Bringing Formal Property Verification Methodology to an ASIC Design

Erik Seligman, Ram Koganti, Kapilan Maheswaran, and Rami Naqib
Advanced Components Division., Intel Corporation
Hillsboro, OR
{erik.seligman, ram.p.koganti, kapilan.maheswaran, rami.a.naqib}@intel.com

Abstract— **The Blackford North Bridge design team was the first chipset project in Intel's Advanced Components Division to seriously utilize Formal Property Verification (FPV). Over the course of the project we learned a lot about the challenges of deploying FPV to an ASIC team. Overall our use of FPV in Blackford was very successful, having helped us find approximately 24 logic bugs, and significantly increased confidence in our design. We have a number of methodology recommendations for future ASIC projects, including early introduction of FPV; the assigning of central FPV owners; FPV-friendly RTL standards; leaving ownership primarily with each DE; and the encouragement of assertion development through density checks. We think that by learning from our experiences and following our recommendations, other ASIC teams will be able to expand their use of FPV as well, for a significant increase in design confidence.**

I. INTRODUCTION

Formal Property Verification, or FPV, is a powerful technique in which properties of a VLSI design are proven mathematically. This contrasts with traditional simulation-based validation, which only applies specific test vectors—FPV proofs are valid for all possible test vectors, theoretically providing 100% coverage for the properties verified. Many design teams have reported finding important bugs using FPV techniques.([1],[2],[5]) We began using FPV on our chipset team for our recent Blackford North Bridge (BNB) chipset design, and based on our experiences, believe that it should become an accepted part of ASIC design methodology.

As one would expect, there are significant challenges in using formal property verification; there has been a lot of debate over whether FPV offers sufficient ROI for most real-life design problems.([3],[4]) Specifying the properties to be proven may require the designer to learn a new specialized language. The tools themselves have been difficult to use, requiring specialized choice of proof engine parameters and involved property decomposition techniques by the users in order to obtain useful results [4]. And once the tool reports that it has disproved a property, it takes detailed analysis by the user to determine if this is a real design bug.

However, recently the EDA industry has moved towards simplifying the FPV process, allowing "push-button" runs to prove embedded assertions in RTL designs. The Foresight tool from Intel's Design Technology team is an example of these trends; in fact, we believe that the current generation of FPV tools are indeed feasible for use by an ASIC group like ACD. It allows the user to supply an RTL model with a set of properties as an input, and uses heuristics to make good choices of proof engine parameters that can handle a wide variety of designs. The Blackford North Bridge chipset design is the first design project in Intel's Advanced Components Division (ACD) to make a serious attempt at using FPV.

We chose to concentrate on proving embedded assertions in the RTL, which has been an increasing emphasis on processor design teams, and enables our engineers to easily add properties applicable to both simulation and FPV as part of the design process. Since our design language was Verilog, our assertions were specified using the Accelera OVL library ([8],[9]), which has recently become a standard in this area. We worked with the Foresight owners to add full OVL support to the tool.

Initially, our design team was provided training on the Foresight tool, and then left free to use FPV as much as desired during the design process. We soon found that a dedicated minority of design engineers went on to use FPV very effectively, but many others still found the tool difficulties too daunting and barely used the tool at all. This revealed a major issue that had not been previously discussed: what is a good team-wide methodology for using FPV? In the area of simulation, nobody would think of just releasing a simulator to the team and then saying "go ahead and validate"—there are coverage metrics, test plans, regressions and similar methodological rigor. Yet in FPV, we had not considered these issues.

At this point, we added a central owner to drive and evaluate FPV, and made several improvements in our methodology. We developed a wrapper script to simplify some of the aspects of tool usage that were challenging for the team. A regression run was added, so we would have a clear idea which blocks had many failing assertions, which had no assertions at all, and which were crashing the tools. We also initiated a "property push" on the most at-risk block, our memory controller, in order to increase our confidence in the design.

