Using Formal Techniques to Verify System on Chip Reset Schemes

Kaowen Liu, Penny Yang
Design Technology Division
MediaTek Inc.
HsinChu, Taiwan
Kaowen.Liu@mediatek.com
Penny.Yang@mediatek.com

Jeremy Levitt, Matt Berman, Mark Eslinger
Design & Verification Technology
Mentor Graphics
Fremont, U.S.A.
Jeremy_Levitt@mentor.com
Matt_Berman@mentor.com
Mark_Eslinger@mentor.com

Abstract - **Complete verification of SoC (System on Chip) reset is a fundamental requirement and, in practice, a prerequisite to ensure correct operation. Today's large SoC designs integrate many design IP (Intellectual Property) blocks, each with its own implementation of reset. This is a significant challenge for design teams, since there is little or no standardization of block level reset circuitry. Moreover, system level reset strategies become increasingly complex as they combine various sources of reset.**

SoC designs have multiple sources of reset, such as power-on reset, hardware reset, software reset, and watchdog timer reset. Simulation alone cannot efficiently verify all reset scenarios. We present a highly automated methodology using Formal verification, instead of simulation, to completely verify reset schemes without the significant manual effort required for simulation-based verification.

Adding to the challenge of reset verification are the fundamental differences in the way an X-State is interpreted in RTL (Register Transfer Level) simulation versus how it is treated by synthesis tools. Synthesis interprets X as *don't care*; RTL simulation interprets X as unknown. This difference can result in silicon that behaves differently than observed during simulation of the RTL. Our methodology combines Formal techniques with automatically generated SystemVerilog assertions to detect X-States that result from bugs in the reset and general design logic.

Increasingly, Formal verification techniques are being employed to supplement simulation-based verification in targeted areas. At MediaTek, we have found that SoC reset validation is an area where Formal technology is more efficient than simulation-based verification methodologies. Along with Mentor Graphics, we have developed and deployed Formal applications that design teams can use without knowledge of SVA (SystemVerilog Assertion) or Formal techniques.

Keywords—Formal Verification; Reset Scheme; SoC; SVA

I. Introduction

When real silicon comes back, it is the most exciting moment to watch as the chip is powered on and starts to work as expected with the first basic test pattern. It can also be the most frustrating moment if the chip doesn't work and the design team realizes that the design bug could have been found in the early RTL (Register Transfer Level) simulation stage.

Mike Turpin's paper [3] in 2003 raised awareness of X issues in the design flow and how X-bugs can be missed by RTL simulation and equivalence checking. To avoid X-bugs, designers are supposed to follow good RTL coding guidelines. Unfortunately, even though designers try their best to avoid X related issues, we still see X-bugs in real chips. These silicon bugs reflect the challenges we face with the traditional simulation-based verification methodologies used in the design flow at MediaTek.

The purpose of this paper is to discuss how Formal techniques can be applied at the RTL verification stage to detect design bugs in reset schemes and initialization that result in X-States which cause errors in silicon. Although simulation-based verification is capable of detecting these design bugs, it is time consuming and requires a significant manual effort. We show how to efficiently tackle this verification challenge using Formal techniques.

Formal technology has tremendous promise to supplement traditional simulation-based methodologies, but the technology has its own challenges: among them, how to generate correct SVA (SystemVerilog Assertion)? Instead of educating design teams about Formal techniques and teaching them how to write and debug SVA, the DT (Design Technology) division in

MediaTek has developed several Formal applications which automatically generate SVA for specific targeted areas. The DT division has been doing this since 2010. In this paper, the SVA generated to addresses reset scheme validation and X-State detection is discussed in detail.

II. Global Reset Verification

A. *Global Reset is Complicated and Needs to Be Verified*

SoC designs have multiple sources of global reset, such as power-on reset, hardware reset, software reset and watchdog timer reset. These are top-level signals that should be connected to all asynchronous resets in the design, i.e., all asynchronous resets in the design should be asserted when the global reset is asserted. Unfortunately, a missing reset connection, or one that is blocked from propagating due to the mode of operation of the design, can easily go undetected. It is hard for simulation to verify all reset scenarios, to verify that every source of reset propagates to all the intended storage elements in the design under all the proper conditions.

