Practical Approach Using a Formal App to Detect X-Optimism-Related RTL Bugs

Shuqing Zhao, Shan Yan, Yafang Feng
Mobile and Wireless Group
Broadcom
Irvine, California, USA
{shuqing.zhao, syan, yafang}@broadcom.com

***Abstract*— "X-optimism" behaviors in standard RTL simulation remains a serious threat to ASIC tape-outs. It is not practical to rely on gate-level simulations to detect all related bugs. We propose a holistic approach centered on a formal X-propagation application to detect X-optimism issues early in the RTL verification cycle. The formal app reads in the RTL, analyzes the design, and then automatically implements assertions to check for all X occurrences on targets such as clocks, resets, control signals and output ports. If the formally proved X occurrences are determined by user to be unexpected, it usually implies they were masked in RTL simulation due to X optimism. We use an X-sources-driven approach to help improve productivity by identifying X sources and then using this information to determine the appropriate scope to apply the formal tool. This also helps improve the possibility of the formal tool achieving full proofs instead of bounded proofs. For example, we use formal reset analysis to identify uninitialized registers from the RTL design. This analysis helps us to apply the formal application on the key design blocks with the best ROI. When bounded proof is unavoidable, we use a simulator with an X-propagation feature to complement the formal method. We discuss results of our approach using two case studies, a power management controller module and an audio processing module, both of which have design bugs masked due to X-optimism.**

Keywords—X propagation; X-optimism; X sources; RTL; Verilog simulator; formal verification; bounded proof

I. INTRODUCTION

Verilog HDL [1] and SystemVerilog [2] use a 4-value logic to model digital circuit behavior. The four values are the 0 and 1 boolean values, *x* for "unknown," and *z* for a high-impedance or open circuit. The standard definition of how *x* is interpreted in expressions and statements causes any simulator following the standard to exhibit two phenomena: X-optimism and X-pessimism. To our knowledge, these two terms first appeared in [3]. Some publications, as in [4], may define the two terms differently, but in this paper we adopt the widely accepted definitions in [3][5]. The rationale for the standard to have these two limitations can be mainly attributed to simulation performance versus modeling accuracy tradeoff. Nonetheless, the lack of complete understanding of these issues and the dearth of a widely accepted comprehensive solution have resulted in many post-silicon functional bugs that cost IC design companies precious debugging time and resources, and possibly expensive chip respins. With the new verification technologies and multiple lessons learned, we believe we have found a holistic methodology that works reasonably well for us as a SoC team.

In Section II, we define what the X-optimism problem is and the known solutions proposed by others. In Section III, we list all the X sources potentially causing RTL bugs. Section IV outlines our X-source-driven formal verification methodology. We discuss two case studies in Section V before we conclude the paper in Section VI.

Disclaimer: We mention names of EDA tools used in our flow in this paper with no intention of any endorsement. The methodology is generic enough for other tools with similar capabilities to produce comparable results.

II. X-PROPAGATION ISSUES AND EXISTING SOLUTIONS

This paper focuses primarily on X-optimism, but for purposes of completeness, we first briefly describe X-pessimism.

A. X-pessimism

X-pessimism is a simulator behavior in which an *x* value, instead of a deterministic *0* or *1* value as in silicon, propagates to the next HDL program execution step. X-pessimism occurrences can be categorized into two groups: single-bit operation and multiple-bits operation. In single-bit operation, X-pessimism occurs due to operator semantics defined by the Verilog standard. For example, assuming *a* is a single-bit variable, the results of (*a & ~a*) and (*a | ~a*) are always *x* if *a* equals *x* in the simulation. However, in real silicon, the value of (*a & ~a*) is *0* and the value of (*a | ~a*) is always *1*. The second source of X-pessimism comes from multiple bits operation involving X. The Verilog standard specifies that for the arithmetic operators, if any operand bit value is the unknown value *x* or the high-impedance value *z*, then the entire result value shall be *x*. For example, in the simulation you see this:

3'b000 + 3'b01x = 3'bxxx

while the following result is more accurate:

3'b000 + 3'b01x = 3'b01x

X-pessimism is an undesirable simulation behavior, because it propagates excessive *x*'s that are time-consuming to debug. It usually does not, however, mask RTL design bugs.

B. *X-optimism*

X-optimism, on the other hand, is a simulation behavior that can mask RTL design bugs. It allows a deterministic value of *0* or *1* instead of an unknown value *x* as in the real silicon, to propagate to the next step of HDL program execution. The simulator is doing nothing "wrong" by just following what the Verilog standard defines as the semantics of language constructs, such as *if*, *case*, *negedge*, or *posedge*. For an *if* statement, only when the *if* condition is true (defined as a nonzero known value) will the true branch be executed. The false branch, if it exists, will be executed when the condition expression is false (defined as 0, *x*, or *z*).

Note: When the *if* condition is a value *z* instead of *x*, the result is the same. In this sense, the "X-optimism" term is not accurate, in our opinion. Nonetheless we will continue using this term in this paper due to its wide acceptance.

