

FEV's Greatest Bloopers: False Positives in Formal Equivalence

Erik Seligman, Joonyoung Kim

Digital Enterprise Group

Intel Corporation

Hillsboro, OR

erik.seligman@intel.com, joonyoung.kim@intel.com

Abstract— Formal Equivalence Verification (FEV), also known as Equivalence Checking (EC), is a powerful technique that formally proves the equivalence of a pair of design models. (Here we are referring generically to FEV tools, not to a specific tool such as Cadence Conformal or Synopsys Formality). FEV is used at numerous places in the design flow, including verification that a netlist was properly synthesized from RTL, verifying that a timing change did not modify logic, and verifying that an ECO was hand-implemented properly. In all cases, passing FEV runs give high confidence that the design is correct: since it is a formal method, theoretically it finds 100% of bugs. But can we fully rely on FEV to give a correct answer in all cases?

In fact, there have been a number of situations at Intel where FEV has given a false positive, claiming designs are equivalent when they really are not. Needless to say, this has been rather embarrassing for the validation teams involved, who had previously reported full confidence in the design cleanliness. These false positives have been extremely costly, resulting in bug escapes that need to be found in downstream flows closer to tapeout, or in the worst cases, in dead silicon. Rather than simple tool bugs, these issues resulted from a combination of misunderstood tool functionality and incorrect usage. They ultimately stem from causes including misuse of constraints, incorrect understanding of tool properties, gaps in verification of library cells, incorrect usage of Verilog, overly aggressive black-boxing, and poor tracking of excluded 'unreachable' points.

In this paper, we describe several cases of FEV false positives at Intel from the past few years, in both processor and ASIC designs. Based on our learnings, we provide a number of recommendations for safely running FEV and reducing the risk of similar situations in new projects. Taking our experiences into account, we believe any design team can run FEV with much greater confidence in the future.

I. INTRODUCTION

Formal Equivalence Verification (FEV) is an essential CAD capability used to ensure correct implementation of VLSI designs, by verifying the equivalence between the implementation (for example schematics) and the specification (for example RTL) of the designs. This capability is also widely used when logical changes are done on a design, and the functional equivalence between the original and modified circuits must be guaranteed. We refer to the original circuit as the specification (spec) model and to the modified circuit as the implementation (imp) model. Traditional methods for doing FEV are based on combinational equivalence verification (CEV) which requires complete correspondence between the state elements, such as latches and flip-flops, of the models under comparison. Once this is guaranteed, a tautology-checking procedure (e.g. BDD [1] or SAT [2] based solver), can be employed to detect functional equivalence. In this case, the two compared designs are normally decomposed into combinational sub-circuits.

FEV has become an essential part of the design process at Intel and similar companies ([3],[4]) for a couple of major reasons.

- **FEV is efficient:** Although it can fail due to time or memory in rare complexity cases, FEV between RTL and a netlist or between two netlists is usually a tractable problem, as long as the state elements correspond as described above.
- **FEV is reliable:** Since techniques such as BDD or SAT are used to mathematically prove equivalence, a passing FEV run is ideally able to guarantee 100% equivalence between hardware design models.

However, as with many CAD tools, fully “push-button” automation is not quite possible. There is a certain level of input and control required from the design engineer. Among the user inputs required are:

- **Design Models:** Typically FEV operates on designs specified in RTL languages such as Verilog or VHDL, and schematic models in Verilog or various backend modeling languages. While design models are a fairly obvious input to the FEV process, we mention them here because ambiguous or incomplete specification can make a difference in accuracy of results.
- **Mapping,** or describing to the tool how state elements in one design correspond to similar elements in the other one. In fact, combinational FEV can be, in its simplest form, viewed as formally comparing combinational logic bounded between mapped state elements. Generally in the paper we use “/spec/a -> /imp/b” notation where signal “a” is a node in the specification side and “b” is a node in the implementation side.
- **Cell Libraries:** The engineer needs to provide logical representations of the design’s primitive elements, such as AND/OR logic gates or flip-flops. While many modern FEV tools can extract transistor-level logic, doing this repeatedly for standardized cells would be wasteful and cause an explosion in runtime.
- **Assumptions,** or telling the tool what “impossible” cases it can rule out. For example, in Figure 1, a backend engineer may take advantage of the INVERSE between A and B, and tie C off to 0 instead of placing the AND gate in Block 2 while editing the layout. Assumptions are written by design engineers on the RTL description of the design to achieve two things; 1) They work as “assertions” and their status is checked continuously during simulation of RTL model, and 2) During FEV, they are used to constrain the design under verification

