The Day Sherlock Holmes Decided to do EDA

Andreas Veneris
University of Toronto, ECE Dept. & CS Dept.
Toronto, ON, Canada
veneris@eecg.toronto.edu

Sean Safarpour
Vennsa Technologies, Inc.
Toronto, ON, Canada
sean@vennsa.com

ABSTRACT

Semiconductor design companies are in a continuous search for design tools that address the ever increasing chip design complexity coupled with strict time-to-market schedules and budgetary constraints. A fundamental aspect of the design process that remains primitive is that of debugging. It takes months to close, it introduces costs and it may jeopardize the release date of the chip. This paper reviews the debugging problem and the research behind it over the past 20 years. The case for automated RTL debug tools and methodologies is also made to help ease the manual burden and complement current industrial verification practices.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Simulation, Verification*

General Terms

Design, Verification

Keywords

Debugging, Error Localization, Verification

1. INTRODUCTION

In the past decade, there has been an exponential increase in the cost and time required for verification and debugging of VLSI systems. Verification checks the correctness of a design and if faulty, debugging identifies the root-cause of the problem. Although debugging manifests itself in every step of the design cycle, in this study, we are interest in functional Register Transfer Level (RTL) debugging.

It is a well-accepted fact that debugging and verification take up to 70% of the chip design time. With debugging contributing to as much as half of this time, it directly results in millions of dollars in non-recurring costs and may jeopardize the release date of the end product. To make things worst, silicon prototypes today are rarely bug-free. Functional bugs may escape pre-silicon verification only to be discovered during in-system silicon validation. It comes as no surprise that more than 60% of design tape-outs require at least one re-spin and more than half of the failures are not due to power, timing or manufacturing defects but due to logical or functional errors not discovered or properly fixed during verification [15].

Without a doubt, verification and debug are major bottlenecks. This burden is expected to increase 675% by 2015 as reported by `EETimes` [10]. There are a number of reasons to justify this trend. Firstly, the modern semiconductor design flows and verification methodologies are complex in nature. The processes of design and verification are comprised of heterogeneous components implemented at multiple levels of abstractions (procedural, behavioral, synthesizable, etc.) using different languages (Verilog, System Verilog, VHDL, PSL, etc) and ever changing standards and protocols. Interacting with such components adds layers of complexity and overhead while hindering transparency and efficiency. The lack of a unified and centralized verification environment makes the debugging pain a growing challenge for the end engineer.

Further, design specifications often described in abstract models, may not directly correspond to signals and transactions at the design level. For example, the specification can be described in a plain document, a Matlab model or in a software language such as C/C++, whereas the design is implemented in cycle-accurate Verilog or in VHDL. The separation between these two layers can result in misinterpretations and usually complicates verification/debugging efforts. Additionally, the ever increasing size of modern devices poses challenges both to Electronic Design Automation (EDA) tools and engineers alike. Typical design blocks grow beyond the 400,000 synthesized gate mark and error-traces extend past thousands of cycles. Task outsourcing and geographical dispersed teams only add extra layers of communication overhead to these processes.

In 2006, the International Technology Roadmap for Semiconductors (ITRS), issued its new set of needs for the current and next generation design semiconductor processes. Although most topics saw minor numeric revisions, the roadmap contains a major fourteen-page update in design verification with a strong emphasis in debugging. The report [11] states that *"technological progress depends on the development of rigorous and efficient methods to achieve high-quality verification results ... and techniques to ease the burden of debugging a design once a bug is found ... without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further semiconductor progress"*. Without a doubt, the roadmap depicts a grim yet realistic picture that establishes an urgent need for scalable automated debugging tools and methodologies.

To a certain extent, the tremendous growth of the semiconductor industry over the past decades can be partially attributed to the amount of automation provided by the EDA community. Most manual steps of the design flow (synthesis, placement, routing, test, verification, etc.) have been automated to help close designs faster and cheaper. Unlike these processes, debugging remains a time-consuming and resource intensive manual task where graphical navigators and waveform viewers allow engineers to perform simple "what-if" analysis. With no form of exaggeration, the engineer today resembles the famous detective Sherlock Holmes who searches for needles in a haystack and relies on a "hunch" or "gut feel" to localize the culprit bug when verification fails.

