How to Kill 4 Birds with 1 Stone: Using Formal Verification to Validate Legal Configurations, Find Design Bugs, and Improve Testbench and Software Specifications

Saurabh Shrivastava, Kavita Dangi, Darrow Chu, Mukesh Sharma
Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124
saurabh.shrivastava@xilinx.com, kavita.dangi@xilinx.com,
darrow@cadence.com, mukesh.sharma@xilinx.com

ABSTRACT

This paper describes how we are able to simultaneously use formal verification for four different purposes: to configure and validate legal set(s) of IP configurations with a SVA assertion library that captures the operating parameters and/or limitations in configuration of the device, to eliminate bugs that can arise with X-assignments and bus contention, to help improve the block and system-level simulations by flagging illegal configuration(s), and to help improve the executable specification for software that will ensure any assemblage of IP blocks by users in the FPGA are correct by construction.

This new methodology has completely replaced a largely manual and error prone simulation-based process that could only verify a handful of representative configurations. It was also very difficult to discover illegal configurations. The holes in the coverage space left us exposed to potential bugs and downstream ECOs.

Results from this new, exhaustive, formal-based methodology resulted in the discovery of completely unexpected bugs, as well as illegal configurations that were discovered early and corrected at very minor cost. Otherwise, some of these bugs might not be caught altogether, or the cost of fixing them in later stages of project would have been more. This also helped us free up resources for more critical work in the later stage of the project. According to our estimates this flow saved us atleast one month in the verification schedule.

KEYWORDS:

Assertions, SVA, ABV, Formal Analysis, Configurations, Parameters, Contention

1. – INTRODUCTION

By definition, the FPGA fabrics that we design & verify are highly configurable. However, it's not like Lego where any block can be arbitrarily connected to any other in any configuration: each block has a set of basic connectivity parameters in addition to dynamic, operational behaviors that are only compatible with certain modes / other IPs / protocols mapped on the device. Consequently, one of the highest priority design implications is the need to check for bus contention and X-propagation amongst all possible legal configuration scenarios. With each new generation, the configuration space is becoming larger in terms of size and

complexity. And consequently, it becomes ever more difficult to keep track of the legal configuration space(s) for design and verification of the fabric device, and making legal sets of configurations available to software and FPGA end users.

2. – PREVIOUS, SIMULATION-BASED APPROACH

Our baseline process was to apply a common directed test and constrained-random testbench simulation methodology to verify multi-block assemblies. We would sweep the inputs for a variety of scenarios, and check for X-assignments and/or bus contention. Wherever possible we also used static linting to root out bus floating and multi-driver issues. While this would catch a few bus contentions and related issues, as we would learn from downstream validation it certainly didn't catch all of them. Indeed, while our process was highly automated, it suffered from an inherent limitation of constrained-random in that it's very challenging to hit all possible statements and use cases (at least within the given project schedule.)

Another issue was that there was no mechanism in the design's RTL to check for deviations from legal operating parameters. In short, this challenge was beyond what a Perl or C program could easily and clearly generate. This limitation carried over to our design generation software which has the ultimate task of enabling FPGA users to configure our device in a legal configuration.

3 - THE NEW APPROACH: FORMAL ANALYSIS

Since we were seeking a more complete, ideally exhaustive exploration of the state and configuration space, our thoughts turned to applying formal tools.

We were already beginning to build SVA assertion libraries in RTL designs to check for illegal configuration settings. Using a simple coding-style trick we converted these checkers into constraints for formal without making changes in our RTL code. This technique involved supporting a pre-compiler directive in our assertion library to convert assertion-checks in simulations to assume-constraints in formal. Here is an example code snippet:

```
`ifdef FORMAL
   Assume_prop: assume property ((@posedge
clk) (cond));
`else
   Assert_prop: assert property (@(posedge
clk) (cond));
`endif
```

Next, we leveraged a formal tool capability that reads in block-level RTL and these SVA constraints to perform what our vendor calls "Automatic Formal Analysis" or (AFA) (others use the term "assertion synthesis") for items like X-check, bus-contention, FSM checks, etc. We then re-used these machine generated assertions for the four sub-flows described in the subsequent sections. A high-level view of the complete flow is shown in Figure 1 below.

