A Scan Pattern Debugger for Partial Scan Industrial Designs

Kameshwar Chandrasekar*, Supratik K. Misra+, Sanjay Sengupta* and Michael S. Hsiao+
kameshwar.chandrasekar@intel.com, supratik@vt.edu, sanjay.sengupta@intel.com and mhsiao@vt.edu
*Intel Corporation, Santa Clara, CA 95054 (USA)
+ECE Department, Virginia Tech, Blacksburg, VA 24061 (USA)

Abstract- **In this paper, we propose an implication graph based sequential logic simulator for debugging scan pattern failures encountered during First Silicon. A novel Debug Implication Graph (DIG) is constructed during logic simulation of the failing scan pattern. An efficient node traversal mechanism across time frames, in the DIG, is used to perform the root-cause analysis for the failing scan-cells. We have developed an Interactive Pattern Debug environment (IDE), viz. scan pattern debugger, around the logic simulator to systematically analyze and root-cause the failures. We integrated the proposed technique into the scan ATPG flow for industrial microprocessor designs. We were able to resolve the First Silicon logical pattern failures within hours, which would have otherwise taken a few days of manual effort.**

I. Introduction

In today's scenario, where changes in functional blocks during hardware designs have reduced the dependency on functional coverage from legacy content of previous products; Scan DFT enables a systematic methodology for making design changes to insert scan-cells, generating scan content, estimating / achieving test coverage goals, and recently diagnosing process issues during Low Yield Analysis (LYA).

A typical scan pattern delivery / debug flow is shown in Figure 1. Scan patterns are generated by an Automatic Test Pattern Generation (ATPG) tool that operates on a gate level abstraction model of the design. The environment for the functional block – including clocks, interface signals, partially implemented blocks and initial sequential values - are typically specified as ATPG constraints. These ATPG constraints are based on design assumptions / specifications and are taken as inputs to the ATPG tool.

In this work, we focus on addressing the pattern issues in First Silicon, i.e., debugging the silicon failures and fixing the incorrect ATPG collateral assumptions made during pattern generation. These ATPG collateral issues could arise from incorrect clock / constraint assumptions, incorrect sequential initialization, modeling issues, etc. Please note that the goal is NOT to perform diagnosis on a large number of Si samples. The diagnosis is typically performed after the pattern debug step, i.e., after the patterns have been cleaned and stabilized. The debug engineer's visibility is also limited to the internal signal values in the ATPG model only and does not include *all* the corresponding values on silicon.

The contributions of this work are:

1. We introduce a novel Debug Implication Graph (DIG) that is dynamically constructed during logic simulation of the failing pattern.
2. We propose a failure analysis technique to root-cause the Silicon failures using the DIG.
3. We developed an Interactive Debug environment (IDE), viz. scan pattern debugger, for the debug engineer to analyze the root-cause and fix the issue.

We integrated the proposed techniques into the scan ATPG flow for industrial microprocessor designs. The IDE enabled the debug engineers to successfully root-cause all the logical failures, understand the cause and clean the patterns.

II. Previous Work

The first systematic debug approach for First silicon debug issues was proposed in [1] based on a fault injection approach. In this approach, a fault is injected on each of the candidates and the response is compared against the failing silicon response. If the faulty response matches the failing silicon response, then the score for the candidate is incremented. The technique has been successful in identifying a set of potential candidates based on this analysis. It requires multiple fault simulations and the performance is dependent on the number of potential candidates identified for analysis. It does not provide any transparency into the internal signals for debug analysis. This is necessary to understand the issue and fix the failure.

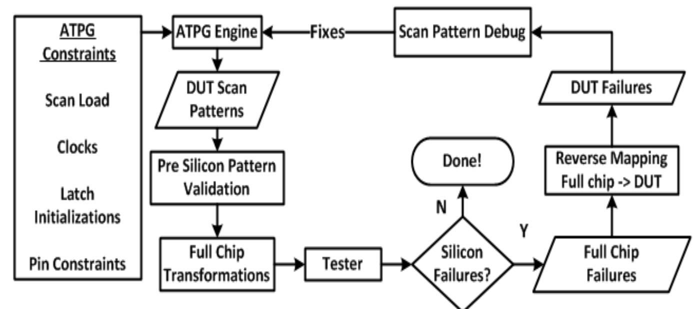


Figure 1: Scan Pattern delivery / Debug flow

Another approach to debug pattern mismatches was presented in [2], albeit, for debugging pattern mismatches during the pattern validation step (please refer Figure 1). The ATPG pattern is simulated with a timing model and the signals in the logic cone of the failing scan cell are dumped. This approach is dependent on the internal signal values that are available only during the pattern validation step and not for silicon debug. It has significant potential to identify modeling and race issues pre-silicon that can alleviate the First Silicon failures that might otherwise be encountered.