B. *Watchdog Reset Verification*

Taking watchdog reset verification as an example, a directed test, specially crafted to verify if the watchdog reset operation works correctly, needs to trigger the watchdog reset condition in the middle of the simulation and check if both (1) the watchdog reset has been propagated to the intended flip-flops and (2) verify that the design can resume normal operation mode after that. While checking that the design resumes normal operation mode is fairly straightforward, verifying in simulation whether the watchdog reset has actually been propagated to every intended flip-flop is very tedious, since there are a lot of flip-flops to check. And, if the watch dog reset is not correctly connected to a flip-flop, the directed test will still pass as long as the reset value happens to match the value on the unconnected flop at the point in simulation when the watchdog reset operation occurs. Thus, a passing test does not mean that the watchdog reset has been verified. Without completely verifying that the watchdog reset has been propagated to every intended flip-flop, simulation-based watchdog reset verification is incomplete, and bugs may still sneak into the design, cause system failures in lab testing, and require a silicon re-spin to fix.

C. *Applying Formal to Verify Global Reset Is More Efficient*

Global reset failures can be caused by missing or wrong reset connections, design bugs residing in the logic between the global reset source and the reset pin of the intended storage element, sequencing errors – there are many potential sources. Instead of crafting simulation-based tests (directed or constrained random) for all possible scenarios to verify that the global reset is correctly propagated to the intended storage elements and indeed resets them, it is much more efficient to add SVA on the reset pins of the intended storage elements and let the Formal engine prove that these local reset pins will be asserted whenever the designated global reset condition is met. An example SVA is shown as the following.

```
global_reset_check: assert property
(@($global_clock) `GLOBAL_RESET |-> ##`DELAY _rst);
```

`GLOBAL_RESET defines the global reset condition which is in the form of an SVA expression. This is the antecedent of the implication. `DELAY is determined by the actual cycle delay between the global reset condition being satisfied and the local reset signal being asserted. To eliminate unnecessary false firings, a maximum delay could be used when the exact cycle delay between the global reset condition being satisfied and the local reset signal being asserted is not critical. "_rst" is the signal which is directly connected to the local reset pin of the intended storage element. Our objective is to verify the connectivity and logic between the global reset and the local reset pin of the intended storage element, thus, by not leaving any logic between the "_rst" signal and the reset pin of the intended storage element, we let the Formal engine verify the entire path. The same is true of the `GLOBAL_RESET part: except for the required signals and expression which make up the global reset condition, we should leave as much design logic as possible for the Formal engine to verify.

III. X-State Detection

A. *X-optimism in RTL Simulation Can Hide Design Bugs*

The Verilog X value can be intentionally used by designers to express a *don't-care* for logic synthesis in order to achieve a better netlist, although it does not necessarily generate the minimum netlist as discussed in [3], and in RTL simulation to express *don't-care/wildcard* in *casex/casez* comparison. It can also be unintentionally introduced by designers through a design bug, and represents *unknown* in RTL simulation.

Design teams mostly rely on RTL simulation to functionally verify their design, before they send their design further downstream in the design flow. As shown in [3], RTL simulation treats an X value optimistically by just taking one if/case branch, and that means design bugs can be hidden by the X-optimism of RTL simulation. Gate level simulation may expose some X issues but it is not applicable for regression testing because of its low performance. Equivalence checking tools are mainly relied upon to verify that the gate level netlist is correct, but they verify that the RTL and the gate level netlist match under synthesis semantics; they do not consider the behavior of X values in simulation. Thus, an effective and efficient way to detect unintentionally introduced X values in RTL is needed.

