Debugging Strategies for Mere Mortals

Valeria Bertacco
Department of Computer Science and Engineering, University of Michigan
valeria@umich.edu

ABSTRACT

Recent improvements in design verification strive to automate error detection and greatly enhance engineers' ability to detect functional errors. However, the process of diagnosing the cause of these errors, and subsequently fixing them, remains one of the most difficult tasks of verification. The complexity of design descriptions, paired with the scarcity of software tools supporting this task lead to an activity that is mostly ad-hoc, labor intensive and accessible only to a few debugging specialists within a design house.

This paper discusses some recent research solutions that support the debugging effort by simplifying and automating bug diagnosis. These novel techniques demonstrate that, through the support of structured methodologies, debugging can become a task pursued by the average design engineer. We also outline some of the upcoming trends in design verification, postponing some the verification effort to runtime, and discuss how debugging could leverage these trends to achieve better quality of results.

Keywords

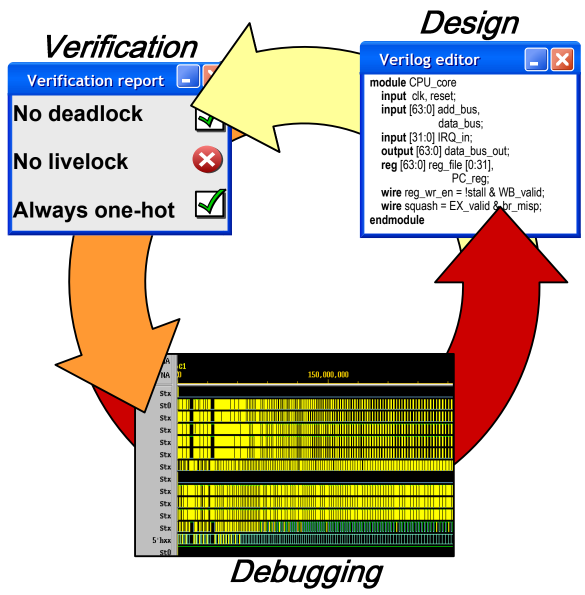
Design verification, validation, error diagnosis, error correction.

1. INTRODUCTION

Digital integrated circuit design has reached unparalleled levels of complexity. In this context, verification has become a pivotal aspect of electronic design automation. In fact, various estimates indicate that functional errors are still responsible for 40% of failures at first tape-out, and that verification accounts for two thirds of the design cycle and effort [4, 18].

Resolving design bugs in the early development stages is, at the same time, a sophisticated and time-consuming activity, as well as a crucial task for the project development and for the success of a design team. In the past few decades, much research has been dedicated to improving the quality and the effectiveness of verification, however, much less effort has been devoted to supporting a design team in resolving a functional bug, that is, finding the root cause of the bug and devising a modification to the design that corrects it. A few commercial software applications are available that provide minimal debugging support, for the most part in the form of visualization tools that can connect a signal transition observed in simulation to a specific location in the source Register-Transfer Level (RTL) description [22]. While these aids are valuable when investigating a bug, they are far from solving the problem, particularly when the problem manifests itself through a bug trace, several millions cycles long, producing an erroneous outcome at the end. As a result, bug diagnosis and correction is an extremely time-consuming challenge, with some bugs imposing delays of several days, or even weeks, to the development schedule. Occasionally, the correction of a bug may affect so many components of a circuit, that the design team may choose not to pursue it. This is particularly prone to occur in the late development stages of a system, or if the effects of the bug under evaluation may be countered through other means (such as microcode or compiler patches). Figure 1 shows a schematic of the design flow, highlighting how *debugging, that is, bug diagnosis and correction*, is an integral part of the design/verification loop, often disregarded in high-level flow diagrams and when planning a development schedule, but almost always the most time-consuming component.