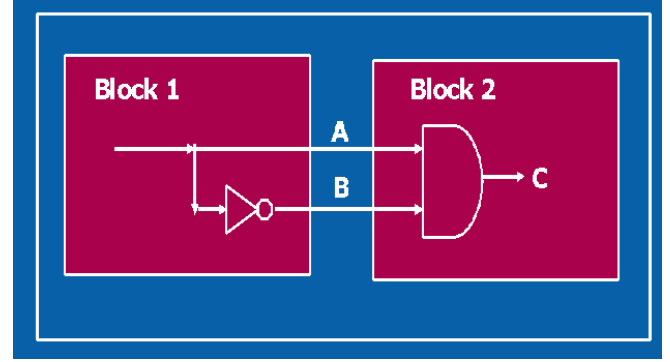


Figure 1: Assumption INVERSE(A,B) May Be Needed for Correct Block2 Verification

Despite that fact the techniques for formally verifying the equivalence between two hardware designs are reliable, incorrect usage of the tool is possible. The user can fool FEV into incorrect behavior through subtle issues in the mappings, the assumptions, or the cell libraries, or can misuse it in other ways. The most dangerous of these situations is the “false positive”: when we have a false positive, it means that we have declared a design verified when it is actually incorrect, and the issue must be discovered in a more expensive (possibly exponentially more expensive, in dead silicon cases) downstream flow. Thus, even an experienced engineer must think carefully about how he is performing FEV. In the remainder of this paper, we will highlight some of the dangerous false positive cases we have encountered at Intel, and describe how Intel and similar companies can avoid these issues in the future, gaining much better overall design confidence.

II. PROCESSOR CASES

A. *Conflicting Assumptions*

On one of our large Intel processor designs a few years ago, we taped out with all blocks FEV clean, and waited eagerly for the A0 silicon to come back for testing. It turned out, however, that our initial silicon would not boot—and using gate-level simulation (GLS), we found that in one block that had been edited just before tapeout, the RTL and netlist gave radically different results. This was a block that had passed FEV!

What could have happened here? After some intense investigation, we finally figured out what had made the FEV run invalid. It turned out to be an issue of conflicting assumptions.

As mentioned above, two of the main inputs to Intel’s traditional FEV flow for CPUs are assumptions and the map file. Common RTL constructs for assumptions include “MUTEX”, “INVERSE” and “FORBIDDEN.” Map files are used to instruct the FEV engine about equivalent points (generally primary input/output and state points).

In general, assumptions are needed more commonly for hand-drawn blocks (at transistor level) than synthesis blocks,

as human designers tend to take advantage of the micro-architectural properties of the design and use them to optimize the schematics. If the circuit is designed with these properties in mind, FEV will not pass unless these assumptions are used to constrain the tool.

The mapping syntax we support does not stop at simple one-to-one mapping; there are situations where we need to group multiple nodes in one bucket and map them to other signal(s) on the other side. E.g. `"/spec/a -> /imp/b1 /imp/b2"` This will mean that the two nodes "b1" and "b2" in the IMP side both correspond to "a" in the SPEC side. Such situations can often result when flops are replicated for timing improvements. In this case, FEV compares both pairs: a vs. b1, and a vs. b2. Note that if both pairs are proven to be equivalent, then we have also just proven, using transitivity, that "b1" and "b2" are also equivalent.

