The future of formal model checking is NOW!

Leveraging formal methods for RAPID System On Chip verification

Ram Narayan
Hardware Advanced Development
Oracle Labs
Austin, Texas. U. S. of A.
ram.narayan@oracle.com

***Abstract*—This paper describes the experience of using formal model checking for verifying portions of the RAPID System On Chip (SoC). We describe the evolution of the RAPID verification plan to leverage formal methods. We outline some of the infrastructure that was developed to make this a productive effort. This paper documents the results and impact of these approaches on RAPID. The purpose of this paper is to encourage readers to explore the use of formal verification for their projects.**

Keywords—formal; verification; SoC; System On Chip; model checking

I. INTRODUCTION

A. SoC Verification

The goal for any hardware verification project involves answering these two seemingly simple questions [8]:

- Does the design work?
- Are we done?

The success of all projects hinges on the ability to answer these questions adequately in a timely manner. Neither the quality nor the timeliness is an easy target in its own right. When combined together as an objective, they present a significant challenge to the best in the industry.

Verification of System On Chip (SoC) designs have some unique challenges posed due to design sizes, number of design units (IPs), complex system interconnect fabric, multiple embedded processors, multiple power and clock domains, analog and mixed-signal content and the software layer that controls much of the behavior on the chip. The stringent time to market demands do not make the challenge any easier.

The approach to verifying SoC's is often a fragmented one.

- IP verification to verify the design units at the block level
- Interconnect verification to verify the system interconnect
- SoC level verification to verify the interplay between the different units and other system functions
- System integration to co-verify hardware and the software functionality and performance.

This verification challenge has fostered significant innovations and collaboration in the industry over the last two decades. Significant among those is the emergence of standards such as SystemVerilog (SV), SystemVerilog Assertions (SVA)[9] and Universal Verification Methodology (UVM)[6].

B. Formal Model Checking

Assertion-based verification techniques [2] have enabled design teams to not only enhance their productivity in simulation debug, but also enabled them to explore formal solutions to solve verification challenges that would otherwise take an inordinate amount of time with simulation. Formal verification is steadily gaining acceptance among design teams. However, only a quarter [1] of the design teams take advantage of these methods. Perception that these methods are complex to adopt and need special skills and expertise to comprehend and adopt is prevalent. In addition to formal model checking, we are witnessing the introduction of automatic formal checking [11] in the industry. These solutions enable the use of formal methods to reveal design issues without the need for any intent specification (assertions) or simulation testbenches.

There are two strategies for applying formal model checking [13] to any design to be verified. The Assurance strategy relies on solely formal model checking to thoroughly verify the unit. This entails proving adequate properties about the design to claim completely correctness. This strategy is applicable for units whose functionality can be completely specified using assertions. Formal model checking can also be applied with a Bug Hunting strategy with the purpose of finding bugs in the design. This strategy is typically targeted at corner cases in the design that are harder to cover with simulation. In this case we do not necessarily try to cover the entire state space of the design with properties. So, the bug hunting approach is often used in conjunction with simulation to completely verify the unit.

C. Project RAPID

Project RAPID is a hardware-software co-design initiative in Oracle Labs that leverages a heterogeneous hardware

architecture combined with architecture-conscious software to improve the energy efficiency of database-processing systems. This paper is a by-product of using formal verification to verify parts of the RAPID SoC.

D. Paper Organization

Section II discusses the evolution of the RAPID formal verification plan from an opportunistic one to a planned deployment of the technology. The infrastructure developed to sustain the entire formal flow is described in Section III. In Section IV, we talk about our experiences with applying formal tools with multiple strategies on RAPID. The results of those experiences and the impact on RAPID are shared in Section V. Finally, this paper concludes by sharing some insights learned from this project in Section VI.

II. Planning for Formal Verification

The initial RAPID verification plan largely relied on a constrained-random simulation environment developed using UVM. Most newly designed units had their own unit level verification environment to thoroughly verify the unit prior to integration into the RAPID SoC Verification Environment. The SoC verification was largely used to verify the interaction between the different units under various operating modes.

A. Initial Outlook towards Formal

At the outset of the project, there were no specific plans to use formal verification on RAPID. We decided to explore the use of formal methods to verify the connectivity [10] at the SoC level. This included

- Connectivity of events like interrupts across the SoC
- Connectivity of Design for Test (DFT) signals across the SoC

Our goal was to catch trivial design errors through formal methods without having to rely on lengthy and in some cases, random SoC simulations. We would have been satisfied if we just verified these SoC connectivity checks with formal tools. We did not have any infrastructure in place for running any formal tool on the project. None of the members on the team had any noteworthy experience with using formal verification tools. Needless to say, our initial expectations were modest.

B. Focused Plans for Formal Verification

Our modest expectations quickly grew wings with the early success of the SoC connectivity checks and we became more ambitious. We decided to explore the use of formal verification to verify some of the custom IP being designed for RAPID. Each of these units were targeted with a different formal strategy depending on design complexity.

