Developing an Integrated Verification and Debug Methodology

Akitoshi Matsuda
Department of Automotive Science,
Graduate School of Integrated Frontier Science,
Kyushu University
Fukuoka, Japan
matsuda_aki@slrc.kyushu-u.ac.jp

Tohru Ishihara
System LSI Research Center
Kyushu University
Fukuoka, Japan
ishihara@slrc.kyushi-u.ac.jp

***Abstract*—As design complexity of LSI systems increase, so does the verification challenges. It is very important, yet difficult to find all design errors and correct them in a timely manner. This paper presents our experience with a new verification and debug methodology based on the combination of formal verification and automated debugging. This methodology, which is applied to the development of a DDR2 memory design targeted for an FGPA, is found to significantly reduce the verification and debug tasks typically performed.**

Keywords-system LSI; verification; debug; methodology

I. INTRODUCTION

Formal verification tools use advanced mathematical analysis techniques to prove and disprove the validity of design properties. These tools can be very powerful aids in the verification effort as they can find bugs early in the design cycle without the presence of a test-bench or stimulus to activate bugs. This fact makes formal verification especially attractive to verification engineers and managers as it can significantly reduce the verification time.

Unfortunately, once bugs are found by formal verification tools, their localization and root cause's analysis remains a manual burden. What makes these tasks even more complicated than usual is that engineers may be unfamiliar with the counter example scenarios generated by the formal verification tools, since such tools often find corner case bugs that escape the designers. Thus the sequence of events captured in the counter example can at times appear unexpected. Furthermore, many counter examples found by formal verification tools can be false negatives, where it does not expose a bug but a lack of constraints on the primary inputs of the design. For example, a protocol violation on the inputs can result in a protocol violation on the outputs.

To establish an efficient verification process, the power of formal verification must be leveraged while reducing the debugging effort. We propose a methodology that does just that. This methodology makes use of a new class of automated debugging tools that help engineers quickly identify the root cause of errors and remove the bugs based on the counter example. The debugging solution reduces the complexity associated to understanding a bug and therefore reduces turn-around-time (TAT) required by aggressive FPGA based design.

Together the efficient use of formal verification and automated debugging tools, result in a reliable and efficient verification process. In this paper, we use a case study in the development of a DDR2 memory design from specification to FPGA implementation to demonstrate the effectiveness of the debugging methodology without affecting the high performance and high quality of the final FPGA design.

II. BACKGROUND

Today, debug tools to aid engineers are primarily waveform viewers, visualization and schematic tools, source code navigation aids, and simulators with built-in debug features. All these require manual intervention and provide no automation, making debug a manually driven and painstakingly difficult task. Debugging tasks include tracing back from the point of failure in the source code through countless signals and drivers while referencing the waveform at every step. It is the repetitive nature of such small tasks that makes debugging a long and time consuming process.

Formal verification tools are very effective at quickly finding complex bugs that often escape the engineers. Even when simulation has been applied, the design may contain errors that have not been discovered. Missing such errors can result in serious bugs appearing in the fabricated or implemented design, which are much more expensive to correct. Finding critical bugs with formal verification and quickly locating and correcting them is very important. The debugging methodology focuses on improving the overall verification quality as well as finding the most critical bugs in the design [1].

III. DEBUGGING SOLUTION FLOW

The debugging methodology requires the tight integration of the formal verification and the automated debugging tools. The formal verification tool is provided with the HDL design, all assertions and assumptions, and a control file. The control file specifies to generate a counter example for all the failing properties. A reset sequence is also provided to ensure that all properties are checked after initialization.

A simple script picks the failure results of the formal verification tool and calls the automated debugging tool for each counter example. For each failure, a report is generated

978-3-9810801-7-9/DATE11/ © 2011 EDAA

with a list of suspect locations. Each suspect location is a line in the HDL source file where the bug may reside. The entire process is automated: when the engineer is ready to debug, he reads the debugging report that points to the file and line number where the fix can be made. The locations are also ranked to focus the engineer quickly on the most effective location to make a correction. It is found that this verification and debug flow is more efficient than the traditional process. Fig. 1 shows the integrated debugging methodology.

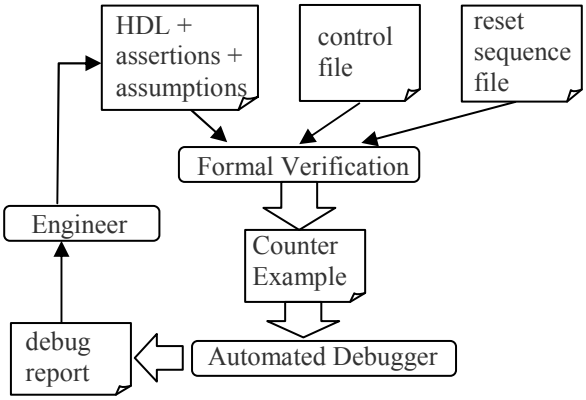

Figure 1. Verification and Debugging Methodology.