It should be noted that in our one-to-many mapping case, we are in essence allowing the user to write an implicit assumption, that the targets on the "many" side of the mapping are equivalent to each other.

In the case of the bad design described above, we found that the FEV run contained both of the following:

- Assumption: INVERSE(b1, b2)
- Mapping: `/spec/a -> /imp/b1 /imp/b2`

Since the mapping implies that b1 and b2 are the same, these two assumptions are conflicting with each other. Conceptually this will reduce valid search space to nil, as no input combinations will be valid (meaning cannot satisfy both assumptions). What happens when formal tools face this is obviously implementation-dependent. What happened at the time is that this caused the formal tool to declare every node that is affected by the two assumptions to be equal! There were indeed circuit design errors which went unflagged because of this.

So how did we fix this case? A later release of our FEV tool had added "vacuity checks", or checks for conflicting assumptions. Once these are found, they are flagged before any formal verification is done so that we can fix it earlier in the verification cycle. In this example, if we had turned on this new feature and used it properly, the issue would have been detected. Once we upgraded to the tool release with this feature, this issue was prevented for future steppings.

But in general, when there are multiple ways to describe assumptions for an FEV tool, we must be very careful for cases where the design space is restricted more than we intend. If, for example, the assumption had been merely "NOT (b1 & b2)", both signals would be constrained to 0, making the verification trivial but not technically a conflict.

Thus, in addition to improved tool support, this case also offers some lessons for the FEV engineer. First, vacuity checks should be run, if available in the FEV tool being used, as an important initial safety net. Second, all assumptions must be carefully reviewed, including indirect assumptions generated by mapping or other tool-specific

directives. Finally, even though FEV offers high confidence, basic gate-level simulation runs should always be required before tapeout, as a smoke test for possible FEV misuse.

B. Missing Circuit Assumption Verification

Unfortunately, there was another CPU project at Intel that also came up with dead A0 silicon. As in the previous case, this was a major surprise, since it was on a design that had passed FEV. In this case, though, the issue stemmed from a somewhat different cause.

To understand this issue, we must first describe what are known as "schematic assumptions". When designing a library of standard cells, it is often possible to save transistors and area if certain assumptions are made about the logic.

For example, in Figure 2, we see a simple cell.

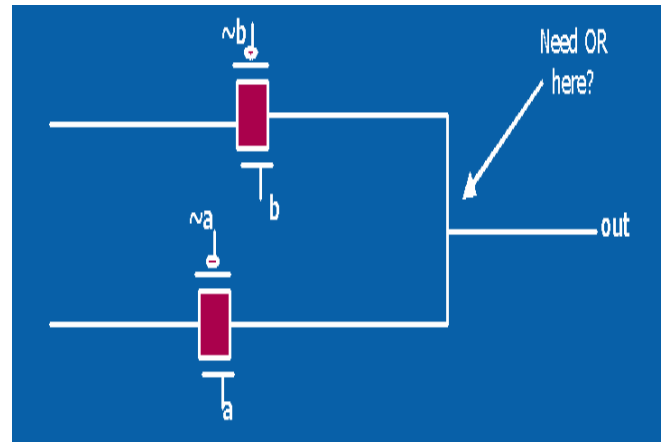


Figure 2: Cell Needing Schematic Assumption

In this Figure, there is clearly a contention issue if it happens that both pass gates are open, so some additional gate is needed where the nets come together. But if we can guarantee that only one pass gate can be open, with an assumption such as INVERSE(a,b), this design becomes perfectly safe.

Due to issues like this, various library cells supported here at Intel have come with internal assumptions. These are delivered with the cells, and no circuit containing such a cell can be considered to be truly valid and FEV-clean unless that assumption is formally proven. To accommodate this, in the FEV flow we have a step called "IMP assumption verification." This is a step where we check whether the built-in assumptions in library cells are violated by the surrounding logic.