1) Assurance Targets

Some of these units, (Event Count Monitor (ECM), System Interrupts Controller (SIC), Least Recently Used Arbiter (LRU) and SRAM Parity Error Register (SPE)) fit the criteria for good candidates[5] for formal verification very well. It was very reasonable to expect to be able to prove the complete functionality of these units formally. We decided to target these units with the Assurance strategy. In addition to proving these units with formal verification, we decided to add a few SoC simulation sanity tests to demonstrate the operation of these units in a broader scope. These simulations were largely to verify the access to these units as specified in the RAPID register map.

The initial architecture for the SRAM Parity Error Register (SPE) did not lend itself well to applying formal methods. The distributed nature of the SRAMs would have warranted the verification of this unit to be distributed across multiple scopes and the development of error injection mechanisms along with the UVM-based verification environments. This was budgeted to consume close to 6 weeks in our verification schedule. Encouraged by the success of formal verification on previous units, we modified the architecture of this unit to make it better suited for Assurance with formal verification. This included partitioning the register function from the error detection and propagation functions. We reduced the time to verification closure to 2 weeks.

2) Partial Assurance Targets

The SRAM controller (SCR) interconnect protocol was more complex and would have made it harder to get thorough formal proofs. The logic beyond those interfaces, however, made a great target for the Assurance strategy. We decided to verify the core logic of the SCR with formal methods using an Assurance strategy and resort to simulation to verify the interconnect logic using SoC tests. These SoC simulations would not need any additional unit level infrastructure and was a fairly small effort.

3) Bug Hunting Targets

The Fuse Controller (FUSE) had two interfaces to control the access to the Fuse array. One is a peripheral bus and the other is an interface (BISR) to the Built In Self Test logic on the SoC. SoC simulation tests were used to verify the functionality of the FUSE through the peripheral interface. We wanted to find bugs in BISR interface well before the BIST logic and the rest of the SoC was ready. We were not very confident about this goal since the FUSE had a state machine that exceeded 500 clocks in depth. Nevertheless, we decided to pursue a Bug Hunting strategy for this unit.

The Memory Interface System (MIS) would have been a good target for the Assurance strategy. However, schedule and resource constraints prevented us from exploring this option. The Bug Hunting strategy was adopted to augment the simulation based methods to accelerate the path to verification completeness.

C. Time to Closure

Our goals for every unit we targeted with formal verification were:

- Find bugs early
- Find all the bugs in the logic targeted by the strategy

While most other units on RAPID had intermediate milestones for verifying basic functionality, the unit we targeted with formal verification had just the one milestone of

verification closure. Unlike traditional simulation based methods, we did not have any code or functional coverage to track if we were completely verifying the unit. A big concern was "How do we know that the assertions are adequate to make claim for verification closure?" We mitigated this risk with diligence and reviews. Table I. shows a summary of the formal verification plan for RAPID. For each unit, we try to outline the formal strategy used, the role simulation played to complement the formal strategy and the time it took us for executing the formal strategy. For the units targeted with Assurance, we only ran some sanity simulations to demonstrate the system behavior or verify the correct system address mapping of the registers in the unit in the SoC. For the SCR, we did relied on simulation as the primary means to verify the interconnect logic. The Bug Hunting targets had their own simulation based verification environments prior to the application of formal verification.

TABLE I. RAPID FORMAL VERIFICATION PLAN

Unit	Formal Strategy	Simulation Role	Time for Formal
SoC	Connectivity	Sanity System Test	1 week
SIC	Assurance	Sanity System Test	2 weeks
ECM	Assurance	Sanity System Test	2 weeks
LRU	Assurance	None	1 week
SCR	Partial Assurance	Primary (interconnect)	3 weeks
FUSE	Bug Hunting	Primary (pre Formal)	1 week
MIS	Bug Hunting	Primary (pre Formal)	4 weeks
CCU	Bug Hunting	Primary (pre Formal)	3 days
SPE	Assurance	Sanity System Test	2 weeks

III. INFRASTRUCTURE

As the application of formal verification grew on RAPID, it was evident that we needed an efficient environment for property specification, formal compile, formal execution, debug and regression. We developed a python based environment to perform the aforementioned tasks and more. Each formal test was specified in Python with the ability to embed raw SystemVerilog code.

A. RAPID SVA Property Library

We defined an SVA property library that was adequate for over 99% of the assertions specified on RAPID. The library included some of these patterns:

- connect_p: Property to verify connectivity between two points in the design. In addition to verifying connectivity, this property is also used for equivalence checks between reference models and the design being verified.
- cond_p: Property to verify conditional connectivity.
- imply_p: Implication property with a user specified delay from the antecedent to the consequent for each instance of the property.
- imply_range_p: Implication property with a user specified window within which the consequent is expected to occur.
- eventually_p: Similar to the imply_p, but the consequent can be specified to occur any time after a specified delay after the antecedent. This is used for specifying liveness properties.
- stable_p: Property to assert that the specified condition did not change beyond some event. This is useful for constraining the programmable control registers in the units from changing during the formal properties. The formal tool is still free to program them in an unconstrained manner at the beginning of the test.
- mutex_p: Property to verify mutex properties between a set of events.
- prop_p: Generic property where the user could specify any SystemVerilog expression to be asserted.

