Architectural Formal Verification of System-Level Deadlocks

Mandar Munishwar[1], Naveed Zaman[1], Anshul Jain[2], HarGovind Singh[2], Vigyan Singhal[3]
[1]Qualcomm Inc., San Jose, USA
Oski Technology, [2]Gurgaon, IND, [3]San Jose, USA

***Abstract*- Formal verification applications have been evolving over the past few decades to address problems at increasingly higher levels of hardware design complexity. The latest step in this evolution has been the introduction of architectural formal verification methodologies to target system-level requirements. This paper describes our architectural formal verification process, including the development and use of architectural models, and we present a case study of how this method was applied to verify the absence of deadlocks in an industrial design.**

I. INTRODUCTION

Traditionally, design teams have relied upon System-on-Chip (SoC) Register Transfer Level (RTL) simulations and in-circuit emulation to verify system-level requirements such as safety, security, coherence and absence of deadlock. These verification approaches model and test the entire chip at a relatively low level of abstraction. Due to the complexity of modern designs, the coverage of corner-case scenarios using these methods is predictably low and it leaves us with the possibility of subtle, system-level bugs surviving the verification process.

Verification of control-oriented problems, such as deadlock, are a perfect fit for formal verification. Formal verification can provide complete coverage, equivalent to that achieved by simulating all possible scenarios, thus leaving no bugs behind. On the other hand, formal model checking suffers from the exponential challenge associated with PSPACE-complete problems [1]. Thus, proving absence of deadlock on the RTL model of an entire system with formal technology is impractical. Instead, we move to a higher level of abstraction in order to produce meaningful results.

Fig. 1 illustrates how formal verification applications have been evolving over the past couple of decades to address problems at increasingly higher levels of hardware design complexity. Formal verification began being used in the 1990s to provide a static sign-off flow for synthesis by comparing the logic of the RTL to that of the gate level netlist, one flip-flop at a time. In the 2000s, Assertion-Based Verification (ABV) came into use to formally verify small cones of sequential logic, such as those which controlled one-hot encoding of an FSM or a hand-shake interface protocol. More recently, formal sign-off methodologies have emerged, which enable formal verification of entire design blocks using end-to-end formal checkers with abstraction models [2].

The next big leap in the evolution of formal verification is to the system-level. Architectural formal verification is a novel approach which on one hand, leverages the exhaustive analysis capability of formal to explore all corner cases, but on the other hand, uses highly abstract architectural models to overcome complexity barriers and enable deep analysis of design behavior. This forms a powerful combination which enables effective system-level requirements verification that is especially useful to target areas that are not well covered by traditional verification methods, such as deadlock.

Since the methodology does not rely upon the availability of RTL models, one additional benefit is that it can be deployed very early in the design phase. This allows architectural bugs to be detected and fixed before they are propagated throughout the implemented design. In contrast, fixing late-stage architectural bugs that are found through full-chip simulation or emulation may require many RTL design changes, since fixing these types of bugs often has a ripple effect that spans out to many blocks. It is vastly preferable to avoid that type of code churn, which can result in a significant setback in verification maturity and lead to costly project delays.

