Speedup Silicon Issue Debug with VC Formal

Wayne Yun

Advanced Micro Devices, Inc.

Markham, Ontario, Canada

www.amd.com

ABSTRACT

Observability is one of the most difficult challenges that slow down the debug of functional issues found during silicon bring-up. A waveform which is generally available during functional verification is impractical to be captured from silicon. Without knowing the complete series of events leading to a failure, designers have to come up with a hypothesis, and prove it. Typically, this process is desperate and requires significant effort. A formal property verification tool can find a trace which shows how an event happened given certain constraints. The formal tool can be setup to generate the invisible steps proceeding the failure. Following such a methodology, this paper is a user experience of using the Synopsys VC Formal tool to trace down the root cause of a silicon issue. It describes the flow used in the debug process, and suggests best practices. The formal approach helped saving time and effort in identifying the mechanism of the failure compared to the traditional method.

Table of Contents

1. Introduction 3
2. Formal Verification Methodology for Silicon Issue Debug 3
2.1 Formal Property Verification 4
2.2 FPV Applied on Silicon Issue Debug 4
2.3 Introduction to VC Formal 4
2.4 Typical Flow of Formal Silicon Issue Debug 5
2.4.1 Reproducing Stage 5
2.4.2 Fixing Stage 6
3. VC Formal on a Real Silicon Issue 6
3.1 The Issue 6
3.2 Chosing Nestlist over RTL 7
3.3 Initial Constraints 8
3.4 Creating an Assertion 8
3.5 Abstracting DUT 8
3.6 Refining Constraints 8
3.7 Un-black-box 8
3.8 New Assertion 8
3.9 After Thoughts 8
4. Conclusions 9
References 10

Table of Figures

Figure 1 Reproducing Stage 5
Figure 2 Fixing Stage 6
Figure 3 Related Design Blocks 7

Table of Tables

Table 1 Effort Comparison between Formal and Simulation 9

1. Introduction

It is always an exciting time when the first silicon comes back to the lab. Failures and observations are often found during bring-up and validation. Designers have to find the root cause, and decide if it is a bug and the action for it. Usually, this debugging process is in the critical path of a project. Formal verification could speed up this debugging process.

A typical front-end debugging process heavily relies on waveforms. The waveform is generated by a failing simulation. Every event over the course of the simulation can be saved in the waveform file. Debug is an iterative process which finds the immediate cause of current failure, then takes the cause as next failure, and eventually lands on the root cause of the simulation failure. Since every event is recorded, this routine always succeeds in theory. However, in reality the debug of silicon bring-up issue faces challenges.

Once a bring-up failure is reported, designers collect all available descriptions of the failure like symptom, stimuli, configuration, debugging trace and oscilloscope waveform. The waveform is not even close to the one generated by simulation in terms of completeness, because of the limited observability into internal nodes of the design. The consequence is that the chain of events is difficult to construct. It is often not obvious how the failure was caused and reached.

Then designers have to think of different hypotheses, test and collect evidence of them on the silicon, and simulate them in the original development environment to find and confirm the root cause. This is a backward reasoning process which starts from the result and ends at the cause. Simulation based technology does not provide a good support of this type of reasoning. Human imagination and creativity are key factors for tracing toward the root cause. It takes not only time, but also the hair of the designers.

In contrast to simulation, formal technology provides a way of finding all intermediate events between the initial state and the final result. A waveform is generated to show these events. Designers can analyze the failure in their familiar environment and, by similar process, from simulation.

In the rest of this paper, Chapter 2 starts with an introduction to formal verification, highlights Formal Property Verification (FPV), details principles of using FPV on silicon issue debug, and ends with a brief on the Synopsys VC Formal tool and a typical flow. Chapter 3 records steps the author took while debugging a silicon issue, along with some discussion. Chapter 4 summarizes results and recommendations.