Overall we consider our use of FPV in Blackford a success, having used the tool to find approximately 24 logic bugs, and significantly increased confidence in our design. We have a number of methodology recommendations for future ASIC projects, including early introduction of FPV; the assigning of central FPV owners; FPV-friendly RTL standards; leaving ownership primarily with each DE; and the encouragement of assertion development through density checks.

We think that by learning from our experiences and following our recommendations, other ASIC teams will be able to expand their use of FPV as well, for increased design confidence.

II. RUNNING FPV

Before we start discussing the details of what we did on our project, we should give a basic introduction to how formal property verification is run on an ASIC design, and the basic debug process. In terms of the methodology for running the tools and debugging issues, FPV is different from some other design flows, and this is a major cause of the usage challenges. Note that we are concentrating here on the type of simplified flow that was used for our designs; microprocessor teams use more elaborate techniques like proof guidance and property decomposition [2] as well.

The major input to FPV is a design description with a set of properties, in our case Verilog RTL models with embedded OVL properties. (It is also possible to run FPV on more abstract models or with externally specified properties, but we did not do that in Blackford.) Some of the properties must be designated by the user as assertions, targets that need to be proven. The rest will be assumptions, or constraints, properties that the tool should take as a given. Usually the assumptions are properties on inputs, or properties whose proof would depend on logic external to the model.

The level of hierarchy where FPV is run has to be chosen carefully: the tools can quickly run out of memory on complex models, and we had to run numerous designs at unit level instead of cluster level. Often it will take a few attempts (in which the tool crashes due to excessive memory consumption) in order to identify a reasonable level to run.

Once the proper hierarchy is selected, the user can effectively run the FPV tool and analyze the results. The output of the tool will be a file classifying the assertions into four basic categories:

- **Proven**. This means that the tool has proven the property for all possible cases, under the current set of assumptions. This is the ideal result.
- **Bounded Proof for <n> cycles**. For complex properties, FPV will often prove them only for a certain number of cycles after reset. While this is not quite as good as a full proof, it is still a very useful result, showing that there are no possible violations for the given number of cycles after reset. In designs with complex state machines or deep pipelines, there may still be an error lurking beyond the given number of cycles; if a user knows this to be the case for their design, they may want to rerun with higher memory or timeouts, or at a lower design hierarchy.
- **Disproved**. For properties that it determines to be false, an FPV tool will generate a counterexample "simulation trace" showing how they are violated. The user must then analyze the trace, to determine if it is a real design bug or the result of a missing assumption that needs to be added. Typically, the majority of these cases turn out to be missing assumptions, which is a major challenge to users: only a small number of the disproved properties from FPV turn out to reveal useful design bugs.
- **Unknown**. These are cases where FPV was possible on some subset of the model, but not the full model, due to the complexity of the assertion's input cone. These cases should usually be rerun on a simplified model or at a lower design hierarchy to get usable results.

After all the counterexamples are understood, the user must add the appropriate assumptions to the model, and run again. It is also important to follow through by verifying the newly added assumptions: either add them as assertions to check using FPV on the driving block, or run the project's validation (simulation) test suite as a sanity check. A common pitfall is the addition of assumptions that are too restrictive, or overconstraining: they enable the proof of the assertions, but rule out valid counterexamples. Thus simulation must still be run as a complement to FPV, to help detect cases where FPV owners generated too-broad input assumptions and possibly invalid proofs.

Overall, FPV is an iterative process: after each FPV run, counterexamples are analyzed and assumptions are added. Unlike the generation of simulation tests, where the designers usually write code to explicitly generate valid test vectors, FPV assumption generation requires that the user think about all possible test vectors, and write constraints that will exclude invalid ones. While this can be a lot more work for the design engineer, it has the significant advantage that coverage is achieved for all possible test cases, instead of the small set of explicit ones used in simulation.