On the other hand, a *scan pattern debugger* is necessary to understand the root-cause and fix the failures. It can also serve complementary to the above approaches by providing

978-3-9810801-8-6/DATE12/©2012 EDAA

visibility into the internal signals of the circuit under test (CUT) in the ATPG model for fixing a Si failure. We use an event driven logic simulation [3] based method and construct an implication graph to build the scan pattern debugger. We refer the reader to [4], [5], [6] and [9] for details on the implication graph and some of its applications in the Test domain. In general, the implication graph provides a very strong data model to keep track of the simulation values and their root-cause.

III. DEBUG IMPLICATION GRAPH (DIG)

A DIG is the back-bone of our scan pattern debugger. The implication graph is dynamically constructed during event driven logic simulation of the failing pattern with minimal performance over-head. An efficient backward traversal algorithm on the DIG is then used for failure analysis during debug.

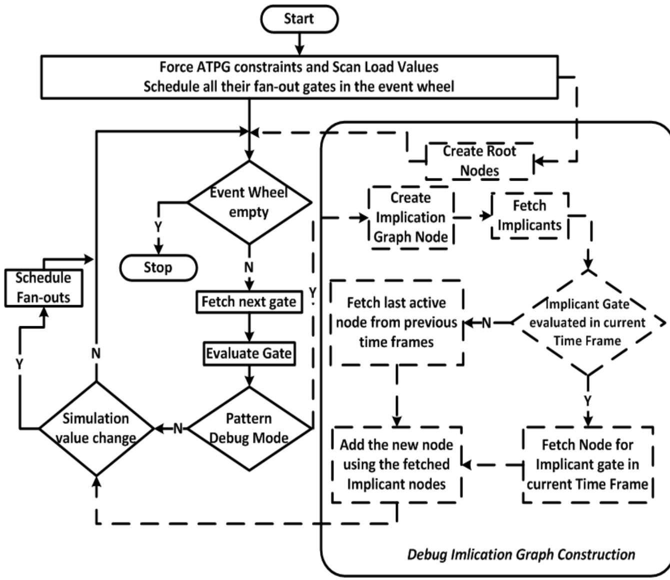


Figure 2: Debug implication graph construction

In Figure 2, the blocks outside the solid box correspond to the base line logic simulator. Each gate in the event-wheel is evaluated and the fan-outs are scheduled <u>only if the gate's value changes</u>. A subtle, but important, fact is that the logic simulator holds this property across time frames while simulating a pattern as well. We dynamically construct the DIG during logic simulation without disrupting these fundamental properties to keep the performance benefits of event driven simulation

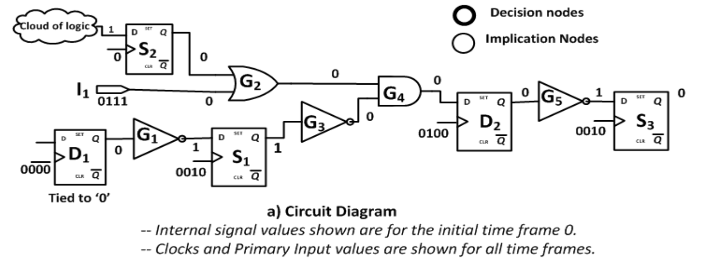
A. Debug Implication Graph (DIG) Construction

The DIG construction, during event driven logic simulation, is shown in dotted lines inside the box in Figure 2. We start the implication graph construction by creating the *root nodes* for each of the scan load values, constrained pins, clocks, internal constraints and latch initialization values. After each gate is evaluated, we create an *implication node* for the gate. The implicant nodes are added as fan-ins to the implication node to represent the implication relations.

We illustrate the implication graph construction with a circuit fragment shown in Figure 3a, for a test vector with a test cycle width of 4 time frames. S_1, S_2 and S_3 are *scan latches* with clock waveform 0010, D_1 and D_2 are non-scan latches that are initialized to 0. D_1 and D_2 get clock waveform 0000 and 0100 respectively. The primary input I_1 is constrained to 0111. The values for all gates corresponding to the four Frames from 0 to 3 are shown in the Figure 3a from left to right. We use the following convention in DIG: (i) the root nodes are represented by thick circles and the implied nodes are represented by thinner circles. (ii) Each graph node has the gate name inside it and the time frame is denoted by the super-script. The nodes are represented by node numbers in order of gate evaluation / node creation.