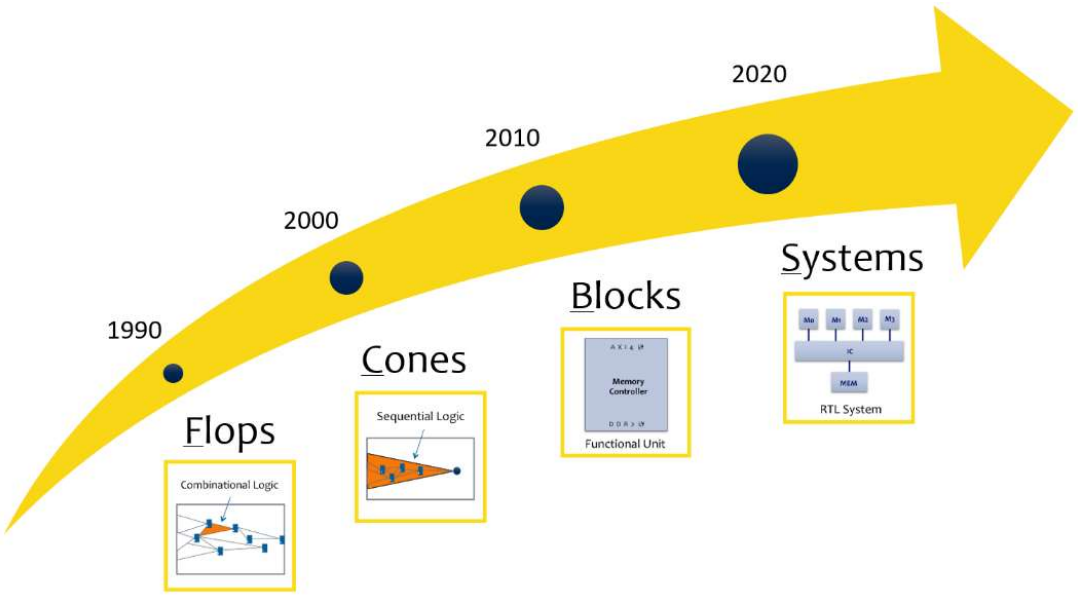


Fig. 1. The evolution of formal applications over the decades.

II. Architectural Formal Verification Methodology

The following sub-sections describe the three major steps of the architectural formal verification methodology.

A. Block-Level Architectural Modeling

In the first step, an Architectural Model (AM) is created for each block that contributes to the system-level requirement that is being verified. In any hierarchical design, system-level requirements must be decomposed and distributed amongst the block-level components. This effectively forms a contract in which it is incumbent upon the blocks to deliver specific functionality that contributes to satisfying the system-level goals.

The block AMs include only a small slice of the functionality of the blocks. This slice of behavior models the contract for a specific system-level requirement. A collection of block AMs forms a set of abstract models for which all other block-level design details are excluded, except for those required to achieve the verification goal.

To verify the absence of system-level deadlocks, the block AMs include only the functionality related to forward progress. They would model behavior such as the passing of control through Start/Done messages, wait states that can be encountered due to limited availability of resources and flow control at interfaces through backpressure or credit-based protocols.

The block AMs are coded with combination of SystemVerilog Assertion (SVA) properties and SystemVerilog (SV) RTL code. A key aspect of the AMs is that the output response of the blocks is controlled using SVA properties that are set as assumptions. These assumptions are subsequently used in Step 3 to verify the implementation.

The block AMs also include non-deterministic models for the latency through the blocks. This allows for a variable range of timing options to be explored and for discovery of bugs related to corner-case combinations of block latencies. A side benefit is that block level timing specifications can be developed by discovering the limitations of acceptable timing parameters at the system-level.

B. System-Level Requirements Verification

In the second step, the system-level architectural model is constructed from a collection of block level AMs. Then the formal sign-off methodology, which is routinely employed for block level verification, is used to prove that the system level requirements hold for this architectural model.

The formal sign-off process is depicted in Fig. 2 and it can be broken down into four parts which are described below.

End-to-end Checkers. End-to-end checkers are quite different than typical assertion based checks. They model the end to end behavior of the block under test, and predict what the output should be based on the input, much like a scoreboard.

For example, a forward progress checker might be coded to check that when some activity is observed at the input to the system, then a corresponding output should arrive within a finite time, when accounting for acceptable blocking conditions.

The end-to-end checking logic is typically coded using specialized abstraction techniques that makes them suitable for formal verification. This is important so that they don't add too much complexity to the state space of the model which is being formally verified.

Constraints. By default, formal verification will explore all possible input stimulus, so it's important to filter out the illegal input space with constraints and eliminate false failures of the checkers. However, care must be taken to verify that there are no over-constraints which could mask real failures and cause bugs in the design to be missed. The constraints can be verified through formal methods such as assume-guarantee and by running the constraints as assertions in the system-level simulation environment.

Dealing with Complexity. Even though we are working with very abstract AMs, formal verification may still hit complexity barriers for system-level architectural designs, especially when dealing with end-to-end checkers. Thus, the requirement exists for the next piece of the methodology, which are the abstraction models. Abstraction models reduce the state space or latency of the design so that formal verification can explore beyond its default threshold.

For example, reset abstraction is a commonly used technique that allows formal analysis to reset the design to very deep sequential states. Design behavior can be explored around those deeps states that would otherwise be unreachable when starting formal verification at the default reset state.

Formal Coverage. Formal coverage is a key component of the sign-off methodology. Much like in simulation, formal coverage measures controllability - how many of the design states have been explored by the input stimulus. But, it also has a unique ability to report observability coverage - that is how many of the design states are checked by the end-to-end checkers. This makes formal coverage a very strong metric for measuring progress and closing verification holes.