III. INTROUDUCING FPV IN BLACKFORD

Intel's Advanced Components Division began developing methodologies for Blackford Server Chipset design in June 2002. It was already decided that we would be using assertions extensively. The benefits of assertions are well documented [7] and had already been successfully used in the TwinCastle Server Chipset.

With assertion checks already available in the design, using them for Formal Property Verification seemed to be a logical next step. The benefits of using Formal Verification on existing assertions are manifold.

- FPV does not require any simulation support or test benches to be written, thus helping in producing good quality RTL early.
- The FPV process would help in correcting assertions. A common problem with assertions is that initial assertions are often wrong. Either the properties are not well thought or they are captured wrongly. FPV would help clean them up, for correct execution during simulation.
- The FPV process would lead to more assertions. Another issue with assertions is that we can think of only a few. While running FPV, counterexamples provide us more properties that can be captured as assertions or assumptions.
- Examining counterexamples and identifying new properties leads to the designer having a better understanding of the design.
- Some simple logic, like large counters and one shot triggers, can be very hard to test with simulation, especially if the logic needs to be exercised by a very high level validation environment. An alternative is to emphasize low-level validation or use FPV on properties written to capture the design intent.
- A formally proved list of properties serves as a valuable safeguard against introducing bugs at a later stage in the design

The process of running FPV on embedded assertions in Verilog RTL code was not fully supported by our installed tools during the early stages of Blackford, though commercial tools such as Blacktie from Verplex (now part of Cadence) were available. Until then Formal Verification within Intel primarily took place on the CPU development teams. A separate Formal Verification Team would write the specifications in a formal language (like PSL or ForSpec) and apply them to the design. They would use many specialized and time-consuming techniques (like pruning, ordering) to exhaustively prove the design properties. Our approach differed significantly from them in three ways.

- We did not have the resources to follow this approach, especially with regard to learning the specialized formal verification techniques. The people who ran FPV were Design Engineers and they would rather concentrate on design.
- Our primary benefit from the process was the counterexamples from which we derived more properties, fixed bugs, corrected existing properties or clarified the design better. Rigorousness in proofs was preferable but not necessary.
- We needed a really easy-to-use interface to the FPV tools and methodology. Otherwise designers and their managers would simply ignore them.

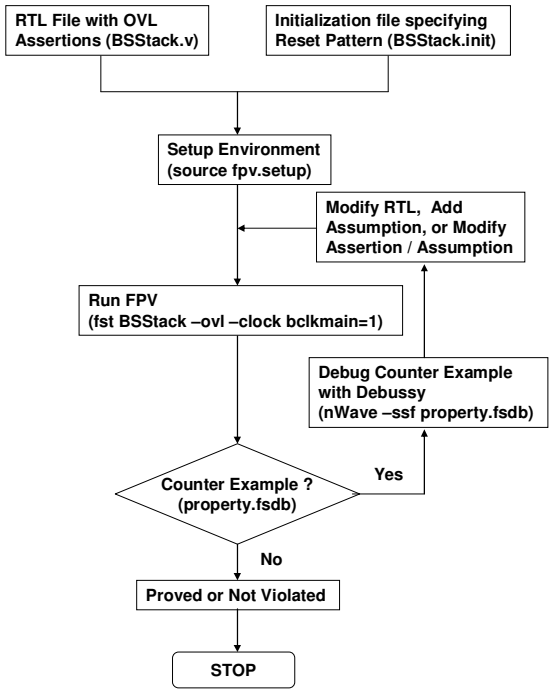

Figure 1: Basic Foresight FPV Flow

Given the above constraints, we worked with the Intel's Design Technology team to guide the development of a new formal verification tool, Foresight. This involved changes to the older, complex flows: directly parsing embedded OVL assertions; providing a straightforward command-line interface; adding simple clock/reset parameters; automatically providing good engine defaults; and generating counterexamples in Debussy waveform format.