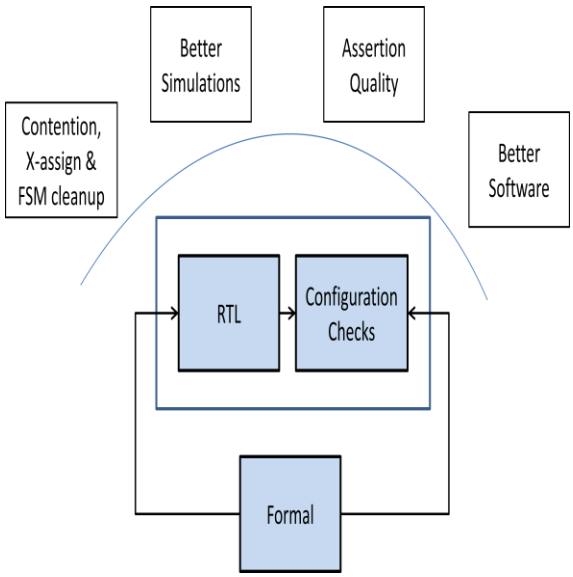


Figure 1:
The overall flow and its resulting benefits.

As shown above, the formal tool effectively takes the DUT's RTL itself and the configuration spec as input, automatically generates assertions from these inputs, and then formally verifies these assertions which capture the legal properties to remain valid when the given DUT IPs are configured and assembled into the finished product.

4 – ASSERTION LIBRARY CREATION

The first step of the process is to create a reference library of assertions for a given block that enforces legal set of configuration space(s). These assertions are then used by our formal flow as input constraints. The assertions themselves are coded using an internal SVA assertion library coded like macros. Here is a definition of one such library assertion that checks for one hot value on a variable "a":

```
`define LIB_ASSERT_ONE_HOT(INST,MSG,a)
   `ifdef FORMAL \
      assume_one_hot_``INST``: assume
property (@(posedge clk) $onehot0(a)); \
   `else \
      assert_one_hot_``INST``: assume
property (@(posedge clk) $onehot0(a)) \
       else begin $error ("%s: one_hot
fails for a=%h\n",MSG,a); \
   `endif
```

Here is an example of how this library assertion macro is used in a block RTL:

```
reg [3:0] varA;
…
`LIB_ASSERT_ONE_HOT(22,"varA value is not
compliant",varA)
```

These assertion checks are used for flagging illegal configuration space values in dynamic simulations. Here is what a failure looks like:

```
ncsim: *E,ASRTST (lib_assert_macros.sv,17):
(time 17 NS) Assertion
top.dut.modA.assert_one_hot_22 has failed

varA value is not compliant: one_hot fails
for a=b
```

These assertion checks are converted into assume constraints for formal under-the-hood in the library

- `iev -f filelist.f +define+FORMAL`

This checker library is then used for our highest priority/concern bus issues: floating-bus, multi-driver / bus-contention, X-assignment, FSM checks.

For example, consider an X-assignment case below, where {sel3,~sel2,sel1,sel0} is used as a select for the multiplexer output "mux_out". The default condition for this case statement is a catch all should the select lines comes up with any value other than one-hot.

```
case({sel3, ~sel2, sel1, sel0})
   4'b1100: mux_out = 1'b0;
   4'b0000: mux_out = latsr;
```

```
        4'b0110: mux_out = latsr_s36;
        4'b0101: mux_out = fifo_rstram;
        default: mux_out = 1'bx;
    endcase
```

The assertion for unreachability of X-assignment statement above is generated internally by the formal tool instead of us having to hand-write it for all such scenarios in our design. And if there is a missing or incorrect one-hot constraint on {sel3,~sel2,sel1,sel0} in the formal proof, that assertion would fail – in short, it's exactly the type of bug that we are trying to catch. Here is how this failure looks like in the tool:

