Formal verification of Macro-op cache for Arm Cortex-A77, and its successor CPU

Vaibhav Agrawal
Vaibhav.Agrawal@arm.com
Arm Inc
5707 Southwest Pkwy, Ste 100
Austin, TX 78735

Abstract: **For Arm Cortex-A77 and its successor core, which are high end Arm A-class CPUs, formal verification was strategically introduced to verify a disruptive and critical new design feature, "macro-op cache", which was added to the instruction fetch unit. In this paper, we describe the complexity of the verification problem, and how formal was applied effectively to augment existing verification techniques to find bugs in the bring up phase and late in the project. The paper talks about methods used to reduce verification complexity for formal, along with effective ways to write checkers and constraints to make them more efficient for formal verification. Data on the bugs is also presented.**

I. Context and Motivation

Background: Cortex-A77, a high-performance A-class CPU for mobile and client markets was the first ARM CPU ever built with a macro-op cache at the front end of the pipeline to help improve performance of the instruction fetch and decode functionality. Here is a very high-level block diagram of the DUT:

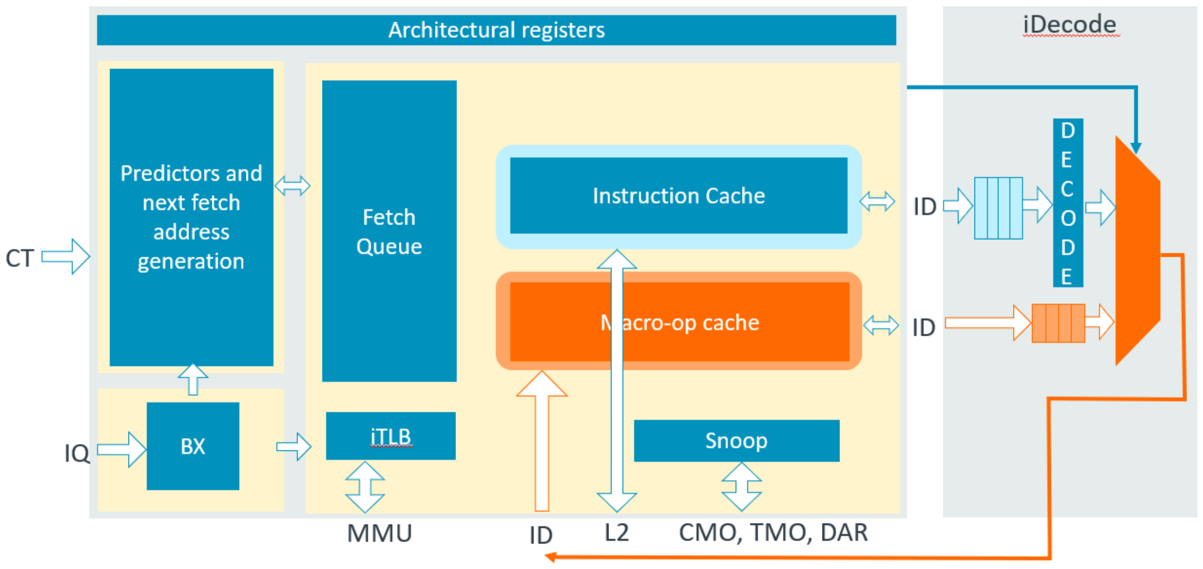


Fig. 1: Instruction fetch unit block diagram

Instruction fetch unit is a control intensive block, which contains RAMs for storing both tag and data, a queue to store transactions received from branch predictors, and an instruction TLB to store address translations. Even before the macro-op cache was added to store macro-ops, there were several pipelines already present in fetch RTL. A macro-op cache was added which would be filled from decoded macro-ops received from decode unit. This resulted in additional complexity including addition of new RAMs for the new cache. Two additional pipelines were added, one each for tag and data for macro-op cache. Also, more overlapped modes and replays had to be introduced between the overall increased number of pipelines in the block, causing increased potential for bugs to creep in. On top of these changes, an dditional requirement of “coherence” between instruction cache and macro-op cache was also added for Cortex-A77’s successor core.

Summary: In this paper, we present how model checking using Cadence’s Jaspergold FPV app was successfully applied to enable all the different stimuli in formal testbench, and then to verify “macro-op cache - including coherence”, which was a highly disruptive change to existing RTL. Macro-op cache was important for additional performance uplift and was on the critical path in terms of schedule. **We started this work knowing fully that we would not be able to get full proofs for properties due to the complexity of the design. The intent was to:**

1) Completement simulation-based verification (**strengths of formal are orthogonal to simulation**)
2) Add efficiency to the overall verification process (**find bugs sooner**), and
3) Ensure quality of the design (**find critical bugs before the first customer delivery milestone**)

II. FORMAL TESTBENCH ARCHTECTURE

The design is logically split into 3 components: the branch execution unit, the branch prediction logic, and the fetch logic. The fetch logic consists of both instruction and macro-op caches, the instruction TLB, and several other micro-architectural structures for performance. Based on the functional changes, it was decided that application of formal would be most beneficial when done on the fetch logic alone.