In the particular processor case we are discussing here, the design engineer did not fully understand the significance of this assumption verification step. Thinking that proving properties was a "bonus" check where we expect lots of failures anyway (as is usual in formal property verification runs, see [5]), the engineer decided to pronounce the design

clean without getting this step to pass, since the main Equivalence Checking step of FEV was passing.

This provides us another lesson in the area of assumptions. It is important for any engineer running FEV to understand why we have assumptions in the design, and to make sure that such assumptions are fully verified. It must be crystal clear that no part of the FEV flow is “optional”, and if any step is to be waived, an FEV expert must examine the situation very carefully.

III. CHIPSET / ASIC CASES

A. The Dangers of Unreachables

Our first ASIC example comes from a platform validation chip designed in Oregon. During the design flow, some bonus logic was added that was not logically connected to anything in the design. This is a common technique used by designers to provide spare gates and logic for use during late ECOs or in post-silicon FIB edits. In many cases, this extra logic can save a design team from having to tape out a new stepping in order to debug an issue, and thus is very important despite being logically insignificant in the initial design.

When using bonus logic like this, we must be careful to recognize the relationship with “unreachable” points. In FEV, some provision must be made for logic that is removed due to being unused or tied off, and thus should be ignored for equivalence purposes. In general, these areas of logic that can never affect any output of a design are known as unreachable points. A naive netlist analysis, in a netlist containing bonus cells, would consider all bonus logic to be unreachable, since it never affects outputs. However, it is important that bonus logic be preserved—so most FEV tools provide some option that tells it to verify any that appear in both the models being compared, and list any unverified unreachables for user review.

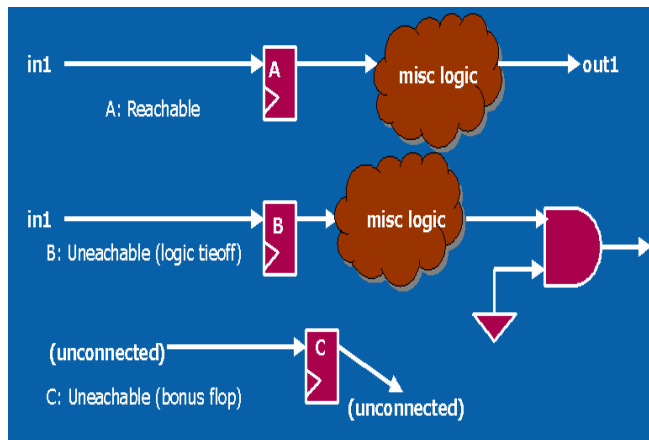


Figure 3: Examples of Reachable and Unreachable Flip-Flops

In this case the designers, after adding bonus logic to an initial synthesized netlist, took the netlist through numerous timing, area, and other backend flows, as is usual in the

process. FEV was used before and after every netlist transformation, to ensure that the design had not changed. In addition, the FEV engineer double-checked that he was using the option to report unreachables in the tool, and reviewing the list of unverified unreachable points for known bonus logic. Yet late in the design cycle, a backend engineer was doing some manual edits to the layout and was surprised to find that some of the bonus logic had disappeared. Despite the constant reports that FEV was passing and reporting none of the bonus logic in the unverified unreachables, they had been eliminated, and FEV did not detect this.

As it turned out, the FEV tool we were using actually had the ability to eliminate unreachables at two different stages of a verification run. While reading a design, it would automatically eliminate obviously unconnected logic—such as the bonus logic, as in Figure 3, flop C. Then, during the mapping and verification process, it would identify a second set of unreachables by analyzing the design for areas that were logically tied off or otherwise unused, as in Figure 3, flop B. It was only this second elimination stage that the FEV engineer was monitoring, and thus the information about the first set of eliminated unreachables was lost.

Once we understood the issue, it was easy to modify the tool options to keep all unreachables during the initial design read, so they would all be there to be reported in the next stage. The main lesson here is that it is very important to understand how your FEV tool handles unreachable points, and be aware of all possible stages where they may be ignored. In modern designs, “unreachable” bonus cells cannot be ignored. We have also noticed that even for non-bonus cells, reviewing the unreachables gave the unexpected benefits of identifying several areas of logic that had been accidentally tied off.