In past years, VLSI design companies have in part alleviated this debugging pain by allocating more verification engineers to the problem. As a net effect, it has been reported, there are two to three times more verification engineers than designers in design teams [3]. It is clear that adding verification engineers cannot provide a sustainable solution as the pain continues to clime. Automated RTL debugging techniques to localize the source of an error have become an urgent necessity if we

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA
Copyright 2009 ACM ACM 978-1-60558-497-3 -6/08/0006 ...$10.00.

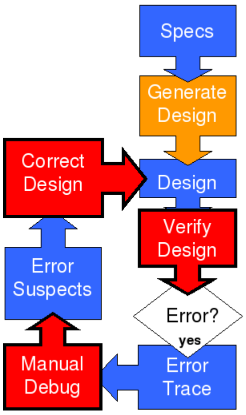


Figure 1: Typical verification and debugging design flow

desire to circumvent the manual problem and drastically improve the verification flow.

This paper builds the case for robust automated RTL debugging tools and methodologies to aid the engineer. We first define the problem and outline its multiple facets in the design cycle. Then we give a historical review of the research in debugging from the early days to state-of-the-art advances and we present industrial case-studies. The paper concludes by re-iterating the need for further research to provide the platform for cost-effective automated debugging tools and scalable methodologies to complement current industrial practices.

2. WHAT IS DEBUGGING?

Every time a design or a silicon prototype fails to adhere to a set of specifications, a debugging problem usually follows. As such, debugging manifests itself in virtually every step of the design cycle. When the design does not meet its power requirements, the engineer has to debug the problem and fix it by optimizing certain portions. When a place and route tool cannot meet timing closure, the designer does it manually by exploiting flexibilities not seen by the tool. When a silicon prototype fails test, silicon debug identifies the error root-cause to fix it so that the re-spun prototype passes test.

In this study we are interested in the problem of RTL debug for functional failures. Once verification fails, it returns with an *error-trace* or a *counter-example* that exhibits the erroneous behavior at some observation points. The input to debugging is the actual design, the set of error-traces V and the correct responses to those error-traces as shown in Fig. 1. A debugging tool localizes the error source or *suspects* with references to the RTL files, gate-level netlists or design schematics.

Note that a debugging tool utilizes a golden or reference model that provides the expected logic values for the erroneous design. In our context, there is a fundamental assumption that this model acts as a "black box", *i.e.,* there is no structural correspondence between internal lines of the model with this of the design. For example, the golden model can be a Matlab program while the design is in Verilog. This complicates the debugging effort dramatically because the solution space explodes exponentially to the number of errors in the design [21]:

$$\textit{solution space} = (\textit{circuit lines})^{\#\ \textit{errors}} \qquad (1)$$

Another implication is that the only observation points to to the debugger are the primary output design signals or the embedded assertions/properties. These may be cycle- or no-cycle- accurate values captured by interface monitors, checkers or assertions. In other words, we are interested in debugging that follows simulation-based verification, formal verification and emulation flows [12]. We do not include combinational equivalence checking in this category since it utilizes structural equivalences to solve the problem [8]. To that end, the problem of functional RTL debug resembles this of fault diagnosis (or silicon debug) [20].

Once verification identifies that a design contains an error(s), debugging usually involves the following questions:

- Is there a bug in the design or is the bug in the testbench?
- Which block in the design and which RTL line(s) should we focus on?
- What is the root-cause of the bug?
- Who should fix the bug and how should it be fixed?

The process that answers these questions today involves an arduous manual task with many iterations to close it. It delays the subsequent steps of the design cycle and introduces significant non-recurring costs.

3. DEBUGGING: THE EARLY DAYS

There is a consensus that the term "design error" is attributed to the paper by Abadir et al. from 1988 [1]. That paper outlines a set of typical errors found in the design flow also known as *design error models*. Essentially, this is a dictionary of possible simple error types such as gate replacement errors, missing or misplaced input gate line errors, etc. In the same work, the authors prove theorems for the test set V to guarantee 100% verification coverage using previous results for stuck-at faults.