The debugging methodologies available today rely, for the most part, on the skill and creativity of individual designers, making it more of an art that only a few gifted people can pursue, than a science that can be taught to the average engineer. While this is a high risk proposition as it stands, the growing complexity of design, verification, and even of simulation traces in validation are quickly



```
module CPU_core
   input  clk, reset;
   input [63:0] add_bus,
                 data_bus;
   input [31:0] IRQ_in;
   output [63:0] data_bus_out;
   reg [63:0] reg_file [0:31],
                  PC_reg;
   wire reg_wr_en = !stall & WB_valid;
   wire squash = EX_valid & br_misp;
endmodule
```

Figure 1: The design, verification and debugging cycle. During integrated circuit development, the system is designed, usually by means of a hardware description language, then verified by a combination of simulation-based and formal techniques. Each time a new bug is exposed in verification, it must be diagnosed and a design fix must be developed for it. This constitutes debugging. Once the fix is deployed, the new version of the design must undergo verification again to detect additional bugs, and/or bugs introduced by the fix. Overall debugging constitutes a major component, in terms of time and effort, of a digital system development cycle.

making this approach crumble. What is critically needed to overcome this situation are tools and techniques to support engineering in debugging, so that the complexity of the task can be reduced and constructive methodologies, accessible by people of ordinary skill, can be developed.

This paper outlines some of the work that is ongoing in our research group at the University of Michigan to address the problem of resolving functional bugs exposed through verification. Specifically, I will present two key projects that support and automate both diagnosis and correction of functional bugs. Both projects share a very low barrier to entrance, that is, they complement the current debugging methodology, without requiring any change to it. Since most verification today still occurs by means of logic simulation, the first solution, called *Butramin* tackles the complexity of debugging based on simulation traces. Such traces are often tens or hundreds millions of cycles long and may take hours or days just to be replayed so to re-create the bug condition. Butramin leverages a number of simulation-based analyses to reduce their size and length (in simulation cycles) by three to six orders of magnitude.

The second solution, called *REDIR*, is an automatic diagnosis technique that considers a set of bug traces and the corresponding correct system responses. It then leverages the RTL description of the design to isolate the root cause of a bug. The diagnosis is presented to the user as a set of signals (wire, registers, etc.) in the RTL that are responsible for the incorrect computation. Note that, "correctness" for REDIR is simply defined with respect to the traces and responses provided by the user. The correct behavior of the system in response to other stimuli is unknown, since no golden model is used, and the RTL contains functional bugs.

Finally, the industry today is becoming aware that design and verification complexities are such that digital systems are bound to be released with latent functional bugs. Thus, researchers are starting to develop correction solutions to be deployed in silicon and that operate at system runtime, being activated only if and when a bug is manifested. While this trend by no means diminishes the importance of debugging, it does allow for trade offs. For instance, if a bug entails such a widespread set of modifications to endanger design stability, it may be wiser to rely on runtime correction. Or, when detailed diagnosis becomes extremely time consuming, a better option may be to simply derive a system-level condition that may trigger the bug, and use that in runtime verification.

2. FOCUSING TRACES ON A BUG

Among the techniques and methodologies available for functional verification, simulation-based verification is prevalent in the industry because of its linear and predictable complexity. A common methodology in this context is *random simulation*, where stimuli are provided by a constraint-based random generator. Such generators can automatically produce random legal input for the design at a very high rate, based on a set of rules (or constraints) derived from the specification document. In random simulation bugs are detected by means of assertion statements, or checkers, embedded in the design. When a bug is detected, the simulation trace leading to it is stored aside and can be replayed at later times to analyze the conditions that led to the failure. Because of the randomized nature of this methodology, and because it is usually applied in late design stages (when simple bugs have already been flushed out), it is very common for the bug traces generated to be lengthy and complex. Another family of techniques, attracting increasing attention from industry, is that of semi-formal verification. These tools combine a mix of formal and simulation-based techniques with the goal of producing high-coverage verification results on complex designs [1, 14, 12]. While semi-formal tools are a promising direction in terms of high-quality verification for industrial size designs, little concern has been given to the reduction in complexity for the bug traces generated. As a result, once a bug is found, a copious amount of effort goes into tracking it back to its root cause: either an incorrect design implementation or an erroneous property definition.