Figure 1 shows an X-optimism example of an *if* statement being used in the following code snippet. When *count* increments to *7*, *en* goes *x* for one clock cycle, causing *count* to remain *7* in the simulation. In real silicon, when *en* equals *1*, count can roll over to a value of *0*. The desired simulation behavior should be *count* getting a value of *3'bxxx*.

```
reg [2:0] count;

always @(posedge clk or negedge rstn)
begin
    if (~rstn)
        count <= 0;
    else if (en)
        count <= count + 1;
end
```

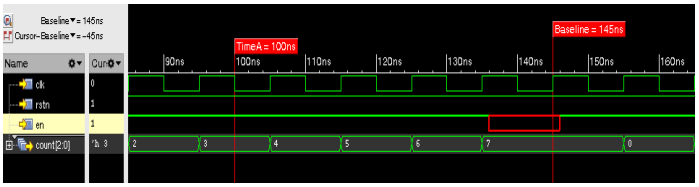

Figure 1. X-optimism prevents the real X from propagating

Not all X-optimism behaviors are undesirable. For the same code above, the reset simulation would not work without the help of X-optimism, as shown in Figure 2. The transition of *rstn* from *x* to *0* causes "*negedge rstn*" to evaluate to true, triggering the reset assignment to be executed in simulation.

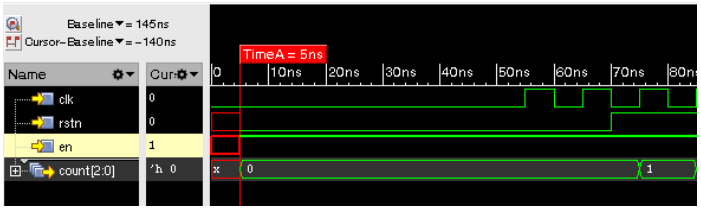

Figure 2. X-optimism makes an asynchronous reset work

C. *Existing X-optimism solutions*

In the following subsections, we discuss (to the best of our knowledge) the methods previously used to address X-optimism issues in RTL simulation.

1) Gate-level simulation

Usually X-optimism behaviors do not show up in gate-level simulation. The culprit language constructs *(if, case, posedge, negedge*) get synthesized into the gate-level net list, which comprises combinatorial gates and flip-flop/latch primitives or UDP. These gate-level models do not exhibit X-optimism behavior. Unfortunately, for any reasonably sized SoC design, the cost of running gate-level simulation for the full RTL simulation regression suite (including tests developed at the block, IP, subsystem, and SoC levels) is prohibitive in terms of labor or schedule. One reason is that X-pessimism becomes more prevalent for gate-level simulation. It is very time-consuming to go through iterations of debugging and fixing (depositing or forcing known values) until the sea of red X's is receding in waveform. This, plus the much slower simulation performance, makes the productivity very poor when compared to running the RTL simulation. Practically, only the minority of the RTL simulation test suite is regressed at the gate-level simulation, which almost guarantees that some X-optimism issues could be missed. Even if RTL bugs can be found at this stage, it usually would be costly to do ECO or resynthesis, jeopardizing the project schedule.

2) Coding style change to prevent X-optimism

It is possible to change the RTL design coding style to detect and avoid X-optimism. One coding style change is recommended in [5] to use the ternary conditional operator *?* to replace *if* statements. For complex nested *if* statements, this change would suffer from very poor readability. Another coding style recommended by some people is to do explicit X interception and propagation. For example, the previously mentioned counter-example can be recoded as it is below, so that when *en* is *x*, the result of *count* turns *x* also.

```
always @(posedge clk or negedge rstn)
begin
    if (~rstn)
        count <= 0;
    else if (en)
        count <= count + 1;
    else if (en === 1'bx)
        count <= 3'bxxx;
end
```

This coding style is also impractical for designers to cover all *x* intercepting conditions, particularly when signals with multiple bits are tested in a *case* statement. Other negative reasons include poor readability, simulator performance penalty, etc., as listed in [3].

3) 2-state logic simulation

Some people [3] have resorted to the use of a 2-state logic simulation instead of the standard Verilog 4-state logic simulation. The key idea is that instead of using X, a random known value is used in any particular seed of the simulation run. The fundamental flaw of this approach is that it is not

possible to cover all combinations for multiple bits of X signals. For instance, if there are 32 register bits that are not initialized, 4 billion test runs need to be done. Of course, not all combinations make sense, but the question is how to determine the meaningful set of combinations that must be covered.

4) Traditional model checking with 4-state logic

Many formal model checking tools support 4-state logic natively. This means that when proving an assertion, the tool takes into account both 0 and 1 cases when a signal can be an *x*. For example, if value 1 for a noninitialized register can make an assertion fail, the tool will pick this value to generate a counter-example for users to debug. The issue with this approach is that usually there is no guarantee that the assertions proven cover the full functionality of the design.

5) Model checking with automatic X checkers insertion

There are a few commercial formal tools, such as [9], that can do automatic X checker generation and use special formal engines to do the proof. The main issue with using this type of tool without following a good methodology is that design complexity and tool capacity often lead to the classic formal proof convergence problem. In fact, the contribution of this paper is to make this tool usage more effective and productive.