The initial flow for running FPV is shown in Figure 1. The notable characteristic is the simplicity of the flow.

The FPV tools were presented to the Design Team in a half-day workshop. The workshop included an introduction to formal verification, a presentation on the FPV flow and a lab session giving the designers hands-on experience.

The next two subsections describe some of the specific successes we had in using this flow for design verification.

IV. FPV Case Study #1: Address Decode Unit

The assertion-based FPV flow was initially tested on the address decode unit (*BCDcd_Addr*) to iron out kinks in the flow. *BCDcd_Addr* handles processor and PCI Express address decode based on the system address map and routes transactions appropriately. The possible targets are memory (*MemCyc*), configuration (*CfgCyc*), PCI Express (*PexCyc*) and other FSB (*IntCyc*); it could also be internally aborted. The system address map is based on numerous configuration registers which have yet to be fully randomized in the simulation environment even today. These configuration values are decoded by the targeted logic blocks in parallel, but it is important that only one of the blocks is activated at any given time. For these reasons the decode unit was a good candidate for FPV.

FPV was used as the first form of unit level validation on the block and it was followed by simulation. The original property that was proved was that the cycles (*MemCyc, CfgCyc, PexCyc* and *IntCyc*) were mutually exclusive and at least one was asserted for every transaction. More properties were added and proven after simulation found problem areas.

FPV found three initial problems and two subsequent ones. For example, one issue was related to an address decode boundary condition where a '>' was used instead of the intended '>=':

```
   assign MEM[5] = ((AddressA >=
`AddrWi'h 00_0010_0000) && (...)) ? 1'b1
:  1'b0;
```

Another was related to request encoding where there was overlap in the decode of transaction types, causing false request types to be output. The encoding was done in parallel and FPV was used to guarantee that each decode was adequately qualified. This would have taken much longer to find using simulation.

All the bugs that were found by FPV could have been uncovered by simulation, but the property that was proven gives higher confidence in the unit, since the simulation environment lacks randomized configurations. The bugs were found without requiring a complex environment to be written; the complexity was in coming up with the correct assumptions.

FPV provided a good understanding of the corner cases in the decode logic, including illegal conditions. We added 8 basic input assumptions and 85 configuration register related assumptions in the process of proving the above mentioned property. One of these assumptions turned out to be overconstraining and was hiding a bug that was uncovered by simulation, where memory region that was made visible by moving an adjustable region caused assertion of *PexCyc* instead of *MemCyc*. This assumption had to be relaxed and property reproved. The configuration assumptions were very useful in bringing up the initial simulation configurations and verifying later configuration changes.

Some tips and tricks based on this experience include:

- Be careful not to use a simple 'mutex' property when something more complex is meant. This can be an issue for naive assertion authors.
- Run at a higher-level module to incorporate more logic, reducing the need for assumptions. A couple of properties in the inbound address decode (*BCDcd_InbAddr*) required FPV to be run on the parent module (*SBDcd*) so that additional assumptions can be avoided due to the dependency on the sibling unit *BCDcd_Addr*.
- Be careful about cases where a sudden reset reassertion can invalidate a property. Reset had to be pulled low (inactive) when proving a property that required decode outputs to be held stable for 2 clock cycles.
- Keep track of previous FPV results for sanity-checking. A global summary file proved useful in keeping track of when a property was last proven and what assumptions were made.

V. FPV Case Study #2: Memory Controller

FPV was used successfully in several of the Memory Cluster (MC) units, including the Memory Arbiter (MRA), and the Memory Protocol Engine (MPE). There are two branches in the MC, and each branch has its own MRA and MPE units. The MRA arbitrates between reads coming from 4 different queues and writes coming from 2 other queues, and the MPE stacks and sends the commands issued by MRA and sends them in the appropriate FBD (Fully Buffered DIMMs) format. Additionally, the MPE applies various back-pressuring signals on MRA to make sure that different commands do not conflict with each other and that various DIMM restrictions are accounted for.