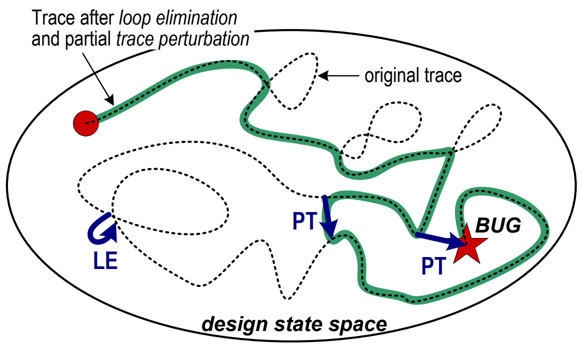

Trace after *loop elimination* and partial *trace perturbation*
original trace
PT
BUG
LE
PT
design state space

Figure 2: Two trace minimization techniques used by Butramin. The schematic shows a bug trace within the design state space (dashed line), starting from an initial state (circle) and ending at a bug state (star). Butramin attempts to remove "loops" within the trace using its *loop elimination* (LE) technique, it then explores possible *trace perturbations* (PT) by eliminating cycles and input events. These perturbations may lead to a shortcut in the trace. The solid line in the picture represents a trace obtained after cycle elimination and after discovering one of the PT shortcuts indicated.

The solution we set forth to address diagnosis of complex bug traces is called Butramin ("BUg TRAce MINimization") [7]. The objective of Butramin is to consider a bug trace and the corresponding checker (or property) that triggers the bug, and seek a much shorter and simpler trace to falsify the same property. Previous work in this area has been mostly centered on using formal techniques to simplify a property's counterexample [20, 9]. In a separate context, the problem of trace minimization has also been addressed in software verification [11, 13]. We instead focus most of the effort on simulation-based techniques so that we can apply our solution to complex modules and designs, such as those developed in the industry. At the same time, we do not strive to obtain a minimum-length bug trace, but simply one of manageable size for debugging purposes. However, in our experimental evaluation, we find that in practice our minimized traces are most often extremely compact, of similar size than those obtained through formal minimization techniques.

Butramin simplifies a trace by iteratively eliminating redundant portions of the trace. For instance, it checks if there are redundant sequential steps, or sequential loops that can be removed. It also checks for possibly redundant combinational input events. In addition, it attempts to "perturb" a trace by eliminating one full simulation cycle, and/or one input event. The perturbed trace obtained is re-simulated to check if it still exposes the original bug (the original user-provided assertion is used for this purpose). If the test is successful the new trace replaces the original one, otherwise it is discarded. Often, perturbation leads to fairly different traces that may expose the bug much earlier than the original ones. When these mechanisms are exhausted, Butramin further simplifies a trace by using X-value simulation to determine which input signals are essential in exposing a bug. In the final stage, a SATisfiability (SAT)-based, fixed-window bounded model checker seeks additional "shortcuts" in the trace obtained so far, typically already much smaller than the original one.

As an example of some of the techniques Butramin uses, Figure 2 shows schematically how *loop elimination* (LE) and *trace perturbation* (PT) work. Loop elimination removes sequential loops in a trace by hashing the states encountered during re-simulation and detecting when a trace enters the same state twice. Trace perturbation uses a trial-and-error approach, whereby a trace is modified by either removing one of more simulation cycles or eliminating input events. Then Butramin checks if the new trace still reaches a bug state through re-simulation.

We found experimentally that Butramin can reduce the size of a bug trace by three to six orders of magnitude in terms of cycles in the trace and, consequently, size of the trace. We note that traces generated by constrained-random simulation are more susceptive to benefit from Butramin, and also that traces derived from more complex designs (which usually entail more events and longer traces) present more opportunities for reduction. The impact of Butramin appears to be uncorrelated with the frequency of occurrence of the bug configuration targeted by the trace, that is, the number of distinct design states that expose the bug. In many cases we could reduce traces of several million cycles down to a few tens or a few hundreds cycles, that is, a trace that is much simpler to analyze and to re-simulate. In terms of execution time, we did not focus on optimizing the performance of this solution, but gave top consideration to the quality of the results, since the engineering time saved by the latter well outweighs the execution time of the software. We envision a deployment scenario where Butramin is run overnight to prepare simplified traces to be analyzed, and we found that all of our execution times are well within this limit: most commonly just one, or a few, hours. Butramin can be deployed in practically any simulation-based and semi-formal verification methodology with no effort: it is simply applied to bug traces generated in verification to greatly reduce them before they are analyzed for diagnosis.