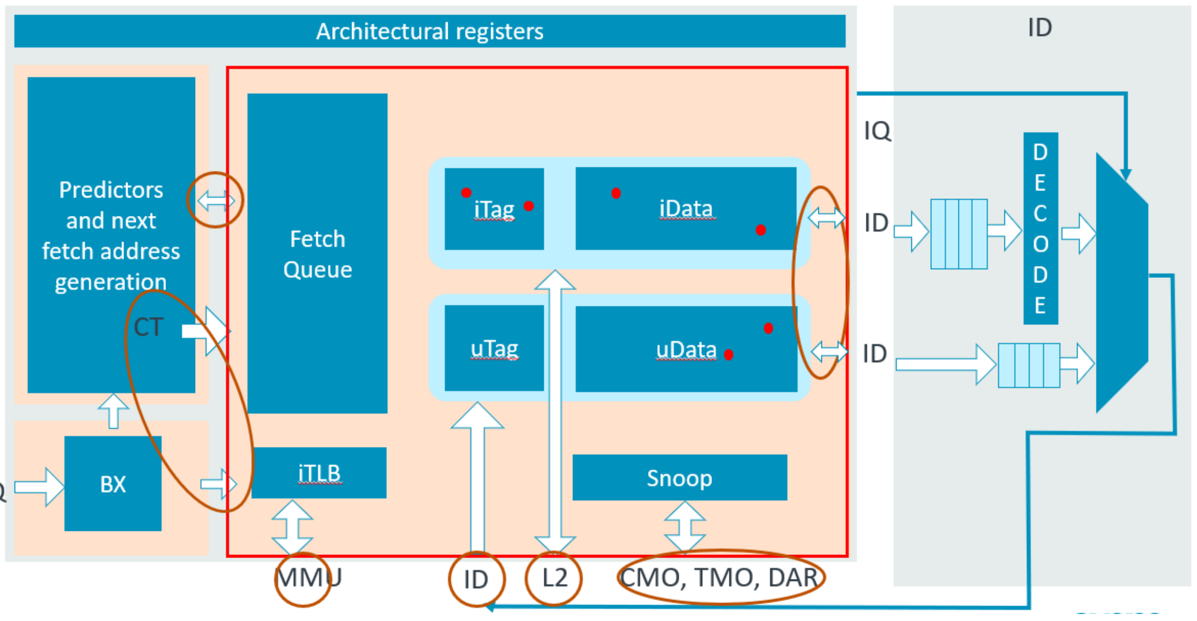


Fig. 2: Instruction fetch unit formal testbench boundary and interfaces

All the interfaces surrounding the fetch logic were modeled and relevant constraints were added. Parity error injection was also enabled. In addition to constraints for fetch logic, all interfaces also contained interface assertions on the outputs of fetch logic. Even though these assertions did not require end to end tracking, they still ranged from simple combinational properties to sometimes complex sequential relations between signals.

Any end to end formal checkers were added in separate files, which were not enabled in simulations. The reason for this will become clear in the section below on end to end assertions, but the high-level reason is that use of pseudo-constants makes these assertions un-usable in simulations.

The most complicated interface amongst all the above was the one between branch predictors and the fetch logic. All interfaces whenever possible, were also enabled in simulation environment to validate the constraints.

III. TACKLING FORMAL COMPLEXITY

To make formal verification more effective, it is critical to reduce the complexity of the overall synthesizable design (including both RTL and formal testbench). Effectiveness of formal verification can be improved by **"Improving formal reachability of state space, both in terms of time and sequential depth"**. This was achieved in our case by applying several techniques as enumerated in the Table [1]:

	Technique	Effort	Return
1	Reduce table/hash sizes (caches/iTLB)	High	Extremely High
2	Reduce mop size	Low	Moderate
3	Reset abstraction (preloading in formal)	High	Extremely High
4	Input VA/PA space reduction	Low	High
5	Input data space reduction	Low	Low
6	Specific mode with TLB accesses always a hit	Low	Extremely High

Table 1: Complexity reduction techniques for our testbench

1. Reducing table/hash sizes (including caches and TLB) using tool assisted mutations: Any non-FIFO, but hash like RTL structures, where each entry has a fixed identifier (a key) and data (value) can have entries be allocated / deallocated in random order (e.g. a TLB, or a cache. For such structures, bugs are often found around allocation / deallocation of an entry, and around the replacement policy, and 99% of the bugs are generic in nature and are not dependent on a specific entry. Formal can benefit greatly if the size of such structures can be reduced. An entry count of 2 is usually sufficient for formal to find interesting bugs. Also, the victim selection policy can be removed, and formal can be allowed to (almost) randomly choose the victim.