The DIG constructed in Frame 0 is shown in Figure 3b. Initially, we create the root nodes for the scan cell values S_1, S_2 and S_3, non-scan initialization D_1 and D_2 and, primary input value I_1. These are represented by nodes N_1-N_6 in the graph. During logic simulation, we create the nodes for the remaining gates. For the sake of explanation, let us consider evaluation of gates G_2 and G_4. The value G_2=0 is implied by I_1=0 and S_2=0. We create a graph node N_8 (G_2=0) and add the implicant nodes – N_1 and N_2 as its fan-ins. At G_4^0, either G_3^0=0 or G_2^0=0 can imply the value since it is the controlling value of an AND gate. We can pick either of them without loss of generality. We pick G_2^0 in this case and construct N_{10}. In this way, all graph nodes are created for Frame 0 as shown in Figure 3b.

In Frame 1, I_1 changes from 0 to 1. Correspondingly, the graph nodes N_{12} and N_{13} are created in the DIG, as shown in Figure 4a. The value at gate G_4 remains at 0, however, the value is implied by G_3 and not G_2 anymore. A new graph node, N_{14}, is created for G_4^1 and the graph node from the *most recent frame* for G_3, i.e. N_9, is added as its implicant . Next, D_2 is scheduled for evaluation since the clock value changes in Frame 1 (and not because of G_4). D_2 was initialized to 0 with graph node N_5. Nevertheless, a new graph node is created for D_2 in Frame 1 with G_4^1 as implicant to reflect the latest implicant (the clock graph node is skipped in Figure 4a for brevity).



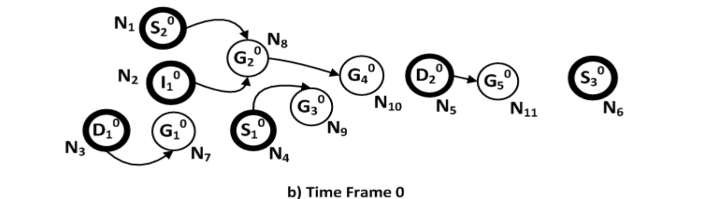


Figure 3: Debug Implication Graph construction

In Frame 2, the capture clocks are fired to capture the values into scan cells. S_1, S_2 and S_3 are scheduled and evaluated in Frame 2 during logic simulation. The corresponding nodes are created in the DIG for the values captured in the scan-cells as shown in Figure 4b.

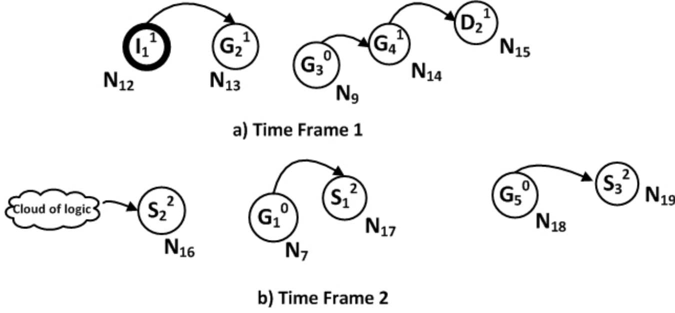


Figure 4: Debug implication graph construction (contd.)

The final DIG is shown in Figure 5. It can be noted that there are discontinuities in the graph (shown in dotted circles) due to the evaluation of a gate to same value in different time frames. This is a characteristic of the event driven simulation to avoid re-simulating the fan-out cone of those gates. It will keep the efficiency of our framework and have an increased impact on multiple capture or sequentially deep test vectors.

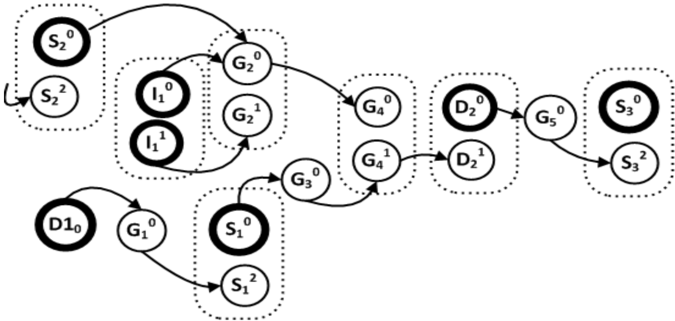


Figure 5: Final Debug Implication Graph

B. *Root-cause analysis with DIG*

For a given gate (g) with value (v) in time frame (t), at least one graph node g^i *s.t.* $i \in (0,t)$ will be created in the DIG while evaluating the gate. Otherwise, the gate would not have been assigned a value to start with. It should be noted that a gate could have been evaluated multiple times in different time frames and different nodes will be created as shown in the dotted circles in Figure 5. The actual graph node responsible for the gate value in time frame t, is represented by,