B. Python Property Constructs

For each of these patterns, we defined a Python construct that the user specified in the Python test file. The Python test file would be executed to generate a SV "checkers" module containing SVA assertions that was bound to the top level of the design under test. Properties that need custom SVA properties are modeled in a side SV file which is included inside the checkers module. This side file is also used to code reference models to aid the process of writing properties. This flow took full advantage of the Python language and made it possible to specify a large number of assertions in a very concise manner. This also eliminated the learning curve for users who were not very familiar with SVA.

Each property specified in the test has the ability to be promoted to the RTL simulation environment. The user is able to specify a "promote" field for each property to enable the property to be a part of another checkers file which is bound to the design in the RTL simulation testbench. This is useful for units with a Bug Hunting strategy. Completely proven properties were not promoted to simulation. This framework also made it convenient for specifying assertions just for simulation.

For each construct, in addition to the expressions for the antecedent and/or consequent and the delay between them, we had the ability to specify the intended purpose for the property. The specified directive can be "cover" or "assume". If no directive is specified, the property is treated as an assertion to be proven. A "cover" directive results in the property being a target for functional coverage. While the formal tool is not a tool for coverage closure, it provides great value in quickly identifying "uncoverable" properties, thereby saving time for the verification engineer trying to achieve closure in simulation. All the "cover" properties are promoted to simulation. The "assume" directive is used to create the SVA property as an assumption. The formal tool treats this as a constraint, and the simulation tool treats this as an assertion to be checked in simulation. Almost all "assume" properties are promoted to simulation. Some of the assumptions are merely

intended to reduce the complexity of the state space for the formal tool and are not promoted. These convenience assumptions are carefully reviewed by the designers to ensure the efficacy of the formal proofs. Fig. 1 and Fig. 2 show examples of a few Python constructs and the resulting SVA code in the checkers module.

```
def_imply (
  name="NO_GRANT_WITHOUT_REQ",
  A="request == '0",
  B = "grant == '0",
  delay=0,
)
```

```
`ifndef DISABLE_NO_GRANT_WITHOUT_REQ
// Source file/line: /import/workdir/user/formal/lru/checks.py +56
NO_GRANT_WITHOUT_REQ: assert property (
  imply_p (
  .clk(lru_clk),
  .rst(!lru_reset_n),
  .inA(request == '0),
  .inB(grant == '0),
  .delayC(0)
  )
) else `formal_error("%m : Cannot have GNT without a REQ")
`endif // ifndef DISABLE_NO_GRANT_WITHOUT_REQ
```

Fig. 1. Example of Python property construct (assert)

```
def_stable (
  name="STABLE_COUNT_START_SEL",
  A="count_start_sel",
  delay=1,
  assume=True,
  promote=True )
```

```
`ifndef DISABLE_STABLE_COUNT_START_SEL
// Source file/line: /import/workdir/user/formal/ecm/checks.py +39
STABLE_COUNT_START_SEL: assume property (
  stable_p (
  .clk(ecm_clk),
  .rst(!ecm_reset_n),
  .inA(count_start_sel),
  .delayC(1)
  )
)else `formal_error("%m : count_start_sel must be stable while counting")
`endif // ifndef DISABLE_STABLE_COUNT_START_SEL
```

Fig. 2. Example of Python property construct (assume)

In addition to specifying the properties, the Python tests also contain:

- Design information (clock/reset/parameters)
- Memory and time limits for configuring the formal run for the server farm
- Other options to pass on to the vendor tool
- Options to add SystemVerilog reference models to the formal testbench.

C. Formal Regressions

The formal tests are run regularly to ensure that the correctness of the designs is maintained through the course of the project. In the world of simulation, the pass/fail status of a simulation can be determined fairly easily. This becomes a little trickier when it comes to formal runs. The outcomes of formal model checking [2] range from vacuous properties to failed properties to bounded proofs to complete proofs. Most vendor tools further classify the properties based on the formal results. A fairly common classification includes [12]:

- Proven: Unbounded proof
- Fired: Disproved assertion which needs to be debugged
- Inconclusive: Bounded proof
- Vacuous: Unjustifiable antecedent
- Possibly Vacuous: Proven property, but antecedent justification still inconclusive
- Covered: Covered "cover" property
- Uncoverable: "cover" property that cannot be covered

A formal test in our environment constitutes a number of properties targeted at the same design under test. So, in our RAPID formal regressions, we save a golden result for all the properties in a test. All subsequent runs are compared to the golden result. If the previously proven properties are no longer proven, the test is deemed a failure. Similarly, if a previously covered property is not covered in the current run, the test is considered a failed test. When we have some previously unproven properties being proven and all other properties maintain their previous status, the current results become the new golden standard for the test. It must be noted that it is possible for some of the properties to be fired in the golden result. This enables us to establish a regression environment through the development and debug stages of the design and formal tests. Our goal is to be able to perform formal regressions in the same manner we run simulation regressions. Fig. 3 shows a flow diagram of a subset of the infrastructure for sustaining formal verification on RAPID.

