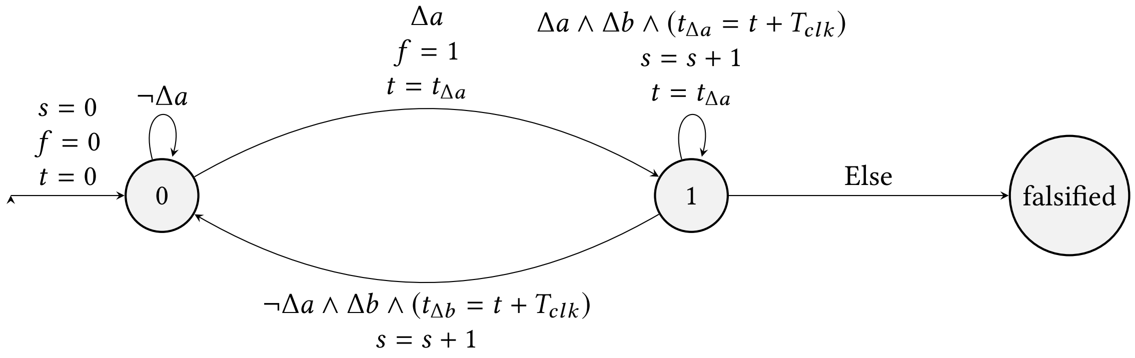
Δa
$f = 1$
$t = t_{\Delta a}$
$\Delta a \wedge \Delta b \wedge (t_{\Delta a} = t + T_{clk})$
$s = s + 1$
$t = t_{\Delta a}$
$s = 0$
$f = 0$
$t = 0$
$\neg\Delta a$
0
1
Else
falsified
$\neg\Delta a \wedge \Delta b \wedge (t_{\Delta b} = t + T_{clk})$
$s = s + 1$

Fig. 4. Automaton that mines the next LTL pattern

3.3.2 Alternating. The automaton in figure 2 mines the alternating pattern and advances support when a Δa then Δb sequence occurs. This pattern is violated if Δa and Δb are both true on the same timestep.

3.3.3 Until. The automaton in figure 3 mines the until pattern. If Δa is seen 2 times in a row without a Δb then this pattern is falsified. Note that if Δa never occurs, this pattern can't be falsifiable on that particular delta trace.

3.3.4 Next. Some hacking is needed with the next automaton in figure 4 to constrain the pattern to only be valid when Δb happens after 1 clock cycle after Δa. The variables $t_{\Delta a}$ and $t_{\Delta b}$ represent the time at which the a and b delta events occured on the transitions in which they are used. Let T_{clk} be the clock period, and t is a state variable to hold the transition time of a.

3.3.5 Eventual. Note that the eventual pattern is never falsifiable on a finite-length trace as seen in figure 5. In addition to simply keeping track of the *support* we also maintain a set of times it took to get to Δb from Δa. This allows us to empirically find cycle limits for the eventual pattern so it can reasonably be applied to finite-length traces.

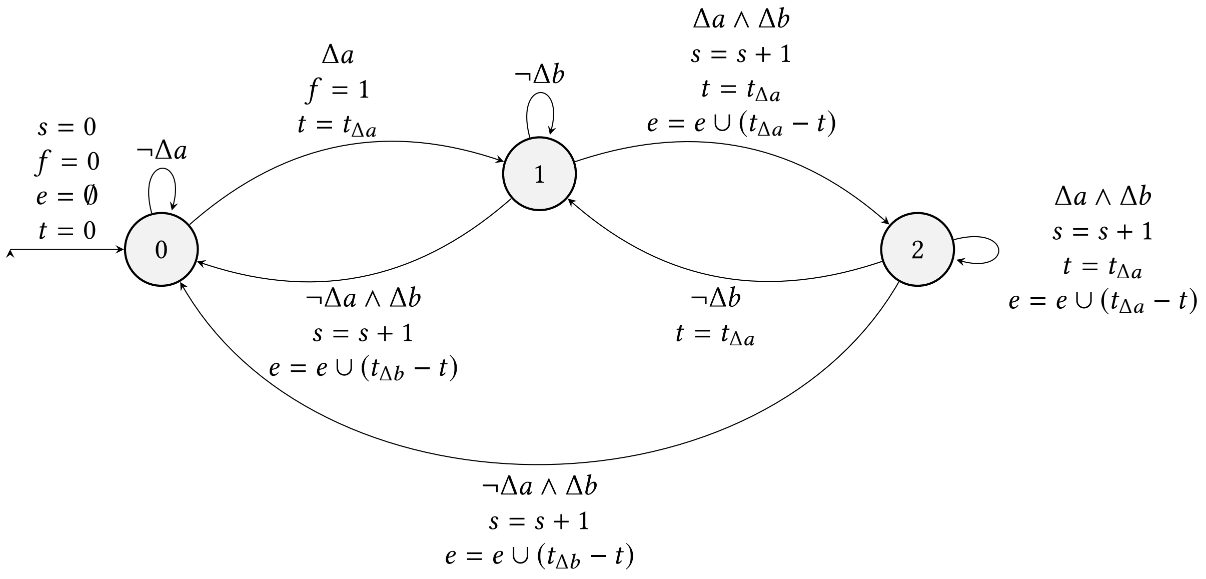
$s = 0$
$f = 0$
$e = \emptyset$
$t = 0$
$\neg\Delta a$
Δa
$f = 1$
$t = t_{\Delta a}$
$\neg\Delta b$
$\Delta a \wedge \Delta b$
$s = s + 1$
$t = t_{\Delta a}$
$e = e \cup (t_{\Delta a} - t)$
0
1
2
$\Delta a \wedge \Delta b$
$s = s + 1$
$t = t_{\Delta a}$
$e = e \cup (t_{\Delta a} - t)$
$\neg\Delta a \wedge \Delta b$
$s = s + 1$
$e = e \cup (t_{\Delta b} - t)$
$\neg\Delta b$
$t = t_{\Delta a}$
$\neg\Delta a \wedge \Delta b$
$s = s + 1$
$e = e \cup (t_{\Delta b} - t)$