Generally, failures can be due to either some false negatives, a bug in the design or a bug in the assertion. In the case of a false negative, the engineer will have to add assumptions to the design based on the signals listed in the debug report. For bugs in the design or assertion, the debug report can lead the engineer to make quick fixes. After all such changes are made, the verification tool must be re-run to ensure correctness of the fixes. If the fixes are not adequate, the verification and debug cycle continues.

IV. EXPERIMENTS

The proposed methodology is carried out on a DDR2 memory design. We used the formal verification tool Solidify [2] and the automated debugging tool by Vennsa Technologies OnPoint [3]. The overall methodology found a number of assertions failing, and the automated debugger found the source of errors directly. For example, the SVA shown in Fig. 2 was one of the failing assertions. One of the solutions found by the debugging tool directly points to the bug as shown in Fig.3. The suspect is related to the transfer of signal "cas_n" to "ras_n". Once the engineer looked at this location in Fig. 4, the fix was easily applied. We removed a pipeline stage and replaced an "always block" with buffers in the error source file.

Comparisons between the traditional simulation-based verification and manual debugging flow are done here. Fig. 5 shows the amount of time spend on running verification upfront to find bugs (simulation or formal), time to analyze the failures and debug, as well as time for the remaining simulation runs. To measure the effectiveness of the proposed flow, two different engineers with similar levels of experience and design knowledge were asked to verify and debug the block based on a small set of specifications. While the traditional technique took 36 hours, the proposed verification and debugging methodology took 21 hours, which corresponds to a reduction

```
assert property (@(posedge clk0)  disable iff(rst)
                    (init_state_r = = INIT_FIRST_READ)
                    |-> ##  [1:50]
                    first_calib_done);
```

Figure 2. Example of failing SVA.

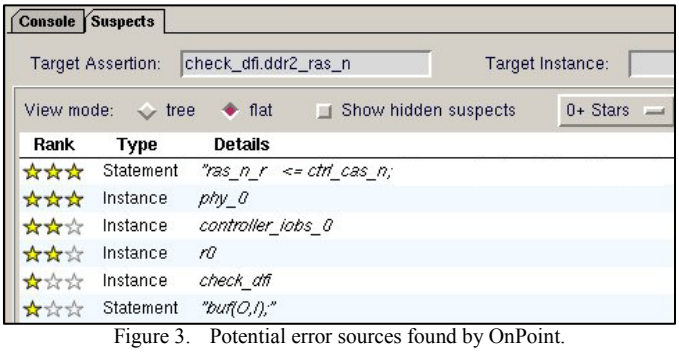

Figure 3. Potential error sources found by OnPoint.

```
always @(posedge clk)  begin
        DDR_RAS_L <= ctrl_ras_n;
        DDR_CAS_L <= ctrl_cas_n;
        DDR_WE_L <= ctrl_we_n; end
=============================================
        OBUF r0( .I (ctrl_ras_n),  .O  (DDR_RAS_L)   );
        OBUF r1( .I (ctrl_cas_n),  .O  (DDR_CAS_L)   );
        OBUF r2( .I (ctrl_we_n),  .O  (DDR_WE_L)  );
```

Figure 4. The fix for the example: replacing the pipeline stage with buffers.

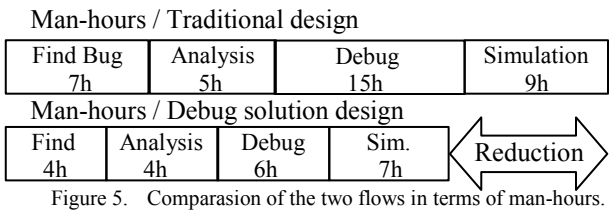

Figure 5. Comparasion of the two flows in terms of man-hours.

of 42%. The reason for the overall improvement is that formal verification can find some bugs quickly and debugging them (including false negatives) is very fast with the help of automated debugger. Similarly false negatives can be quickly ruled out. Without this debug methodology, finding the root of the corner case bugs takes many man-hours.

V. SUMMARY AND CONCLUSIONS

This case study shows that the integrated automated verification and debugging methodology can be very effective. The debugging solution can overcome some of the major challenges presented by formal verification while leveraging its strengths.

ACKNOWLEDGMENT

My appreciation goes to Dr. Safarpour whose comments were valuable throughout the course of my study.

REFERENCES

[1] J. Jan, A. Narayan, M. Fujita, and A. S. Vincentelli, "A survey of techniques for formal verification of combinational circuits," in *Int'l Conf. on Comp. Design*, Oct. 1997, pp. 445-454.

[2] Formal Verification tool "Solidify" of Averant Inc.

[3] Vennsa Technologies Inc., OnPoint, www.vennsa.com/product.html