2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Formal Verification of a Highly Configurable DDR Controller IP

Sumit Neb, Synopsys Inc., Dublin, Ireland (sumit.neb@synopsys.com)
Chirag Agarwal, Oski Technology Inc., Delhi, India (chirag@oskitech.com)
Deepak K. Gupta, Oski Technology Inc., Delhi, India (degupta@oskitech.com)
Roger Sabbagh, Oski Technology Inc., Ottawa, Canada (rsabbagh@oskitech.com)

***Abstract*— In this paper, we describe the formal verification methodology used for the AMBA AXI Port Interface (XPI) controller of the Synopsys DesignWare Cores DDR Memory Controller [1]. The controller is feature-rich and the XPI is a highly configurable design, which varies with both hardware and software configuration parameter settings. This creates a formidable challenge for traditional verification methods as there are many combinations of settings resulting in long simulation cycles to reach the coverage objectives. Contrast this with formal verification, which uses mathematical algorithms to efficiently test design behaviors for all possible stimulus and configurations. We will describe the formal verification flow including configurable formal testbench implementation and formal coverage closure.**

Keywords—Formal Verification, Design IP, DDR SDRAM

I. INTRODUCTION

One of the factors that has enabled the continued growth in complexity of modern SoCs is the widespread use of design IP [2]. Design IP provides many significant benefits including reduced development cost, reduced risk, improved time-to-market and perhaps most importantly, it allows design teams to focus on developing the portions of their designs that offer differentiated value to their customers. However, to be reused across a wide spectrum of applications and to enable architectural trade-offs to be made between performance, power and area, design IP must be highly configurable through both hardware and software parameter settings.

Ensuring the quality and speed of time-to-market of highly configurable designs presents a challenge for IP providers. Formal verification can play a very important role in addressing that challenge. Formal verification uses mathematical techniques to efficiently explore all design configurations in parallel. Thus, it can provide complete coverage, equivalent to that achieved by simulating all possible configurations sequentially.

II. FORMAL VERIFCATION SIGN-OFF METHODOLOGY

In the past, formal verification was largely used to complement simulation in areas where coverage was suspect and corner-case bugs might have escaped the tradition verification flow. However, what started as a bug hunting approach has evolved and matured in recent years to the point where formal verification is now being used to provide complete verification sign-off of some types of designs [3].

Formal verification methods suffer from state space complexity barriers, which are associated with exponentially hard to solve problems. To enable complete design assurance with formal verification, formal technology must be applied in a very strategic manner. The formal sign-off methodology consists of the following primary components, known as the four "Cs":

A. End-to-end Checkers

End-to-end checkers model the end-to-end behavior of the block under test. They must be coded using special techniques that make them "formal friendly" so that they add only the minimal amount of complexity overhead. End-to-end formal checkers act like scoreboard checkers in simulation. Contrast this with traditional ABV formal, in which white-box checkers capture the pin-point behavior of sub-block circuit elements, such as a pair of interface handshake signals or the expected transitions out of a specific FSM state.

B. Constraints Validation

Constraints are required to filter out illegal input space combinations, however they must be verified to ensure bugs are not missed, which can happen when the design-under-test is over-constrained. Constraints can be validated through various approaches including simulation of constraints in higher level simulation testbenches, formal verification of neighboring blocks in an assume-guarantee approach and with formal coverage metrics.

C. Complexity Management

When formally verifying end-to-end checkers on real-world designs, the process will undoubtedly encounter complexity barriers. Abstraction models are used to reduce the state space depth so that formal verification can explore beyond the default complexity barriers. It is only through the effective use of abstraction models that a sufficient depth of analysis can be achieved for verification sign-off.

D. Formal Coverage

Formal coverage [4] measures both the range of stimulus that has been explored as well as the quality of checking performed by the testbench, making it a very strong sign-off metric. Formal coverage helps to grade the quality of the other three "Cs" in the methodology, as follows:

- Checkers

  Formal coverage helps to answer the question of whether the checkers are sufficient to check the entire behavior of the DUT.

- Constraints

  Formal coverage can indicate when the design inputs have been inadvertently over-constrained.

- Complexity

  Although it's not the only criteria [5], formal coverage helps to determine if the depths of bounded proofs are large enough.

Formal coverage comes in two forms: reachability coverage and observability coverage.