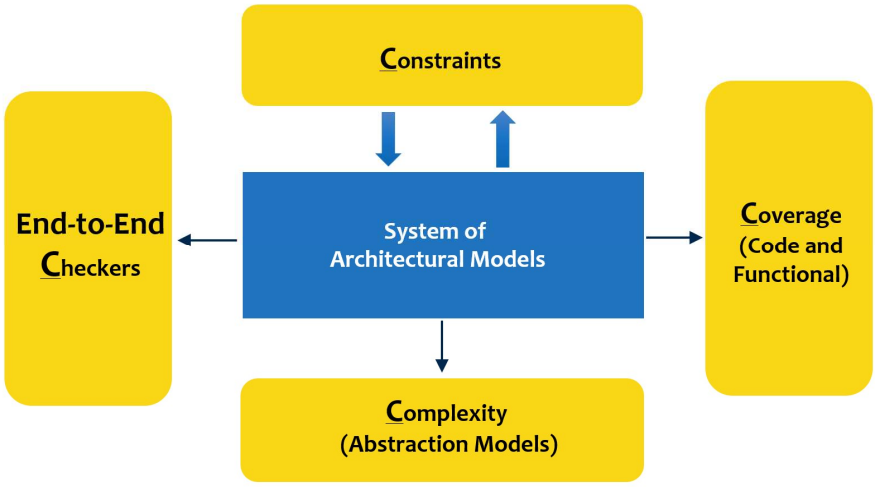


Fig. 2. Components of the formal sign-off methodology

C. Block-Level Implementation Verification

The third and final step in the architectural formal verification flow is to check that the RTL implementation of the blocks will guarantee the system-level contract that was assumed for the AMs. In this step, the modeling logic of an

AM and the SVA assume properties that govern the output behavior are turned into a checker for the RTL code of the block by now asserting those properties. This checker can be used in either simulation or formal verification.

When used in formal verification, this step is a formal equivalency check between the AMs and the RTL model, one block at a time. It serves the purpose of closing the loop on the architectural verification process to ensure that the implementation has not introduced bugs that would cause the design to fail to meet the architectural requirements. Since the problem is decomposed to a sequence of block-level verification tasks, it becomes tractable. A failure observed in this step due to a mismatch between the two models would detect a bug in the architectural specification from which the AM was derived, a bug in the AM itself or a bug in the RTL model.

III. CASE STUDY: WIRELESS SOC PHY LAYER SUB-SYSTEM

The design under test (DUT) in this case study is a wireless SOC PHY layer sub-system. The block diagram of the system is shown in Fig. 3. Wireless packets contain various fields in the preamble and payload which must be concurrently operated on by the corresponding design blocks during the reception and transmission of data [3]. In this system, the central controller (RX CTRL) must sequence the operation and manage the interaction between the blocks, but the blocks also communicate and pass control between themselves. The risk of deadlock arises because of the many corner-case interactions between the various blocks and the dependency upon the control being successfully passed between them under all conditions.