III. Common X Sources Causing RTL Design Bugs

Our methodology is an X-source-driven approach. The Verilog standard defines many cases in which an X value can occur. In this section, we list four X sources known to us as potential causes for RTL design bugs. Some other X sources such as floating input ports or implied latches are less interesting, because they can be easily detected by the lint tool prior to RTL sign-off.

A. Uninitialized Registers

First, the most common X source in RTL coding is uninitialized registers, either latches or flip-flops. Although the best known practice is to reset all registers, designers often choose not to reset some registers for reasons like performance, area saving, ease of routing, timing closure, etc. In this case, those registers have neither asynchronous nor synchronous reset signals hooked up. These registers are also known as "nonresettable" registers. Another case we have experienced is that sometimes the registers use a synchronous reset signal, but the clock is not active when the reset is asserted.

B. Out-of-Bound Array Element or Bit-Slice Access

In many other programming languages, when an out-of-bound array access occurs, a run-time error will happen and the program will either crash or be terminated. The Verilog standard treats this scenario differently — the program continues to run and *x* will be returned as a result. In the following example,

```
reg [3:0] addr;
reg [7:0] data;
```

If the value of bit-select index `addr` is out of bounds, e.g., `addr == 8`, then `data[addr]` returns *x*. Similarly, the following example declares an array to model a 1 KB memory,

```
reg [7:0] mem[0:1023];
```

If the index is out of the address bounds, or if any bit in the address is *x* or *z*, then the value of the `mem[addr]` shall be *x*.

C. X assignments

There are primarily four reasons that designers use explicit X assignments in RTL code:

- Using X assignments to truly model unknown values in the silicon. This is a common usage in the analog IP model, memory model, and gate cell model.
- Assigning *x* values to a signal is interpreted by a synthesis tool like Design Compiler as "*don't care*," meaning that any known value could be assigned during synthesis for logic minimization purpose.
- Assigning *x* values to a signal is used when an error condition occurs. The intention is to propagate X to some observable objects checked by the test bench. We believe this should be discouraged. Assertions should be used instead to report the error condition.
- Assigning *x* values in an *else* branch or *case* default is used to intercept and explicitly propagate X with the intention of fixing the X-optimism issue.

D. Power-aware semantics in UPF or CPF

For low-power SoC design, UPF [6] or CPF format power-intent specification files have become must-haves as companions to RTL HDL code. The UPF standard defines simulator behavior for power-related chip operation, such as power-on/off, isolation and retention, etc. When a power domain is powered down, all the logic nodes within that domain will be corrupted as X and will remain X until the domain is powered back on again. If the powered-down domain has isolation on its output ports, the isolation cells would prevent X from going out to other powered-on cells—the isolation cell output drives a known value specified in the UPF file. To the retention registers, the X corruption still occurs, just as the power down case does, upon entering retention mode. The difference is that the retention register contents prior to going into retention mode are saved and can be restored upon exiting the retention mode.

IV. X-Source-Driven formal X Propagation Method

Ideally, we would like to apply a formal tool [9] to exhaustively prove the SoC is free of X-optimism issues. However, this approach suffers the usual formal-tool-capacity issue. It is not practical to apply this method blindly to the full SoC, or even to subsystems. We believe the best way to reduce X optimism bugs is to avoid X in the first place. In cases where this is not possible, knowing where all the X sources are in the design can greatly help detect X optimism issues. This X-source-driven approach we recommend consists of the following steps.

A. Adopt a Coding Style that Reduces X-Optimism

Many, such as [5], have suggested coding styles that can help reduce X-optimism problems. We agree with some and

disagree with others. Here is a list of the coding style or design choice recommendations that are appropriate and not too aggressive for us as a SoC chip team that must deal with many third-party IPs.

- Use asynchronous reset signal to reset as many registers as possible. There are two parts to this suggestion. One is whether to use synchronous or asynchronous reset style. The other is whether all registers should be reset. On the first issue, asynchronous reset is better than synchronous reset in the sense that it does not depend on whether the clock is running during reset assertion. Of course, the deassertion of the reset should still be synchronized to the clock. On the second question, our belief is that all registers, especially the control-, clock-, and reset-related registers should be reset. The savings on area or the optimization gained on routing or timing closure typically do not compensate for the risks of hiding RTL design-bug primarily caused by X-optimism. For example, a common mistake in many publications is that a clock divider flip-flop like the one in Figure 3 does not need a reset. The claim is that there is no difference whether the output clock starts as 0 or 1. Actually, it depends on what kind of registers this clock output is connected to down the pipe. If it is used as clock for falling-edge triggered flip-flops or latches, whether it starts as 0 or 1 does make a difference. A bug could be hidden because of this.

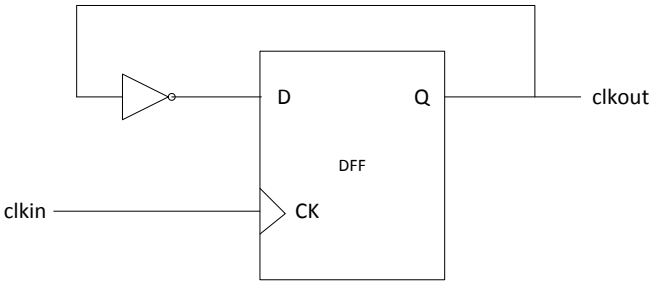

Figure 3. Clock divider

Another similar example can be seen in Figure 4. This is a clock gating cell using a latch. When *phi* powers up as 1, the clock output could start as 0 or 1. When starting as 1, it sends one extra falling edge to its fan out. This behavior could hide an RTL bug, especially when *negedge* is used in RTL code.