Following that work, in the 1990s, a great deal of automated algorithms were developed to tackle the problem using the design error model in [1]. A comprehensive review of those methods is found in [14]. Depending on the underline engine used to drive the algorithm, those techniques can be classified as *symbolic-based* and *simulation-based*.

3.1 Symbolic-based Debugging

Symbolic approaches typically perform diagnosis by generating and solving an *error equation* using Binary Decision Diagrams (BDDs) based on the functionality of both the correct and erroneous circuits. In [9], algorithms for single and multiple design error diagnosis and correction are presented. For single errors, an error equation is generated in turn for each line l in the netlist. For a netlist with inputs X, the error equation for line l is noted $E^l(X,z(X))$, where $z(X)$ is an unspecified function over the circuit inputs. By construction, if there exists some function $z(X)$ that satisfies the equation $E^l = 0$, then replacing the line l with a circuit implementing $z(X)$ will "correct" the behavior of the netlist according to its specification. Moreover, $z^*(X) \in [E^l(X,0), \overline{E^l(X,1)}]$ specifies the family of all solutions $z^*(X)$ that correct the circuit at l.

Essentially, E^l is an algebraic representation of a miter for the two circuits. Although effective for single errors, it exhibits memory problems as it uses BDDs [5] to build the error equation. Furthermore, its applicability to multiple errors is limited to cases depending on their structural proximity [9].

3.2 Simulation-based Debugging

To overcome the excessive memory requirements of BDD-based approaches, debugging with simulation has been extensively investigated. Those methods provide a time/space trade-off as they remain polynomial in the input size but they may require more time to give an answer.

Simulation-based techniques [14] [21] [22] simulate an error-trace and trace backwards from primary output to primary input marking suspect lines using different criteria. For each error-trace, they collect the suspect lines and since the error is present in each one of them, they intersect the results for all runs. Although their memory requirements remain linear to the size of the circuit, the complexity mitigates to the time domain. As the number of errors increases, their performance degrades. For this reason, their applicability to sequential circuit debugging has been rather limited [14].

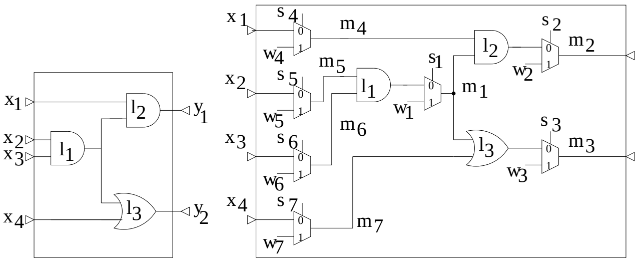
To overcome these obstacles, the concept of simulation-based debugging has been enriched with simulation of unknown values [4] to alleviate the need for an error model. Although practical in some cases, unknowns can decrease the resolution of the solution for large designs or for sequential designs. In the work of Liu et al. [16], an incremental debugging method is proposed. That method outperforms conventional techniques for multiple errors but since it is not exhaustive in the solution space, it may miss finding solutions.

4. DEBUGGING WITH SATISFIABILITY

Recently, the introduction of Boolean Satisfiability (SAT) in the field opened new opportunities for cost effective automated debugging tools.

4.1 SAT-based Debugging

A *SAT-based debugging* technique was proposed in SAT [20] where the problem is formulated as a SAT instance for a conventional solver to return solutions corresponding to suspects. A variety of SAT-based debugging formulations have been proposed building on the initial work [6] [13] [18]. Experiments show that SAT-based debugging outperforms traditional simulation- and BDD-based techniques, in both time and space, sometimes by orders of magnitude.



(a) (b)

Figure 2: SAT-based debugging

To model debugging [20], a multiplexer m_i is added for every gate (and primary input) l_i. The output of this multiplexer, m_i, is connected to the fanouts of l_i while l_i is disconnected from its fanouts. This construction has the following effect: when the select line s_i of a multiplexer is inactive ($s_i = 0$), the original gate l_i is connected to m_i, otherwise, when $s_i = 1$ a new unconstrained primary input w_i is connected. Figure 2 illustrates the above transformation for a combinational circuit.