$$R(g,t) = g^i;\ s.t.\ \ i=max(0,t) \text{ and } g^i \in DIG$$

For example: $R(S_1,0) = S_1^0$ and $R(S_1,3) = S_1^2$ represents the difference between scan load and unload values on S_1.

To find the root-cause of a gate value in a particular frame, we can perform a *smart* backward traversal from the corresponding implication graph node , $R(g,t)$, to the root nodes in DIG as shown in Equation 1.

$$Root\ Cause\ (g,t) = \begin{cases} R(g,t)\ if\ R(g,t)\ is\ a\ root\ node \\ Root\ cause\ of\ R(fanins(g,t));\ otherwise \end{cases}$$

Equation 1: Root-cause Analysis in DIG

When we reach a graph node during backward traversal, we should update to the latest graph node, $R(g,t)$ and the traversal time frame for further traversal. These scenarios are shown by dotted circles in Figure 5. For example, let us say, we need to root-cause the value on S_3 in the unload frame. We will start with R (S_3, 3) = S_3^2, and update the traversal time frame to 2.

During backward traversal in DIG, we will do similar update at D_2 when we reach D_2^0 by updating R(D_2, 2) = D_2^1 and updating the traversal time frame to 1. In this way we can reach S_1^0 as the root-cause for unloading a value 1 on S_3. Note that we will not back-trace further since R(S_1, 1) = S_1^0 is a root node. The DIG can be used to debug any gate value during simulation and is not limited to scan-cell unload values.

IV. Interactive Pattern Debug Environment (IDE)

We use the DIG as the back-bone to create an Interactive Pattern Debug Environment (IDE) for the debug engineer. We have built a set of interactive commands that are suited to debug the First Silicon failures viz. root-cause the silicon failures, understand the pattern behavior in CUT and clean the patterns. Typically, First Silicon debug consists of two high-level steps:

Select a silicon failure: The debug engineer needs to first select a failure i.e. a scan-cell that fails on silicon and the corresponding test vector where it fails. We refer to the <scan-cell, test vector> pair as a silicon failure. To select a failure, we developed a set of simple and effective interactive debug commands that can:

a. Rank failing scan-cells by #times they fail
b. Rank failing scan-cells by size of fanin-logic cone with higher precedence for logic cone from scan-cells than non-scan cells
c. Group scan-cells with similar names and rank them based on criteria *a.* and *b.*
d. Sort all failing test vectors for a scan-cell

To select a failing scan-cell, we use a combination of group and rank mechanism, with a heuristic to (i) select a failing scan-cell that has many other similar failures, (ii) has many failures cumulatively as a group and (iii) has a smaller support set. The first test vector for the selected scan-cell is picked using command *d*. It has been our experience that fixing the selected failure, using the group and rank mechanism, can resolve most of the issues related to the group. A common fanin-cone based grouping would also be generic alternative in place of a name-based grouping mechanism. In some cases, it is possible that the test engineer gives inputs to debug specific failures based on design / tester feedback. In that case, we can by-pass this step and directly start with debugging the silicon failure.

Debug the silicon failure: We first start with clock path debug and then proceed to the data path debug.

Clock path failure debug: The clock path is expected to be the source of the failure: if all scan-cells driven by the same clock source and expecting different load/unload values fail on silicon [7]. This analysis is done interactively on the scan-

cell chosen based on the criteria in Step 1. In addition to this analysis, the IDE enables us to query the DIG and analyze the expected clock values in the clock logic systematically to root-cause the potential cause of clock not firing, understand the internal values of the clock network and fix any issue

Data path failure debug: The data path debug is premised on two interactive commands that perform the following: (i) logic-simulate the failing test pattern to the selected test vector and (ii) to root-cause the value on any gate in the selected test vector. The first command logic simulates to the selected test vector. The DIG construction is triggered for the selected test vector, using the mechanism in Section 3.2, and skipped for the previous test vectors by default. The second command uses the concepts in Section 3.3 to identify the root-cause for any gate value in the test cycle.