Hash structure size reduction can be achieved either by parameter manipulations (if available), or by using tool assisted mutations (available in **Cadence Jaspergold**), which can be implemented by understanding the microarchitecture of the design around these structures, and then performing surgical operations as follows:

1) Randomize victim selection to remove the logic of the victim selection policy:
```
stopat -env {u_fp.u_itag_ctl.victim_ent_id_c2[3:0]}
```
2) Restrict victim selection to effectively reduce the size:
```
assume -env {u_fp.u_itag_ctl.victim_ent_id_c2 < 4’h2}
```
3) Legalize victim selection, e.g. entry hit in last cycle cannot be victimized (c3 follows c2 in the pipeline):
```
assume -env {u_fp.u_itag_ctl.victim_ent_id_c2 != u_fp.u_itag_ctl.hit_ent_id_c3}
```

The process is slightly different for RAMs (memories). Most designs instantiate RAMs in a separate hierarchy, making it easier to black-box the instance of the RAM module, and then replacing these instances with synthesizable CAM (content addressable memory) models, such that only a limited number of cache-sets can be tracked in the abstraction model. The CAM models are CAM-ed against the set (address) itself. In case the RAMs are not a separate instance by themselves, adding ifdefs (for formal) around the memories, followed by binding the specialized CAM based models, can achieve the same result.

A complete description of how a CAM based RAM abstraction works is out of the scope of this paper. The basic idea is that for any sets (addresses) that are tracked, the model behaves like a normal memory, but for any untracked sets, data read out from the model is randomized.

Impact: Formal found a few bugs around the replacement policy of an entry in one of the design structures (because RTL has completely randomized the replacement policy, hence allowing formal to do whatever it wanted). After the design was fixed, an appropriate constraint was added around the replacement policy so that formal was not completely free in picking the victim. This bug could not have been found without the abstractions mentioned above.

RAM abstraction for caches was essential for formal to get to reasonable sequential depth because of the sheer number of memory elements in the RAMs.

2. Reducing macro-op size: While the v8 ISA ARM instructions are only 32 bits wide, the decoded macro-ops are much wider. However, **there is no macro-op transformation done in the instruction fetch unit**. This implied that for formal, the size of macro-ops stored in the RAM models could be much smaller. The control portion of the macro-op, which is critical to correct functionality, had a small number of bits, and was left untouched for formal, but the number of bits in the data portion was reduced to 2. This is depicted in **Figure [3]** below. This allowed for the number of bits in the RAM abstraction model for macro-op data RAMs to be much smaller, hence reducing the complexity for formal. On the read path out of the data RAMs, the abstracted-out bits were tied off to all-zeroes.

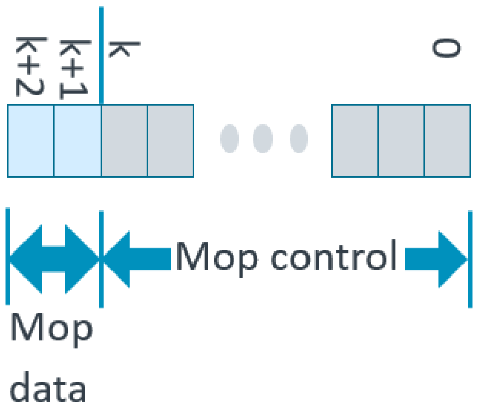


Fig. 3: Macro-op size abstraction

Note that such an optimization could not be applied to the instruction cache due to predecoding of the instructions.

Impact: The impact of this was not significant (about 20% improvement in terms of the time it took to hit some interesting covers and fails).

3. Reducing sequential depth using IVAs (Initial Value Abstractions, or preloading): The macro-op cache designer was very interested to quickly verify the core functionality of the macro-op cache, even though the rest of the fetch RTL wasn't ready early enough. In the minimum, this required the stimulus to be setup such that addresses from predictors could always hit the macro-op cache. This required that the macro-op cache to be preloaded. However, the simulation unit testbench tracking took a while to get to the point where addresses could be generated in way that they would always hit the macro-op cache.