This construction is later constrained with the input/output values of the expected primary output responses for the particular error-trace. A potential correction on line l_i is indicated when the select line s_i is assigned to 1 under which condition the correction value is stored in w_i. The SAT solver can assign any value {0,1} to the s_i and w_i variables such that the resulting Conjunctive Normal Form (CNF) satisfies the constraints applied by the vectors V. To force the SAT solver to find a specific number N of error locations, further logic is added to activate at most N select lines [20]. Thus for $N = 1$, a single s_i is set to 1 which *corresponds* to candidate error location l_i, etc. Finally, the construction is repeated and constrained for each error-trace before given to a SAT solver.

Following the initial formulation, a variety of advances have been proposed to improve performance. More notably, the work in [18] borrows the concepts of abstraction and refinement from formal verification to ease the debugging effort. The authors in [2] use Quantified Boolean Formula (QBF) Satisfiability to “compress” the memory required by replicating the CNF for the different error-traces. Finally, orthogonal to [20], the research in [7] and [19] introduces SAT-based techniques to reduce the length of the error-traces and further reduce the memory requirements for existing debugging methods.

4.2 QBF-based Debugging

The backbone of the SAT-based formulation proposed by [20] when applied on sequential circuits is the repetition of the combinational circuitry for a number of cycles equal to this of the error-trace. This is also known as the Iterative Logic Array representation or time-frame expansion [17]. Clearly, replicating a half million gate block for possibly thousands of cycles of industrial-size error-traces, it may require prohibitively excessive memory resources. Evidently, more compact representations of sequential debugging problems are required to ensure scalability with no sacrifice in performance.

To that end, [17] presents a parameterizable encoding for debugging using QBF Satisfiability. In Boolean SAT all variables in the CNF are existentially quantified. QBF is a generalization of SAT that also allows for universal (∀) quantification of the variables. A QBF formula in *prenex normal form* is written as:

$$Q_1\mathcal{V}_1 \quad Q_2\mathcal{V}_2 \quad \cdots \quad Q_r\mathcal{V}_r \quad | \quad \Phi \tag{2}$$

The design debugging formulation using QBF is given by the following equation:

$$\begin{aligned} &\exists e, s^0, s^1, \ldots, s^k, X, Y \ \ \forall t \ \ \exists s, s', x, w, y \ \ | \\ &\bigwedge_{j=1}^{k} t^k(j) \rightarrow [(s = s^{j-1}) \wedge (s' = s^j)] \wedge \\ &\bigwedge_{j=1}^{k} t^k(j) \rightarrow [(x = x^j) \wedge (y = y^j)] \wedge \\ &T_{en}(s, s', \langle x, w, e \rangle, y) \wedge \Phi_C(s^0, X, Y) \wedge \Phi_N(e) \end{aligned} \tag{3}$$

where e are the error location select lines, $s^0 \ldots s^k$ are state elements for the k-cycle error-trace, and X (Y) is the set of design primary input (output). Although the intricate details of the encoding are beyond the context of this paper [17], pictorially the hardware construction that corresponds to Eq. 3 is shown in Figure 3. In that figure, T_{EN} is a single copy of the combinational circuitry (*i.e.,* transition relation) from Figure 2(b). Intuitively, the QBF formulation mitigates the space expansion of the circuit into time using universal quantification. Experiments shows the favorable nature of this encoding as it achieves a dramatic 92% reduction in space when compared to SAT and sometimes outperforms it in terms of run-time.

5. INDUSTRIAL CASE STUDIES

With the help of recent advances, automated debugging tools for industrial problems are within reach. In this section, we present three case studies representative of common bugs found in the RTL. For each of the cases, we show a code snippet in Verilog containing the bug as well as the correct implementation shown in comments (*i.e.* `//`). We present how a state-of-the-art industrial automated debugger using the methodologies from Sections 4 and 4.2 efficiently tackles the problem in Table 1.