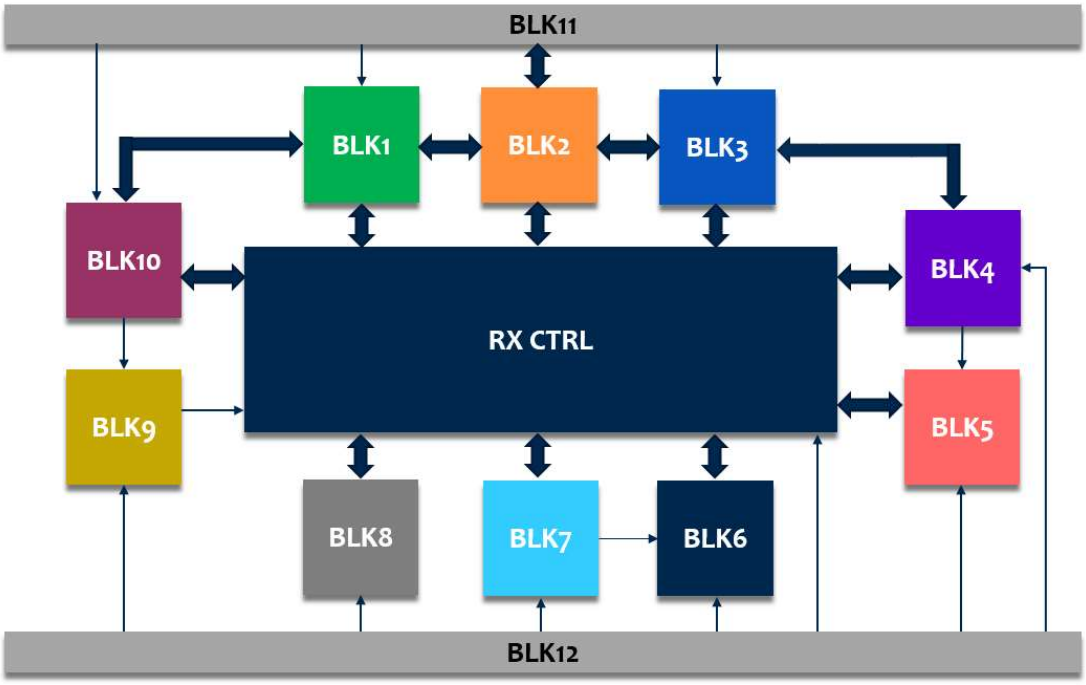


Fig. 3. Wireless SoC PHY layer sub-system

The following sections describe the process of deploying the three steps of the architectural formal verification methodology described in section II.

A. Block-Level Architectural Modeling

Block level architectural models were developed using a finite state machine (FSM) to track the control state of each module. A set of SVA properties defined the output behavior of the block based on the inputs and the internal state.

The AM for BLK8 forms a simple example as shown in Fig. 4. The AM has only two internal states, IDLE and ACTIVE. In the IDLE state, the block waits for the *end_packet* input to assert. Then it transmits the data within a variable time-window after activation, asserts the *blk8_complete* output pulse and returns to the idle state.

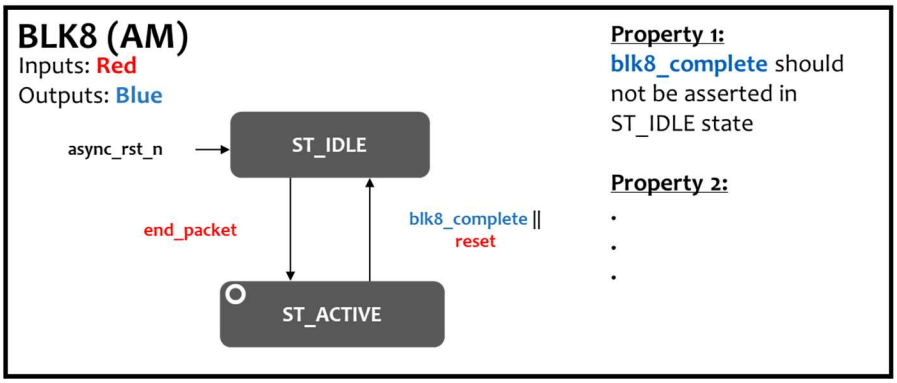


Fig. 4. BLK8 Architectural Model

The AM for BLK4 is depicted in the state diagram of Fig. 5. The FSM is composed of seven states. The states with the circle markings indicate the states in which the block has "the ball". In these states, control of forward progress has been passed to this block from a neighboring block and the neighboring block is waiting for control to be returned. The block outputs marked with up and down arrows indicate a positive or negative edge transition of the signal.