2. Formal Verification Methodology for Silicon Issue Debug

Formal verification is defined in many ways. The definition used in this paper is that formal verification is the process of mathematically checking whether a design satisfies some requirements. The design usually is the DUT in simulation or a portion of it, and requirements are almost identical to checkers of simulation. However, no stimulus is specified for formal verification. Formal verification considers possible stimuli unless one is excluded explicitly. This feature is the source of exhaustiveness and quick turnaround of formal verification.

If a design doesn't satisfy a requirement, formal verification typically finds a counter example. The counter example can be converted to a trace or waveform. The waveform is most helpful for debug since it is more humanly readable. The process of identifying the root cause is the same as simulation.

The branch of formal verification applied to silicon issue debug is formal property verification.

2.1 Formal Property Verification

FPV is also called model checking or property checking. A requirement is a property, and FPV checks if the design is a model satisfying the property. If the design satisfies the property, the property is proven. If the design doesn't satisfy the property, the property is said to be falsified and a counter example is generated as a waveform.

Designs are typically required to be synthesizable. Properties can be expressed in many languages and in different ways. SystemVerilog Assertion (SVA) and Open Verification Library (OVL) are two common examples.

There are three types of properties used in FPV. The first one is assert property. An assert property is a statement of the design which is expected to be true for any stimulus. The waveform is an example of violating the assert property.

The second is assume property. An assume property is a constraint on stimulus. Only stimuli satisfying all assume properties are considered.

The third is cover property. A cover property is a statement of a scenario that could happen. The waveform is an example of satisfying the cover property.

All three types of properties are used for silicon issue debug.

2.2 FPV Applied on Silicon Issue Debug

The task of silicon issue debug is mapped into an FPV problem mainly in two areas. Then solving the FPV problem will generate a waveform example of how the issue happened.

The first area concerns the design. The amount of logic for a chip is too large to fit the capacity of an FPV tool. Ideally, a minimum portion of the chip having all logic related to the issue should be cut out and used as the design for FPV. Since the root cause is unknown at this step, it is impossible to accurately identify related logic. Good judgment is needed to include all logic contributing to the issue with high confidence. If the design is too small to contain the faulty logic, the issue won't be reproduced, which indicates a need to include more of the design.

The second area concerns properties. Assumptions should be created for clock, reset, neighboring blocks, scan logic, etc. An observation describing the issue should be converted to a coverage or an assertion.

If a coverage is created, it is the abnormal behavior observed from the silicon. The FPV setup will find a trace if it is possible to happen. Then this trace can be saved as a waveform, from which the root cause can be found.

If an assertion is created, it is the expected behavior of the design. The FPV setup will falsify it if a scenario violating the assertion can be found. This scenario can be saved as a waveform, and be confirmed if it is same as the silicon issue. If it is, the root cause can be found in the waveform.

The FPV tool used was Synopsys VC Formal.

2.3 Introduction to VC Formal

VC Formal [1] is the latest FPV tool of the Synopsys Verification Compiler series. It supports synthesizable RTL, SVA, etc. Its commands are TCL based and many of them are shared with other VC series tools. Its debug interface is widely used Verdi-based RTL and waveform visualization. Much existing knowledge and skills of tools and languages from simulation can be applied directly into formal verification.

2.4 Typical Flow of Formal Silicon Issue Debug

The flow is divided into a reproducing stage and a fixing stage.

2.4.1 Reproducing Stage

Figure 1 shows a typical working flow using formal verification methodology to reproduce a silicon issue.