In came formal! The designer was able to point to a control signal in the design which would always have a certain set of values when the addresses hit in macro-op cache. Formal was able to constrain this signal, and then back propagate it backwards so that addresses generated by the tool would always hit in the macro-op cache. In addition, the macro-op cache was pre-loaded suing reset abstractions. It took a few weeks to understand and code all the constraints around the control and data portion of macro-ops as stored in the macro-op cache, to ensure that any pre-loading was legal. This was very useful in reducing the overall sequential depth of interesting corner cases.

E.g., extending the scenario presented in the "reducing table/hash sizes" section above, reset abstraction can be done in **Cadence Jaspergold** as following:

1) Allow design structures to be valid at reset, ie, effectively remove the "if (reset)" clause from flops:
```
// allow entries to be marked valid right out of reset
abstract -init_val {u_fp.u_itag_ctl.vld_ent[15:0]}

// allow entry contents to be non-zero right out of reset
for {set i 0} {$i < 16} {incr i} {
    abstract -init_val u_fp.u_itag_ctl.ent_addr_c2\[$i\]
}
```

2) Add a constraint valid only on the first cycle, to keep the size of the structure limited to 2:
```
assume –bound 1 {u_fp.u_itag_ctl.vld[15:0] == 16'h0003}
```
3) Add any additional constraints to ensure that any valid preloaded addr_c2 values are legal w.r.t. each other right out of reset:
```
for {set i 0} {$i < 15} {incr i} {
  for {set j (i+1)} {$j < 16} {incr i} {
     assume -bound 1 {u_fp.u_itag_ctl.vld_ent[$i] & u_fp.u_itag_ctl.vld_ent[$j]
     |->
     u_fp.u_itag_ctl.ent_addr_c2[$i] != u_fp.u_itag_ctl.ent_addr_c2[$j]}
  }
}
```

While the actual abstract commands shown above have to be added as tcl commands, the constraints themselves (assume -bound 1) can also be added as regular SVAs in verilog files.

Impact: IVAs were necessary and the most effective technique that we used to enable formal to reach deeper design states in a reasonable amount of time to find interesting bugs. Without IVAs, we would not have been able to find even 20% of the bugs. With both instruction cache and macro-op cache preloaded, and with some other design structures preloaded as well, it was possible for the design to be brought up in deep interesting states right out of reset.

4. Input VA/PA state space reduction: All interface properties, and design coded assertions were not aware of tracked sets in the RAM abstractions for caches (unlike formal end to end checks), and would therefore generate false fails as these abstractions would prevent Read - Write consistency for addresses that fell in untracked sets. As a result, the addresses generated by branch predictor / fetch interface were limited to only those addresses which were tracked in the caches. This required care to not over-constrain away meaningful stimulus, e.g., scenarios for VA crossing a cache line boundary, a page boundary, and to allow capacity evictions from caches. Therefore, the virtual addresses generated were constrained to lie in the tracked sets of the caches. This had a positive side effect of reducing the time it was taking to reach interesting covers. Also, the last and first set in the caches were always tracked to allow page crossings.

Another interesting technique which yielded high benefits was to reduce the VA to PA mapping. I.e., instead of allowing all possible mappings from VA to PA (through the TLB), the PA was constrained to be a function of the VA, so that formal tool would not have to try all possible mappings.

Impact: VA space reduction led to about 50% improvement in runtimes for interesting covers. VA to PA mapping space reduction led to more than 100% runtime improvement for some relevant scenarios (not all of them though).

5. Input data space reduction (data in this case is instructions or macro-ops): The number of instructions was limited (for both preloading and cache fills) to a small subset of the legal instruction set. We chose to have a set of approximately 10 instructions, including static branches of different types. A negative side effect of limiting instructions to a smaller subset was, that we could not verify some assertions added by the designer which were intended to check the pre-decode functionality, but this was an acceptable trade off.

As mentioned earlier, the number of bits in the data portion of macro-ops were reduced to 2, which automatically meant that the maximum number of different macro-ops that could be preloaded (or provided as part of a fill) was limited to 4, and this was enough for our purpose.

Impact: Coding a synthesizable constraint to capture the entire legal set of instructions would have taken additional work, whereas, it was easier to pick a small subset of instructions. Also, the intuition is that the much bulkier constraint capturing the entire legal instruction set would have made the tool's job more difficult, but we don't have any experimental data to support this claim.