There is a very tight handshake between MRA and MPE on each branch, but the two branches are independent of each other in normal running conditions. There is a configuration, however, "mirror mode", where the two branches become intertwined. In this mode, write commands (and associated data) that are issued on one branch need to be sent to the other branch at the same time. This creates a rather complicated handshake and requires special handling.

Verifying assertions on one branch proved to be relatively straightforward. However, proving the same assertions in mirror mode where both branches come into the picture proved to be a demanding job. The ideal way of verifying those assertions would have been running FPV on the whole MC cluster and letting the tool test all the MRA and MPE logic as well as other interfacing units to ensure that the assertions are not violated. The size and complexity of the MC cluster made this option impossible: FPV kept

crashing during the runs. The second best option was to try to combine the logic of both branches into one wrapper and use FPV to try to prove the assertions at that level. Because of the interdependency between MPE/MRA and the two branches, we decided to create two levels of wrappers: one around MRA and MPE, and another around the two branches.

Furthermore, a global assumption file was added to the top wrapper with 25 assumptions on interfacing units. After adding the wrappers, we verified a total of 82 MPE/MRA assertions, none of which were violated. When running simulation regressions on a new model late in the project, the added assumptions caught 3 potential problems in other interfacing units (where FPV had not been fully debugged). No real problems were found in MPE/MRA itself, but two minor modifications were made just in case other interfacing units were not covering special corner cases.

A. *MPE Electrical Throttling Example*

In this example, the specifications require that DIMM activations be limited to a programmed value on each rank on a branch. The programmed value is essentially the width of a sliding window, and the specifications require that each rank should not receive more that 4 activations at any time within that sliding window. Once the threshold of 4 activations is reached, the MRA starts electrical throttling on the rank in question by asserting ElecBlockRank, which blocks further activations until the window slides further in time and the very first activation steps outside the window. Assertions similar to the one below were written for all 8 ranks:

```
  assert_never EThrotViolation0
(bclkmain, resetnn, (ElecBlockRank[0] &
ActDone & (ActRank==3'd0)));
```

The ActDone & ActRank above could originate from Branch[0] but are sent to both Branch[0] & Branch[1] in mirror mode. This extra complexity made these assertions very difficult to prove on a single MRA. However, once the wrappers described above were added, FPV was able to use the logic of both branches and managed to prove the above property.

B. *MPE Configuration Command Example*

The MPE schedules configurations reads and writes to be sent on the FBD. It receives a request (IssueCfgTxn) from the Memory Address Decode unit (MAD). Along with the request, the details of the configuration command (CfgDecAddress) are supposed to stay stable until and acknowledgement is sent back by the MPE (MpeCfgSent).

In initial FPV runs, many invalid counterexamples were generated, where the CfgDecAddress input was illegally changing in the middle of a command, before the MPE acknowledgement. We realized that an explicit assumption was needed in order to specify that this address had to remain stable. The OVL expression below captures this handshake:

```
  assert_win_unchange #(1,1)
ASSUME_TxnUnchange0 (bclkmain, resetnn,
IssueCfgTxn, CfgDecAddress, MpeCfgSent);
```

This was needed to prove other assertions in MPE and was coded as an assumption since it applies to an interfacing unit (MAD), and cannot be proven based on the MRA/MPE logic only. Therefore, it was added at the top-level wrapper and was applied to both branches. Once this was added, many previously failing assertions were proven in FPV.

This assumption later fired in simulation, though, when running the model release regressions after a new RTL release. It was found the MAD samples the buffer ID from the Data Management (DM) cluster one cycle late, and that ID can change before the MPE asserts the MpeCfgSent.

Note the symbiotic relationship between simulation and FPV in this case: while simulation ultimately detected the error, the assumption that was violated in simulation was generated as part of the FPV debug process. This illustrates the advantage of using a general assertion library like OVL, which applies to both the simulation and FPV environments.