B. Gaps in Library Verification

Our next example comes from a chipset component design. As mentioned in previous sections, our synthesis process includes the use of a library of primitive cells, which is delivered in multiple formats for use by various tools. In particular, our cell library is delivered in two different formats, a logical Verilog model and a proprietary backend language that specifies detailed electrical properties. During FEV, it is important for the tools to read the libraries in order to understand the logic in the synthesized netlists. This information is used to build the equations which are being tested by FEV.

Taking advantage of a feature offered by our FEV tool, some of our engineers were reading the library in the proprietary format, and using this to verify netlist edits. Normally this is perfectly fine, since this format contains a logical representation of every cell. Late in the design flows, however, we did some more runs after standardizing on the Verilog library, and noticed that some of our 2-flop synchronizer cells had been changed to single flops in the netlist during a power reduction flow, and nobody had previously reported an FEV error.

Upon examining the library files more closely, we discovered that in the logic equations in the proprietary format, these 2-flop synchronizer cells were represented as single flops. The Verilog representations of the same cells had the expected pair of flops. We ran a library cell FEV just to be sure, comparing the two models of the libraries, and confirmed that these synchronizer cells (and fortunately, only these cells) were logically different between the two versions.

This did not make much sense, and after some digging, we discovered the reason. Apparently some backend tools do not accept multiple flops in a single cell in the logic equations, when using the proprietary format. The library team, assuming that logic flows such as FEV would be using the Verilog representation anyway, made the necessary hack to stop tools from crashing and moved on.

This case teaches us two important lessons. First, the obvious lesson, that authors of library cell models used by other projects should be very careful about any assumptions they make. In this case, the equations in the backend models were more significant than initially thought. Secondly, if multiple representations of the same logic are allowed by a project, they should be FEVed against each other. If we had done an initial FEV of the two library representations, we would have prevented this issue from the beginning.

C. *Obscure Corners of Verilog*

This is another example from the same chipset design. In this case we synthesized a large cluster, then when running FEV, found a mismatch between the new netlist and the RTL. This occurred in a gigantic cone of 900+ gates, nearly impossible to debug by hand for all practical purposes. Suspecting an FEV tool issue, we re-verified with a second FEV tool, the one supplied by our synthesis vendor. This tool said that the netlist and the RTL were equivalent. At this point, several team members were leaning towards declaring the netlist clean, and assuming the mismatch was a bug in the original FEV tool.

However, we were nervous about this approach, due to the fact that we would be relying on FEV and synthesis from a single supplier. They assured us that the FEV tool was developed by a separate and fully independent team, but could we really be confident in one tool from a vendor to bless the results of another tool from the same vendor? We continued to investigate the bug for several months, trying to narrow down the ultimate source of the problem.

Finally, the synthesis vendor came up with a small code fragment that demonstrated the issue:

```
wire foo = {a, {(2-WIDTH){1'b0}}, b};
```

In this particular example, the WIDTH parameter was exactly 2. What exactly does the above line mean in this case? The design engineer's intent, and the behavior specified in the SystemVerilog standard, would treat this as equivalent to:

```
wire foo = {a, b};
```

In other words, the 0-length replication would disappear from the vector. However, according to the synthesis vendor, that code was equivalent to the following:

```
wire foo = {a, 1'b0, b};
```

In their interpretation, any replication had to have at least one element. As it turns out, in some versions of the Verilog standard, this is an unspecified case—so technically **both** interpretations were correct!

Once we identified the cause of this issue, we modified the RTL to eliminate the ambiguity, and after a resynthesis our RTL and netlist were FEV-clean. If we had not fixed this, then our silicon would have implemented logic different from the design intent and possibly different from the results of our RTL simulation. This would have been very dangerous.