1) Reachability Coverage

Formal reachability coverage is a measure of the ability of the formal engines to explore all possible states of the DUT, as defined by the coverage model. The coverage model may include structural code coverage items, such as line and condition coverage, as well as functional coverage expressed in the form of SVA cover directives or covergroups.

Reachability coverage serves multiple purposes:

a) Detecting dead code

Coverage goals that are proven to be unreachable may be an indication of a bug or redundant code which should be fixed. On the other hand, they may be due to parameter settings or coding style, in which case they can be waived.

b) Detecting over-constraints

Formal tools will report coverage goals that are proven unreachable due to the constraints. This is an indication that the input constraints are too tight and not allowing the formal analysis to fully explore the DUT, which in turn could mask design bugs.

c) Qualifying the depths of bounded proofs

Coverage goals which are only reachable at depths beyond the bounded proof depths are a red flag that indicate that the bounded proof depths are insufficient. In this case, additional abstraction techniques can be used to enable these design states to be reached at shallower depths of analysis.

2) Observability Coverage

Formal observability coverage measures the amount of logic that is being checked by the assertions in the formal testbench. Coverage holes in these metrics indicate that the assertions in the formal environment are insufficient, in quantity or quality, to verify the complete design.

Observability coverage comes in three forms:

a) Cone-of-influence (COI) coverage

COI coverage is purely a structural metric that reports DUT registers that are not in the structural fan-in cone of any formal testbench assertion. In some cases, this logic is not required for the configuration under test, and it can be safely waived. In other cases, it may indicate a missing check which must be added, otherwise a bug in the flagged logic would not be detected by formal analysis.

b) Formal engine coverage

Formal engine coverage reports the amount of COI registers that are not required for the proof or bounded proof of any of the assertions. This metric reflects the automatic abstraction process that is performed by the formal engines to trim the COI. It is a much more accurate, but more computationally expensive metric as compared to COI coverage.

c) Formal mutation coverage

Formal mutation coverage measures the set of design mutations that can be exposed by the formal verification testbench. The premise of mutation coverage is that if it is possible to insert any bug in the design that is not detected by an assertion, then the formal verification cannot be considered complete.

As mentioned, formal verification requires a separate tool run to generate coverage reports for each H/W configuration of the DUT. Hence, formal coverage metrics must be generated, and coverage closure achieved, for each H/W configuration.

III. AMBA AXI PORT INTERFACE CONTROLLER

This paper describes how the formal sign-off methodology was applied to address the challenges of verifying a highly configurable IP design. The design under test (DUT) in this case study is the AMBA AXI Port Interface (XPI) controller. The block diagram of the XPI is shown in Figure 1. The XPI has dozens of software configuration registers which could potentially be programmed in as many as 10^{16} different combinations of settings. In addition, there are over 100 hardware parameters that configure the controller and enable various modes and features, such as:

- Address and Data bus widths
- Burst lengths
- Command support
- Parity / ECC protection
- Dual channel support
- Interleaving
- Upsizing and downsizing

A. Hardware Configurability in Formal Verification

Hardware configurability of design IP is achieved with compile-time settings, such as parameters and `define macros in SystemVerilog. These settings must be taken into consideration when constructing the formal testbench. The checkers, constraints and coverage points must automatically adjust to match the changing hardware in the DUT.

State-of-the-art formal verification tools are unable to automatically explore multiple H/W configurations in a single tool run. Much like simulation, the formal verification strategy involves running each configuration

separately, however, unlike simulation where hundreds of tests must be run for each H/W configuration, formal can explore each in a single run.

B. Software Configurability in Formal Verification

Software configuration registers in the IP can be set as cut-points in the formal tool to allow formal to directly drive the register values as though they were inputs and explore all possible configuration settings. However, each H/W configuration typically supports only a sub-set of S/W configuration settings. This legal sub-set must be defined through formal constraints that are tailored by the H/W configuration control settings.

