Sign-off with Bounded Formal Verification Proofs

NamDo Kim, Junhyuk Park
Samsung Electronics
Giheung, South Korea

HarGovind Singh, Vigyan Singhal
Oski Technology
Gurgaon, India and Mountain View, CA, USA

Abstract— **Formal property verification (also known as model checking) is a powerful methodology that can be used to find corner-case bugs, improve verification efficiency and reduce the verification cycle. However, inconclusive formal analysis results or bounded proofs have been hindering adoption of formal technology in the industry. This paper describes a formal sign-off methodology in the presence of bounded proofs. With an understanding of the design-under-test and a systematic analytical approach, we can qualify the bounded proof depths and use Abstraction Models to achieve the required proof bound for formal sign-off.**

Keywords—Formal verification; sign-off; abstractions; proof depth; bounded proof; formal coverage

I. INTRODUCTION

Most verification sign-off in the industry are based on simulation. Good metrics exist to track verification progress, such as percentage of checks written, bug rate (test-bench bugs and RTL bugs), coverage targets; and sign-off is defined as the meeting of these metric-driven goals. However, simulation is often not enough to verify functional correctness for today's complex designs. Formal verification (model checking) can cover all state transitions, thus proving exhaustive functional correctness of designs with complex corner case scenarios often harder to cover with simulation.

Formal technology can be used in different applications:

- automatic formal checks (also called super-lint or formal lint) can detect dead code, pragma violations, constant nets/registers and state machine deadlocks
- a recent category of Formal "Apps", which target specific applications, such as clock domain crossing verification, pre- and post-clock gating equivalence checking, X-propagation verification, can solve specific verification needs
- assertion-based-verification (ABV) can find bugs by verifying local assertions as well as prove compliance with standard interfaces
- end-to-end formal verification can replace block-level simulation by building reference models and proving the complete functionality of the blocks

Of all these applications, end-to-end formal results in the hardest proofs, but also offers the most benefits that make formal sign-off possible.

Regardless of the types of formal applications, for each checker verified by the formal tool, the result is either conclusive (either an unbounded proof or a failure) or inconclusive (accompanied by a proof depth of *N* cycles from reset). Traditionally, users have ignored the results when tools report "inconclusive", especially when used in a bug-hunting mode. In fact, commercial tools reinforce the worthlessness of this result by reporting "inconclusive" or "explored", instead of "bounded proof". In reality, the proof depth N guarantees that the shortest failure will be longer than N. If we can determine all interesting design behavior is observed within N cycles, the inconclusive bounded proof is equivalent to a full proof, no less useful than an unbounded proof.

We describe a methodology to use such proofs in a tapeout-worthy sign-off process, by qualifying the bounded proof depths with an analysis. Section II describes the notion of "sign-off" used during design verification, and sign-off requirements for formal verification. Section III describes the formal verification search process, and the use model. Section IV describes end-to-end checkers and the complexity challenge created by such checkers. Section V describes our methodology used to compute the required proof bound for a sign-off. Section VI discusses how to achieve formal sign-off using Abstraction Models with the understanding of the required proof bound. Sections VII and VIII demonstrates the effectiveness of the methodology to achieve formal sign-off using industry designs. Section IX offers concluding remarks.

II. VERIFICATION SIGN-OFF

Since the costs of making design changes rise exponentially as a design gets closer to tapeout, and beyond (Fig. 1), "sign-off" is used to denote a critical milestone when a particular department (e.g. timing, power, functional verification departments) commits to having reached a measurable level of completeness with respect to its respective metric(s).

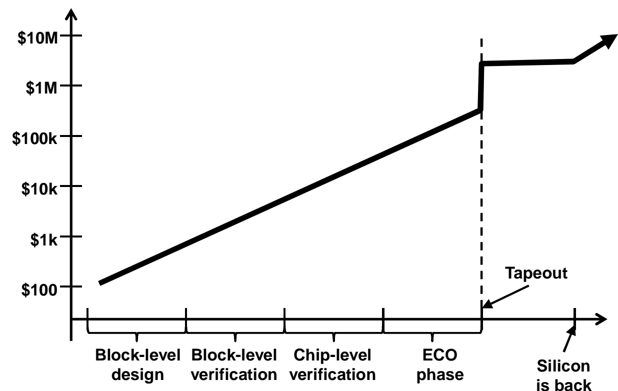

Fig. 1. The cost of fixing a bug rises exponentially

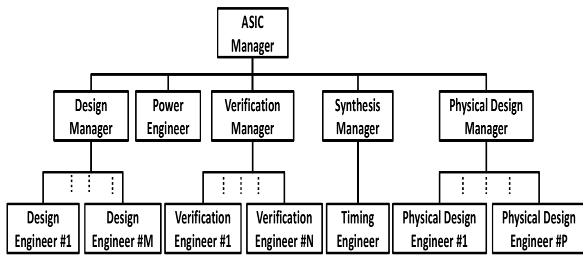

Fig. 2. Each department must sign off before tapeout

The origin of this word is from netlist sign-off at the point when an ASIC customer hands off the netlist to its ASIC vendor; at this time, the customer has met the timing and functional requirements in the netlist, and any change required by the customer thereafter is very expensive. In this traditional usage, sign-off was used for the timing analysis and equivalence checking departments. However, given the rising cost of changes as the design reaches certain milestones, (e.g. "final netlist", tapeout, or production milestones), the term sign-off is now also used to get commitments from other departments (Fig. 2), such as the functional verification department, which commits that the RTL has been thoroughly tested, and that the cost of continued verification has reached a point of diminishing returns, compared to the cost of lost market opportunity incurred by delaying the tapeout.