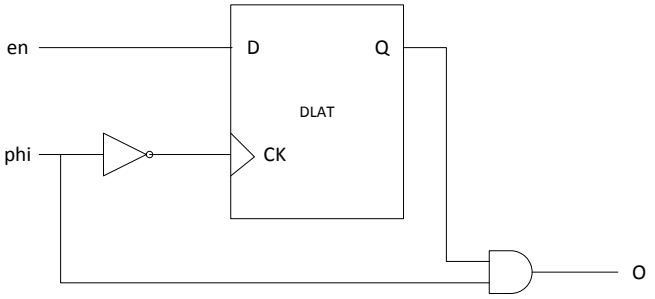

Figure 4. Clock gating cell

- Avoid using a *negedge* flip-flop or latch when possible. With the above two examples mentioned, it should be clear why this has something to do with X-optimism.
- Avoid using X assignments whenever possible. As discussed in Section III.C, X assignments sometimes are used as "*don't cares*" by designers to assist logic minimization. The area-saving benefit of such a technique is really questionable [5] in today's billion-transistor SoC chips that we are working on, unless it is a timing-critical path and there is empirical evidence demonstrating the obvious improvement. If the purpose of using X assignment is to help catch an unexpected state entry, assertion should be used instead. With our methodology, there is also no need to do X-explicit intercepting and propagating.
- Avoid having floating input or wires. This is a well-known good practice, yet some designers misuse the synthesis tools' capability of optimizing out unused logic because of floating input. In our experience, there should be very few exceptions where floating input can be used, e.g., some analog IP ports are required to be left unconnected.
- Avoid intentionally using an out-of-bound bit select or array element reference as an X source, either for an error -indication or logic-optimization purpose.
- Avoid using *casex* and *casez*. The subtlety of these two statements is just too much for average designers to comprehend (Sorry, no offense!). For details, see [5].

B. Use lint to Identify X Sources

A Verilog lint tool such as Spyglass [7] can be used to eliminate some X sources easily. Here is a list of problematic areas the tool can check:

- Floating input ports, dangling wires, or *reg* type variables with no driver
- Signals having multiple possible drivers

In addition, a lint tool should be used as first level of defense to enforce the coding style mentioned above and, in addition, to detect other X-optimism-related issues before any simulation is run. Of course, the usage of a lint tool does not diminish the importance of designer self and peer code review.

C. Perform "formal" Reset Analysis

A lint tool can detect reset issues, such as no reset signal present on any registers. But this is done by a structural analysis of the RTL code without actual simulation. This could miss issues such as a reset polarity mistake, the clock being stale during synchronous reset assertion, and so on. Some formal tools can use a user-provided reset vector to generate an analysis report of what registers remain as *x* after the reset period ends. This report is very useful for us to identify the most common X source: uninitialized flip-flops or latches. You will see in Section V.A how this helped in identifying some suspicious X sources caused by no-reset flip-flops.

D. Run Formal Structural Property Analysis and Proof

Some formal tools have the structural property generation or synthesis [10] feature that can analyze RTL code and automatically extract properties (assertions and coverage) for dead code check, arithmetic overflow check, FSM reachability check, and the checks we want to emphasize in this paper, namely array out-of-bound indexing and X assignment reachability. These two types of checks can help detect and analyze X sources (as mentioned in Section III.B and III.C) very early in verification cycle—as soon as RTL code is compile-ready, and before any test-bench development starts. Ideally, this should be part of the RTL signoff process executed by designers who know the RTL code best. You will see in Section V.B how the array out-of-bound indexing check helped detecting the unintended X source. Our experiments also showed these types of formal proofs are less expensive in terms of machine run-time and memory footprint than other formal proofs like the X propagation check that we shall discuss next.

E. Run the Formal X-Propagation App

Once we identify and confirm all the X sources found in the three steps mentioned above, we must determine the target Verilog modules that we need to do further X-optimism analysis. The basic principle is that we select the Verilog module that is the immediate container, within which X source(s) are produced and consumed, as the target for further X-optimism analysis. Our experience has shown that these types of formal checks are subject to the classic formal tool capacity issue. The smaller the scope is, the more likely the proofs can achieve convergence.

We use a formal x-propagation app [9] to do X-propagation properties extraction and proof. The properties (assertions or coverage) essentially checks whether X can reach any of the target signals of interest to the user. The target signals can be either user specified or automatically extracted common critical signals such as clocks, resets, test conditions (e.g. the *en* signal in Figure 1) for if/case statements, and primary output ports. These signals usually should be free of X and when X is detected on any of them it implies X optimism could have occurred in the passing simulation tests. See [11] for more details about the formal tool generated properties.