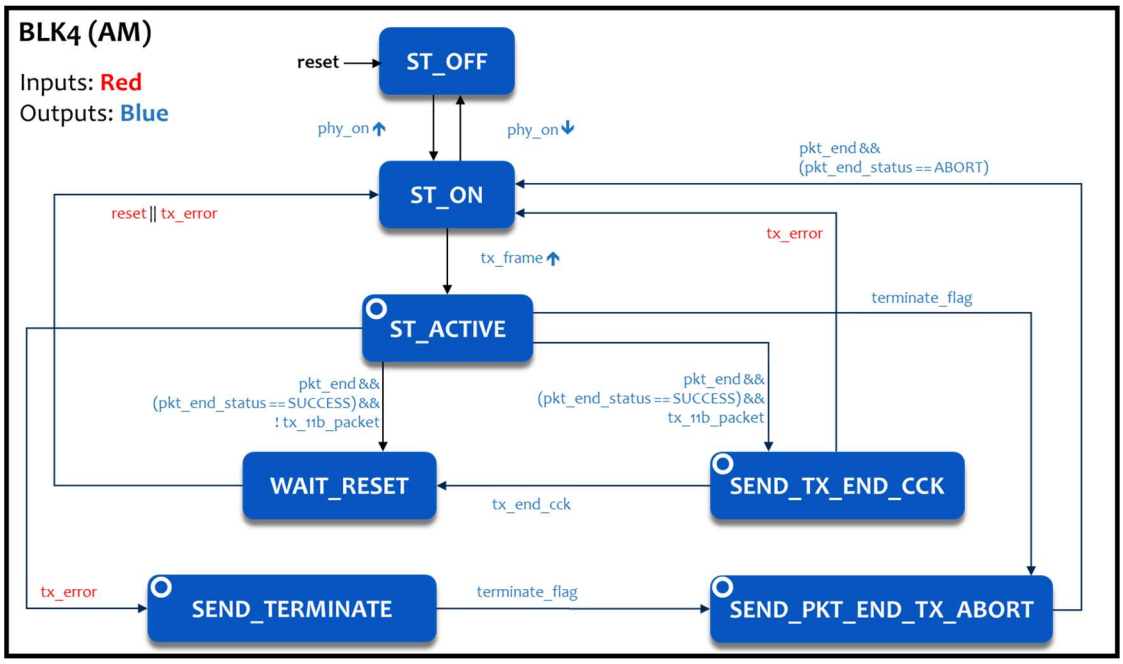
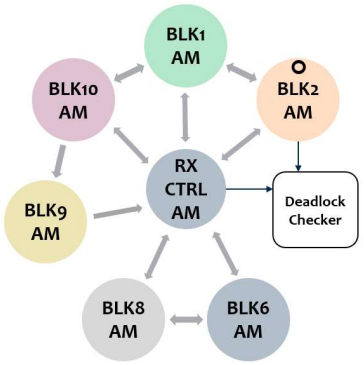


Fig. 5. BLK4 Architectural Model

B. System-Level Requirements Verification

The complete system of AMs was then assembled and end-to-end checkers for deadlock were developed as shown in Fig. 6. The system allows for all possible wireless packet control flow operations with communication between the blocks and the central controller.



Note: For simplicity, a subset of AMs is shown

Fig. 6. System of AMs and End-to-End Checkers

Fig. 7 describes one possible path of packet control flow through the system. The system-level deadlock checker was implemented as a safety property. We used a counter which started at observation of the start of a new frame i.e. posedge of sm_findnxtframe, shown as Step 1 in Fig. 7. The counter counts up until one of the following happens, shown as Step 8 in Fig. 7:

1. The frame is completed successfully and a request to start searching for frames is made
2. The frame is terminated prematurely due to an error

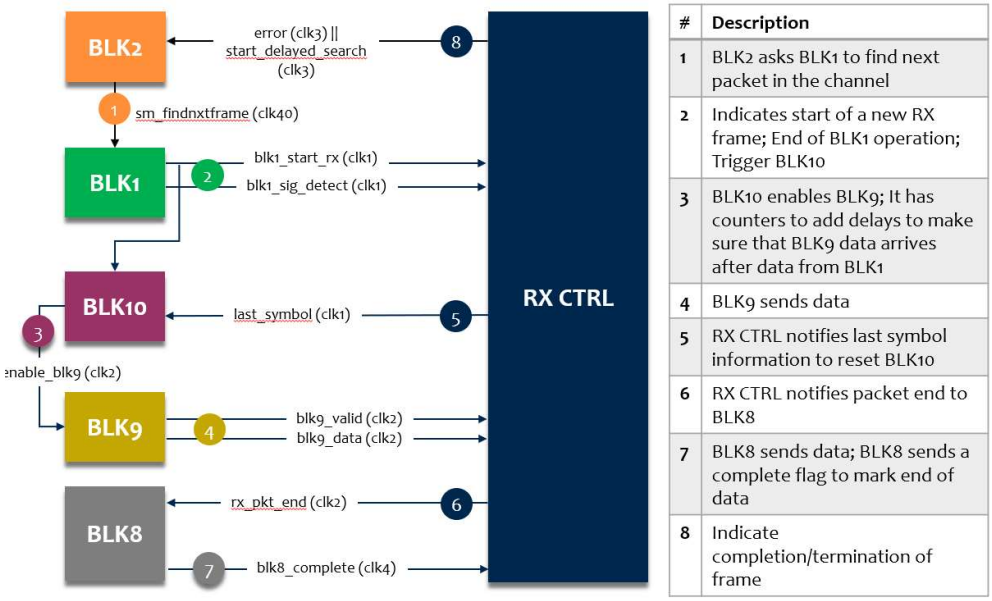
There is an upper bound on the length of a WiFi frame, therefore the frames are expected to be completed, and the system is expected to transition from Step 1 through to Step 8, within a limited time. We used an assertion to check that the counter never crossed this upper limit. The real upper time limit on the frame would be too long for formal to handle, since it is on the order of microseconds. The time taken is distributed across different blocks, therefore, we scaled down the upper time limit proportionally for all the blocks. This abstraction reduced the limit to the order of a couple of hundred clocks for the system level deadlock checker. The below code shows how the counter and the assertion were implemented:

```
localparam TIMEOUT = 200;

reg [$clog2(TIMEOUT) : 0] counter;

always @(posedge clk) begin
   if (rst) begin
      counter <= 'd0;
   end
   else begin
      // Reset counter on arrival of either of the following
      // 1. error
      // 2. start_search
      if (error || start_delayed_search_ofdm) begin
         counter <= 'd0;
      end
      else begin
         if (tx_frame) begin
            counter <= 'd0;
         end
         else if (sm_findnxtframe) begin
            counter <= 'd1;
         end
         else begin
            counter <= (counter == 'd0) ? 'd0 : (counter + 1'd1);
         end
      end
   end
end

phy_deadlock_a: assert property (
   @(posedge clk) disable iff (rst)
   (counter < TIMEOUT)
);
```