The biggest lesson here is always to be very careful about independence between synthesis and FEV tools. In this case, our synthesis tool supplier had some general assumptions about Verilog that were apparently part of the corporate culture, despite their attempts to keep the synthesis and FEV teams separate. If we had trusted their FEV tool, we would not have caught the issue.

In addition, this highlights the importance of good linting (code quality checking) of RTL code. While we do require linting checks as part of our design process, they did not catch this issue. If we had stronger checks to prevent designers from using any unspecified corners of the language, that would make errors like this much less likely.

D. *Multiply Driven Signals*

One of our test chips recently had an issue where late in the design process, a manual inspection revealed a connectivity bug: a submodule pin, the VCC input to a blackboxed analog submodule, was supposed to be connected to logical 1 at the top level, and was in the RTL, but this connection had been lost in the netlist. This was again a surprise, since the block in question had been declared FEV-clean, and FEV should easily detect such disconnects.

This case was a little tricky at first, until we realized that the disconnected submodule pin was declared as an 'inout'. This meant that logically, the top-level net to the submodule was considered to be multiply driven (multi-driven) in the RTL. What does FEV do in the case of multi-driven nets?

A verification engineer might have the intuition that FEV tools should check the set of drivers for any multi-driven net, and report the designs matching only if a corresponding identical set of drivers can be found. However, this would cause the tool to report false negatives in cases where there was truly "wired logic" intended, and the logic was rearranged in synthesis or other flows. So another option is to treat multi-driven nets as implicit wired AND or OR gates.

We were surprised to find that for the FEV tool we were using, the default behavior of multi-driven nets was to treat

them as wired AND. This makes it clear how we got the false positive in the case above: the extra connection to logic 1 in one model was meaningless, since it just added an ‘AND 1’ to the logic equation of the multi-driven net, which did not change the logic.

This issue was fixed by modifying the submodule VCC net to be an input rather than an inout, eliminating the multi-driven condition. Another fix, and a recommended practice in general, would be to set the FEV tool options so multi-driven nets are not treated as wired gates, and instead all drivers are evaluated.

This example shows that it is important to understand how FEV tools handle special cases, such as multi-driven nets, where it may not be obvious how to construct logic equations. A verification engineer should always double-check that a reasonable set of options is being used for these situations.

E. Incorrect Analog Black Boxing

Our final ASIC example is another case that escaped to post-silicon, and thus caused a major financial impact. Once again, we taped out our design with confidence that it was fully FEV-clean. Then, unfortunately, we found out in the lab that there was a real logic error that had escaped all our tests and flows. Upon further examination, we discovered that this logic error was inside an FEV 'black box', an area excluded from the FEV verification due to heavy analog logic.

In general, blackboxing analog submodules is an accepted and required part of FEV flows: at some level, the design simply depends on electrical properties that cannot be modeled by digital logic tools. Such modules must be verified through other methods, such as detailed circuit simulation. But, on the other hand, analog blocks usually come with a small amount of "glue logic", digital logic that is at an RTL/netlist hierarchy just above the analog parts of the design. While the analog logic must be blackboxed, designers need to be careful to blackbox only what is necessary, and include as much glue logic as possible in FEV runs.

In this case, we determined that the error was a simple hookup in the glue logic. This piece of the design could have easily been included in our FEV runs; we had made the mistake of blackboxing at too high a level. If we had included the glue logic in FEV, the error would have been easily detected before tapeout.

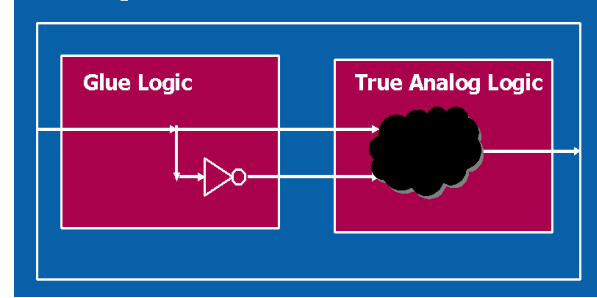


Figure 4: Example of “analog block” that contains some non-analog glue logic.