B. *Design Bugs Cause X-State*

Designers can easily unintentionally introduce X-States. Here is an example which shows how a design bug caused an X-State:

```
reg [9:0] w_slot;

assign y_data = w_slot[x_select[3:0]];
```

In the above RTL code, the designer used a 4-bit *x_select* selector to select one element from the *w_slot* vector which has 10 elements, and then assigns the value of the selected element to *y_data*. As long as the 4-bit *x_select* selector ranges between 0 and 9, *y_data* is assigned to a deterministic value from one

element of the *w_slot* vector (assuming that the selected element already has a deterministic value). However, the designer negligently allowed the 4-bit *x_select* selector's value to exceed 9, the upper bound of *w_slot* vector's index, and *y_data* ended up assigned to an X-State.

Here is another example which shows how a design bug caused an X-State:

```
always @(posedge clock)
begin
  if (update && select)
    reg_to_be_read <= data;
end
```

To save silicon space, designers use filp-flops without reset circuitry. In the above RTL code, *reg_to_be_read* starts off at time zero in an X-State before it is assigned a deterministic value from *data* when *update==1'b1* and *select==1'b1*. In RTL simulation, it is hard to prove that *reg_to_be_read* will always be in a deterministic state before it is read when the design is in a normal operation mode.

These X-State values may or may not cause test failures in RTL simulation depending on factors such as whether the X-State is masked out by simulation X-optimism or if the test checkers compare the propagated X.

C. *Resettable Flip-Flop Should Not Output X-State After Initialization Is Completed*

It is acceptable for an X-State to persist in a design for a while, as long as it does not break the design's normal operation. For example, the shift register shown in **Figure 1** may contain an X-State even after the design's reset sequence is completed. As data is fed into the shift register, the content of this shift register will gradually become deterministic. As long as the reader of the shift register starts to read the shift register after its data has become deterministic, the X-State causes no harm to the design. Actually, to accommodate a variety of design needs, this is a common design practice.

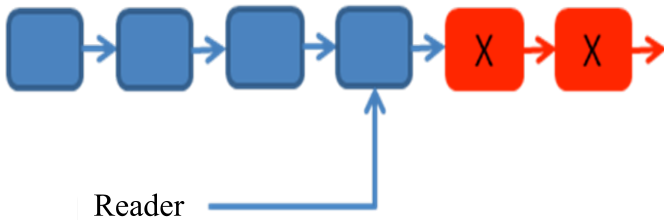

Figure 1 – Shift Register

On the other hand, every synchronous digital design relies on its reset sequence to put itself into a known initial state where all FSM (Finite State Machine) and control registers are at the predefined start point intended by the designers. Thus, all resettable flip-flops should be in a known state, either 1'b1 or 1'b0, after the reset sequence is completed. This is, of course, usually done by asserting/de-asserting the reset pin. The following is an example SVA property to assert that all resettable flip-flops are in a known state.

```
x_check: assert property
(@(posedge _clk) ((^_reg) !== 1'bX) |=> ((^_reg) !== 1'bX));
```

IV. Formal Verification Flow

Increasingly, Formal verification techniques are being employed to supplement simulation-based verification in targeted areas. At MediaTek, we have found that SoC reset validation is an area where the strengths of Formal verification techniques can be more efficient than simulation-based verification methodologies. In the past, we relied upon directed tests, which we knew had limited coverage. With constrained random simulation, the manual effort required to verify SoC reset schemes with assertions and coverage is high. Thus, we developed Formal solutions with Mentor Graphics to automatically generate SVA to tackle various verification targets. Furthermore, these solutions are push-button, so that design teams can benefit from them without any knowledge of SVA or Formal techniques. With this new approach, we leverage the strength of Formal verification techniques to exhaustively explore all possible conditions with respect to SoC reset schemes.

Figure 2 shows the Formal Verification Flow for Global Reset Verification and X-State Detection.