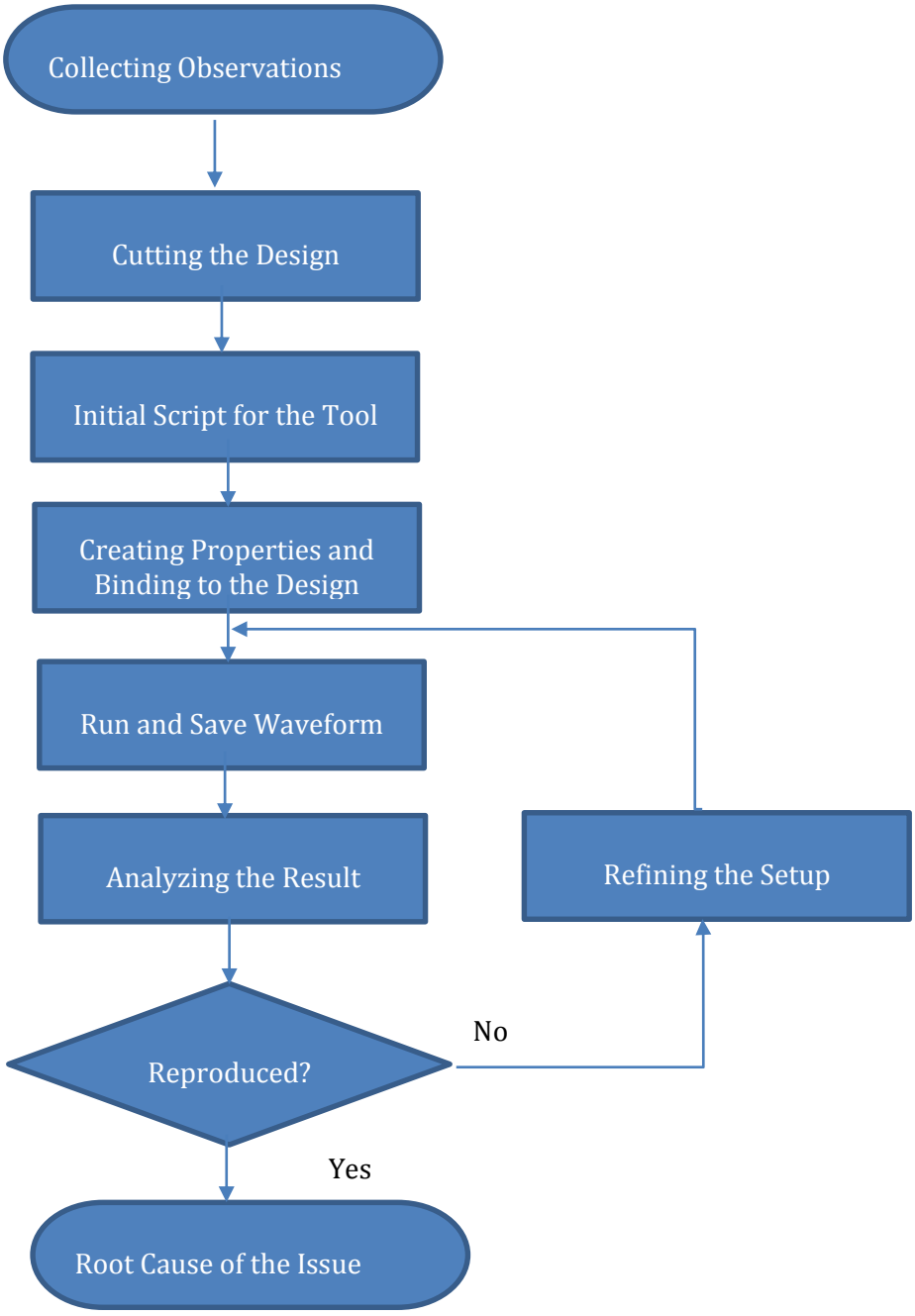

Figure 1 Reproducing Stage

The generated waveform is analyzed at "Analyzing the Result" step and checked against real world scenarios. The reproducing stage ends once the root cause of the silicon issue has been identified. The next stage is the fixing stage.

2.4.2 Fixing Stage

The fix of the issue is verified formally at the fixing stage. This stage is optional but recommended. The setup at this stage leverages the one from the reproducing stage. Figure 2 shows the typical steps.