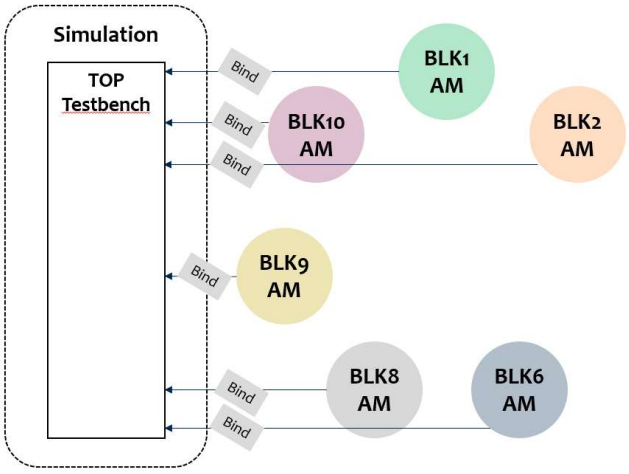
#	Description
1	BLK2 asks BLK1 to find next packet in the channel
2	Indicates start of a new RX frame; End of BLK1 operation; Trigger BLK10
3	BLK10 enables BLK9; It has counters to add delays to make sure that BLK9 data arrives after data from BLK1
4	BLK9 sends data
5	RX CTRL notifies last symbol information to reset BLK10
6	RX CTRL notifies packet end to BLK8
7	BLK8 sends data; BLK8 sends a complete flag to mark end of data
8	Indicate completion/termination of frame

Fig. 7. Example Packet Control Flow

Formal coverage was used to ensure that the depth of bounded proofs for the system-level checkers was sufficient for sign-off on these design properties [4]. One of the key inputs to the process of determining the Required Proof Depth (RPD) comes from measuring the maximum depth of reachability witness waveforms for the functional and code coverage goals in the design.

C. Block-Level Implementation Verification

The final step was to verify the RTL implementation against the flow-control contracts for the blocks that were assumed in the AMs. AMs were used either in the top-level simulation environment or in formal verification at the block-level. Binding the models to the RTL in the simulation environment required very little effort. A trade-off had to be made between exhaustive verification in formal versus the extra time required to create the formal testbench. Fig. 8 shows how the AMs were used in the top-level simulation environment.



Note: For simplicity, a subset of AMs is shown

Fig. 8. Verifying RTL Matches the AMs in Simulation

Due to the simplicity of the AMs of most blocks, using them as a checker in the top-level simulation environment was deemed to suffice. However, a few of the more complicated blocks, such as the main controller (RX_CTRL) were verified in formal. Fig. 9 shows the formal verification environment for the RX_CTRL block.