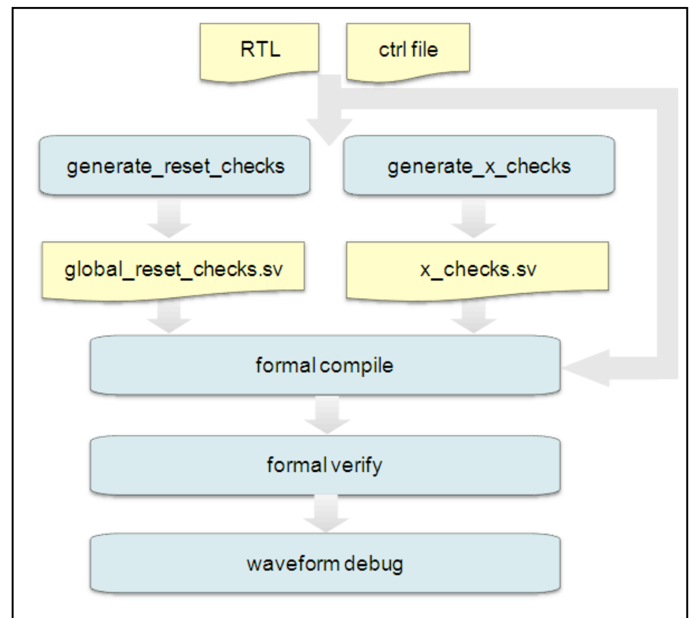

Figure 2 - The Formal Verification Flow

A. *Global Reset Verification*

1) Generate reset checks: Use Questa Formal's "generate_reset_checks" application to parse the design's RTL code and search for all asynchronous reset signals. As shown in **Figure 3**, SystemVerilog assertions (global_reset_checks.sv) are automatically created to verify that the global reset is correctly connected to all asynchronous reset signals in the SoC.

```
module global_reset_checks ();
`define GLOBAL_RESET    \
  this_signal_must_be_defined_by_the_user
`define DELAY 0
`define CHECK_GLOBAL_RESET_MACRO(_rst)      \
   @($global_clock) `GLOBAL_RESET |->    \
     ## `DELAY _rst

// GLOBAL_RESET checks for async reset used by
// 1 register:
// my_dut.A0.o
wire async_reset_id0 = (( ! my_dut.A0.arst_n) === 1'b1);
global_reset_check_id0: assert property
(`CHECK_GLOBAL_RESET_MACRO(async_reset_id0));

…………

// FILE: ./questa_sva_global_reset_checks.sv
// GENERATED: Wed Oct 19 16:00:00 2012
// 2 distinct async reset signals
// 2 registers with async reset
endmodule
```

Figure 3 - Global Reset Checks

2) *Formal compile, verify and debug:* Use Questa Formal to prove or fire the global reset SystemVerilog assertions. The Formal results can be listed in the GUI as **Figure 4** shows. And the debug waveform, schematic and source code are shown in **Figure 5**.

Properties

		Name	Type
⊙	F	global_reset_checks.global_reset_check_id0	sva
⊙	F	global_reset_checks.global_reset_check_id2	sva
⊙	P	global_reset_checks.global_reset_check_id1	sva
⊙	P	global_reset_checks.global_reset_check_id3	sva
⊙	P	global_reset_checks.global_reset_check_id4	sva

Figure 4 - Formal Results of Each Property

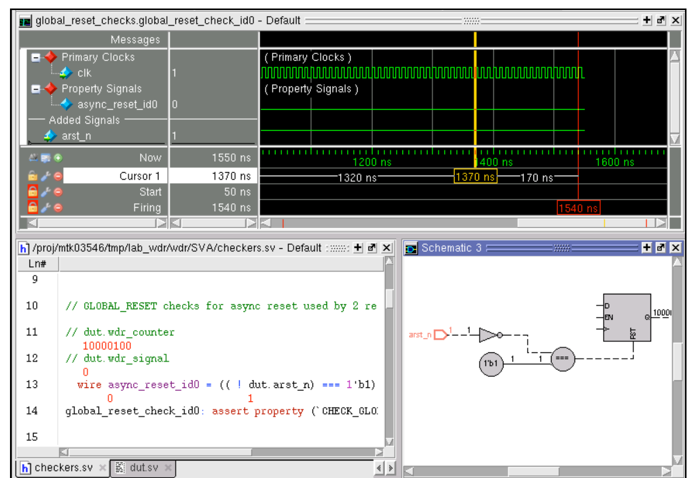

Figure 5 - Debug with Waveform, Schematic and Source Code

B. *X-State Detection*

1) *Generate X checks:* Use Questa Formal's "generate_x_checks" application to parse the design RTL code and search for all asynchronous reset flip-flops, synchronous reset flip-flops, registers with enable and primary outputs as **Figure 6** shows. Asynchronous reset flip-flops should never be in an X-State once the reset assertion/de-assertion procedure is completed. The other three types of design elements should be checked under proper conditions, such as reset with the active clock edge, or the enable condition with the active clock edge. The corresponding SystemVerilog assertions (x_checks.sv, as **Figure 7** shows) are automatically created to detect any propagation of unknown values after the registers have been initialized.