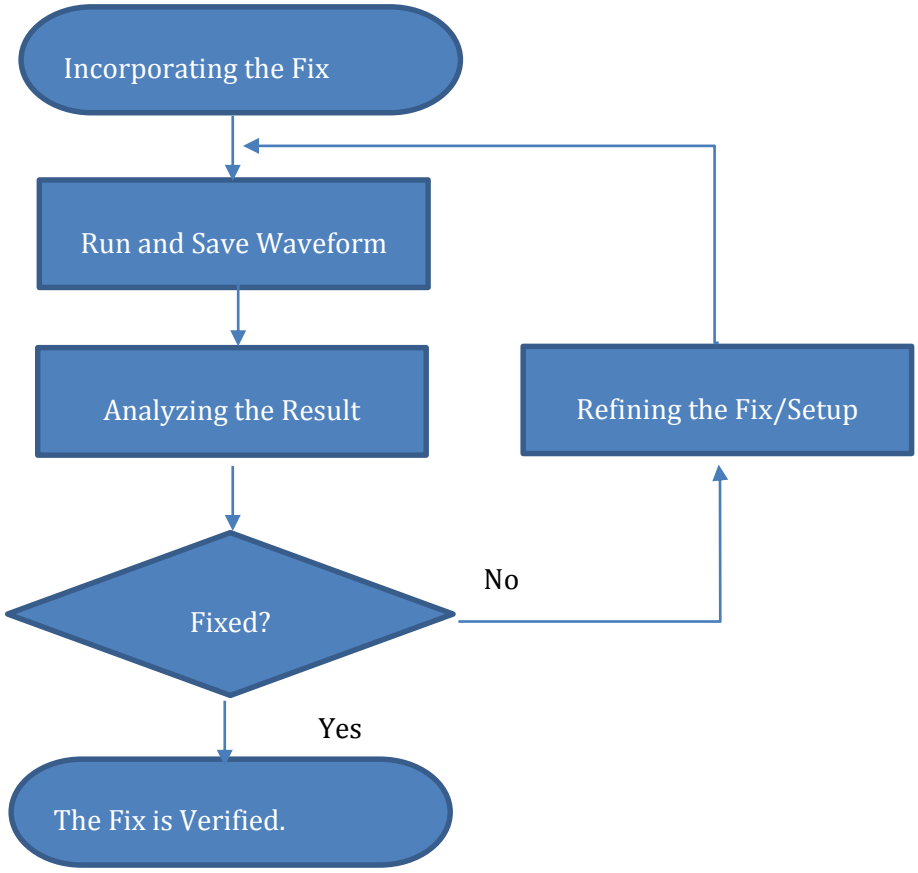

Figure 2 Fixing Stage

Flows could be customized based on the specific situation. Reference [2] is an example of enriching the reproducing stage.

3. VC Formal on a Real Silicon Issue

3.1 The Issue

Figure 3 shows related design blocks. The Reset block generates the reset sequence for all other blocks. The Arbiter block coordinates startup sequences and resource accessing requests from functional blocks F_A and F_B. All blocks have control and status registers. Reset and startup are driven by writes to these registers.

The Reset block provides the reset signal to all blocks. When reset is released, all blocks enter their startup sequences. The Arbiter block initiates a few rounds of handshakes with F_A and F_B. At one step, flip-flop Rel_B is set and enables a further startup sequence at F_B.

The observation was that F_B didn't transition to its working state after reset, though F_A did. The status register of F_B indicates that F_B either didn't complete its startup sequence or was stuck right before reaching its working state. The on-die-debug-system didn't provide enough clues.