3. AUTOMATIC DIAGNOSIS

One of the most difficult aspects of debugging is diagnosis, that is locating the error source within a design. *REDIR* (RTL Error DIagnosis and Repair) [8] is a scalable and powerful RTL error diagnosis and correction system, which adopts some of the hardware analysis techniques prevalent at the gate-level into the more designer-friendly and succinct RTL descriptions. The approach is significantly more accurate than previous software-based solutions in that it can analyze designs rigorously using formal verification techniques. At the same time, it is also considerably faster and more scalable than gate-level diagnosis because it models errors at a higher abstraction level (RTL), and thus there is a smaller number of candidate error sites to be evaluated.

The inputs of REDIR include a design containing one or more bugs, a set of simulation vectors exposing them, and the correct responses for the primary outputs over the given test vectors (usually generated by a high-level behavioral model). Note that REDIR only requires the correct responses at the primary outputs of the high-level model, not at any internal node. The correct output responses could be the primary outputs of the design, or the outputs of a set of checkers in the context of assertion-based verification. REDIR can then output a minimum cardinality set of RTL signals that should be corrected in order to eliminate the erroneous behavior. We call this set the *symptom core*. When multiple cores exist, REDIR provides all of the possible minimal cardinality sets. In addition, the framework suggests several possible ways of modifying the signals in the symptom core to help in the correction of the design. Our empirical evaluation shows that REDIR can diagnose and correct multiple errors in design descriptions with thousands of lines of Verilog code (corresponding approximately to 100K cells after synthesis), a typical block size developed by individual engineers. As a result, REDIR can assist in everyday debugging tasks and fundamentally accelerate the design development.

The objective of error diagnosis is to identify a minimal number of variables in the RTL description that are responsible for the design's erroneous behavior. Moreover, errors can be corrected by modifying the statements related to those variables. Each signal affecting the design's correctness is called a *symptom variable*. Correcting all the symptom variables that contribute to a bug would eliminate it. A key idea in REDIR is error modeling: we embed additional constructs in the RTL design to evaluate a number of possible variants of the design and determine which of these variants would produce the expected output responses for each input test vector. Gate-level solutions for automatic diagnosis used a similar concept applied at the gate-level.

```
module half_adder(a, b, s, c);
  input a, b; output s, c;
  assign s = a ^ b;
  assign c = a | b;
endmodule
module half_adder_MUX_enriched(a, b, s_n, c_n,
s_sel, c_sel, s_f, c_f);
  input a, b, s_sel, c_sel, s_f, c_f;
  output s_n, c_n;
  assign s = a ^ b;
  assign c = a | b;
  assign s_n = s_sel ? s_f : s;
  assign c_n = c_sel ? c_f : c;
endmodule
```

Figure 3: An RTL error-modeling example. Module half_adder shows the original code, where c is erroneously driven by "$a \mid b$" instead of "a & b". Module half_adder_MUX_enriched shows the corresponding MUX-enriched version. Differences are marked in boldface. (*Figure reproduced from [8]*)

To model errors in the design, we introduce conditional assignments for each RTL variable, as shown in the example in Figure 3. Note that we insert only one conditional assignment even if the variable contains multiple bits. These assignments allow the REDIR framework to locate sites of erroneous behavior in RTL. Suppose that the output responses of the design are incorrect because c should be driven by "a & b" instead of "$a \mid b$". Obviously, to produce the correct outputs, the behavior of c must be modified. To model this situation, we insert a conditional assignment, "assign $c_n = c_{sel}\ ?\ c_f : c$", into the code. Next, we replace all uses of c in the code with c_n (but not the assignments to c). Variable c_{sel} allows simulation of the design using c_f instead of c; moreover c_f is what we call a *free variable*, that is, we can assign it as deemed necessary to achieve the correct output response. If we can identify the $_{sel}$ variables that should be asserted, and the correct signals that should drive the corresponding free variables to produce the desired circuit behavior, we can diagnose and fix the errors.