Fig. 5. Automaton that mines the eventual LTL pattern

3.4 Naive Mining Algorithm

The algorithm we used to mine LTL properties on VCD files is very naive, but is still relatively fast due to the sparsity of the compressed delta traces and the limited set of signals that can fill the roles of a and b in the LTL templates.

Algorithm 1 Naive Spec Miner

```
1: procedure MINER($[\tau_{\Delta 1,c}, \dots, \tau_{\Delta N,c}], M$)          ▷ $M$ is the RTL module hierarchy
2:     $P \leftarrow \emptyset$
3:     for $[\tau_m] \leftarrow$ MODULARIZE($[\tau_{\Delta 1,\dots,N,c}]$, M) do          ▷ Split traces by module
4:         $[\tau_{m,trim}] \leftarrow$ TRIM($[\tau_m]$)          ▷ Only keep signals with width $\leq$ 5 bits
5:         if $\neg$ ISLEAF(m) then          ▷ Only consider port signals except for leaf modules
6:             $[\tau_{m,trim}] \leftarrow$ STRIPINTERNAL($[\tau_{m,trim}]$)
7:         for $(\tau_{\Delta i}, \tau_{\Delta j}) \leftarrow$ PERMUTATIONS($[\tau_{m,trim}]$, 2) do
8:             for Miner $\leftarrow$ [A, N, U, E] do
9:                 $P \leftarrow P \cup$ MINER($\tau_{\Delta i}, \tau_{\Delta j}$)
10:    return $P$
```

The algorithm begins by splitting delta traces along module boundaries via DFS and furthermore constrains the signals under consideration to be below 5 bits in width and in the module's port list (except for leaf modules). Usually this technique keeps the final length of $\tau_{m,trim}$ to below 10 which makes iterating over their permutations ($O(n!)$) have reasonable runtime. We maintain a set of mined properties that is updated as the miner proceeds along different modules.

Table 1. Module hierarchy, number of signals under consideration for mining for `riscv-mini`, and mined properties for 2 ISA tests

Module	Number of Signals	Props Mined on add	Props Mined on sw
TOP	4	45	45
TOP.Tile	15	791	1529
TOP.Tile.icache	18	1113	1113
TOP.Tile.dcache	4	43	1965
TOP.Tile.core	12	565	973
TOP.Tile.core.dpath	23	2149	3308
TOP.Tile.core.dpath.regFile	4	43	43
TOP.Tile.core.dpath.immGen	1	0	0
TOP.Tile.core.dpath.csr	16	849	1080
TOP.Tile.core.dpath.brCond	8	261	261
TOP.Tile.core.dpath.alu	3	24	24
TOP.Tile.core.ctrl	9	333	403
TOP.Tile.arb	11	430	683

3.5 Merging Properties and Checking

For each VCD we add a property to the mined set P if the miner reports it as *falsifiable* and is either *falsified* or has positive *support*. Properties mined from each VCD are the merged together into a final property set by discarding any properties that were *falsified* at any time and aggregating the *support* for any properties that were never falsified.

The final property set can be fed into a checker which will run the the miner for each property on a VCD trace and verify that none of them are falsified.

3.6 Challenges

Most of the challenges for this project were engineering-related. We had to hand construct automaton for each of the LTL templates and check them against many test inputs to verify they accurately captured the property. The automaton had to work with compressed delta trace inputs which complicates some of the checks for the next and until patterns.

4 RESULTS

We apply our specification mining engine to `riscv-mini` which has the module hierarchy and number of signals in table 1.

We also record the number of properties mined for each module for 2 different tests that stress different parts of the processor. The `add` test does not stress the memory system and data cache as much as the `sw` test and this can be seen in the modules where the discrepency between the number of mined properties is highest.