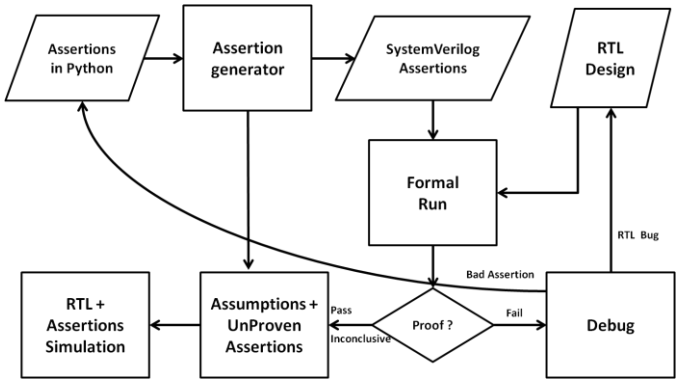

Fig. 3. RAPID Python based formal flow

IV. RAPID FORMAL EXPERIENCES

This being the first formal verification experience for the team, the infrastructure and practices evolved to meet the needs of the different units.

A. SoC Connectivity

SoC Connectivity checks were written to verify the correct connectivity between critical SoC events like interrupts. These checks are trivial to define and are of high value. Proving these connections saved us significant cycles in SoC simulations.

SoC Connectivity checking also included Boundary Scan (BSR) connectivity tests to prove drive, sample and high impedance properties of each I/O cell. The RAPID Nodewatcher functionality was also targeted with formal verification to verify the connectivity of thousands of internal signals to a selectable set of I/O pins. These are conditional connectivity checks based on the configuration of the Test Data Registers (TDR). TDR related checks included properties to verify the JTAG overrides that cause the RAPID clock control unit to bypass some states or pause. Some of these SoC checks went beyond just the point to point connection between SoC events and verified the correct configurability and functioning of certain global functions on the SoC.

To make the SoC Connectivity proofs easier for the formal tool, we specified directives to the formal tool to blackbox most of the units in the SoC. This reduced the time to prove the connectivity between these units significantly. In the absence of these blackbox directives, the formal tool would have had to justify the generation of '1' and '0' at the source of the connections.

B. IP Assurance

The System Interrupt Controller is the global interrupt router on RAPID. It routes interrupts from N different sources to M different destinations. The enablement and clearing of interrupts is managed through programmable control registers. The status of the interrupts is also available through interrupt status registers. These control and status registers are accessible through a system interconnect. The incoming interrupt from each source can be configured to be rising edge, falling edge, active level high or active level low. This behavior can be configured through the control registers. **Error! Reference source not found.** shows a generic block diagram of the SIC. This diagram is typical of most units in an SoC.

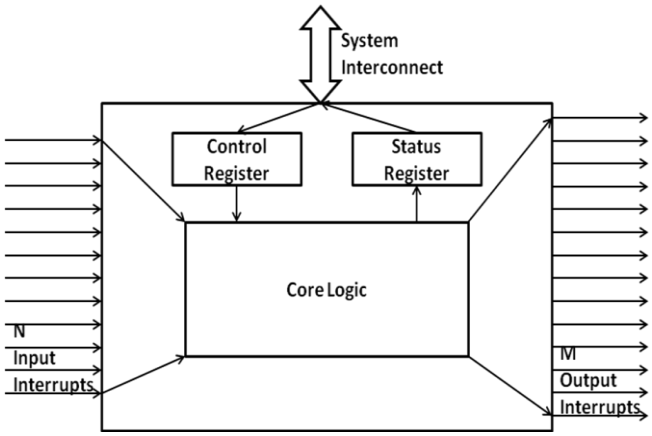

Fig. 4. System Interrupt Controller Block Diagram

We can partition the unit verification into two areas.

1) Register Access Verification

The questions that are answered here are:

- Are the control registers being written to correctly through the system interconnect?
- Are the status registers being read from correctly through the system interconnect?

This verification requires the system interconnect interface to be constrained to ensure that the formal tool only generates legal transactions. We took advantage of vendor provided constraints to constrain these interfaces. The verification IP also included checks to ensure that the SIC adhered to the protocols of the interconnect. We developed some sequences and properties to be able to write to and read from the registers based on the interconnect protocol. These sequences accounted for any wait states in the protocols and did not constrain the response latencies from the slave at all. We used these properties to prove that a write to each address specified in the architectural spec for the unit caused the appropriate control register in the design to receive the data that was written. Reserved bits were masked from the comparison. Similar properties were used to ensure that the data in the status registers were read correctly. The status registers were constrained to hold a stable value during the read protocol to prevent the hardware from writing to them and causing the read properties to fail.

In the SoC context, we added simulation tests to ensure the correct channeling of requests through the SoC system interconnect fabric to these registers. This was done more for SoC verification and less for the unit verification. While these interconnect protocol properties were easy to implement for some interconnects like the one in the SIC, it was not a trivial approach for more complex protocols. In those situations, we just relied on simulation to verify the correct functioning of the registers. Fig. 5 shows the formal testbench for the SIC unit.