We have also seen another issue related to handling of analog logic, where points can be labeled "unreachable" when they are actually important to the design. This can happen when rather than being fully blackboxed, a small analog cell is replaced with a logic representation for FEV. This can increase the accuracy of the verification, in cases where the overall organization of the logic has major differences between the RTL and the netlist. In most cases this is a safe and useful method.

The issue occurs when we use a representation of an analog cell that only uses a subset of its inputs. For example, a piece of analog design may have a "vccbias" input for electrical reasons, but viewed as a purely logical entity, it acts as an inverter on its "data" input. If it is replaced with the logical representation "out = !data" for FEV purposes, it is possible that logic driving the "vccbias" pin might be incorrectly detected as unreachable. In general, when replacing analog logic with abstract representations, we must take care to account for all inputs that are driven to the block, and explicitly force verification of any that will logically disappear in the analog modules.

The main lesson here is to be very careful when blackboxing analog logic. It is important to verify as much as possible by FEV, and only to exclude the truly unverifiable parts of the design. And if analog logic is replaced with simplified equations for FEV purposes, the verification engineer must be very careful that all inputs are covered.

IV. SOME COMMENTS ON FALSE NEGATIVES

In this paper, we have been emphasizing false positives, since these are the most dangerous situations for FEV. But another concern of verification engineers is FEV false negatives—cases where we pronounce a design dirty and reject it, but it is actually logically correct. This is usually much less expensive than false positives, since an issue is flagged and the engineers involved know they have to debug it, but is an area that deserves attention as well.

In general, all the issues we have described for false positives can lead to false negatives as well, with slightly different twists.

- Wrong or missing assumptions can cause us to be verifying a different design space than we intended. Missing assumptions often happen due to the need to propagate assumptions across rows of state elements for use in the following logic cone.
- Assumption verification may fail because other assumptions, not yet added, are required in order to prove the assumptions we want.
- The tool may fail to identify an unreachable point as unreachable, due to complex logic or incomplete constraints. Then some equivalence failures will occur on something that should not matter anyway.
- Incorrect library logic equations can cause designs to fail that should actually be passing.
- Misuse of Verilog or other design languages, causing the meaning of RTL to become ambiguous, can cause false negatives as well as false positives.

We should also mention another class of (possibly) false negatives that has been a major challenge in FEV. These are cases where we use aggressive synthesis tools on designs with large logic cones, in order to meet timing. In some rare cases, FEV then fails, indicating that either we have a synthesis bug or there is an FEV tool problem. But sometimes the size of the logic cones is gigantic—1000+ originating (“support”) flops, and thousands of intermediate gates, in a single verification cone—and it is very hard to analyze the result. Is the error in the FEV tool, the synthesis tool, or both? Debugging by staring at a schematic with standard debug tools on logic cones of this size is nearly impossible.

Typically we try to debug these by manually over-constraining a subset of flops to simplify the logic, but this often leads to a very limited insight. We are also working on a “tiebreaker” script to use a third-party simulator (not supplied by the FEV or synthesis vendor) on the extracted failing cones, to determine whether the FEV or synthesis tool is reporting the correct results. But in general, we believe a CAD breakthrough here is needed, to truly enable efficient debugging of these cases. In the worst cases, resynthesis with slightly modified parameters sometimes seems like the most viable solution.

V. CONCLUSIONS AND RECOMMENDATIONS

We have seen that despite the high quality of the tools, incorrect FEV usage can create false positives in both processor and chipset projects. In the worst cases, this led to dead A0 silicon for some processors.

Formal techniques have a number of dependencies on engineer input, including the specification of assumptions, mappings between key points, and provision of pre-verified library cells. From our particular examples at Intel, we have

seen a number of ways to cause FEV tools to generate incorrect results: Conflicting assumptions from multiple control files can confuse the tool. Assumptions critical to a verification may be neglected. Points may be omitted from verification. Library cells may have incorrect or invalid logic that is used by FEV. Obscure aspects of RTL languages can be treated differently by different tools. Corner cases of tool behavior may be specified incorrectly by the user.