We mined properties on all VCD traces produced by the riscv-mini ISA test suite and merged them together. When merging properties, many of them are falsified and we are left with only **3251** properties at the end. Here is an exerpt of the mined properties with the highest support:

```
Until TOP.Tile.core.ctrl.io_A_sel -> TOP.Tile.core.dpath.io_ctrl_imm_sel, support: 6113
Next TOP.Tile.core.dpath.brCond.eq -> TOP.Tile.core.dpath.brCond.neq, support: 5457
Eventual TOP.Tile.core.dpath.io_ctrl_imm_sel -> TOP.Tile.core.dpath.regFile_io_wen, support: 4466
Until TOP.Tile.icache.io_cpu_req_valid -> TOP.Tile.icache.is_idle, support: 4359
Until TOP.Tile.icache.io_cpu_req_valid -> TOP.Tile.icache.is_read, support: 4359
Eventual TOP.Tile.icache.io_cpu_req_valid -> TOP.Tile.icache.is_read, support: 4336
Until TOP.Tile.icache.io_cpu_req_valid -> TOP.Tile.core.dpath.csr_io_stall, support: 4318
```

Some of these properties are interesting, but ultimately many of them are redundant or obvious (such as mining next on a shift register). Some of the properties that a human designer would identify as valuable have lower levels of support, such as:

```
Next TOP.Tile.core.dpath.csr.isEcall -> TOP.Tile.dcache_io_cpu_abort, support: 45
```

The reader may notice that these properties appear to be mined across module boundaries, contrary to the algorithm as stated. This is because many signals are aliased across module boundaries and the mined properties printout just shows one alias of the signal.

4.1 Bug Localization

We introduce a bug in Control.scala that mimics a copy/paste bug:

```
  io.csr_cmd := ctrlSignals(11)
- io.illegal := ctrlSignals(12)
+ io.illegal := ctrlSignals(11)
```

This bug caused the sub ISA test to hang, so after letting it time out, the failing VCD was checked against the mined properties.

```
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal -> TOP.Tile.icache.io_cpu_req_valid
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal -> TOP.Tile.icache.io_cpu_resp_valid
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal -> TOP.Tile.core.ctrl.io_A_sel
```

The violated properties directly point to something wrong with the io_illegal signal, and effectively localize the bug.

We introduced another bug in Cache.scala:

```
- hit := v(idx_reg) && rmeta.tag === tag_reg
+ hit := v(idx_reg) && rmeta.tag =/= tag_reg
```

This bug does not affect most ISA tests but a multiply benchmark failed via hanging. After checking the VCD against the mined properties, we found these violations ranked by time of violation:

```
ERROR on property Until TOP.Tile.arb_io_dcache_r_ready -> TOP.Tile.dcache.hit with support 24 at time 418
ERROR on property Until TOP.Tile.dcache_io_nasti_r_valid -> TOP.Tile.dcache.hit with support 24 at time 418
ERROR on property Until TOP.Tile.dcache.is_alloc -> TOP.Tile.dcache.hit with support 24 at time 418
ERROR on property Until TOP.Tile.arb.io_dcache_ar_ready -> TOP.Tile.arb_io_nasti_r_ready with support 4173 at
↪ time 640
```

Yet again, the violated properties point to something wrong with the hit signal and localize the bug.

4.2 Future Work

There are quite a few things to improve:

- Add additional LTL templates that mine on boolean signal traces and delta event traces
- Prune away mined properties that are statically provable (such as shift registers)
- Develop a faster VCD parser to allow mining over larger VCD files
- Generalize the miner to take any LTL formula and automatically construct an automaton to mine the template
- Construct traces of atomic propositions from user-defined events (such as ready/valid transactions firing)
- Introduce more subtle bugs in `riscv-mini` and check whether they can be caught

5 COURSE-RELATED STUFF

Donggyu provided the inspiration and motivation for this project and gave suggestions on what types of bugs to introduce in `riscv-mini`. Vighnesh implemented the spec mining engine and evaluated it on `riscv-mini`.

This project makes use of LTL specifications and conversion of an LTL formula to an automaton. We thank Prof. Seshia for introducing a wide variety of topics in this course and for his advice on our project.

Our project is available on Github.

REFERENCES

[1] Christopher Celio, David A. Patterson, and Krste Asanovic. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report. https://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[2] Donggyu Kim. 2019. Simple RISC-V 3-stage Pipeline in Chisel. https://github.com/ucb-bar/riscv-mini

[3] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2018. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL).* IEEE. https://doi.org/10.1109/fpl.2018.00021

[4] Wenchao Li. 2014. *Specification Mining: New Formalisms, Algorithms and Applications.* Technical Report. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-20.html

[5] Andrew Waterman. 2019. Unit Tests for RISC-V Processors. https://github.com/riscv/riscv-tests