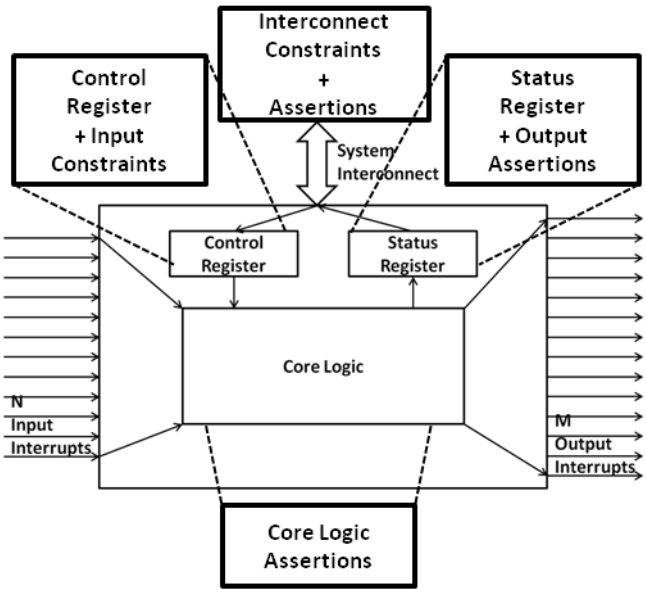

Fig. 5. System Interrupt Controller Formal Testbench

2) IP Core Logic Verification

The questions that are answered here are:

- Are the control registers being correctly used by the unit as per the design specification?
- Is the design being reset correctly?
- Are the inputs being interpreted correctly?
- Are the outputs being generated appropriately?
- Are the status registers being updated correctly?

The Register Access Verification did not verify the intent of the registers or the architectural appropriateness of the data written to the register. However, it did establish controllability and observability of the registers in the unit from its interface. The IP core logic verification could now safely use the control registers as inputs to properties on the rest of the logic they drive. In additional to these registers, we chose a few internal nodes in the design as observation and control points in our properties. These points gave us additional controllability and observability to the design and reduced the complexity of the cones of logic being analyzed around them. We proved the correctness (observability) of these points prior to enjoying the benefits of using them (controllability) for other properties. The formal tool ensured the non-vacuity of all the properties by ensuring that the antecedents are reachable by stimulus involving only the primary inputs. This approach made it easier to write properties on the entire unit without any compromise on the efficacy of the overall unit verification.

While defining the set of properties to verify the core logic, we had to constrain the control registers to allow only legal values as defined in the architectural spec. These constraints were promoted to the RTL simulation environment to ensure that they were not violated in the system tests that used this unit at the SoC level. If the other inputs to the design needed to be constrained, assumptions were added and promoted accordingly. Additional assertions around internal nodes were used as needed for convenience with the same discipline as when using the control registers for properties. Exhaustive properties were written to ensure the correctness of the primary outputs of the unit and the correct updates to the status registers. To be thorough in our verification, we added checks to verify the reset state of some key registers as specified in the micro architecture. The Python infrastructure described earlier made this entire process of generating assertions seamless and was critical to our successful deployment of the methodology.

The Event Counter Monitor and SRAM Parity Error Register experiences were similar to that of the System Interrupt Controller. An additional aspect of the ECM and SPE verification was the reuse of the core logic units in both the cases. The formal testplan for the SPE and ECM employed a hierarchical approach with a mix of assurance of reused design blocks and checks to verify their connectivity across the SoC. To handle that, we added connectivity checks between the independently verified units and/or the unit registers. In the case of both the SPE and the ECM, designing for formal verification was a consideration in the architecture of the units.

The Least Recently Used Arbiter is a parameterized unit to grant access to the least recently granted requester. To verify this unit, we wrote a few standard properties to ensure that multiple requestors were not granted simultaneously and at least one requestor was granted. To verify the correctness of the LRU scheme, we wrote a simple reference model in synthesizable SystemVerilog and verified that the requestor granted by the designed unit matched that by the reference model. We proved fairness and bounded waiting properties for the LRU. This exercise eliminated the need for functional coverage in multiple units in the SoC where this LRU arbiter was used.

The SRAM Controller interfaces to a more complex system interconnect that is not as convenient for verifying formally. We used a vendor provided monitor to constrain the interface. The core of the SRC was rigorously verified using formal properties. The SRAM controlled by the SRC was blackbox'ed and the correctness of the control, address and data inputs into the SRAM were verified to be correctly generated or calculated by the controller logic. As discussed earlier, we used internal nodes (outputs of the interconnect logic) as inputs to the properties describing the intended core behavior. So, while some interconnect verification was achieved as a byproduct of the core logic verification, it was not adequate to claim the interface verified. The verification of the register accesses was done through SoC level simulation tests. In addition to the register accesses, we deferred the verification of the protocol along with its corner cases to simulation tests at the SoC level. Since accesses to the SRAM are defined in the system memory map, these tests were fairly trivial to write.