Once errors are modeled as described, we rely on a Pseudo-Boolean solver or RTL symbolic simulation to perform the diagnosis and infer which design signals are responsible for incorrect output behavior. By forcing the Pseudo-Boolean solver to find a set of assignments that satisfy all the given <test vector, output response> pairs while minimizing the number of $_{sel}$ variables assigned to true, we obtain the complete set of signals that concur to the bug and that must be thus corrected.

The diagnosis capability of REDIR has been evaluated on a number of microprocessor modules and designs. The bugs injected in the design ranged from using an incorrect operator or complemented operand or simply wrong operand, to incorrect data forwarding and to incorrect state transition or execution of an instruction. For one simple microprocessor design, we had available a number of buggy versions [5] with bugs that were present in the design since its development and had been fixed in validation. Thus, for this testbench, no artificial bug injection occurred. The experimental evaluation indicates that REDIR could isolate the bug sources in each case, and that, because it addresses the problem at a higher abstraction level (that is, RTL) it can cope with much more complex designs than gate-level diagnosis solutions. The computation time of this diagnosis solution is also very practical, always less than one hour, even for systems sufficiently complex that a gate-level analysis would take more than two days, possibly running out of memory. One of the limitations of REDIR, however, is that in performing diagnosis it may detect several distinct sets of signals that can independently be modified to correct the bug. Thus, a user would have to manually determine which of these sets entails the minimum amount of source code modification.

Most techniques that have been proposed so far in this space target the gate-level description of a design [6, 17, 24, 25, 27] or even the transistor-level [16]. However, most debugging effort occurs in the early development phases of a design, when the system is described by an RTL model. The lack of powerful and automatic debugging tools at this level greatly reduces designers' productivity when fixing even very simple errors. Recently, a few techniques that work directly at the RTL have been developed. Some of them [15, 19, 21] employ a software analysis approach that implicitly makes use of multiplexers to identify statements in the RTL code potentially responsible for the errors. These techniques can suffer from a large number of false positives, and return too many candidate error sites. To address this problem, recent work by Staber *et al.* [23] inserts multiplexers explicitly into the RTL code. This enables the use of hardware analysis techniques and greatly improves the accuracy of diagnosis. Other techniques, such as [10], analyze an RTL description and failed properties using state-transition diagrams and model checking. REDIR is similar to several successful gate-level methods [2, 3, 6, 24, 27] in that it only requires test vectors and output responses to diagnose a functional error.

4. CONCLUSIONS

Looking forward, bug diagnosis can leverage and benefit from current trends in verification. Until recently, the goal for verification was to achieve complete functional correctness before tape-out. However, today, due to the unattainable complexity of verification, this task is becoming more selective. In recent trends in academia and, in lesser measure, in industry, engineering teams strive to verify the most common system's behaviors, and then they complement this effort with runtime detection and correction techniques for functional correctness. These techniques provide the advantage of shortening the digital system development cycle, but also come at a price in chip area and performance [26]. Thus, verification is striking a new trade-off between development effort/time and runtime performance, and development teams can choose to halt verification, once they can guarantee than any residual bug would occur with less than a specified frequency on average.

Bug diagnosis and correction can also benefit from this trend by choosing to only fix bugs for which a correction is known that does not run the risk to jeopardize the stability of a design close to tape-out. Or, by budgeting the time that can be spent in finding a bug, or a class of bugs, based on their criticality, frequency of occurrence, *etc.* All the techniques discussed above and the trade-offs enabled by novel runtime solutions contribute to taming the complexity of bug diagnosis and make it a task that can be approached by mere engineering mortals.

5. REFERENCES

[1] M. Aagaard, R. Jones, and C.-J. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment", in *Proc. DAC*, 1998, pp. 538–541.