6. A specific mode with TLB access always a hit: TLB misses were not required to catch a lot of these bugs, and formal was doing a lot of thrashing around TLB misses. Therefore, a special mode was added under which all TLB accesses were hits, which was used as the default mode, while TLB misses were allowed only under a special mode. This was quite easy to implement using some simple constraints and cut points, as follows:

a) Under a pseudo-constant, all the TLB entries were black-boxed, along with logic for determining TLB hit/miss. The TLB hit signal was tied to a high. A **pseudo-constant** is a signal that is constrained to remain stable throughout any trace and is free to take up different values across traces – hence the prefix "pseudo".

b) All TLB faults were disabled under the same pseudo-constant.
c) The VA to PA translation was constrained to honor the VA to PA mapping that was mentioned earlier. There is nothing sacrosanct about choosing a specific mapping. To start with, a flat mapping can suffice.

Impact: This had a huge impact on reachability (for some of the covers, time was reduced to less than a quarter of what it was before). However, the downside was that we now had some unreachable logic and properties due to TLB miss being disabled, but this was acceptable. To mitigate the risk here, we also did separate runs with the TLB misses enabled.

IV. FORMAL FRIENDLY CHECKER DESIGN BASICS

Often, end to end formal verification checkers are written in a way that is not friendly for formal, and bad checker design increases formal complexity. Most of the time, an elaborate tracking and score-boarding is not needed for formal checks. There is usually an efficient way to do the same checking with much lower impact to overall complexity [1].

A general rule of thumb for a formal testbench (as a base line) is to minimize the number of flops, including both the design and the testbench. The previous section enumerates how this was achieved for the design. But it is equally important that the testbench doesn't contribute (any more than needed) to formal complexity. And I have seen very often that this cardinal rule is violated by unsuspecting formal engineers, especially those who come from simulation background.

For instruction fetch unit, it is important to have an end to end check that the instructions (or macro-ops) being delivered to decode unit are correct. In simulation-based testing, this is achieved at a high level by having a cache model in the testbench and keeping track of an address space map for any addresses that are fed from the branch predictors. If a checker design strategy similar to simulation is used in the formal testbench, then that will end up more than doubling the state space complexity for formal because of all the tracking and score-boarding that will need to be done. For formal, the foundational thought process needs to be:

a) How can we use the design itself to do the checks, without doing elaborate tracking in the testbench?
b) Instead of (or in addition to) end to end checks, can we write several smaller microarchitectural checks?

With (a) above in mind, an ordering checker was designed using **pseudo-constants**, **tracked addresses**, **tracked data**, and **coloring technique**.

Pseudo-constants: These are variables (single or multi-bit) which are constrained to be a constant throughout a given trace, e.g. by adding a $stable(variable) constraint. Note that this still gives formal the freedom to assign different values across traces, but in any given trace, the value will remain constant.

Tracked address: A tracked address is a pseudo-constant (potentially constrained to lie within an allowed set of values across traces). In our case, these were virtual addresses, and also the corresponding physical addresses were based on a constrained VA to PA map.

Tracked data: A pseudo-constant data value in a design structure, potentially constrained to lie within an allowed set of values across. In our case, this was the tracked instruction (or tracked macro-op).

Coloring technique: When a design structure with a specific attribute (like an ID, or an address) is constrained to have a specific value, then the ID or address is said to be "colored" with that value. In our case, we tied off specific addresses to specific instructions or macro-ops.

V. END TO END ORDERING CHECKER DESIGN

Towards checking correct instruction (or macro-op) delivery to decode, an ordering checker can be designed as follows:

i) Have 2 tracked Virtual addresses, VA1, and VA2.
ii) Constrain the predictor / fetch interface so that VA1 must be followed in program order by VA2. This implies, if VA1 is a predicted taken branch, then VA2 must be its target.
iii) A coloring constraint was added for preloading, so that if a cache line containing VA1 is present in the **instruction cache**, then it must be **colored** by a specific tracked instruction. A similar constraint was added for VA2. Let these colored instructions be **tracked-instr(VA1)** and **tracked-instr(VA2)** respectively**.** Also, these colored instructions were constrained to not be present at any other addresses.
iv) A constrained was also added that **tracked-instr(VA1) != tracked-instr(VA2).**

v) A coloring constraint was added for preloading, so that if a cache line corresponding to VA1 is present in the **macro-op cache**, then it must be colored by a specific tracked macro-op. A similar constraint was added for VA2. Let these colored macro-ops be **tracked-macro-op(VA1)** and **tracked-macro-op(VA2)** respectively**.** Also, these colored macro-ops were constrained to not be present at any other addresses.
vi) A constraint was also added that **tracked-macro-op(VA1) != tracked-macro-op(VA2).**
vii) A coloring constrained similar to (iii) above was also added for cache line fills.
viii) A coloring constraint similar to (v) above was also added for macro-op fills.
ix) Finally, the ordering check can be: If **tracked-instr(VA1)** or **tracked-macro-op(VA1)** is being delivered to decode, then it must be followed (in program order) by **tracked-instr(VA2)** or **tracked-macro-op(VA2).**
x) Enable the checker, and related constraints under an oracle (explanation coming up below).