The sign-off requirements for simulation-based functional verification usually include tracking various metrics like number of open bugs, number of functional tests written vs the planned list of tests, and percentage of code and functional coverage targets achieved. To achieve similar sign-off commitments from the formal verification team, we need to quantify the utility of the "inconclusive" results from the formal tools, especially since most end-to-end formal proofs rarely give conclusive results. This will be the focus of the paper.

III. Formal Verification Use model

The inputs supplied to a formal verification tool are:

- the design-under-test (DUT),
- a set of constraints,
- a set of checkers (or assertions), and
- optionally, a set of manually written Abstraction Models (described further in Section VI) that reduce the complexity of the formal search.

In a given run-time, for each checker, the formal tool returns one of three possible answers:

1. (unbounded) pass, or a full proof, indicating there is a guarantee that no counter-example is possible in the entire search space;
2. fail, along with a counter-example trace the user can debug; or
3. an inconclusive result, or a bounded pass, along with a proof depth *N*.

The proof depths achieved by formal are often orders of magnitude smaller than depths reached by ordinary simulation tests. An oft-asked-question that needs to be answered is: If I achieve a bounded proof of N cycles on a given design, how much of the design functionality have I verified? In other words, for a given design, how many cycles do I need to cover with formal?

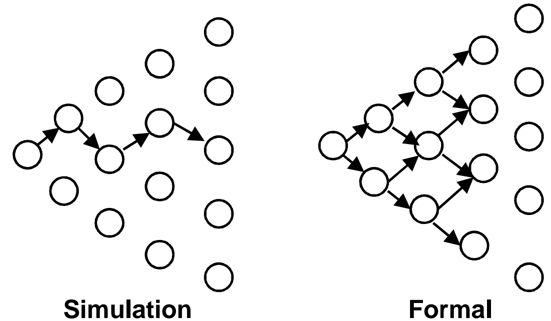

Fig. 3. Comparing state space searches

A formal tool performs an exhaustive state space search on the design, starting from the reset state. In Fig. 3, we contrast the state space searched via a simulation test, versus state space searched by formal verification (up to a proof depth of 3, in the figure). Whereas each simulation test covers a single but deep sequence of states after reset, the formal search performs a breadth-first search from a reset state, or from an intermediate state, in case of hybrid formal. Sometimes, usually for smaller DUTs or easy checkers, the tool can guarantee an unbounded proof. On the other hand, often for larger designs or end-to-end proofs, we face an exponential complexity barrier (Fig. 4). In such cases, for a given run-time, the tool reports a bounded pass.

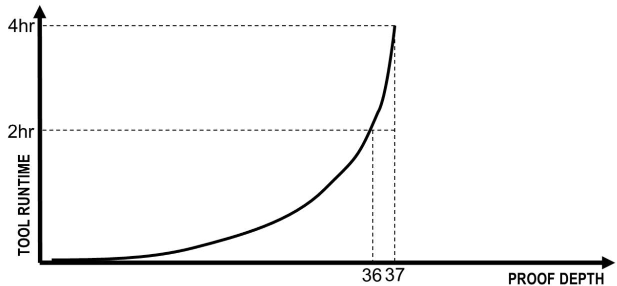

Fig. 4. Proof depth vs. Run time

IV. Complexity of End-to-end Proofs

Both the complexity of running formal verification proofs and the completeness of such proofs in fulfilling verification sign-off requirements depend heavily on the scope of the checkers being proven.

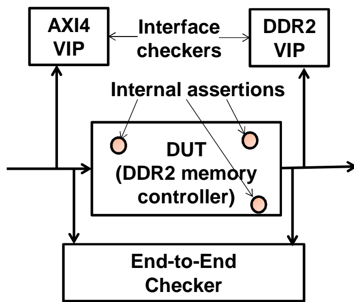

Fig. 5. Scope of formal checkers

Such scope can vary widely, as discussed earlier [1]. Formal can be used to verify (a) local assertions, (b) interface checkers, or (c) end-to-end checkers. Local assertions, easiest to verify, include assertions written inside the RTL as well as checkers like clock domain crossing (CDC) checkers. Interface checkers verify the compliance of the design to standard, or custom, interfaces such as ARM AMBA AXI [9], sometimes verified with the use of a formal assertion VIP. While verifying local assertions and interface checkers is useful and can uncover some hard-to-find bugs, clearly this does not replace the task of verifying the DUT completely with a sign-off commitment. If that is a motivation to apply formal, one must build end-to-end checkers (Fig. 5) as well. As one would expect, the complexity of proving end-to-end checkers is significant, and we rarely expect unbounded proofs.

In absence of an unbounded proof, we require every end-to-end checker to reach a *required proof bound*. This requirement is similar to the coverage metric in simulation. As argued earlier, if we can determine all interesting design behavior is observed within *N* cycles, the inconclusive bounded proof is equivalent to a full proof, no less useful than an unbounded proof. Just like the coverage metric in simulation, this metric provides confidence that we have exercised all corner cases in RTL. By using this metric, formal verification is now transformed into a target-oriented methodology. Based on the difference between depth achieved on a checker and the required proof bound, a formal engineer can make informed decisions on whether to add Abstraction Models (Section VI), or increase machine effort. With this metric in hand, formal sign-off is possible.