[2] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir and R. Drechsler, "Post-verification debugging of hierarchical designs", *Proc. ICCAD*, 2005, pp. 871–876.

[3] M. F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith and M. Abadir, "Debugging sequential circuits using Boolean satisfiability", *Proc. ICCAD*, 2004, pp. 44–49.

[4] J. Bergeron, *Writing testbenches: functional verification of HDL models*, Kluwer Academic Publishers, 2nd edition, 2003.

[5] "Bug UnderGround", `http://bug.eecs.umich.edu/`

[6] K.-H. Chang, I. L. Markov and V. Bertacco, "Fixing design errors with counterexamples and resynthesis", *Proc. ASPDAC*, 2007, pp. 944–949.

[7] K.-H. Chang, V. Bertacco and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement", *IEEE Transactions on CAD*, Jan. 2007, pp. 152–165.

[8] K.-H. Chang, I. Wagner, V. Bertacco and I. L. Markov, "Automatic error diagnosis and correction for RTL designs", *Proc. HLDVT*, 2007, pp. 65–72.

[9] Y. A. Chen and F. S. Chen, "Algorithms for compacting error traces", in *Proc. ASPDAC*, 2003, pp. 99–103.

[10] G. Fey, S. Staber, R. Bloem and R. Drechsler, "Automatic fault localization for property checking", *IEEE Transactions on CAD*, Jun. 2008, pp. 1138–1149.

[11] P. Gastin, P. Moro and M. Zeitoun, "Minimization of counterexamples in SPIN", in *Proc. SPIN*, 2004, pp. 92–108.

[12] S. Hazelhurst, O. Weissberg, G. Kamhi and L. Fix, "A hybrid verification approach: getting deep into the design", in *Proc. DAC*, 2002, pp. 111–116.

[13] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proc. ISSTA*, 2000, pp. 134–145.

[14] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Proc. ICCAD*, 2000, pp. 120–126.

[15] T.-Y. Jiang, C.-N. J. Liu and J.-Y. Jou, "Estimating likelihood of correctness for error candidates to assist debugging faulty HDL designs," *Proc. ISCAS*, 2005, pp. 5682–5685.

[16] A. Kuehlmann, D. I. Cheng, A. Srinivasan and D. P. Lapotin, "Error diagnosis for transistor-level verification", *Proc. DAC*, 1994, pp. 218–224.

[17] J. C. Madre, O. Coudert and J. P. Billon, "Automating the diagnosis and the rectification of design errors with PRIAM", in *Proc. ICCAD*, 1989, pp. 30–33.

[18] P. Rashinkar, P. Paterson and L. Singh, *System-on-a-chip verification: methodology and techniques*, Kluwer Academic Publishers, 2002.

[19] J.-C. Rau, Y.-Y. Chang and C.-H. Lin, "An efficient mechanism for debugging RTL description", *Proc. IWSOC*, 2003, pp. 370–373.

[20] K. Ravi and F. Somenzi, "Minimal satisfying assignments for bounded model checking," *Proc. TACAS*, 2004, pp. 31–45.

[21] C.-H. Shi and J.-Y. Jou, "An efficient approach for error diagnosis in HDL design", in *Proc. ISCAS*, 2003, pp. 732–735.

[22] SpringSoft, `http://www.springsoft.com/`

[23] S. Staber, G. Fey, R. Bloem and R. Drechsler, "Automatic fault localization for property checking", *Springer-Verlag LNCS 4383*, 2007, pp. 50–64.

[24] A. Smith, A. Veneris and A. Viglas, "Design diagnosis using Boolean satisfiability", *Proc. ASPDAC*, 2004, pp. 218–223.

[25] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation", *IEEE Transactions on CAD*, Dec. 1999, pp. 1803–1816.

[26] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians", *Proc. DATE*, 2007, pp. 743–748.

[27] Y.-S. Yang, S. Sinha, A. Veneris and R. Brayton, "Automating logic rectification by approximate SPFDs", *Proc. ASPDAC*, 2007, pp. 402–407.