While most of the bugs manifested as firings of the properties targeted, some of them were revealed due to vacuous proofs or design check violations that were reported by the formal tool as a by-product of the model checking. The inability of the formal tool to justify the antecedent of the property could be due to an undriven signal or some other condition that would prevent the property from being proven or

fired. Such failures may be trickier to debug because of a lack of a waveform to describe the failure. From our experience, it is important to not ignore them. They are likely masking a bug. It would be beneficial to identify some of these bugs earlier concurrently with the design process using automatic formal checks.

C. IP Bug Hunting

Our objective in this strategy is clearly to drive to verification closure by finding the remaining bugs in the design as fast as possible. Emphasis is less on complete proofs and thorough verification of the unit through a set of formal proofs. In this approach, bounded proofs are tolerable. Such a strategy always complements simulation or is complemented by simulation. Traditional simulation-based code and functional coverage become the metrics for verification closure. Formal verification is just a catalyst to accelerate to closure. On RAPID we applied the Bug Hunting strategy in three different situations.

1) Fuse Controller

The FUSE unit has a deep state machine and hence we did not target this with the IP Assurance strategy. Besides, most of this unit was verified in the early stage of the project before formal verification had made its mark on RAPID. The FUSE unit was largely verified through its system interconnect interface using SoC level simulation. Through those simulations, we had verified the ability to program and read the fuse array. The BISR interface to the FUSE was yet to be verified. Our goal was to iron out this access mechanism to the FUSE unit prior to the BIST and BISR interface being designed.

We just wrote two properties to verify that fuse array read and write requests through the BISR interface would be completed within the expected number of cycles. The read/write operations take over 500/1000 clock cycles respectively. We were a little skeptical about the applicability of formal verification to explore the state space of the controller to these depths. These two assertions proved highly valuable by highlighting a few bugs in the unit which could very likely have been missed in probabilistic SOC level simulation. The formal tool highlighted issues in the state machine being able to handle back-to-back reads/writes. These failures occurred around 1000 cycles after reset. Once the bugs were fixed, we were able to get unbounded proofs for these properties. The run times for these proofs were very reasonable as well (less than 2 hours). It would have required us an elaborate random stimulus generator for the BISR interface to probably find these bugs. The SoC environment does not lend the controllability for such random stimulus. At the very least, this formal effort saved us the longer debug times in the simulation environment.

2) Clock Controller Unit

The CCU controls the clock generation to the various units on the RAPID SoC. Towards the end of the project, we found a bug using random SoC simulations in the CCU with one of the clock modes the SoC was expected to operate in. The challenge was to ensure that this bug did not occur in any of the other clock modes. Doing that in simulation would have been impractical. We decided to explore the use formal verification to give us that confidence. We described the property that the bug would have violated and ran formal model checking on the CCU. Through this effort we were able to:

- Confirm the bug that was found in simulation
- Prove that this bug only occurred in the clock mode that was reported in simulation.

That proof gave us confidence in the health of the design and expanded our options to fix or ignore the bug.

3) Memory Interface Subsystem

The Memory Interface Subsystem (MIS) includes a localized Memory Access Arbiter (MAA) and its interface (MSYS) to the system interconnect. The units in this subsystem were already being verified in a unified UVM-based unit level environment. We decided to accelerate the verification closure of these units by using formal verification for finding the remaining bugs. We asserted some general invariant properties about the design. We also implemented properties to assert some corner cases. In order to increase our chances to find bugs, we verified the MAA and the MSYS in separate formal environments. Fig. 6 shows the formal testbenches for the units in the MIS.

Like in the previous units, we used vendor IP to constrain the system interconnect. We strictly followed the Assume-Guarantee [3][4] formal proof methodology in this situation. The assumptions that were made to verify properties for the MSYS unit became targets for the MAA verification. This bug hunting exercise revealed a few design issues including a critical one that potentially saved us a respin of silicon. Although we did not get unbounded proofs for all these properties, we were able to achieve our goals of driving to verification closure on these units.

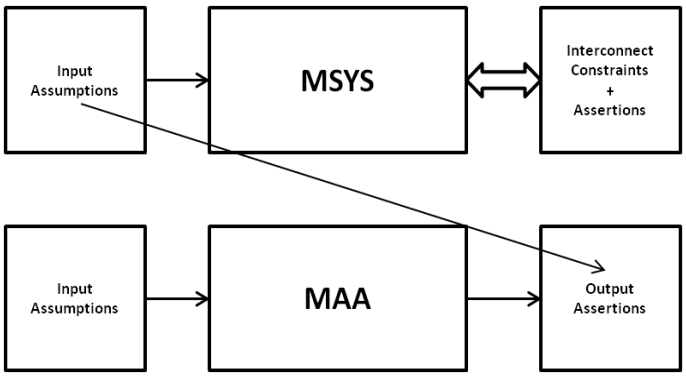

Fig. 6. MAA and MSYS Formal Testbench

Subsequent to the initial bug hunting effort, the verification methodology for this unit has evolved into using formal methods for performing early bug hunting for all the new design features designed. This has proven to be a very valuable exercise with bugs being found within minutes after

the designer has implemented the design. This is as productive as it can get when the design and verification responsibilities are split between different teams.