V. DETERMINING REQUIRED PROOF BOUND

We determine the required proof bound using following steps (many of which are subjective in nature):

A. Latency analysis of the design

B. Micro-architectural analysis

C. Covers for “interesting” corner-cases

D. Formal coverage

E. Failures seen during formal verification

F. Safety nets like bugs found in simulation and/or in hybrid regression runs

The first three steps are executed in the initial stages of the formal verification work, whereas the last three steps are carried out while executing formal verification project, and at the end.

A. Latency Analysis

This involves analyzing the latency from input to the relevant output port of the design. This initial analysis provides a lower bound for the required proof bound. In this step, we layer in additional proof depth due to design initialization, multiple input streams, long input packets etc.

For most designs, the latency number can be obtained by writing covers on output data valid ports. These estimates can change greatly as input constraints are developed. An unexpectedly large latency number obtained in the initial phase can also be profitably used. For example, out of reset, design may be performing automatic hardware initialization sequence. During this period, design will not accept any input, nor will it generate any output. Latency number observed for the design will highlight this, and verification engineer can tackle hardware initialization right in the beginning by either short-circuiting the initialization process (by applying cut-points and constraining – if it is irrelevant to the functionality under test), or by providing the post hardware initialization design state as the initial state to the formal tool.

B. Micro-architectural Analysis

This involves identifying major design structures e.g. state machines, counters, FIFOs, RAMs, linked lists. This can be done with the RTL designer’s help, or by an analysis of the RTL design. Architectural information is augmented by RTL code information e.g. deeply nested if-then-else, or case statements. Formal cover properties can be written to put each design structure in an interesting state to get an estimate of required proof depth. For example, we can find out the minimum cycles required to fill a FIFO, or traverse all states of a state machine.

As an example, consider a design that implements multiple FIFOs sharing a single memory for storing data, with supporting memories to store head/tail pointers for each FIFO, linked list and free list. Apart from writing covers for filling each FIFO, we can also determine the minimum cycles required to write to (or read from) each address of each memory. This information can be used to separate out randomly accessed and sequentially accessed memories, and the formal verification engineer can then determine if design size should be reduced (if design is parameterized), or Abstraction Models would be needed for the design.

C. Covers for Interesting Corner-cases

This step is similar to the analysis that is done to create the list of functional coverage targets to be met in simulation. This involves brain-storming the interesting scenarios to exercise different corner cases of RTL e.g. an internal arbiter, back-pressuring the input request path due to lack of output interface credits, causing the input request FIFO to get full. However, we need to carefully filter out the corner-cases not relevant to the RTL. This is similar to avoiding mistakes while

coding functional cover points for simulation. For example, a FIFO getting full along with a counter rolling over seems like an interesting corner-case; but the FIFO and counter may be completely unrelated design-structures, and targeting a proof bound where both structures reach their individual interesting states (simultaneously) may not be useful.

Note that it is often useful to under-constrain the inputs of a DUT to allow more generic behavior than code system-specific constraints that specify the exact input constraints. This scenario is possible if the DUT is designed to handle more generic behavior than possible in the real system. The use of such under-constraints prevents a large required proof depth than may otherwise be necessary – see the discussion of under-constraining number of AXI read/write transfers in our second case study in Section VIII.

D. *Formal Coverage*

Formal coverage is a relatively new feature offered by some commercial tools. This feature was proposed recently [11] as a metric very similar to simulation-based coverage with the same goals (to measure the completeness of the verification), albeit to measure the completeness of the formal verification work. The idea is that when formal verification runs are complete, a formal user can measure the code coverage targets that are hit with the formal testbench, within the proof bounds achieved in the formal regression runs.

The code coverage targets are identical to the targets used by simulation tools, e.g. line coverage, expression coverage, or FSM coverage.

The flow is to first run the end-to-end checkers and determine the proof bounds reported by the formal verification tool. Then, in a separate formal coverage run, the smallest proof bound is supplied as an input, and the formal coverage tool reports the code coverage targets reached within that bound. Any code coverage targets not reached indicate portions of the code that are not formally verified within the achieved proof bounds. These targets must be analyzed manually. It is also possible that some of these targets are not reached because of intentional or unintentional over-constraints in the formal testbench.

E. *Bugs Found during Formal Verification*

Usually, a single end-to-end checker finds most (or many) bugs. For every counter-example (both real bugs as well as false failures due to formal test-bench issues), we track the number of counter-examples reported for every depth (Fig. 6). The graph shown in the figure is very typical. For the data in the figure, if formal reported the first counter-example at a depth of N cycles, and later proves that there are no failures at a new higher depth (e.g. N+6 in Fig. 6), *usually* this higher depth is the required proof depth. This analysis has to be balanced with some exceptions: for example, if the design has multiple modes of operation, each resulting in a different latency from inputs to outputs, or if the latencies can be multiples of some number larger than 1, we have to ensure that we have covered all modes of operation. As an example, consider a design containing an arbiter, which arbitrates between multiple internal requestors, with one of the requestors being special – capable of generating requests only at cycle numbers which are multiple of 4. Goal is to prove that the design is deadlock free. Due to the presence of the special requestor, it is possible that our checker fails only at N+8 now. However, formal may have hit complexity limits and checker is bounded proven to N+7 cycles only. Due to this, while inferring required proof bound from a number of counter-examples seen for each proof depth, we must be aware of detailed design functionality.