asynchronous reset DFF:
always @(posedge clk or negedge rstn)
  if (~rstn) reg <= …;
synchronous reset DFF:
always @(posedge clk)
  if (rst) reg <= …;
registers with enable:
always @(posedge clk)
  if (enable_expression) reg <= …;
primary outputs:
output po1, po2;

Figure 6 - The Classification of Registers and Primary Outputs in X-Checks

```
module X_check_s #(parameter WIDTH = 1) (
   input _clk,
   input [WIDTH-1:0] _reg);

x_check : assert property
    (@(posedge _clk) ((^_reg) !== 1'bX) |=> ((^_reg) !== 1'bX));

…
endmodule
```

Figure 7 - The Format of X Checks

2) Formal compile, verify and debug: Use Questa Formal to fire the assertions with X. The GUI mode of Questa Formal can be used to debug the firing as we described previously in section "*A. Global Reset Verification*". For our designers who are accustomed to using Verdi [5], we automatically extract a waveform and an rc file so that the firing can be debugged in the environment they are most familiar with, as displayed in **Figure 8**.

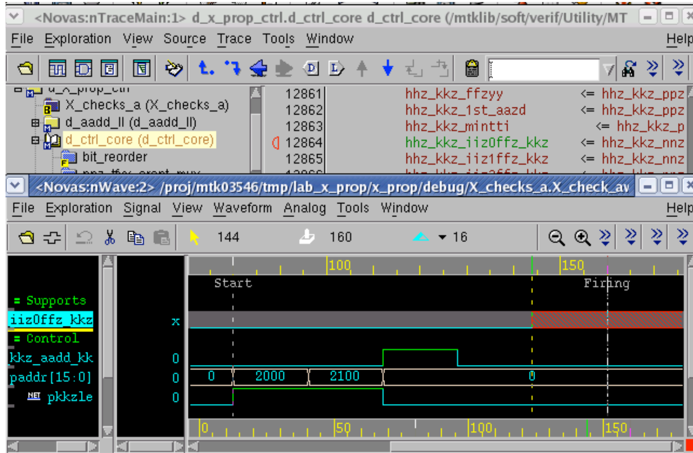

Figure 8 - Debug Formal Results with Verdi

V. Results and Summary

The global reset verification and X-State detection methodologies described above have been deployed on multiple projects at MediaTek. They have caught many design bugs and proven to be of immense value.

A. Results of Global Reset Verification

The design under verification is an SOC with 3738047 register bits. With Questa Formal version 10.1b, 14835 assertions were generated in 61 minutes, compiled in 6.5 hours and analyzed in 2.3 hours without any inconclusive checker. 14373 assertion properties were proven, 462 assertion properties were fired, and 248 bugs were found among 3 different sources. Most of the firings were due to connection bugs. However, some firings were caused by the design's testing logic.

TABLE I highlights the results of using the global reset verification flow on our SoC design. It is also impressive progress that Questa Formal 10.1b, compared to previous versions, can prove a whole SoC size design in hours without any inconclusive properties.

TABLE I - Results of Global Reset Verification on the SoC

Design size	3738047 register bits
SVA	14835 assertions
Run time	gen_sva: 61 min compile: 6.5 hr prove: 2.3 hr
Formal result	fired: 462 proven: 14373 inconclusive: 0
Bug	248 connection errors from 3 modules

B. Results of X-State Detection

X-State Detection flow can be applied at different levels in the design hierarchy. Here, 3 designs under verification are shown for demonstration purposes. The smallest one has 2357 register bits; 319 assertions were generated and they were completed within 1 minute, with 2 bugs found out of 24 firings. The medium size design has 14136 register bits; 1874 assertions were generated and they were completed in 48 minutes, with 2 bugs found out of 38 firings. The largest design has 167632 register bits; 19907 assertions were generated and they were completed in 4 hours and 20 minutes, with 6 bugs found out of 195 firings.