VI. Driving FPV Centrally

About three quarters before tapeout, we realized that only a minority of design engineers were actively using Foresight to check their designs. Part of the problem was that FPV had no real owner for our project: once the tool was available, it did not appear in any official design completion checklists, so there was little management support for its use. To address these issues, a formal verification owner was assigned for BNB, to analyze and improve our usage of FPV throughout the project.

This owner's first task was to analyze our Foresight runs for common project-level issues, and implement central solutions to make it easier for individual engineers to run the tool. He identified a number of common issues that resulted in invalid counterexamples or failed tool runs, and pursued several solutions: a wrapper script to hide tool details, central assumption templates to quickly enable workarounds for common problems, and working with the tool authors to improve compatibility with BNB designs. Among the issues simplified by the wrapper script were:

- **Tool invocation details**. Even simplified FPV tools like Foresight offer a bewildering set of options for a novice user, and it is often unclear which ones need to be set explicitly and which have reasonable defaults.
- **RTL Issues and Compiler Limitations.** Numerous parts of Blackford initially could not be consumed by the FPV compiler (Esperanto), due to slight deviations from the IEEE Verilog standard that broke the FPV compiler but were tolerated by simulation.
- **Mid-test reset assertion.** Often a counterexample would be seen to be reasserting reset in the middle of

some complex protocol. While this is technically possible, it is not interesting from a debug point of view. An example of this is discussed in Case Study #1 above. We wanted to rule out all such cases in FPV runs by default.

- **Configuration registers**. Many designs have a set of config registers, which are set once and never change values without another reset. Like a mid-test reset assertion, a mid-test config change is technically possible but not interesting from a debug point of view in most cases.
- **Stability assumptions**. Most BNB inter-unit signals are synchronized to rising clock edges, but by default Foresight generates counterexamples with signals changing in both clock phases.

The next task was to implement a project-level FPV regression. Once this was implemented, we began regular runs on our full design, so we could assess which clusters were making effective use of FPV, and begin deciding where to concentrate more detailed efforts. We found that initially about 70% of our assertions were proven in the regressions, and also identified several clusters with almost no assertions. Running the regressions regularly enabled us to identify cases where previously passing assertions began to fail, so we could analyze them for further debug.

Along these lines, we also began running improved assertion density checks. Previously, assertion density had been measured using the traditional metric of assertions per line of code. We added a new method provided by the 0in Check tool from Mentor Graphics: functional coverage of flops by assertions. This check measured how far each flop is from the nearest assertion, enabling us to find cases where seemingly assertion-dense units (measured by assertions per line) actually had large areas of logic that were uncovered.

We then identified the cluster considered the most at-risk due to logic complexity, our memory controller (MC), and began a Property Push, where the FPV owner and other experts worked with individual design engineers. In cases where units were poorly covered by assertions or most assertions were not yet proven, the experts helped the DEs write new assertions, run FPV, and debug the results. Initially we were hoping that the central experts could take over most of the FPV work for these blocks, but it quickly became apparent that very detailed design expertise is needed to create the correct assumptions—the assumptions created by non-authors of a design almost always missed some subtle issue and were violated in simulation. Thus we decided to revert to a more DE-driven method, with the central owners encouraging FPV use and helping designers with tool usage and debug issues. In the end, we considered the Property Push a success; it found five interesting logic bugs that had been missed in simulation, using about 10 engineer-weeks of effort.

Finally, we began some experiments with a vendor FPV tool, Mentor's 0in Search, that takes an interesting alternative approach. This tool combines simulation with FPV in the same run: it analyzes simulation traces to find interesting states, then runs FPV from that point. Thus it has the potential of finding errors many cycles after reset (beyond the proof radius of tools like Foresight) using real-life test cases as a guide. While some companies have reported good results with this and similar tools ([1],[6]) we were disappointed. There were too many tool bugs and minor issues to get it working on more than one cluster by tapeout, and in the end 0in Search found us no additional bugs. Later we also tried the similar Magellan tool from Synopsys, but were not able to find any new bugs using that technology either.