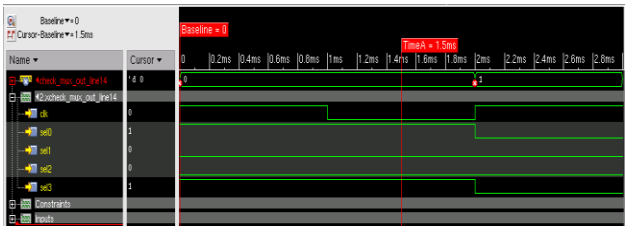


Figure 2: Assertion failure that identifies a bug; in this case, the top waveform clearly flags the issue

The tool can thus identify any bug in the RTL code above, or if there is a missing or incorrect assume property (which is a bug as well in our flow).

5 – VALIDATION OF DESIGN CONFIGURATOIN CHECKERS

We converted our configuration assertions library into formal analysis constraints without making any changes in the design. Consequently, they prevent and/or guide the formal engine from coming up with illegal configuration values. If these checkers are incomplete or missing, then AFA checks would fail (if the related configuration variables are in cone of logic) because of incorrect constraints. Hence this flow allows us to clean-up these checkers. (And if not for the machine-generated AFA checks, we would have to completely rely on time-consuming, painful simulations for this clean-up step which may not find all such issues.) Additionally, the downstream simulation-based design & verification methodology comes up with random configurations for blocks, these SVA-assertions help catch any potential illegal configuration settings.

Here are some examples of SVA assertions from the library coded using macros:

```
`LIB_ASSERT(mode_chkA,"Dev modeA is
incorrect",(BITEN==1 && (WIDTH==16 ||
WIDTH==32)))
```

This library assertion makes sure that when user programs BITEN with a value 1, then the WIDTH value should be either programmed to 16 or 32.

```
`LIB_ASSERT(mode_chkB,"Dev modeB is
incorrect",(BITEN==0 || WIDTH==64 ||
WIDTH==128))
```

This library assertion makes sure that when user programs BITEN with a value 0, then the WIDTH value of only 64 or 128 is possible.

```
`LIB_ASSERT_RANGE(range_chk,"Param range is
incorrect",PARAM,0,100))
```

Above library assertion checks for user programmed variable PARAM being inside values 0 and 100.

6 – ASSERTIONS AS INPUT TO THE END-USER CONFIGURATOR UTILITY

The SVA assertion library created above can also be read by the end-user's IP configuration and assembly software to enable FPGA users to configure their device in a legal set of configurations. In effect, this assertion library acts as an executable specification originating from hardware and consumed by software.

For example, imagine a DSP design where the customer wanted to place an adder in sequence with a register, then a multiplier. Naturally the bit widths of each element should be the same. For small designs this is a relatively trivial parameter check. But for larger fabrics of logic, with multiple layers of hierarchy that often include thousands of instantiations of IPs, such trivial parameters can rapidly get misaligned. What is worse is the complexity of figuring out what went wrong when we hit a bug, is it caused due to incorrect parameter or if it is a real design bug. Now consider a more complex case where the various IPs each have a long menu of low power control logic and the potential for mismatches and error is considerable.

Having an assertion library that is consumed by both hardware verification and software eliminates a lot of miscommunication would arise if that whole configuration is communicated through any means other than an executable specification.

7 – ASSERTIONS AS INPUT TO FORMAL ANALYIS TOOLS

The fourth use of the master assertion library is to apply the assertions as targets for formal verification, where the above design configuration checker methodology inherently describes the legal configuration space for the design(s). An added benefit is that the design and setup is cleaned-up for any subsequent formal effort on the design blocks. In general, if the formal tool reports a failure, we know immediately there is either a bug in the design, or the design is under-constrained.