In the first example, shown in Fig. 4, the 0 and 1 input of the multiplexer are incorrectly connected. This type of mistake is very common, as it is easy to mix-up the signals or forget the polarity of the condition. In this case, the bug is detected by a self-checking testbench when the output of the memory controller is found to be different than the expected value.

In the second example, shown in Fig. 5, a single clock fifo module, `vga_fifo`, is erroneously instantiated instead of dual clock fifo module, `vga_fifo_dc`. This type of error can be caused by missing details in specifications that do not explicitly require a dual clock fifo. This bug is caught by an end checker when the pixels generated by the vga controller do not match those of a golden C model. In this case, it can be very hard to narrow down the problem from the controller output all the way to the fifo module.

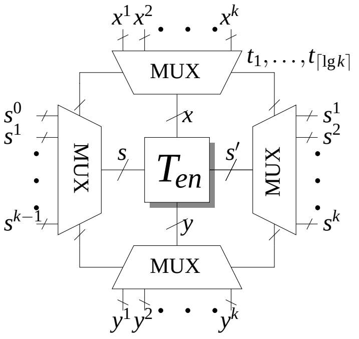


Figure 3: Design debugging construction

```
always @(posedge clk)
   // correct line:
   //wb_data <= #1 ‘MC_MEM_SEL ? mem : rf;
   // bug below:
   wb_data <= #1 ‘MC_MEM_SEL ? rf : mem;
```

Figure 4: Case study 1: misinterpretation of condition in memory controller

```
// correct block:
//    vga_fifo_dc line_fifo(
//        .rclk  ( clk_p_i           ),
//        .wclk  ( wb_clk_i          ),
//        .rclr  ( 1’b0              ),
//        .wclr  ( ctrl_ven_not      ),
//        .wreq  ( line_fifo_wreq    ),
//        .d     ( line_fifo_d       ),
//        .rreq  ( line_fifo_rreq    ),
//        .q     ( line_fifo_q       ),
//        .empty ( line_fifo_empty_rd ),
//        .full  ( line_fifo_full_wr  )
//    );
//
// bug below:
    vga_fifo  line_fifo (
        .clk  ( clk_p_i            ),
        .aclr  ( 1’b0              ),
        .sclr  ( ctrl_ven_not      ),
        .wreq  ( line_fifo_wreq    ),
        .d     ( line_fifo_d       ),
        .rreq  ( line_fifo_rreq    ),
        .q     ( line_fifo_q       ),
        .empty ( line_fifo_empty_rd ),
        .full  ( line_fifo_full_wr  )
    );
```

Figure 5: Case study 2: wrong fifo instantiation in vga controller

The third example, shown in Fig. 6, contains multiple errors where the `if` conditions for signals `RX_W` and `RC_W` are misinterpreted. In this large communication block, the errors affect control circuitry which triggers an assertion to fail dozens of clock cycles after the bug has been excited.

```
always@(posedge CLK)
  // correct line:
  //if(RX_W == 1’h1)begin
  // bug below:
  if(RX_W == 1’h0)begin
    NTY_CTL <= RX_CTL;
    NTY_DATA <= RX_DATA;
  end
  else begin
    // correct line:
    //if(RC_W == 1’h1)begin
    // bug below:
    if(RC_W == 1’h0)begin
      NTY_CTL[63:48]<= RC_CTL[63:48];
      NTY_CTL[7:5]<= REG_RES_FRAME[2:0];
    end
  end
```

Figure 6: Case study 3: multiple wrong conditions in communication block

For the three case studies, Table 1 shows the design type and size of the circuit in primitive gates in columns one and two respectively. Columns three and four present the time required by the debugger to localize the suspects, and the total number of suspects returned to the engineer. In all cases, the debugger eliminates more than 99% of the code in a few seconds or minutes. The suspects returned point the engineer to a small set of Verilog lines where the design can be rectified and the bug removed. The relatively small run-time of the tool makes the automated debugger a powerful and practical tool to aid engineers in the daunting debugging task.

6. CONCLUSION

Debugging of RTL designs remains a manual and resource-intensive task today. This paper outlines the rich research in debugging over the past 20 years. It also builds the case for novel automated debugging methodologies for industrial applications. In the near future, these tools can help reduce the manual debugging pain as well as the overall verification effort.