We used counter abstraction methods for the packet length and wait counters when formally verifying RX_CTRL. This method abstracts the 2^n-state graph of an n-bit counter to a few states, e.g. 0, 1, at-least-one, at-leastzero [5]. This shortened the sequential depth of otherwise deeply embedded corner-case states and allowed us to cover them with formal analysis.

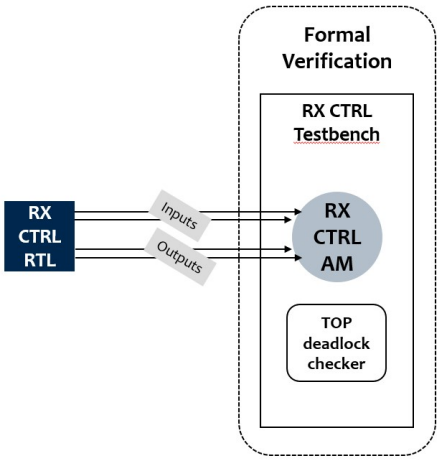


Fig. 9. Verifying RTL Matches the AMs in Formal Verification

Once again, we used formal reachability coverage analysis to validate the RPD of the block level checkers.

IV. RESULTS

The architectural formal verification process uncovered design issues during each of the three major steps of the flow.

During the first step, simply the act of creating the abstract block-level models exposed some architectural specification issues, even before formal verification was run on the models. This is typical of any process that takes a design description and captures it in executable form. For example, this effect is often observed when documenting design intent with assertions. In this case, we found that the PHY specification clearly described the recovery mechanism for shutdown during reception of an Rx frame. However, the recovery mechanism following shutdown during transmission of a Tx frame was omitted. Our investigation uncovered a system-level inconsistency that spanned the function of both the MAC and PHY, since the design of the MAC falsely assumed that shutdown during Tx frames would never occur.

During the second step of formally verifying the system of AMs, numerous bugs related to system-level deadlock were discovered. For example, the counterexample in Fig. 10 shows how a sequence of events causes the main controller not to be notified when an 802.11b packet is terminated. The RX_CTRL block becomes stuck in a wait state that will never terminate, resulting in a deadlock situation in the PHY.

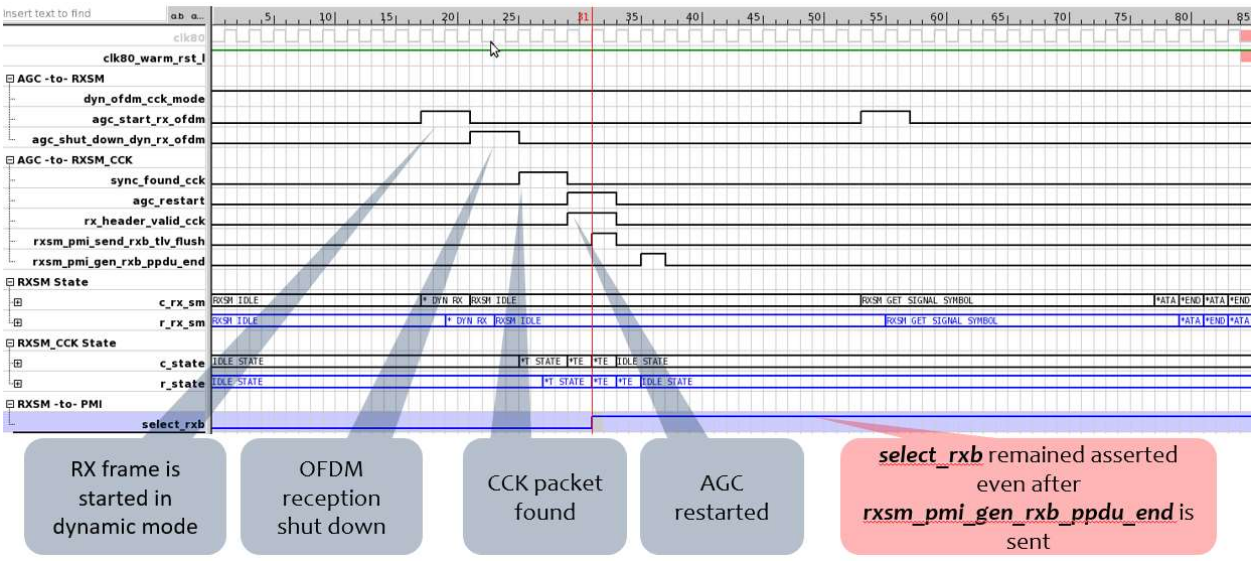


Fig. 10. RX_CTRL not Notified of Terminated Packet

During the third step of testing the RTL models against the AMs, the formal verification process uncovered additional bugs. Fig. 11 shows one example of a corner-case bug that was discovered in the RX_CTRL block. Here, the packet end and bf_fail conditions occur at the same time during transmission, causing the RX_CTRL to become deadlocked in the TX state.