As the macro-op data portion was reduced to only 2 bits, we were able to use the value 2'b01 for **tracked-macro-op(VA1)** and 2'b10 for **tracked-macro-op(VA2).** All the other addresses were colored with 2'b00. A few checkers required 3 tracked addresses, for which 2'b11 was also used. In the instruction cache, as the number of instructions were constrained to a small set of real instructions, **tracked-instr(VA1)** and **tracked-instr(VA2)** were allowed to be picked out of this small set, and for all the untracked addresses, any of the remaining unpicked instructions from the same set were used.

Adding checker specific constraints: Several checks were designed using similar, but with slight variations of coloring scheme depending on the intent of the checker. In formal language, an **oracle** is a variable that is used like a coin flip (may or may not be a pseudo-constant), the value of which helps the tool make a decision. As conflicting constraints were needed across different checkers, this was managed by qualifying both constraints and checks with pseudo-constants (oracles). In the example below, the checks chk1 and chk2 have conflicting constraints (assume_3 and assume_4 respectively), but this can be managed using oracles oracle_enable_chk1 and oracle_enable_ch2:

```
  assume_1:  ##1  $stable  (oracle_enable_chk1);  //declare  pseudo-constant  oracle  for
chk1
  assume_2:  ##1  $stable  (oracle_enable_chk2);  //declare  pseudo-constant  oracle  for
chk2

  assume_3: oracle_enable_chk1 |→ input_a == 1'b1; // conditional constraint for chk1
  assume_4: oracle_enable_chk2 |→ input_a == 1'b0; // conditional constraint for chk2

  assert: oracle_enable_chk1 |→ chk1;
  assert: oracle_enable_chk2 |→ chk2;
```

Even though the consequents of assume_3 and assume_4 are conflicting with each other, the presence of independent oracles as preconditions allows formal to proceed with checking both chk1 and chk2.

VI. MICROARCHITECTURAL COHERENCE CHECKING BETWEEN L0 AND L1 CACHES

The coherence requirement for Cortex-A77's successor CPU's instruction cache did not need a full coherence protocol implementation, and ended up being an "inclusivity requirement", implying that that if a macro-op for a particular address is in the macro-op cache, then the corresponding instruction for that address must also be present in the instruction cache. This inclusivity check can also be implemented using coloring techniques:

i) Have a tracked Virtual addresses, VA1
ii) A coloring constraint was added for preloading, so that if a cache line corresponding to VA1 is present in the **instruction cache**, then it must be **colored** by a specific tracked instruction. Let this instruction be **tracked-instr(VA1).** Also, this colored instruction cannot be present at any other address.
iii) A coloring constraint was added for preloading, so that if a cache line corresponding to VA1 is present in the **macro-op cache**, then it must be colored by a specific tracked macro-op. Let this macro-op be **tracked-macro-op(VA1).** Also, this colored macro-op cannot be present at any other addresses.
iv) A constraint was added for preloading, so that a cache line corresponding to VA1 can be present in macro-op cache only if the corresponding line is also present in the instruction cache. This enforces the inclusivity requirement at reset.

v) A coloring constrained similar to (ii) above was also added for cache line fills.
vi) A coloring constraint similar to (iii) above was also added for macro-op fills.
vii) The checker can be: If **tracked-instr(VA1)** is not present in the instruction cache at any time, then **tracked-macro-op(VA1)** must not be present in the macro-op cache.
viii) Enable the checker, and related constraints under an oracle.

VII. ENSURING FORWARD PROGRESS (CHECKING FOR HANGS)

Two separate approaches were used to find forward progress related issues in the design:

1. Liveness properties: One of the strengths of formal is that it can incentivize the stimulus to check for hangs when liveness assertions are used, which is not possible in simulation based approach. Liveness assertions check that "eventually, something must happen".

```
state_of_interest |→ s_eventually(forward_progress_seen)
assign forward_progress_seen = real_progress_seen || flush;
```

Adding `flush` conditions to the consequent, when determining the logic for `forward_progress_seen` allows the tool to exclude checking for cases when the design is "off the hook" because things were "restarted", hence preventing false fails. Liveness assertions requires external dependencies to be explicitly resolved, and this can be achieved by adding "fairness constraints", which are "liveness properties" (requirements) on adjoining blocks.