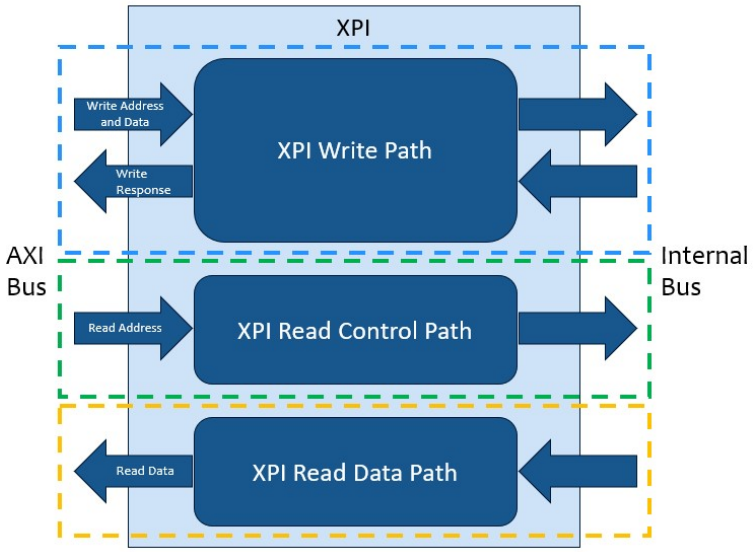

Figure 1. XPI Block Diagram with Formal Verification Partitioning

IV. RESULTS

We developed a configurable formal verification testbench for the XPI which allowed multiple hardware configurations to be tested. For each hardware configuration, the formal constraints allowed the full legal range of software configuration parameters to be tested.

We partitioned the formal testbench into multiple sections that independently covered:

- the write path, including the write response,
- the read control path, and
- the read data path.

The following sub-sections describe how each element of the 4Cs methodology was implemented for this design.

A. End-to-End Checkers

Over 70 formal checkers were developed and verified against the RTL. This included the reference models and the end-to-end checks required to verify the data transfer between the AXI interface and the internal bus.

Some of the key design requirements that were formally verified included:

1) Data Integrity

Data integrity checking verifies that write data is transferred correctly from AXI interface to the internal bus, and that read data is transferred correctly from the internal bus to the AXI interface. It includes checks such as:

- Data should not get dropped or duplicated inside the design
- There should not be any spurious data transfer

These kinds of checks require that the formal testbench include a reference model of the transfer function of the design, which takes into account the parameterized data bus widths on both interfaces.

2) Forward Progress

Forward progress checking verifies that the design does not suffer from deadlock, livelock or starvation issues. It includes checks such as:

- If there is no backpressure, write data should get transferred from AXI to the internal interface within a bounded number of clock cycles

Configurability of Checkers

To make the testbench configurable, we used various software register inputs and hardware parameters in the modelling of the reference models.

H/W configurability examples:

1) A particular block in the design is used only for a particular HW configuration. In that case, the functionality of that block must be coded in the reference model in such a manner that it is used by the checkers only when corresponding HW configuration is ON.
2) The data width at the input and output side is defined by H/W parameters. In that case, reference model must be modelled such that it can handle different values of data width in different H/W configurations.

S/W configurability examples:

1) The signal block_en is a software-controlled input which enables/disables the block. If this signal is de-asserted, the DUT will not send any outputs even if there is input traffic. In that case, the formal reference model must use the block_en signal in modelling so that it does not expect any output. If it does not use the signal, then it will expect an output and the corresponding checkers will fail.
2) The transaction burst length at the interfaces is governed by S/W controlled registers. The reference model must use those register inputs so that it can work with different burst lengths.

B. Constraints

In formal verification, all inputs of the DUT are free and the formal tool can drive any value to these inputs. Illegal values of these input signals may lead the design to misbehave. To stop the illegal combinations at the input, constraint properties are written.

For the XPI design, we used the ARM AXI protocol VIP [6] at the AXI interface so that the formal tool would follow the legal AXI protocol while driving the AXI inputs of the DUT. We also wrote constraints on the interface with the internal bus to ensure that the internal bus protocol was followed.

For constraint validation, we used the constraint properties as the checkers of the neighboring blocks to the XPI. A failure of any of these properties would indicate a possible over-constraint. An over-constraint in the formal setup may disable some functionality of the DUT and may lead to bugs being missed in the verification process. Hence none of the constraint properties should fail when run as checkers of the neighboring blocks.

C. Formal Coverage

We used formal verification coverage metrics to sign-off on each hardware configuration that we tested. Formal coverage was used to check for over-constraints, validate the depth of bounded proofs and to measure the completeness of the set of checkers.