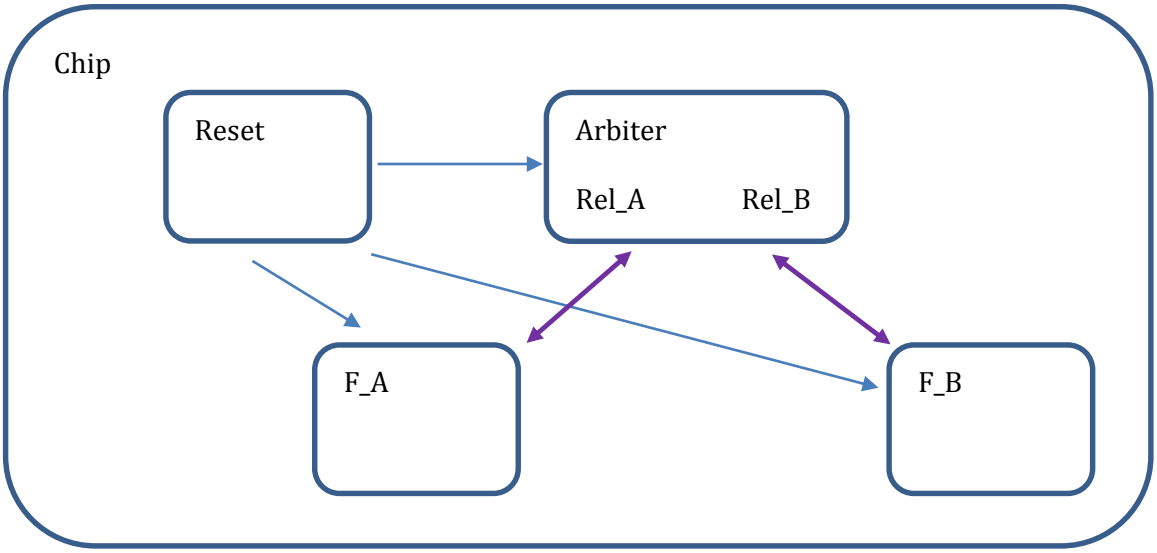

Figure 3 Related Design Blocks

The formal effort was started with limited information about the issue.

3.2 Chosing Nestlist over RTL

Generally, a netlist is preferred over corresponding RTL since the netlist is a more accurate model of the silicon than the RTL. There are issues only reproducible with the netlist but not the RTL. Another advantage of a netlist is that the design usually is only one file which consumes less time to setup. The disadvantage of a netlist is that no higher level logic exists, the formal tool has to deal with bits rather than words. This disadvantage is not noticeable if the logic is bit based at the RTL, which is the typical case for most control type logic.

The first difficulty appeared even before any code was written. There is a wrapper around all related blocks shown in Figure 3 along with a few others in the RTL. This wrapper would have been the ideal design for FPV. However, the wrapper was dissolved at the back-end and related blocks are scattered over the tile. On one hand, since connectivity in the netlist is reformed from the RTL, there is a chance of introducing errors. This factor favors the netlist. On the other hand, the design for FPV has to be increased to the tile which has more than two million instances. This factor disfavors the netlist.

The decision was to go with the netlist based on reasoning that unrelated blocks can be black-boxed later. As such, the formal flow will cover every possible failing mechanism, and the complexity is also under control. RTL could work too, just it won't find any issue caused by the difference between RTL and netlist, then another round of effort is needed and the issue is found later.

3.3 Initial Constraints

With hints from designers, the code was inspected. Some basic constraints are identified, like clocks, resets, scan signals, etc. They are defined and controlled from a VC Formal script.

3.4 Creating an Assertion

The observation of the issue is too general, it doesn't detail a sequence of signals. An Assertion is more intuitive because the startup sequence is well defined and every block is expected to give certain signals at certain steps. While looking for these expectations from the code and an RTL simulation waveform, it was noticed that flip-flop Rel_B had to be set at roughly mid-point of the startup sequence. If it had failed, the symptom would have been same as the observation. And it doesn't involve block F_A or F_B. So, the SVA assertion is the setting of Rel_B after reset is released. Block F_A and F_B are to be black-boxed. Below is the pseudo code of the assertion:

```
Assertion_Rel_B: assert property ($fell(firmware_reset)|->##3 Rel_B);
```

3.5 Abstracting DUT

With the initial setup, the hope of reproducing the issue quickly was broken when the tool kept running at the formal model creation stage for more than two hours. It was realized that the design was still too large for the tool to digest. With all second level modules that are not directly related to the prolbem black-boxed, the runtime was reduced to about 20 minutes, of which only 3 minutes were used by formal proving.

3.6 Refining Constraints

The first a few runs had the assertion falsified. Each waveform was analyzed, and each time a missing constraint was found. Most of them were on scan logic and clocking. This process iterated for about 6 times, then the assertion was proven. This indicates flip-flop Rel_B works as expected.