Here is a high-level English description of liveness checks coded for fetch logic:
i) Instruction / macro-op must be delivered for an un-cancelled address granule
ii) On a breakpoint, instruction or macro-op before the breakpoint must eventually be delivered
iii) An abort detected in fetch (or communicated from predictors) must eventually be passed on to decode
iv) Forward progress on parity errors in RAMs (including stuck-at faults)
v) A snoop request from L2 must eventually be completed
vi) A DSB sync request must eventually be completed.

Here is an outline for the breakpoint checker in (ii) above: Use 2 tracked VAs, such that VA2 follows VA1 in program order, and the breakpoint is on VA2. Data coloring is performed for both "instructions and macro-ops" for these tracked addresses. All the constraints can be added under the precondition of an oracle, which is a pseudo-constant. Then, the liveness check can be coded to say that "eventually the colored instruction or macro-op corresponding to VA1 must be delivered to decode". For completeness of the breakpoint check, a safety assertion will also need to be added which says that the tracked instruction or tracked macro-op corresponding to VA2 must not be delivered to decode, due to the breakpoint being on VA2.

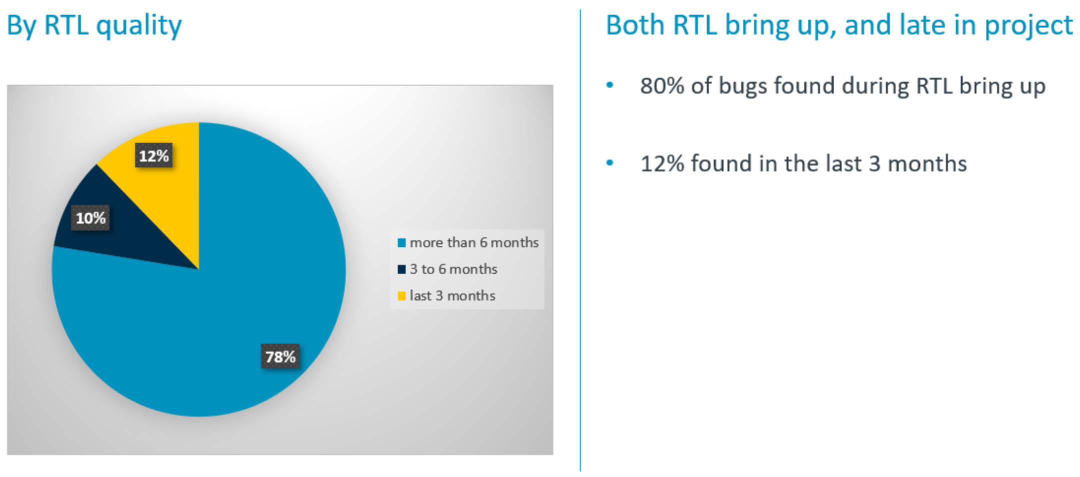
2. Smart counter-based approach: To be used if liveness properties do not converge. Create a counter that measures "lack of forward progress" and use a safety property which fails when the counter reaches a threshold. The counter is called "a smart counter" because it has configurable "reset" and "stall" conditions:

```
logic clock, reset
logic [N-1:0] counter, counter_ns;
logic counter_reset, counter_stall;

assign counter_reset = forward_progress_seen || any_uninteresting_stimulus_or_state;
assign counter_stall = any_unresolved_external_dependency || any_safe_state;

assign counter_ns = (counter_reset)? N'b0: (counter_stall)? counter: counter + N'b1;

always @(posedge_ff clock)
      if (reset)
            counter <= N'b0;
      else
            counter <= counter_ns;
assert property (@(posedge clk) disable iff (reset)
                  (counter[N-1:0] < threshold[N-1:0]));
```

The counter is reset either when forward progress is seen (e.g., instruction on macro-op seen delivered to decode, or a flush is seen, etc.), or for conditions that we explicitly do not want to check due to any reason (e.g. a TLB miss). The counter is stalled when the DUT is waiting for an external dependency to resolve before it can make any progress (e.g. waiting for a cache fill to arrive on a cache miss), or conditions and state that we know to be safe. By stalling rather than resetting the counter on these "safe states" allows formal to get into these states, enabling the tool to explore what happens after these states are exited. What conditions are kept in reset vs stall are entirely up to the intuition of the verification engineer. As a starting point, the signals **`any_uninteresting_stimulus_or_state`** and **`any_safe_state`** can both be assigned to 1'b0. In the process of debugging counter examples, and as design knowledge improves, the verification engineer can then assign specific conditions to both these signals.