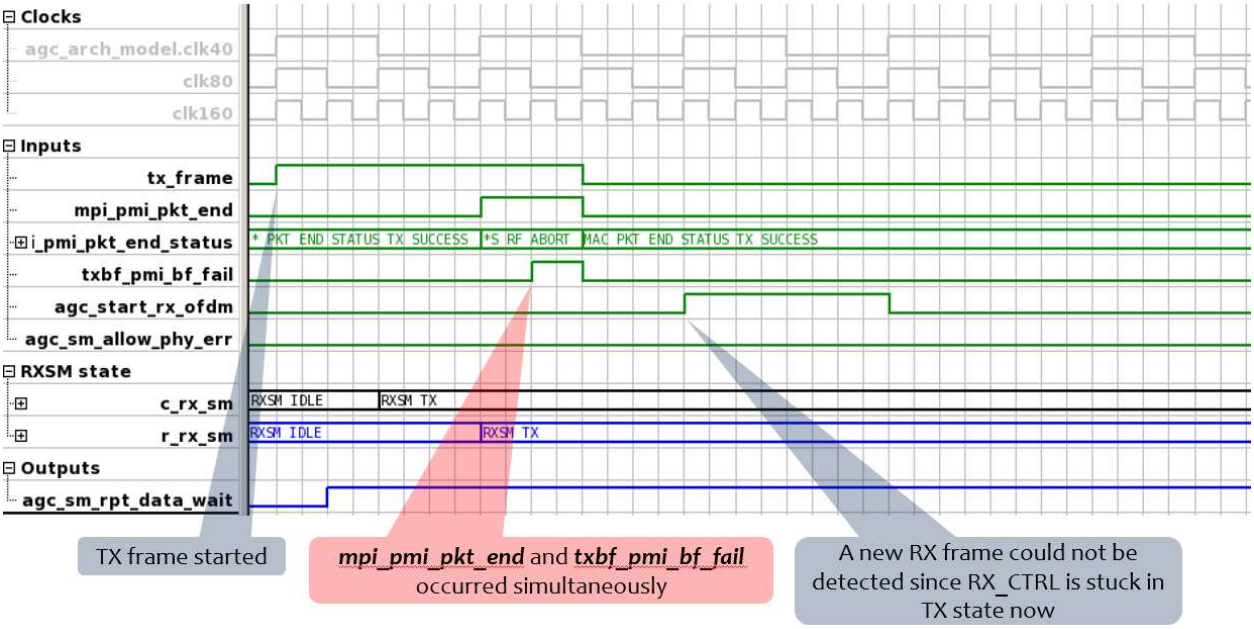


Fig. 11. RX_CTRL Deadlocked in the TX State

V. CONCLUSIONS

Verification of system-level requirements is a critical challenge for today's design teams. The problem is not well addressed by traditional verification methods and subtle bugs that slip through to production can have disastrous consequences. In this paper, we have described an architectural formal verification methodology which addresses these concerns.

Architectural formal verification is the next big leap in the evolution of formal applications. It can be performed at an early phase of design development, before RTL coding has begun, which improves the quality of the architectural design. Once the RTL code is developed, the RTL models can be tested against the contract required of them at the system-level, which improves the quality of the design implementation.

The methodology was deployed on a wireless SoC PHY layer sub-system. Design quality was improved in a measurable way as nine difficult to find bugs were discovered, many of which could have led to an unplanned chip re-spin.

ACKNOWLEDGMENT

The authors are grateful for the support of Cedric Choi, the designer of RX_CTRL.

REFERENCES

[1] A. Aziz, V. Singhal, and R. Brayton. “Verifying Interacting Finite State Machines: Complexity Issues.” Technical Report UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1993.

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

[3] IEEE 802.11 https://standards.ieee.org/about/get/802/802.11.html

[4] N. Kim, J. Park, H. Singh, and V. Singhal. “Sign-off with Bounded Formal Verification Proofs”, DVCon 2014.

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