3.7 Un-black-box

The proven assertion of Rel_B shows that the startup sequence works as expected until that point. The issue happens after, and more logic, especially block F_B, most likely participates in. As such, block F_B was un-black-boxed.

3.8 New Assertion

A new assertion was created to a later point of the startup sequence of block F_B. It was the expected behavior toward completion of the startup sequence of block F_B. It should cover the period from reset to successful completion of the startup sequence. The assertion looks like:

```
Assertion_F_B: assert property ($fell(firmware_reset)|->##6 F_B_up);
```

The runtime was increased to about one hour, only five minutes of which were for formal proving. About 5 iterations were spent in tuning the assertion, and eventually the symptom of the observation was reproduced.

The bug was in the Arbiter block. It was triggered if block F_B was started later than block F_A for more than a certain number of cycles.

3.9 After Thoughts

The effort took totally about 2.5 days. Among all decisions along the way, some were good and saved

time, some worked out but could have used an alternative to reduce the total effort.

1. The decision of starting with flip-flop Rel_B saved time at the later refining constraint stage. A smaller design and a simpler assertion reduced the turnaround time of each iteration.
2. Using more intuitive assertions rather than coverages saved debugging time too.
3. That assertions used internal signals in antecedents avoided constructing register accessing cycles. Pseudo code is below:
   ```
   Astn: assert property (reg_output |=> result);
   ```
4. More modules should have been black-boxed at the very first run.
5. Better understanding of DUT/Scan, more experience with assertions and formal verification would have generated positive contributions.
6. Save/restore might save compile time. However, assertion would not be able to be written in SVA since it needs to be recompiled for every change.

The issue itself is also an excellent candidate for formal verification because the failure state is not far away from the reset.

4. Conclusions

The formal debug effort was started after, and went in parallel with, simulation. Both of them ended at about same time. Table 1 compares these two efforts.

Table 1 Effort Comparison between Formal and Simulation

Item	Formal	Simulation
Calendar Time	2.5 days	5 days
Total Effort	3 man-days	7 man-days
Verification Effort	2.5 man-days	5 man-days
Design Effort	0.5 man-day	2 man-days
Skill of Verification	Formal	Simulation

A few observations can be drawn from above table:

1. If formal were started at the same time as simulation, the calendar time for debugging the issue would have been reduced by 50% from 5 days down to 2.5 days.
2. If formal were started at the same time as simulation, and both were run in parallel, total effort would have been about same as simulation alone.
3. If formal were run alone, total effort would have been reduced by 57% from 7 man-days down to 3 man-days.
4. If formal were run alone, design effort would have been reduced down to a quarter.

Generally, designers work on specific blocks. On the opposite, formal verification engineers are not specific to any block, though deeper knowledge does help. However, it is not mandatory. This feature may provide managers with some extra flexibility in resources allocation.

Formal silicon issue debug requires making instantaneous decisions under tremendous pressure. To make it worse, not much external help can be used. This venue definitely is not the first exercise for anyone who just started with formal verification. For those who made up their mind to proceed, below are suggested:

1. Using a netlist to cover more possibilities, especially when cell definitions had been checked

and are compatible with the formal tool before started, otherwise RTL.
2. Choice between assertion and coverage. Write the one that fits known facts.
3. Using internal signals. Avoid specifying complicated sequences at ports.
4. If uncertain about completeness of constraints, start with a simple property.
5. Black-boxing as many modules as possible at beginning.
6. Be prepared. Practice before deployment.

May every reader gain full benefit of formal silicon issue debug!

References

[1] Synopsys, "VC Formal Verification User Guide", 2015.09

[2] C. Richard Ho, Michael Theobald, Brannon Batson, J.P. Grossman, Stanley C. Wang, Joseph Gagliardo, Martin M. Deneroff, Ron O. Dror, and David E. Shaw, "Post-Silicon Debug Using Formal Verification Waypoints," DVCon 2009