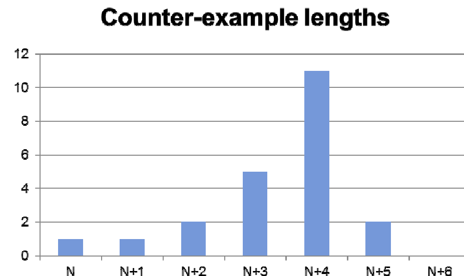

Fig. 6. Proof depth vs. Number of counter-examples observed

F. *Safety Nets*

Many of the steps described so far are based on manual analysis, and may possibly under-estimate the required proof bound. While they have worked very well for us, resulting in post-silicon success on many designs, we also like to use a few methods as backup safety nets:

1. *Bugs found by simulation.* Usually formal verification is run in parallel with simulation or emulation running at a higher-level (the DUT for formal is contained in the DUT for simulation). If a bug is found in the formal DUT by simulation or emulation, we always try to reproduce the bug in the formal testbench. This process could show that our analysis for the required proof depth was incorrect. We have experienced a situation where our original analysis for the required proof depth required rework when a simulation trace revealed a micro-architectural condition we had overlooked.
2. *Hybrid formal search.* Formal tools have the ability to search from deep states in the design [10]. Once the formal testbench is mature, we often set up nightly regression runs, with some runs running in a hybrid mode. If our proof depth analysis is incorrect, it is possible the hybrid formal search hits a bug, requiring re-analysis.
3. *Proof engines that are not strictly breadth-first.* Some recent powerful engines do not always search in a breadth-first manner [3]. While these engines cannot easily be used for a deterministic sign-off process, they may uncover a bug that shows a flaw in our proof depth analysis.

VI. ABSTRACTION MODELS

When formal cannot reach deep states in a design that are important to verify, use of manually crafted Abstraction

Models can often reduce the required proof depths, and achieve sign-off.

Abstraction Models are used to reduce the state space of a design so that the search becomes computationally less complex [1]. A sound Abstraction Model adds more behavior to a design, and does not remove any. By adding reset states, or state transitions to the design, the depth of state space of the original design can be reduced and hence far away corner cases can be reached in a lesser number of cycles than on the original design. These Abstraction Models help reach proofs, or failures, faster. There is also the possibility of a false failure, but such a failure comes with a trace, which can be debugged to determine if it is an RTL bug, or one due to an over-abstraction.

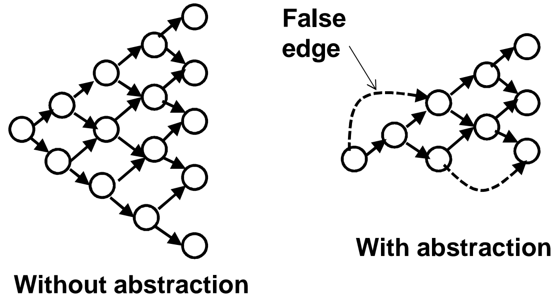

Fig. 7. State space transformation due to an abstraction

Examples of Abstraction Models include:

- *Localization*: adding cut-points in a design [5]
- *Counter abstraction*: replacing deep counters by abstract counters, e.g. replacing the 2^n state space of an n-bit counter by a few states: 0, 1, *at-least-two* [8]
- *Tagging*: simplifying a system consisting of a large number of data types by simplifying the structure with respect to a specific or a symbolic tag [6]
- *Memory abstraction*: replacing the height of a large memory with one line of memory, representing the transaction that is being tracked [1] [13]

Abstraction Models have the effect of both reducing the state space of a design (Fig. 7), as well as increasing the portion of a design that can be covered in N cycles, thereby increasing the reach of formal tools. When the state space is reduced in a sound Abstraction Model, multiple states from the original design map onto a single state, and all original transitions are aggregated, resulting in an addition of behavior and not the loss of any behavior.

Abstraction Models are design dependent, so one first needs to analyze the design-under-test (DUT) to understand where the formal complexity comes from and then craft specific Abstraction Models to overcome this complexity. While using Abstraction Models cannot result in a false positive (missing bugs), a coarse Abstraction Model may result in a false negative (a false failure, and hence not a real bug). Crafting the right Abstraction Model to solve the complexity problem can be an iterative process that leads to a final Abstraction Model, which is neither so tight that it doesn't help with verification closure nor so coarse that it results in false failures. The right Abstraction Model can offer significant improvement to the verification run time, as illustrated in the case study below.

A. Example of an Abstraction Model

Consider Tx portion of PCI Express transaction layer design which takes application layer requests and generates commands for data-link layer. Design contains a tag allocator block (TAB) which attaches a unique tag to each incoming request. TAB takes 'request' as input, and in next cycle, asserts either 'empty' or 'grant' at its output. If TAB asserts 'grant', it indicates granted tag number on its 'grant_tag' output port. When tag returns, 'return' input of TAB is asserted and freed tag is indicated on 'return_tag' input port.

Initially, TAB grants tags in linear order (0 -> 1 -> 2…), but as requests are processed and tags are returned in out of order fashion, TAB may start granting tags in a random order. Additionally, when there are no free tags, requests are back-pressured by asserting 'empty'. Due to the large number of available tags, interesting back-pressure scenarios can be reached only after 1024 cycles, which is way too deep for any formal tool to reach.