D. Challenges

We encountered our set of challenges during the deployment of the formal solutions. None of them posed any significant obstacles to the project, though.

1) Confidence

Some designers were very comfortable with conventional simulation based methods. It needed an initial leap of faith to rely solely on formal verification for verifying their designs. One designer commented "*I understand you are going to prove assertions. I would still like to see some simulations to verify the unit.*" This concern is very reasonable given that we did not have a quantifiable way to articulate the completeness of the formal properties. Another willing but skeptical verification engineer commented "*There are reasons we have traditional methods like scoreboards and functional coverage*." Our experiences on RAPID changed the outlook of each of these engineers towards formal verification. The former's response after the formal experience was "*I did not know that formal verification can verify my unit this thoroughly. I understand my design better because of this formal experience*." The latter engineer went on to verifying a complete unit formally within the scheduled time.

2) Assertion Based Verification Focus

The documentation of design intent in the form of assertions and design coverage by the designers can accelerate the adoption for formal verification. The benefits of formal verification can be fully exploited when the designers make a commitment to specifying assertions as a part of the design process.

3) Infrastructure development

As mentioned earlier, the infrastructure for formal verification was non-existent prior to our undertaking. The development of the infrastructure was an evolutionary process.

4) Assertion coverage

The biggest challenge that is still unresolved is a lack of tangible coverage metric to tell us if the set of assertions written is adequate to claim verification completeness for the units. While there has been some work done on this topic [7], we did not have the bandwidth to adopt any of these methods on this project. We alleviated this concern with reliance on design and verification reviews.

V. RESULTS

TABLE II. shows the results of applying formal verification on the different units on RAPID. The table also shares some attributes about the designs as reported by the formal tool. The number of assertions reflects the number of unique assertions eventually run for the unit. These assertions were broken down into multiple tests for each unit. Our environment enables a definition of a hierarchy of tests for a unit and automatically handles the dynamic scheduling of these formal jobs on the server farm. The run times reflect the total amount of time taken to run all the tests for each unit.

TABLE II. FORMAL VERIFICATION RESULTS ON RAPID

Unit	Strategy	Registers	Assertions	Run Time	Bugs
SIC	Assurance	~4200	~3000	3m	5
ECM	Assurance	~5500	~1500	5m	15
LRU	Assurance	~75	~60	120m	2
SRC	Partial Assurance	~2500	~1500	600m	39
FUSE	Bug Hunting	~4500	2	110m	4
MAA	Bug Hunting	~6700	~2000	600m	6
MSYS	Bug Hunting	~9500	~100	120m	2
SPE	Assurance	~350	~150	2m	8

A. Schedule

The results clearly show the effectiveness of formal verification in finding bugs. What they do not convey is the time taken to find those bugs. Each of the units targeted for IP Assurance was verified well within the original time allocated for those units in the original project plan. These units were qualified for SoC integration well ahead of their schedule. The role that SoC tests played in the thorough coverage of these units was also reduced considerably. For e.g. with the formal verification of the SIC and the SoC interrupt connectivity, it was no longer necessary to verify the correct functioning of every interrupt on the SoC level. Such tests would have consumed MxN tests at significant lower SoC simulation performance to cover the entire matrix of N interrupt generators and M destinations. Instead of these exhaustive tests, just a few SoC simulation tests were written to demonstrate the correct system behavior. This was more a system level task. These tests were not expected to find any bugs and they did not. These schedule savings in the RAPID SoC verification plan has not been estimated and is non-trivial.

B. Quality

For all the units that were targeted with the IP Assurance strategy, a few SoC test were written to demonstrate their behavior in a system. Some of these were highly randomized tests subjecting the units to adverse conditions. The only bugs these tests revealed were in the SRC unit where we had deferred the verification of the system interconnect interface to simulation. This has helped develop more confidence in formal verification and our methods among the RAPID design and verification teams.

C. Benefits

Beyond the schedule and quality impact of formal verification on RAPID, we realized a few intangible benefits of adopting a formal verification.

1) Design – Verification Collaboration

Formal verification success fosters a structured approach to IP verification. Structured planning reviews are conducted to review architectural features to be implemented and document key assumptions, assertions and coverage items. This enables

a consistent understanding of the design prior to the design and verification phase. The designers are able to accommodate design for verification as a factor in evaluation design options. With more familiarity with the formal verification, these considerations evolve into design for formal verification guidelines.

2) Organizational Capability

There is a distinct openness in the organization to leverage formal methods. Designers and verification engineers are looking for ways to use formal verification to verify their units. Even if the entire unit is not a good fit, key sub functions are being considered to be verified formally. With a more mature infrastructure in place, we are in a position to execute on these plans more efficiently. It now takes us less than 10 minutes to define a formal testbench together for a new unit and start writing and verifying properties on the unit.

3) Early Bug Hunting

While the benefits of formal verification are well understood in finding hard to find bugs, it has proven very effective and efficient in finding and verifying most of the "simpler" bugs from the design. The formal tools may take a long time to prove complex properties, but often do not take very long to find a counter example to disprove a property. It does not take very long to put together a formal testbench along with constraints and the necessary monitors. Unlike simulation failures, firings in formal verification are very easy to diagnose to get to the root cause of the failure. This can be a big productivity boost in getting to a fully functional unit faster.