To run the tool, the following steps are needed:

- Set up the design under verification (DUV) environment, similar to constructing a test bench in simulation world. At a minimum, the primary input clocks and reset definitions, as well as the X sources (either static or dynamic) have to be described.
- Prepare the Verilog RTL code for formal tool compilation. All the code has to be synthesizable. If there are nonsynthesizable blocks within, they have to be indicated to the tool as black boxes. For example, a common case of this is the memory models instantiated in the RTL design. For any black box, by default, we constrain it as not an X source to its containing module. The exceptions need to be modeled as constraints to the formal tool. For example, a constraint should be added if a memory model outputs X when a chip select signal is not asserted.
- Compile the RTL code using a specific command option to enable X processing.
- Run the formal tool X properties generation to instrument assertions for the following three targets:
    - Clocks and resets signals
    - Primary output ports
    - Test conditions used in *if* and *case* statements
- Run X-properties proofs using engines known to be more effective for X propagation.
- When the proof job finishes before the specified time limit, three results can occur:
    - Properties proved.
    - Properties failed.
    - Properties proved within a cycle bound.

F. Solutions for Properties with Bounded Proof

There are a few ways to deal with properties getting bounded proof.

1) Experiment with tool timeout limit and engine selection

If you are not lucky enough to get all your assertion proofs converged with the first attempt, the low-hanging-fruit methods are to play with tool setup or host machine selections. Often simply choosing a faster and bigger memory machine to run the proofs will solve the problem, or you can try to increase time-out limit. Sometimes picking different formal engines (i.e., algorithms) can also work.

2) Use manual abstraction techniques

Abstraction is a very common method to reduce DUV complexity in order to help formal tools to achieve proof convergence. The tools can already do a lot of automatic safe abstraction processing behind the scenes. However, sometimes they need human help to apply some abstraction techniques manually. Black-boxing is one of the simplest and yet most effective abstraction techniques, based on our experience. It is intuitive to understand the safety of black-boxing blocks that are not in the cone of influence (COI) for an assertion. It is somewhat counter-intuitive that black-boxing blocks in the COI is also safe in most situations, with a few exceptions like the involvement of clock domain crossing. Figure 6 illustrates how the black-boxing technique can be applied in the context of proving an X-propagation assertion. The X-propagation assertion references an X source originated in module A and terminated in module B. For this specific assertion, we can black-box module C, which is in the cone of influence. Assuming B and C are in the same clock domain, this is safe (no false positives) if the assertion is proved, because the formal tool considers all possibilities of input ports coming from C.