VII. RESULTS

There are three types of results to report from using our FPV methodology. First, there is the usefulness of the DE-driven FPV run on local models. Second, there is the common metric of the number of bugs found. Finally, there are the overall benefits to the project tapeout confidence as a whole.

Qualitatively, local DE-driven FPV runs were reported to be very useful by the design engineers. They were utilized to hammer out basic errors in logic and assertion definitions without needing to put together a simulation testbench environment, in the very early stages of design. The case studies we discussed above describe some of the types of issues that were found. Furthermore, when adding ECOs in later stages of the project, the DEs used local FPV to sanity-check their changes before turning them in, and make needed logic corrections. FPV counterexamples also gave design owners valuable insights into subtle corner cases of their logic.

The most common question people ask, however, is how many bugs were actually found using FPV. We have some difficulty in measuring the results of the DE-driven early and pre-release FPV runs, since these mostly were done on local, private model versions. The designers report finding 10 or so subtle bugs that might have been tricky to catch in simulation. In the later stages of our design and the Property Push, 8 FPV-based logic bugs were filed against Blackford, of which we consider 5 to be interesting bugs that would have been tricky to find in simulation. There were also 6 bugs found in assertion definitions themselves, which are important to debug to maintain accurate assertion checking in simulation.

Finally, there is the question of overall confidence in the project. We were able to implement a total of about 2300 assertions and assumptions in our RTL. Of the 1895 that were assertions eligible for FPV, 1527, or 81%, had full or bounded proofs before tapeout. Spending some time pushing formal property verification in early designs and on our most critical blocks, and achieving an 81% proof rate in our assertions overall, provided significant "peace of mind" benefit: we knew that if we had missed some crucial corner case in our test plan that might violate our assertions, we

would have a good chance to catch it using FPV. Overall, Blackford was a very successful project from a logic standpoint—our A0 stepping booted all targeted operating systems within three weeks of its arrival.

VIII. Summary And Recommendations

Our experiments on the Blackford project, as described above, gave us a good quantity of practical experience in running FPV tools, in addition to leading to the discovery of a handful of interesting logic bugs. Based on what we have learned, there are a number of learnings that should be taken into account by similar ASIC teams as they begin their designs, in order to gain maximum benefit from FPV:

- **Each ASIC project should have a central FPV owner** to analyze project-level issues. While the tools have achieved a level where they are usable by ASIC designers, there are still a number of complex details that should be analyzed centrally by a team's expert, and incorporated into an FPV-launching wrapper script.
- **Select an assertion language that applies to both FPV and simulation.** Simulation can help check for cases where assumptions are too broad or are violated by expected input vectors, while FPV helps ensure greater coverage of cases that might be missed in simulation. Using the two techniques together on the same set of properties is much more powerful than using either alone.
- **Introduce FPV very early in the design process** to take advantage of its usefulness for unit-level debug and design exploration before the simulation testbenches are ready.
- **Create FPV-friendly RTL standards** when beginning a design for which FPV is planned. In addition to the basic tool issues such as adhering to synthesizable verilog standards (IEEE 1364.2001), try to enclose complex protocols in a hierarchy at some level to simplify the FPV process.
- **Each designer should be responsible for FPV on their block.** While a central owner can help diagnose and solve common problems, the complex process of creating assumptions (constraints) can really be done most efficiently by a block's author, due to the subtle and tricky counterexamples often uncovered by FPV.
- **Assertion density checks should be used** to ensure that all designers add sufficient assertions to make FPV worthwhile. Assertion density needs to be measured and some level of assertions enforced in each block, though some types of modules (such as datapaths without much control logic) may need waivers.
- **Run FPV Regressions after RTL releases** to make sure that the project's FPV state is understood, both in terms of the total number of assertions available, and the number that are formally provable. This regression also enables quick detection when a previously proven property is invalidated by a design change.
- **Consider moving external simulation checkers into RTL assertions** to make them useful for FPV as well. This also has the beneficial side effect of transforming the implied requirements of an external checker to a formal specification directly in the RTL.