To reach interesting back-pressure scenarios for the PCIe transaction layer design, we can replace TAB by its Abstraction Model in which a single symbolic, but random, tag value ('sym_tag') is modelled. We create a state machine with two states (Fig. 8)

- H – Indicating that TAB has modelled tag
- D – Indicating that TAB doesn't have modelled tag

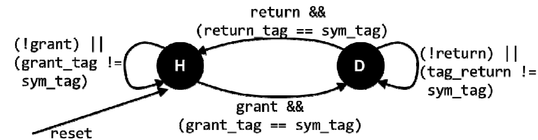

Fig. 8. Abstraction Model for TAB

Following properties were written on TAB outputs (P_{TAB})

- request |-> ##1 (empty || grant)
- (state == H) |-> (!empty)
- ((state == D) && grant) |-> (grant_tag != sym_tag)

Additionally, following properties were written on TAB inputs (P_{SYS})

- ((state == H) && return) |-> (return_tag != sym_tag)
- (state == D) |-> "sym_tag is eventually returned"

After replacing TAB RTL by Abstraction Model (i.e. assume P_{TAB}), all our PCIe transaction layer checks will see back-pressure condition right after 'sym_tag' is granted. It is equally important to prove the correctness of P_{TAB} assumptions. This can be done by running formal on TAB RTL and assuming P_{SYS}.

VII. CASE STUDY 1

For the first case study, we use the open-source version of the Sun Microsystems' UltraSPARC T1 microprocessor [4], [12]. This version, called OpenSPARC T1, is available as download from Oracle [7], under the GNU license. It consists of eight SPARC cores, connected through a crossbar (CCX) to an L2 cache [12] (Fig. 9). CCX is the design we are formally verifying in this case study; the top-level block diagram appears in Fig. 10. It consists of 22,174 lines of RTL code and over 30,000 flip-flops (TABLE I), representing a design that could stress the limits of end-to-end formal verification.

TABLE I DESIGN SUMMARY OF CCX

Parameter	***Value***
Line of Code	22,174
Design units	97
Instances	3,227
Number of inputs ports	59
Number of outputs ports	48
Input bits	1,982
Output bits	2,005

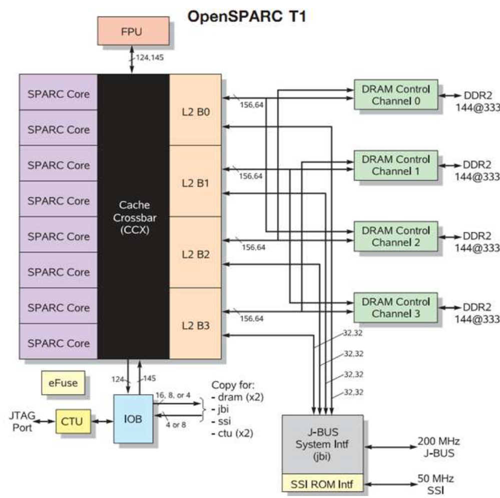

Fig. 9. Block diagram of OpenSPARC T1 processor

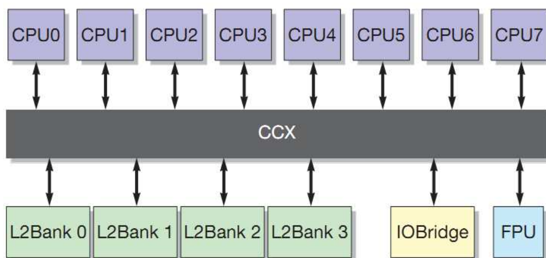

Fig. 10. CCX top-level block diagram

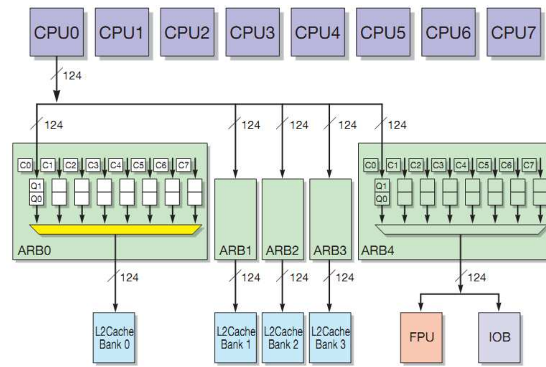

Fig. 11. Internal queues for L2-cache bank0, FPU and IOB

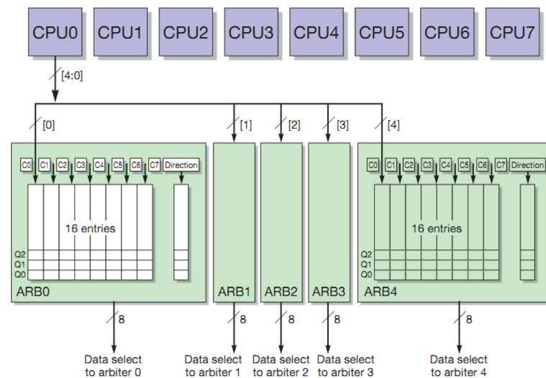

Fig. 12. Checkerboard implementation for CCX arbiter

CCX has one arbiter per destination to schedule requests when source(s) transmit packets to a single destination in the same cycle. A packet for a request that has not been scheduled by arbiter is stored in an internal queue which can store up to two such packets per destination for each of the sources (Fig. 11). Contents from eight sources are effectively stored in a 16 entries (2 entries per source) checkerboard structure (Fig. 12). Each source can possibly generate two-packet requests which consumes both the locations reserved for the source. Additionally, CCX arbiters pop first 2 requests from FIFO in flops to reduce latency, effectively creating an 18-entry structure.