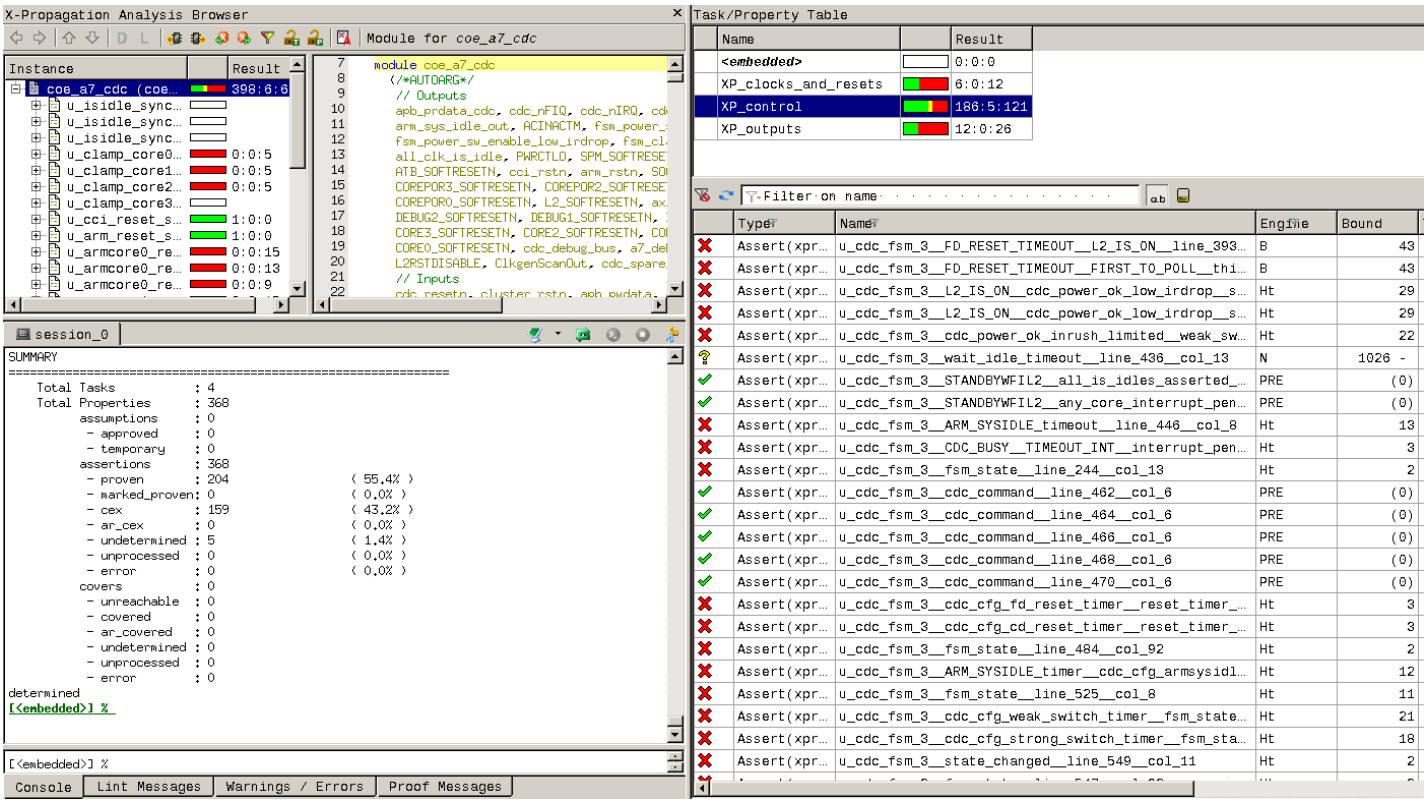

Figure 5. Case Study 1 Xprop Assertion Violations

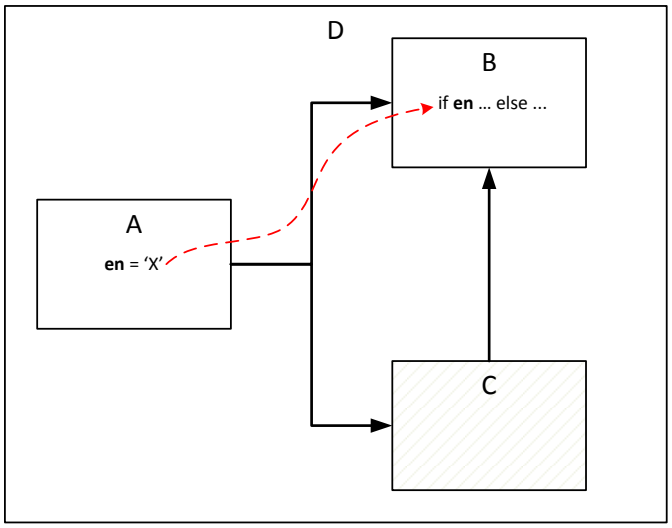

Figure 6. Black-boxing blocks in the COI

3) Use Simulator X propagation Feature

Recently there have been advances in HDL simulators that address this critical weakness in the X-optimism area [8]. The simulator essentially merges the results of both 0 and 1 in the place of an X when executing *if, case, posedge/ negedge* language constructs. We recommend that this feature be enabled in RTL simulations, especially for the Verilog modules/instances that do not have all X properties fully proved by the formal tool. This simulator feature is, in theory, also useful for nonsynthesizable modules, for example, analog models or memory models modeling an unknown using X.

V. CASE STUDIES OF FINDING X-OPTIMISM BUGS

In this section, we describe two case studies that highlight how we came up with the methodology described above. The first one showcases how important the reset coding style and the formal reset analysis is. The second one demonstrates that the best way to deal with X-optimism is to identify X sources and confirm their validity as early as possible in the verification cycle.

A. Case Study One

This is a success story of catching a critical RTL bug using a formal approach before tape-out. The DUV in this case is a power-management controller Verilog module for a quad-core application processor in one of our chips. It controls the dormant entry and exit of each processor core. Needless to say, this block is very critical. We ran very extensive UVM-constrained random simulation at the block level as well as many more directed test cases at the SoC level to test the integration. Spyglass lint analysis had been done and errors/warnings had been waived by the designer.

Knowing the danger of X-optimism, we decided to run a reset analysis using a formal tool and found there were four flip-flops remaining as X only in the first clock cycle after reset deassertion. After the first clock cycle, they were all assigned to known values. We did some very rough RTL code tracing and found one particular flip-flop looking suspicious. That prompted us to spend a few minutes setting up the environment