The preliminary coverage results, before applying abstractions, for a single H/W configuration are summarized in following tables:

Coverage	Total Points [T]	Deadcode [D]	Covered [C]	Uncovered	Inconclusive	% Coverage [%C/(T-D)]
Line	1,223	58	1,163	0	2	99.82
Condition	809	99	693	1	16	97.60

Coverage	Total Points	In COI	Out of COI	% Coverage
Cone-of-Influence (COI)	283	183	100	64.66

Coverage	Total Points	In FE	Out of FE	% Coverage
Formal Engine (FE)	283	170	113	60.07

The 2 inconclusive line points, 16 inconclusive condition points and 13 in COI but out of Formal Engine points were not covered as they were very deep because of the complexity of the design.

D. Complexity

In the XPI IP, there were two main sources of complexity:

1) Multiple deep FIFOs: One important scenario related to FIFO is the rollover of the read/write pointer counters. As the depth of the FIFO increases, the coverage of this scenario required deeper formal analysis, in terms of the number of clock cycles of behavior explored.

2) Token management: The token manager controls the status of tokens and keeps track of whether they are in use or not. It releases the next token which is available with the highest value. In such a case, exploring the design states when the token number zero is released may take a huge number of clock cycles, which is impractical with formal analysis.

To resolve above two complexity issues, we applied reset abstraction models.

1) FIFO reset abstraction: We abstracted the reset values of FIFO pointers such that they were able to take any values out of reset. Care was taken here to ensure that both the pointers tooklegal values out of reset. The rollover scenario of the pointers then came closer to the reset state after abstracting the reset values. Hence the deep inconclusive coverage points related to the FIFO were covered.

2) Reset abstraction on token management: We abstracted the reset value of the status register so that it started with a non-deterministic status out of reset. This allowed the token zero to be released at any time.

After applying the reset abstraction on FIFOs and token status register, the inconclusive points were covered and the overall coverage scores increased.

Improved coverage results are summarized in following tables:

Coverage	Total Points [T]	Deadcode [D]	Covered [C]	Uncovered	Inconclusive	% Coverage [%C/(T-D)]
Line	1,223	58	1,165	0	0	100.00
Condition	809	99	709	1	0	99.85

Coverage	Total Points	In COI	Out of COI	% Coverage
Cone-of-Influence (COI)	283	183	100	64.66

Coverage	Total Points	In FE	Out of FE	% Coverage
Formal Engine (FE)	283	183	100	64.66

We observed the following improvement in the coverage scores:

1) Line coverage scores increased from 99.82% to 100%
2) Condition coverage scores increased from 97.60% to 99.85%
3) COI coverage score was not affected by the abstraction models
4) Formal engine coverage score increased to its maximum value which is equal to the COI coverage score

The following is the justification for the coverage holes in the above scores:

1) One condition points failed to be covered because of a valid constraint. This condition is not expected to be covered with legal stimulus for the configuration under test.
2) 100 points were out of COI as these points were not used in the mission mode function of the given H/W configuration. We reviewed these 100 points with the design team and waived them based on classification into groups, for example debug ports and signals tied-off to 0.
3) 100 points were out of FE as they were out of COI and the points which are OUT of COI can never come in FE. Hence, the expected FE coverage score is always less than or equal to the COI coverage score.

V. CONCLUSIONS

Formal verification of configurable design IP, such as the XPI, provides a boost to design confidence through improved coverage. Formal verification explores all possible software configuration registers settings in parallel. Also, a configurable formal testbench enables multiple hardware configurations to be tested. Finally, formal coverage metrics are a key component of the sign-off criteria.

VI. FUTURE WORK

The formal verification of XPI described in this paper covered two of the most common hardware configurations. Future plans for formal verification include extending the formal testbench to support additional H/W configurations. This includes adding the corresponding S/W configuration constraints for each added H/W configuration.

The coverage metrics used did not include mutation coverage. In the future, we plan to deploy mutation coverage as part of the closure process. We also plan to close coverage for additional H/W configuration.

REFERENCES

[1] https://www.synopsys.com/designware-ip/interface-ip/ddrn.html

[2] https://www.synopsys.com/dw/doc.php/ds/o/ip_accelerated_brochure.pdf

[3] I. Tripathi, A. Saxena, A. Verma, P. Aggarwal. "The Process and Proof for Formal Sign-off: A Live Case Study.", DVCon 2016.

[4] 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.

[5] N. Kim, J. Park, H. Singh, V. Singhal, “Sign-off with Bounded Formal Verification Proofs”, Design Verification Conference (DVCon) 2014

[6] AMBA® 4 AXI4™, AXI4-Lite™, and AXI4-Stream™ Protocol Assertions User Guide