The design is hard to verify with simulation because it has a large number of concurrent operations; all the 8 CPUs, 4 L2 cache banks, FPU and IO bridge operate in parallel. In addition, managing multiple requests to single destination makes the control logic quite complex. It is impossible to simulate all possible scenarios and not miss a bug in the simulation-only verification environment. Formal is the right technology to verify such designs as it performs exhaustive analysis to ensure all scenarios are covered with no missing bugs. Of course, formal has to converge.

To formally verify this design, the formal testbench environment has 11 checkers, 6 end-to-end checkers that model end-to-end behavior of the design, and 5 interface checkers that check the correctness of interface compliance.

One of the key checkers is the data consistency check – called *pcx_data_match_A*, which states that when output data ready is high, output data must match corresponding input data. There are also 16 constraints written to disallow illegal input scenarios.

A. *Determining Required Proof Bound*

1) Latency analysis: We wrote a cover on output data valid port and got a depth of 4 cycles. With a cover aware of two-packet requests, we got an initial proof depth estimate of 5 cycles, representing the minimum proof depth required to see interesting proofs amd counter-examples.

2) Micro-architectural analysis: Each destination has an effective storage of eighteen entries. To cover all corner-case for any potential bug in the storage of the 18th (last) location, at least 21 cycles (4 cycles of initial latency for first request + 17 more requests) are required.

3) Covers for interesting corner-cases: Our micro-architectural analysis driven depth requirement didn't include:

- Two-packet requests
- Presence of stalls

Writing cover which included these conditions gave us a proof bound of 23 cycles. We included one more cycle in our proof bound requirement as a safety margin. This gave us a 24-cycle required proof bound for the end-to-end checker for the CCX design.

4) Formal coverage: Our proof bound requirements were validated using toggle coverage. As shown in Fig. 13, the dotted blue line, 100% toggle coverage target was reached at 21 cycles. Apart from the data consistency check, the rest of the checks required much smaller proof depth.

B. *Complexity Analysis and Abstraction Model*

By plotting 'Proof radius' Vs 'effort' curve (Fig. 14), we estimated that tool would need 991 days to reach our required proof depth of 24 cycles. Clearly this runtime was not reasonable, and we needed a better strategy to achieve sign-off. After looking at the design, we decided to apply two Abstraction Models:

- Data-width Abstraction Model: Reduce the width of data buses. This didn't reduce the proof depth requirements, but reduced the number of flops in the design and hence formal complexity.
- Counter Abstraction Model: Each destination of CCX had an 18-deep storage. The write and read address counters of the arbiter FIFO are zero on reset and hence require many cycles to reach interesting scenarios like writing and reading last location of storage. Incremental nature of write and read address counters are key to design functionality. However, design behavior is immune to out-of-reset value of these two counters – as long as both the counters take same value. Use of this abstraction reduced our proof depth requirement to 9 cycles. This was validated by 100% toggle coverage results achieved in 6 cycles in Fig. 13 (dotted orange line)

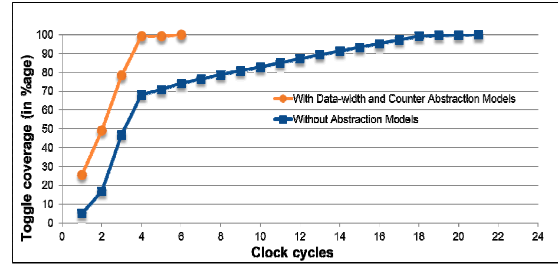

Fig. 13. Clock cycles vs. Toggle coverage

Comparative results of proof depth achieved for the data consistency checker, with and without Abstraction Models, are shown in TABLE II. The results are obtained using Mentor's Questa Formal version 10.2a and 5 CPU cores with a time-out of 8 hours. Required proof depth number helped us in deciding to work on Abstraction Models instead of just running tool for more time, and lead to a 600,000x speed-up in the data-consistency proof by developing two Abstraction Models so as to achieve sign-off on this checker.

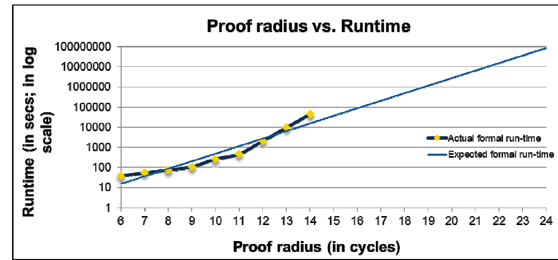

Fig. 14. Proof radius vs. Runtime for CCX data-consistency checker

TABLE II IMPROVEMENT IN CONVERGENCE BY ABSTRACTIONS

	Runtime Comparison with & without Abstraction Models			
	Without Abstraction Models	***With Data-width Abstraction***	***With Data-width and Counter Abstractions***	
Proof Depth	Runtime (in sec)	Runtime (in sec)	Proof Depth	Runtime (in sec)
10	253	175		
11	447	160		
12	2,008	317		
13	9,112	841		
14	Timeout (8 hours)	1,705		
15	Timeout (8 hours)	5,265		
16	Timeout (8 hours)	25,748		
17-22	Timeout (8 hours)	Timeout (8 hours)	7	56
23	Timeout (8 hours)	Timeout (8 hours)	8	101
24 full proof	Timeout (8 hours)	Timeout (8 hours)	9 full proof	149