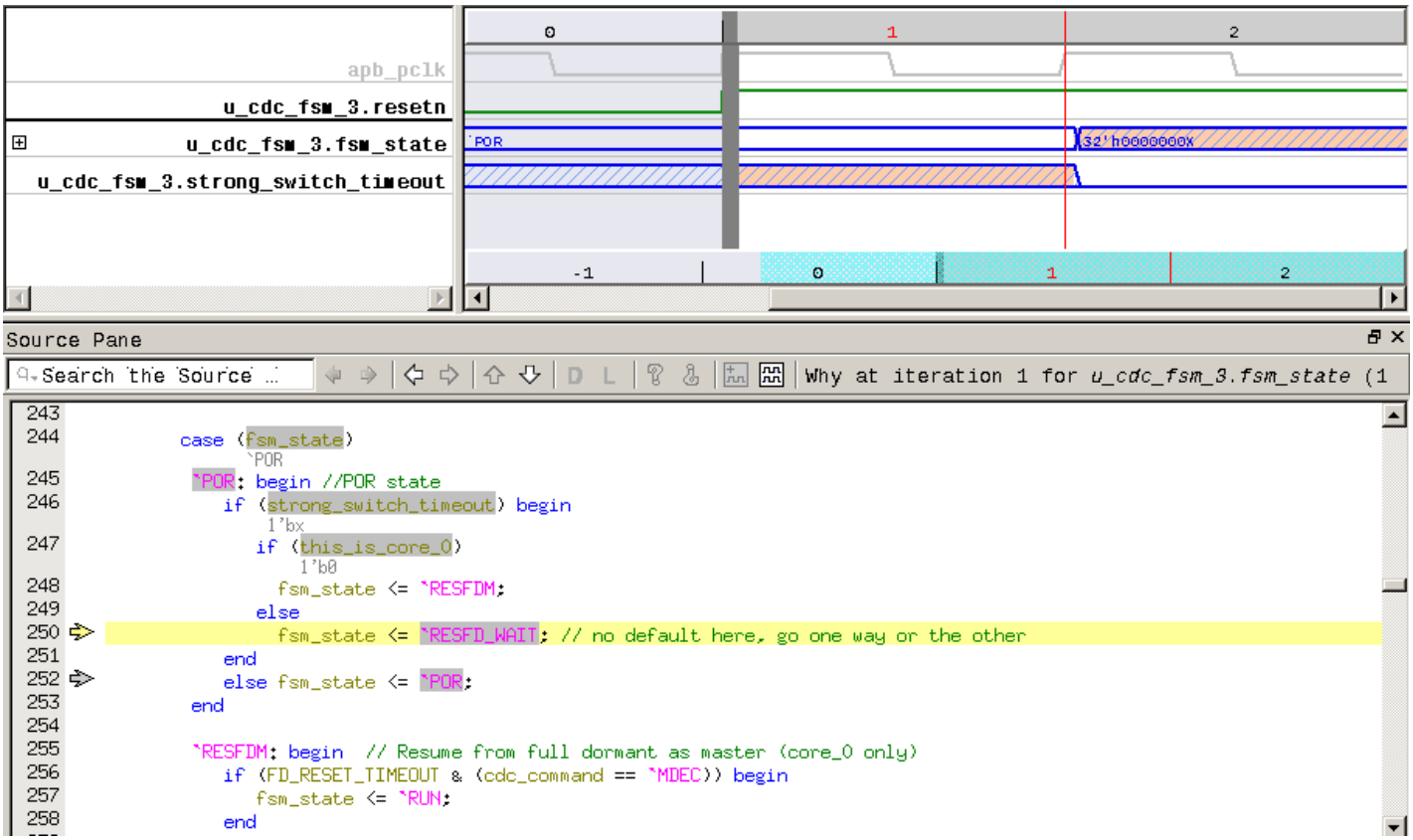

Figure 7. Counter Example to a Xprop Assertion

to run the formal x-propagation app. The tool was able to report multiple assertion violations very quickly, as shown in Figure 5. Upon further debugging, it turned out that all the assertion failures were caused by the X-optimism on that one flip-flop we suspected earlier.

Figure shows a counter example to one of the failing assertions. At the first cycle after power-on reset, the *strong_switch_timeout* signal has the value of X, due to no reset on this flip-flop. At the second clock cycle, it gets assigned a value of 0 but it is too late. Because of X-optimism, the X value of the signal causes the state machine to always stay in *`POR* state at the first cycle after reset. However, in the real silicon the state could have jumped to the *`RESFDM* or *`RESFD_WAIT* state, which was not intended.

B. Case Study Two

The second case study is a post-silicon bug hunting story. The DUV is an in-house developed audio processing IP. In the new chip this "silicon-proven" (be alert whenever you hear this word!) IP supposedly had only minor modifications, one of which is adding a debug channel to the legacy 16 audio data channels. A "delta" verification strategy was deemed appropriate, specifically to run an existing test suite regression and add additional test cases covering the design change. (**Note**: Under increasingly tight schedule and resource constraints for consumer SoC projects, the "delta" verification strategy is usually what can get approved by management, but it does not always work as expected.) The legacy 16-channel test cases at IP level had been extended to support 17 channels, and all RTL simulation regression was passing prior to tape-out. After the silicon arrived and months into testing, a failing symptom was found in which the system would hang whenever channel 16 was used. After many other unsuccessful attempts, we tried to duplicate the failing symptom by running a gate-

Figure 8. Code Snippet of Channel 16 X-Optimism Bug

```
wire [16:0] Arb_reqs;

// Selected Q requesters
assign channel_req = Arb_reqs[channel];

always @(*)
  case (SM_state)
     SM_idle:
       if (Q_request)
              next_SM_state <= SM_select;
       else
              next_SM_state <= SM_idle;
     SM_select:
       next_SM_state <= SM_grant;
     SM_grant:
       if (channel_req)
              next_SM_state <= SM_grant;
       else
              next_SM_state <= SM_idle;
    default:
       next_SM_state <= SM_idle;
  endcase
```

level simulation of the channel 16 test case at the SoC level. This successfully replicated the failure seen in the silicon and brought to our attention the fact that the root cause of the bug was an out-of-bound array indexing issue. As shown in Figure , when *channel* mistakenly gets a value of 17 during run-time, the *channel_req* signal, referencing *Arb_reqs[17]*, goes X and causes *next_SM_state* to always jump to *SM_idle* state due to X-optimism in simulation. This behavior does not match what happens in real silicon—*next_SM_state* can jump to *SM_grant* state.