Does this mean that FEV cannot be relied upon for ASIC verification? Of course not—it is still a very powerful technique, mathematically proving that two models are equivalent, and thus critical for gaining confidence that bugs were not missed due to incomplete simulation. But these examples have taught us a number of lessons about FEV use and misuse, and lead us to a set of recommendations for future projects.

1. **Make sure all assumptions are carefully reviewed, and formally verified whenever possible.** Be sure to review not just explicit assumptions, but also indirect assumptions generated from mappings or other tool directives. Remember that even if a tool detects conflicting assumptions with vacuity checks, it may miss near-conflicts that are logically valid but decimate the possible input space. Also be especially careful to check if the cell library came with built-in assumptions; these may be required in order for the library logic to be valid.
2. **Monitor unreachable points carefully.** It is guaranteed that some subset of unreachable points will have to be omitted from FEV, since modern synthesis tools will eliminate unused logic. So these points will always exist, and must be carefully reviewed to make sure important bonus logic is not lost. Reviewing unreachables also may detect unintentionally tied-off logic due to RTL coding errors, another important benefit.
3. **Make sure all libraries used are verified and understood.** As we have seen, cell library suppliers are often not involved in the higher-level FEV processes of their customers, and this can result in misunderstandings. Also, if using a cell library with multiple representations, it is important to verify that all representations are equivalent.
4. **Make sure all RTL is high-quality and fully linted.** If some ambiguous RTL is used, then all bets are off—different tools may give different interpretations, and it is uncertain what version will end up in silicon.
5. **Be very careful about how analog logic is blackboxed.** While analog logic must be excluded from FEV, be sure to include the surrounding glue logic as much as possible.

Also, if the analog logic is replaced with equations rather than being truly blackboxed, be careful that all inputs are verified.

6. **Be sure tool options are set reasonably for special situations, like multiply-driven nets.** Try to review the tool settings you are using; most FEV tools have various options that define how to interpret different styles of design, and each project must use the appropriate settings.
7. **Ensure independence between synthesis and FEV tools.** The easiest way to do this is usually to choose Synthesis and FEV tools from different vendors. A single-vendor “integrated flow” may be tempting, but as we have seen, can be dangerous.
8. **Double-check FEV-clean logic with gate-level simulation (GLS).** This is an excellent sanity check, in case of errors in FEV setup.

We have seen that while it is a powerful technique, FEV is still very dependent on user input and control. FEV tools have reached the level of usability that nearly every design engineer can run them without much difficulty, but experienced FEV engineers should be supervising the process and ensuring that these dangerous scenarios are avoided. We hope that by documenting our FEV “bloopers” and offering our diagnoses and solutions, we can help other engineers as well as ourselves to avoid these false positives in the future, and to fully leverage this powerful technology to continue producing high-quality VLSI designs.

ACKNOWLEDGMENTS

We would like to thank Daher Kaiss for his many years of FEV support, and his assistance in reviewing drafts of this paper. We would also like to thank many other Intel design, CAD, and verification engineers, too numerous to name here, who we have worked with on the various projects we discuss.

REFERENCES

- [1] R.E. Bryant Graph-based algorithms for Boolean function manipulation, IEEE Trans.Computers, C-35(8), 1986.
- [2] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem proving, Communication of the ACM, Vol 5, pp. 394-397, July, 1962.
- [3] W. McDonald and J. Liao, Logic Equivalence Checking has Arrived for FPGA Developers, DVCon 2006, February 2006.
- [4] C.M.Chang, Verification: Port in a Storm, EE Times, 7/25/2001.
- [5] E. Seligman, R. Koganti, K. Maheswaran, R. Naqib, Bringing Formal Property Verification Methodology To An ASIC Design, DVCon 2006, February 2006.