VIII. Case Study 2

For the second case study, we pick a proprietary, commercial design for a scalable and configurable network-on-chip (Fig. 15), consisting of multiple masters and slaves interconnected using decoders, arbiters and optional register slices (RS). Fig. 16 shows the smallest possible, 2x2 version (2 masters and 2 slaves) of the network with cross-bar components, that we will use in this case study. TABLE III lists the design statistics of 2x2 sub-system. All masters and slaves followed AXI4 protocol on their input and output side respectively. However, in order to support much higher performance in the internals of the design, a proprietary modification was specified on the AXI4 protocol to allow sequences of packets on the five AXI4 channels that would otherwise violate the AXI4 protocol. This modification enabled much higher performance, but created additional corner-cases for verification, and created new challenges, especially for simulation-driven verification. The work described in this section was done using Mentor's Questa Formal tool.

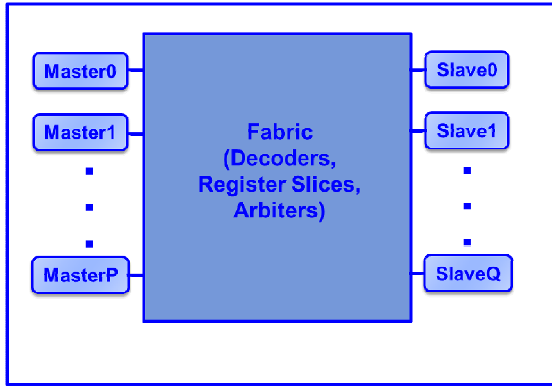

Fig. 15. A P * Q network-on-chip system (P masters and Q slaves)

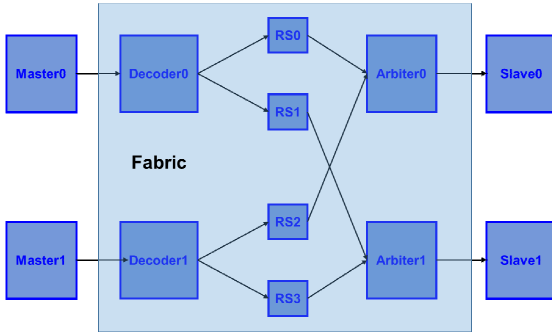

Fig. 16. 2x2 sub-system used for verification

Slaves ensure that the strict AXI4 protocol is obeyed at its output, even though the modified AXI4 protocol was allowed at its input. To this end, it needs to buffer transactions for AXI4 channels, and re-order transactions to comply with AXI4 protocol on its output.

The primary verification goal was to ensure that the network-on-chip does not deadlock under any possible combinations of input scenarios on all AXI4 channels.

TABLE III DESIGN SUMMARY OF 2x2 SUB-SYSTEM

Module	*Number of flops*
Master0	5,217
Master1	5,229
Decoder0	2
Decoder1	2
Register Slices (RS0+RS1+RS2+RS3)	334*4=1336
Arbiter0	382
Arbiter1	382
Slave0	3743
Slave1	3743
Total	20,036

We determined that the master can never contribute to dead-lock and excluded it from scope of formal verification. Further, we divided the remaining requirement into three separate proofs:

- The Fabric consisting of the Decoder-RS-Arbiters network does not deadlock
- The Fabric obeys the modified AXI4 protocol on its output
- Assuming the Fabric obeys the modified AXI4 protocol on its output, the Slaves do not deadlock

We take "Decoder-RS-Arbiters network doesn't deadlock" as an example checker to explain the methodology used to determine and validate required proof depth.

A. Determining Required Proof Bound

1) Latency analysis: We wrote covers on valid and control outputs of the two arbiters (e.g. AWVALID and WLAST signals) to derive a preliminary estimate. These covers were reached in 3 cycles.

2) Micro-architectural analysis: We analyzed the depth of FIFOs and counters used in the Arbiter nodes, especially paying attention to the conditions for getting the FIFOs to be full, and the counters to be empty. This gave us an estimate of 13 cycles.

3) Covers for interesting corner-cases: The AXI4 modification resulted in a few interesting functional coverage targets. We wrote covers for each of these targets. The cover for the deepest of these targets was hit in 29 cycles and was an indication of a lower bound on the required proof depth.

Note that the AXI4 protocol allows up to 256 transfers for read or write data bursts [2]. However, the Fabric design is oblivious to this requirement, and the output of the Fabric design satisfies this if and only if the input satisfies it. So, we deliberately under-constrain the inputs to allow arbitrary

number of, even more than 256, transfers. Since nothing in the design micro-architecture depends on this limit of 256, we are able to achieve a required proof bound far smaller than 256 cycles.