Table 1: Case study statistics

Design type	# gates	Debug time	# Suspects
memory controller	46K	26 sec.	5
vga controller	150K	32 sec.	12
communication block	800K	672 sec.	10

7. REFERENCES

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, “Logic Verification Via Test Generation”, in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 1988.

[2] M. Fahim Ali and S. Safarpour and A. Veneris and M. Abadir and R. Drechsler, “Post-verification debugging of hierarchical designs,” in *IEEE Int. Conf. on Computer-Aided Design*, pp. 871-876, 2005.

[3] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers, 2003.

[4] V. Boppana and M. Fujita, “Modeling the unknown!Towards model-independent fault and error diagnosis,” in *IEEE Intern. Test Conference*, pp. 678-687, 1998.

[5] R. E. Bryant, “Graph–based algorithms for Boolean function manipulation,” in *IEEE Trans. on Computers*, vol. C–35, no. 8, pp. 677-691, 1986.

[6] K.H-. Chang, I. Markov and V. Bertacco, “Automating post-silicon debugging and repair,” in *IEEE Int. Conf. on Computer-Aided Design*, pp. 91-98, 2007.

[7] K.H-. Chang, I. Markov and V. Bertacco, “Simulation-based Bug trace minimization with BMC-based Refinement,” in *IEEE Trans. on Computer-Aided Design*, vol. 26, no. 1, Jan. 2007.

[8] S. Chatterjee, A. Mishchenko, R. Brayton and A. Kuehlmann, “On resolution proofs for combinational equivalence,” in IEEE/ACM Design Automation Conference, pp. 600-605, 2007.

[9] P. Y. Chung and I. N. Hajj, “Diagnosis and correction of multiple design errors in digital circuits,” in *IEEE Trans. on VLSI Systems*, vol. 5, no. 2, pp. 233-237, June 1997.

[10] D. Geus, *ICs will rebound*, EETimes, Mar. 3, 2009.

[11] International Technonology Roadmap for Semiconductors, 2006.

[12] R. Drechsler, *Advanced Formal Verification*, Kluwer Academic Publishers, 2004.

[13] G. Fey, S. Staber, R. Bloem and R. Drechsler, “Automatic Fault Localization for Property Checking,” in *IEEE Trans. on Computer-Aided Design*,, vol. 27, no. 6, June 2008.

[14] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[15] J. Jaeger, *“Virtually every ASIC ends up an FPGA*, EETimes, Dec. 7, 2007.

[16] J. B. Liu and A. Veneris, “Incremental Diagnosis,” in *IEEE Trans. on Computer-Aided Design, vol. 24, no. 2, pp. 240-251,* Febr. 2005.

[17] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti and D. Smith, “A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test,” in *IEEE Int. Conf. on Computer-Aided Design*, pp. 240-245, 2007.

[18] S. Safarpour and A. Veneris, “Abstraction and Refinement techniques in Automated Design Debugging,” in *IEEE/ACM Design and Test in Europe*, pp. 1-6, 2007.

[19] S. Safarpour, A. Veneris and H. Mangassarian, “Trace Compaction using SAT-based Reachability Analysis,” in *IEEE Asian-South Pacific Design Automation Conference*, pp. 23-26, 2007.

[20] A. Smith, A. Veneris, M. Fahim Ali and A. Viglas, “Fault Diagnosis and Logic Debugging Using Boolean Satisfiability,” in *IEEE Trans on Computer-Aided Design*, vol. 24, no. 10, pp. 1606-1621, 2005.

[21] M. Tomita, T. Yamamoto, F. Sumikawa and K. Hirano, “Rectification of multiple logic design errors in multiple output circuits,” in *Proc. of the Design Automation Conf.*, pp. 212–217, 1994.

[22] A. Veneris, and I. N. Hajj, “Design Error Diagnosis and Correction Via Test Vector Simulation,” in *IEEE Trans. on Computer-Aided Design*, vol. 18, no. 12, pp. 1803–1816, Dec. 1999.