Once realizing this is a design bug masked by X-optimism in simulation, we pondered how we could have caught this in pre-silicon verification flow. To make sure in the future we can prevent similar bugs from being masked, we tried a few methodology improvement experiments.

First, we evaluated *VCS Xprop* feature [8]. We enabled X propagation on the synthesizable portion of the design and rerun the channel 16 test cases at the IP level. The test cases now failed because X, generated from *Arb_reqs[17]* reference, can propagate to *next_SM_state* and eventually become observable to the test bench checkers.

To be sure we do not have any other related X-optimism issues, the second experiment we tried was to setup the formal x-propagation app for the exact Verilog module where we know the X was produced and consumed. The tool was able to quickly generate X checker assertions and run formal engines to produce definitive results. Many X checkers failed due to this one out of bound array indexing bug. The tool provided counter examples in waveform showing how the assertions can fail.

This formal approach was also used to guarantee the safety of the fix – a software workaround instead of a hardware ECO in this case. Based on the severity of the bug and the cost of re-spinning a chip, the decision was made to work around the bug by simply disabling channel 16 in software. In *VCS Xprop* simulation the full regression (with disabled channel 16) passed. However the simulation approach is always as good as your stimulus and checkers. To be absolutely certain we added a few constraints to model what software would do to disable channel 16 and reran the formal x-propagation app. All X propagation assertions, including previously failing ones, were proved passing this time. This provided a much higher level of confidence for the software workaround decision.

The formal approach is not without limitations. We realized that it is all in 20/20 hind-sight that we picked this particular Verilog module deep within the IP to run the formal X propagation tool. Had we not known of this bug, it would have been more likely that we would set the verification scope to the IP level. This time, we ran the formal x-propagation app on the whole IP to see if the tool could catch the bug. The results were not as good using the same two-hour run-time limit due to a much bigger DUV size. No violations of X propagation assertions were reported. Only bounded proofs were achieved for all X propagation assertions. In this case this information is not very helpful because we know an X-optimism bug can occur. This is a typical formal proof convergence issue that can be addressed in many ways. In this case we know we may have a better way, which is to first find and validate the X sources to be legitimate or intended before even tackling the X-optimism issue.

The tool we used this time is a formal structural property synthesis (SPS) app as mentioned in Section IV.D, which can detect whether there are array out of bound indexing issues in the design. The tool needed little setup effort other than compiling the synthesizable RTL design of the whole IP. In contrast to the proof convergence difficulty when running the x-propagation app, the SPS app was able to very quickly generate out of bound indices assertions and run formal engines to produce results. Many assertions failed and counter examples were produced to show violation sequences.

Through this experience, we learned that using the structural property analysis tool is a much less expensive way to detect out-of-bound indexing X sources. It can also help designers to analyze the validity of these X sources. In this case, the out-of -bound indexing occurrence was not intended by the designer. This led us to the conclusion that in our methodology, structural properties should be checked prior to running X-optimism formal check.

VI. Summary and Future Work

In summary, the methodology we recommend to address the X-optimism problem is an X-source-driven holistic approach centered around using a formal X-propagation app. Several other methods are used to complement the formal X-propagation tool itself, such as using reset analysis, out-of-bound indexing checks, and X-propagation-enabled simulators. We believe this is the most practical for us as a SoC team having to deal with IPs coming from many sources that we do not have control over.

Our X-optimism methodology is quite new, and we are in the process of experimenting with our current methodology for the next SoC chip. One area we are investigating is whether code coverage data collected from formal engine proof execution can be used to qualify bounded proofs as acceptable signoff criteria. Another major shortcoming we have is that the formal tool needs to understand UPF semantics and generate X accordingly when certain conditions are met. Other improvement areas are expected to be found as well.

Acknowledgment

We would like to thank Jennifer Hwang at Broadcom for the encouragement and support of writing this paper.

References

[1] *IEEE Standard for Verilog Hardware Description Language*, IEEE 1364-2005.

[2] *IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2009.

[3] Lionel Bening, "A Two-State Methodology for RTL Logic Simulation," DAC 1999.

[4] Stuart Sutherland, "I'm Still In Love With My X!," DVCon 2013.

[5] Mike Turpin, "The Dangers of Living with an X," SNUG Boston, 2003.

[6] *IEEE Standard for Design and Verification ofLow-Power Integrated Circuits*, IEEE 1801-2013.

[7] Atrenta, Spyglass, http://www.atrenta.com/solutions/spyglass.htm5.

[8] Synopsys, "VCS Xprop Datasheet," http://www.synopsys.com/Tools/Verification/FunctionalVerification/Documents/vcs-xprop-ds.pdf.

[9] Jasper Design Automation, "JasperGold X-Propagation Verification App," http://jasper-da.com/products/jaspergold-apps/x-propagation-verification-app.

[10] Jasper Design Automation, "Structural Property Synthesis App," http://jasper-da.com/products/jaspergold_apps/SPS_App.

[11] Laurent Arditi, "A Simple and Efficient X-propagation Checking Method Based on Formal Verification," DAC, User Track, 2011