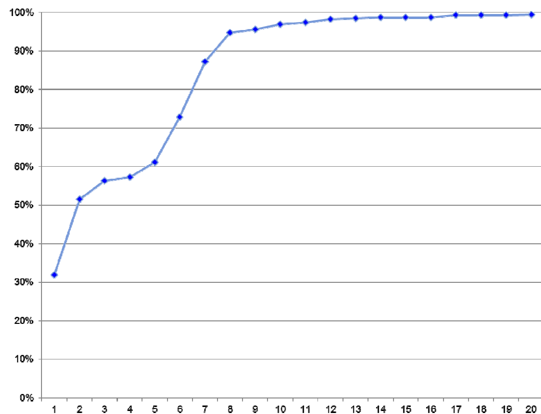

Fig. 17. Clock cycles vs. Coverage

4) Formal coverage: Our proof bound requirements were validated using statement, branch, expression and toggle coverage. As shown in Fig. 17, 100% coverage target was reached at 20 cycles.

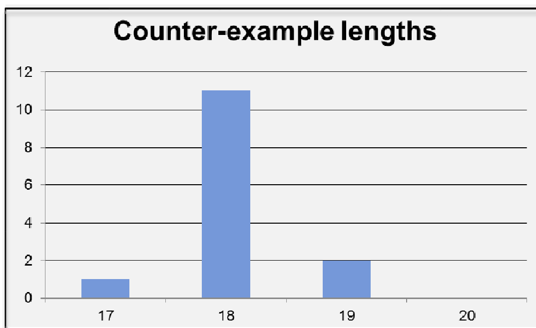

Fig. 18. Clock cycles vs. number of counter-examples seen

5) Failures seen during formal verification: As shown in Fig. 18, during our checker coding effort, we found first counter-example at 17th cycles, and most (11 total) of the counter-examples at 18th cycle and one counter example at 19th cycle. From 20th cycle onwards, no counter-example was seen.

B. Results

We reached a proof depth of 31 cycles for the Decode to arbiter deadlock checker in 12 hours of effort, satisfying our required proof depth target.

IX. Conclusion

Formal verification is a powerful technique to find corner case design bugs. A formal sign-off methodology can ensure no bugs are left behind when taping out. However, for end-to-end formal verification to be applied to realistic and interesting design blocks, expecting full proof is unrealistic. The key is to know what proof depth is sufficient and how to achieve such proof depth when formal tools don't converge. This paper outlined the techniques that we have used successfully to formally sign-off on many design blocks. With this methodology, formal tools will find more use in today's verification flow, and assist in resolving the challenging task of verification faced by the industry.

Acknowledgments

The authors would like to acknowledge support from Mentor Graphics to support their tool and the formal coverage options, especially Roger Sabbagh. The authors also want to thank Ankit Saxena and Prashant Aggarwal from Oski Technology, and designers Junhee Yoo, Jaegeun Yun, Bubchul Jeong and Dongsoo Kang for their work in the two case studies.

References

[1] P. Aggarwal, D. Chu, V. Kadamby, and V. Singhal, "Planning for end-to-end formal with simulation-based coverage," Proc. Formal Methods in Computer-Aided Design FMCAD 2011, Austin, TX, USA, 2011, pp. 9-16.

[2] ARM Ltd., "AMBA AXI and ACE protocol specification, Issue E," 2013.

[3] A. R. Bradley, "SAT-based model checking without unrolling," Proc. Verification, Model Checking, and Abstract Interpretation VMCAD 2011, Austin, TX, USA, R. Jhala and D. A. Schmidt (Eds.), Springer, 2011, pp. 70-87.

[4] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded SPARC processor," IEEE Micro, vol. 25(2), 2005, pp. 21-29.

[5] R. P. Kurshan, "Formal verification in a commercial setting," Proc. Design Automation Conference DAC 1997, Anaheim, CA, USA, pp. 258-262.

[6] K. L. McMillan, "Verification of an implementation of Tomasolu's algorithm by compositional model checking," Proc. Conf. Computer-Aided Verification CAV 1998, Vancouver, BC, Canada, A. J. Hu and M. Y. Vardi (Eds.), Springer, 1998, pp. 110-121.

[7] Oracle Corp., "OpenSPARC overview", OpenSPARC site, http://www.oracle.com/technetwork/systems/opensparc/index.html.

[8] F. Pong and M. Dubois, "A new approach for the verification of cache coherence protocols," IEEE Trans. Parallel Distrib. Systems, vol. 6(8), 1995, pp. 773-787.

[9] C. Saye and J. Sonander, "Formal verification of AMBA 3 AXI bus systems," ARM Information Quarterly, vol. 4(2), 2005, pp. 15-17.

[10] A. Seawright, R. Sathianathan, C. G. Gauthron, J. R. Levitt, K. C. Mulam, R. C. Ho, and P. Yeung, "Selection of initial states for formal verification," U. S. Patent 7,454,324, November 2008.

[11] V. Singhal and P. Aggarwal, "Using coverage to deploy formal verification in a simulation world," in Proc. Conf. Computer-Aided Verification CAV 2011, Snowbird, UT, USA, G. Gopalakrishnan and S. Qadeer (Eds.), Springer, 2011, pp. 44-49.

[12] Sun Microsystems, "OpenSPARC T1 Specification,", http://www.oracle.com/technetwork/systems/opensparc/index.html.

[13] M. N. Velev, R. E. Bryant, and A. Jain, "Efficient modeling of memory arrays in symbolic simulation," in Proc. Conf. Computer-Aided Verification CAV 1997, Haifa, Israel, O. Grumberg (Ed.), Springer-Verlag, 1997, pp. 388-399.