TABLE II contains the results from the application of the X-State detection flow to three designs with different sizes.

TABLE II - Results of X-State Detection in Different Design Scales

Design size (Register bit)	SVA	Firing	Bug	Run time
2357	319	24	2	1 min
14136	1874	38	2	48 min
167632	19907	195	6	4 hr 20 min

From the results, we can approximately see the trend of run time versus design size as **Figure 9** shows. A module-level design can be done in minutes and is easier to debug. As design and SVA size become larger, the run time becomes longer, so we recommend applying the X-State detection flow on module-level designs for shorter run time and easier debugging.

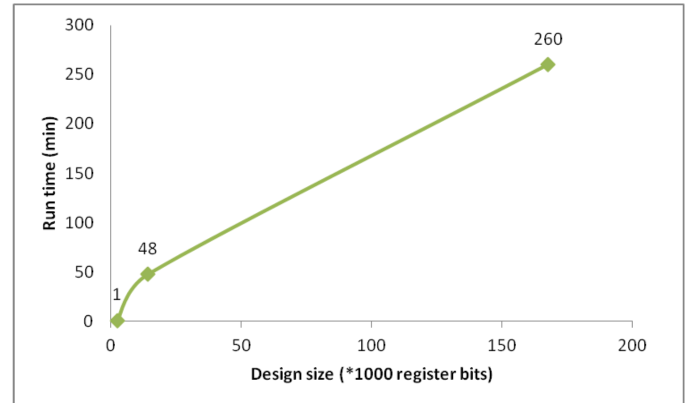

Figure 9 - The Trend of Run Time versus Design Size in X-State Detection

VI. Conclusion

This paper presents highly automated methodologies using Formal techniques to verify the correctness of global reset schemes and perform X-State detection, without the large amount of manual effort required by simulation-based verification. The methodologies described above have been deployed on multiple projects at Mediatek. They have caught

design bugs which would have been missed by the existing verification practices. As the results show, we have found that SoC reset validation is one of those areas where the strengths of Formal techniques can be more efficient than simulation-based verification methodologies. The Global Reset Verification flow is able to complete in hours, without any inconclusive properties, on an SoC size design with Questa Formal 10.1b. To shorten the run time and facilitate debugging, we recommend applying the X-State Detection flow on module-level designs. In summary, the application of Formal techniques and automatically generated SVA to verify reset schemes enables design teams to gain much greater confidence in their designs with less effort.

VII. FUTURE WORK

Formal verification techniques have proven themselves to be a good way to supplement simulation-based verification methodologies. There are many other areas, similar to reset scheme verification, where the application of Formal verification techniques looks promising. We will keep working on different topics. Our goal is to reduce manual effort, raise the degree of automation, and achieve higher verification confidence.

ACKNOWLEDGMENT

We would like to thank Whitney Huang from MediaTek for initiating the request of the Global Reset Verification flow, Joseph Hou from MediaTek for giving comments to enhance this flow, Chunyi Lin for providing the data and help with checking design issues and intentions, and Roger Sabbagh for his help with the writing.

REFERENCES

[1] Lionel Bening, “A Two-State Methodology for RTL Logic Simulation”, Proceeding 36th Design Automation Conference, June 1999.

[2] Lionel Bening and Harry Foster, Principles of Verifiable RTL Design, Kluwer Academic Publishers, May 2001.

[3] Mike Turpin, “The Dangers of Living with an X (bugs hidden in your Verilog)”, Version 1.1, October 2003, http://www.arm.com/files/pdf/Verilog_X_Bugs.pdf

[4] Questa® Formal User Guide, version 10.1b, 2012.

[5] SpringSoft Verdi™ Automated Debug System http://www.springsoft.com/products/verdi