We should also point out that these are recommendations based on one project's experience, and our team is still at a relatively early point in the learning curve of formal property verification methodology. There are numerous related questions that we do not yet have good answers for, or have not yet attempted to address, such as:

- Can we deploy a more formal property tracking database, to automatically track passing and failing properties and property dependencies, and auto-update after every FPV run?
- What proportion of a block's failing properties should we try to debug as a requirement for tapeout? For Blackford, since assertions and FPV were optional overall, we did not enforce any level of FPV debug in most blocks.
- How do we most effectively balance resources between traditional simulation-based validation and FPV? It seems logical that as we become more comfortable with FPV, we might get increased benefit by retargeting some of our current simulation resources.
- Can we get significantly more effective FPV if we try to directly translate the high-level requirements in the component specification into usable RTL properties, and do more architecture-oriented property verification? We did not really try this in Blackford, though this method is common on processor teams. FPV runs on such properties are likely to be very challenging for the tools.
- Can we generate some assertions automatically that will be useful to prove formally? Some vendor tools such as Fishtail Focus, 0in Checklist, and Synopsys Magellan attempt to automatically generate assertions in various contexts.
- Can we make effective use of tools which combine simulation and formal methods in a single run? Intuitively this seems like a powerful concept; the problems with our experiments on early tools in this area should not blind us to the potential for the future.

We hope that other ASIC projects will share their experiences as well; together we can continue to refine the

FPV methodology for this type of design, and further explore the possibilities of this powerful technology.

IX. ACKNOWLEDGEMENTS

We would like to thank the many other design engineers on the Blackford team who helped to pioneer this effort, including Rajesh Pamujula, Dhananjay Joshi, Nick Stasik, Joaquin Romera, and Joe Udompanyanan. We would also like to thank the additional engineers who assisted with FPV activities: Zan Yang, Nandini Sridhar, Latha Rao, and Luis Kida, as well as the Esperanto/Foresight support team, especially Yossef Levy, Dan Jacobi, Ranan Fraer and Yael Zbar. And of course, we should acknowledge our managers, who showed great patience as we explored this new area: Suneeta Sah, Amir Taraghi, Dave Smith, and Subba Vanka.

X. REFERENCES

[1] Frank Dresig, et al., "Assertions Enter the Verification Arena", Chip Design Magazine, December/January 2004.

[2] Tom Schubert, "High-Level Formal Verification of Next-Generation Microprocessors", Design Automation Conference, June 2003.

[3] Rajesh Gupta, et al., "Panel: Formal Verification: Prove It or Pitch It", Design Automation Conference, June 2003.

[4] David Dill et al., "Panel: Formal Verification Methods: Getting Around the Brick Wall", Design Automation Conference, June 2002.

[5] Mike G. Bartley, Darren Gilpin, and Tim Blackmore, "A Comparison of Three Verification Techniques", Design Automation Conference, June 2002.

[6] Remi Francard and Franco Toto, "Take the Next Productivity Leap", Chip Design Magazine, December/January 2005.

[7] Lionel Benning and Harry Foster, "Principles of Verifiable RTL Design", 2nd Edition, ISBN 0-7923-7368-5, Kluwer Academic Publishers, 2001.

[8] Accelera OVL Technical Committee Web Page, http://www.eda.org/ovl/.

[9] Harry Foster, Kenneth Larsen, and Mike Turpin, "Introducing the New Accelera Open Verification Library Standard", Design and Verification Conference, February 2006