8 – BUGS FOUND

There were various types of bugs found with the methodology developed in this paper. The verification using this approach is still underway, but to date the bug count stands at 40. This included issues found in RTL code and SVA library issues. Here are a few examples of types of bugs discovered:

8.1 Bus contention failure due to incorrect SVA assertion: This bug was found in the following snippet of RTL code:

```
Assign out = SEL[0] ? in0 : 1'bz;

Assign out = ~SEL[1] ? in1 : 1'bz;

Assign out = SEL[2] ? in2 : 1'bz;

Assign out = SEL[3] ? in3 : 1'bz;
```

And the culprit library assertion was:

```
`LIB_ASSERT(sel,"Select values are
incorrect",(SEL==0 || SEL==1 || SEL==6 ||
SEL==10)
```

When the formal engine assumes the value of SEL=1, this results in bus contention bug. Under this scenario the formal tool reports a witness waveform where SEL[0]=1 and ~SEL[1]=1, and in0 != in1 (since in0 and in1 are free to toggle). In general, bugs were found because of bus-contention failures when either assertion check (which becomes a constraint for our flow) was missing or was incorrect.

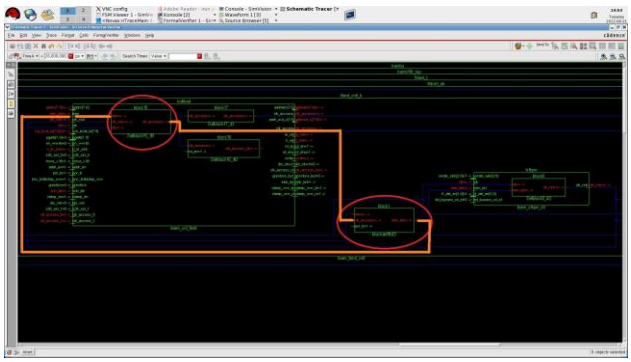
8.2 Combinational loop bug: This bug was caused due to the following code across two Verilog modules. In module1:

```
Assign varA = En & (clk_rst || clk);
```

In module2:

```
always_latch begin
       if (~rst)
              latchA <= 1'b0;
       else if (~varA)
              latchA <= En;
 end
```

This bug was found when a few of the auto-properties came out blocked. This is because formal engine could not converge on any proof on them. The reason was that this combinatorial loop was in a cone of logic of variables involved in the proof. The combinatorial loop shows up like this on the schematic trace of the tool:



9. SUMMARY

Compared to earlier simulation based techniques, utilizing formal analysis and its related methodologies has enabled us to simultaneously "kill 4 birds with one stone" in that:

- We can exhaustively verify device blocks for bus contentions and x-source check
- We have a known-good set of assertions we can embed in the design RTL
- The exact same assertions will check for illegal block configurations by the D&V team when the assembly is instantiated in a downstream testbench simulation.
- The exact same assertions will become an executable specification for the software/customers/design teams trying to configure the device.

We ran this flow on most blocks in our design. The design itself is a combination of old-verified legacy code and new code. It was a significant effort to debug 18k properties (not including 64k deadcode properties) generated using this flow. We found a variety of bugs that targeted different aspects of our design. In one of the important block, which is replicated thousands of times in our design, we found 8 bugs. This block was to be verified using subsequent formal verification effort. To our joy we did not find any issues with this block when we performed verification on it. The AFA setup was available on this block soon after it was released by the design team. The traditional formal verification setup only became available two months later. This allowed us to save significant debug time at the block level verification. Since this block is pervasive at chip-level setup, hence we were also able to save both time and effort in debugging it at the full-chip testbench simulations.

Looking forward, we are considering making this flow a standard part of our design-verification qualification process. Up until now we have been deploying this flow only at the block-level. In future we would like to run it on

multi-block setup. This would help us verify both, configuration assumptions and design constructs that this flow targets, at a bigger design scope. The challenges to accomplish that today are: setup time, tool capacity and tool issues that arise with bigger designs.

10. ACKNOWLEDGMENTS

Thanks to Joseph Hupcey III of Cadence Design Systems, Inc. for his feedback in reviewing the paper.

11. REFERENCES

[1] Incisive Enterprise Verifier (IEV) User Guide, Version 11.10.s020, Cadence Design Systems, Inc.

[2] Incisive Formal Verifier Reference Manual, Version 11.10.s020, Cadence Design Systems, Inc.