VIII. Summary and Results (Cortex-A77)

Data on Cortex-A77 bugs: Due to the disruptive nature of the change caused by addition of macro-op cache, several bugs were found in instruction fetch unit for Cortex-A77 unit. About two fifths of the bugs found at unit level were found by formal. These formal bugs were about one sixth of the total bugs found on fetch unit (at any level of verification). Here is a break-down of the bugs found by formal verification, based on project timeline (Fig [7]) and by type of properties (Fig [8]):


By RTL quality

Both RTL bring up, and late in project

- 80% of bugs found during RTL bring up
- 12% found in the last 3 months

Fig. 4: Bugs found by RTL quality

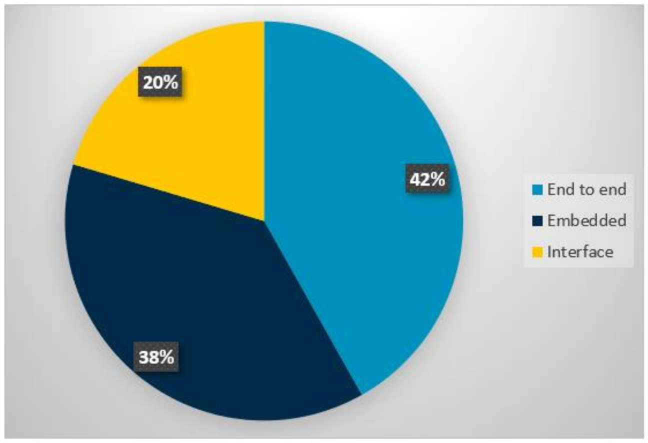


Fig 5: bugs found by property type

Since this was a highly intrusive change, it took some time for the traditional approaches to adjust their testing strategies and stimulus to be able to hit interesting corner cases in the design state space. As a result, there was a huge opportunity for formal to come in early. In addition to this, formal was able to find about 10 meaty bugs in the last 3 months of the project.

Designer quote about macro-op cache RTL bring-up: In this case formal was very useful during macro-op cache RTL bring up even before other supporting functionality around macro-op cache was ready, due to ease of setting up macro-op cache only hits.

Critical bugs: Formal found a few forward progress issues (hangs), and several other bugs out of which about 7 were corner case issues which were otherwise difficult to catch.

IX. Summary and Results (MACRO-OP Cache Coherence)

Data on Cortex-A77 successor CPU: Formal was applied selectively to verify coherence (a new feature) between macro-op cache and L1 instruction cache. The checker for coherence was developed on a priority basis to match RTL development schedule to aid in bring-up, and 7 interesting bugs were found within 2 weeks, 4 of which are mentioned below:

1. Special handling for page crossing: When address granules cross a page table boundary, then corresponding macro-ops were not being invalidated from the macro-op cache.
2. Instruction cache tag invalidation due to parity error colliding with a DVM snoop from L2 was causing a missed invalidation of the corresponding macro-ops in macro-op cache.
3. Instruction cache tag invalidation due to DVM snoop, colliding with an error in L2 response for a cache miss was causing a missed invalidation of the corresponding macro-ops in macro-op cache.
4. Instruction cache line capacity victimization (due a line fill from L2), along with a collision with a macro-op cache fill (from decode) corresponding to virtual address of the victimized line, allowing the line to be installed in macro-op cache, when it should not have been installed.

X. Conclusion

After the first partner delivery, a lot of additional work had been done at various levels of simulation based testbenches, FPGA based testing, and other environments to ensure robust checking. The value that formal had added was primarily seen before the first partner delivery (LAC milestone), where it could come in and quickly knock out some otherwise hard to find bugs, resulting in high quality release of RTL to Lead Partners. I am sure my very talented and experienced colleagues working with traditional verification methodologies would have eventually found most, if not all of these bugs, but it was quite clear that for a high quality LAC, formal played a critical role for this highly disruptive change to instruction fetch unit.

In this work, we demonstrated that formal can quite nicely complement other testing methodologies, due to its complementary strengths of "exhaustiveness", and when applied judiciously, it can add significant value in verification of complex changes to control intensive DUTs.

XI. ACKNOWLEDGEMENTS

I would like to thank my unit designer Yasuo Ishii for his invaluable help. I would also like to thank my unit verification lead Wes Ward for providing me the space to contribute and also for agreeing to modify simulation TB stimulus to match formal constraints wherever it made sense. Finally, I would like to thank my project validation lead Nagesh Loke for trusting me, bringing me to the table, and providing me with this opportunity.

XII. REFERENCES

[1] Prashant Aggarwal, Darrow Chu, Vijay Kadamby, Vigyan Singhal, "End-to-End Formal using Abstractions to Maximize Coverage" (https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD11/papers/inv2.pdf), FMCAD 2011