4) Documentation Verification

Often, the bugs found in formal verification are a result of inconsistencies between the assumptions made by the designer and the design spec. The properties written for formal proofs are based on the specified intent of the design. There will be situations where the spec has to be modified to clearly articulate the functionality of the design to the system engineers and software developers.

5) Confidence and Understanding

During the course of the formal process, we ended up writing a number of incorrect assertions. The tool will rightfully disprove these assertions and generate waveforms to illustrate the failing pattern. While these experiences may seem a waste of time, there is a sense of confidence in the correctness of the design. These are valuable experiences for the designer and the verification engineer to deepen their understanding of the design and become more adept at verifying their unit.

.

VI. CONCLUSION

Our efforts on the RAPID project and the results have demonstrated the role formal verification can play in an SoC project. We have shown how a team that had fairly limited experience with formal verification has adopted the methodology and leveraged it to achieve the project's goals. We have shown the importance of applying the correct formal for each unit. For some units, it can be applied to completely prove the units' functionality and for others, it can be a bug hunting expedition. It is important to be true to the plan for the unit. Any deviation from the plan can lead to an unproductive effort.

Formal verification is a highly effective and efficient approach to finding bugs. Simulation is the only means available to compute functional coverage towards verification closure. We have attempted to strike a balance between the two methodologies to operate within the strengths of each approach towards meeting the projects goals. We have demonstrated that closer collaboration between the design and verification teams during the pre-implementation phase is essential for maximizing the applicability of formal methods.

What we have achieved on this project is nothing new. These methods have been employed by a number of projects across the industry. We wanted to share our experience to allay prevalent concerns about the obstacles to adopting formal verification. More than skill, this approach requires the right attitude – diligence, perseverance and a pinch of humility.

In our industry, verification is widely accepted as the long pole in the project schedule. 67% [1] of projects do not meet their planned schedules. On RAPID, we did not apply formal methods for the sake of applying formal methods. We applied formal methods because we believed it would help us reign in the schedule for the units we targeted. And, we did. Our role as verification engineers and managers is to certify the quality of the designs being taped out within the schedule outlined for the project. It is our responsibility to use available solutions wisely to make those schedules appear less unreasonable. We encourage verification teams to explore the use of formal (and any other) verification to meet your project goals.

A. Future Work

As we continue to develop formal expertise across our organization, there are some areas that merit some additional exploration.

- Quantify the coverage achieved with the proven assertions on any unit. This will provide a metric to indicate completeness of the set of assertions.
- Use automatic formal checks to eliminate design violations prior to simulation and formal verification.
- Use automatic formal analysis of the design to identify unreachable and uncoverable sections of the design. This will help streamline coverage closure and potentially identify design issues early.
- Augment the formal flow to automate the Assume-Guarantee flow and the promotions of inconclusive assertions to RTL simulations.

ACKNOWLEDGMENT

I salute all the formal users and solution providers who have persevered over the years to make this a viable solution for hardware verification. This paper is the result of the trust and encouragement from my management, Doug Good and Charlie Roth. I would like to thank my colleagues, Steve Burchfiel and Sravya Kusam, for sharing their experiences with using formal on their units. The adaptability of the

designers, Ashraf Ahmed, Philip Sams, John Fernando, Joseph Wright and John Coddington, was important to the successful adoption of this approach. The support and responsiveness from our formal verification tool vendor was critical to our success.

REFERENCES

[1] H. Foster, "The 2012 Wilson Research Group functional verification study," Verification Horizons, April-September 2013.

[2] H. D. Foster, A. C. Krolnik, D. J. Lacey, "Assertion-based design," KAP, 2003.

[3] T. A. Henzinger, S. Qadeer, S. K. Rajamani, "You assume, we guarantee:methodology and case studies," CAV98: Computer Aided Verification, pp440-451, 1998

[4] S. K. Roy, H. Iwashita, T. Nakata, "Formal verification based on assume and guarantee approach: a case study," Proceedings of the ASP-DAC 2000.

[5] H. Foster, L. Loh, B. Rabii, V. Singhal, "Guidelines for creating a formal verification testplan," DVCon 2006.

[6] Accellera Systems Initiative:Universal Verification Methodology, Accellera Standards.

[7] V. Singhal, P. Aggarwal, "Using coverage to deploy formal in a simulation world," CAV 2011: Computer Aided Verification, pp 44-49, 2011.

[8] M. Glasser, "Open verification methodology cookbook,", Springer, 2009.

[9] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, IEEE Std. 1800-2005.

[10] K. Ranerup, M. Handover, "Using formal verification to exhaustively verify SoC assemblies," DVCon 2013.

[11] R. Sabbagh, "The top five formal verification applications," Verification Horizons, Vol. 8, Issue 3, October 2012

[12] Questa Formal User Guide, Mentor Graphics, August 2013.

[13] H. Foster, "Planing for formal ABV success," Verification Academy, Courses: Assertion-Based Verification