V. INDUSTRIAL DATA

The scan pattern debugger was integrated into the scan ATPG flow for pattern debug in industrial micro-processor designs in a *production* environment.

TABLE I : INDUSTRIAL DATA FOR PATTERN DEBUGGER

D	Size	Root-cause reported	Debug Fix	Debug time
D_1	711K	HL & PI	HL from non-scan blocks	3.5 hrs
D_2	3.4M	LI	Initialization Seq.	1.5 hrs
D_3	6.2M	LI & DC	Incorrect LI	3.5 hrs
D_4	9.8M	SL & LI	Incorrect LI	6.5 hrs
D_5	1.1M	PI and LI	PI values from other CUT instance	2.0 hrs
LI : Latch Initialization, DC : Design Constraints HL: Hold Latch, PI: Primary Input, SL: Scan Loads				

TABLE II: DIG RUN-TIME AND MEMORY OVER-HEAD

D	Size	Run time over-head		Memory over-head	
	#gates	100 TVs	Avg/TV	100 TV	Avg/TV
D_1	711K	2X	2%	8.6X	8.6%
D_2	3.4M	1.7X	1.7%	6.8X	6.8%
D_3	6.2M	2X	2%	9.2X	9.2%
D_4	9.8M	2.4X	2.4%	7.6X	7.6%
D_5	1.1M	2X	2%	30X	30%
TV : Test Vector					

In Table 1, Column 1 lists the design, Column 2 shows the design size, Column 3 reports the root-cause, Column 4 shows the actual fix done with the debugger and Column 5 represents the debug time for fixing the cause. The debug time involves debugging multiple scan cell failures. In D_1 with 711K gates, the tool reported the primary input and hold-latches as root-causes. After carefully examining the internals of CUT, the hold latch initializations were fixed to clean the pattern. On previous products, First Silicon debug took days, in the absence of the scan pattern debugger. We have reduced the debug time to the order of a few hours from the order of days.

In Table 2, we report the run-time and memory over-head for the DIG for 100 test vectors in Columns 3 and 5 and the average over-head per test vector in Columns 4 and 6.
It is seen that the average run-time over head is 1-2.5% and memory over-head is 8-30% for each test vector in the debug build of the logic simulator. The increased memory over-head, in $D_{5,}$ is attributed to higher activity in the CUT leading to larger number of graph nodes being created.

VI. CONCLUSION

We proposed a scan pattern debugger framework to debug First silicon failures and clean scan patterns. The approach relies on constructing a novel debug implication graph (DIG) during logic simulation to serve as the back-bone of the debugger. An interactive pattern debug environment (IDE) is built around the implication graph to select and debug the scan cell failures. We were able to root-cause and clean all pattern related issues in a matter of few hours, as compared to a few days on previous products.

REFERENCES

[1] D. Nayak, S. Venkataraman, P. Thadikaran, "Razor: a tool for post-silicon scan ATPG pattern debug and its application," VLSI Test Symposium, 2004. Proceedings. 22nd IEEE , vol., no., pp. 97- 102, 25-29 April.

[2] Kun-Han Tsai; Ruifeng Guo; Wu-Tung Cheng, "A Robust Automated Scan Pattern Mismatch Debugger," Asian Test Symposium, 2008. ATS '08. 17th , vol., no., pp.309-314.

[3] Z. Wang, P.M. Maurer, "LECSIM: a levelized event-driven compiled logic simulator," Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE , vol., no., pp.491-496.

[4] P. Tafertshofer, A. Ganz, M. Henftling, "A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists," Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on , vol., no., pp.648-655.

[5] S. Bommu, K. Chandrasekar, R. Kundu, S. Sengupta, "CONCAT: CONflict Driven Learning in ATPG for Industrial designs," Test Conference, 2008. ITC 2008. IEEE International , vol., no., pp.1-10.

[6] M.H. Schulz, E. Trischler, T.M. Sarfert, "SOCRATES: a highly efficient automatic test pattern generation system," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.7, no.1, pp.126-137.

[7] Dmitris Gizopoulos, "Advances in Electronic Testing – Challenges and Methodologies", Springer, 2006.

[8] S. Venkataraman, S.B. Drummonds, "Poirot: applications of a logic fault diagnosis tool, " Design & Test of Computers, IEEE , vol.18, no.1, pp.19-30.

[9] Chandrasekar, K.; Bommu, S.; Sengupta, S.; , "Low Coverage Analysis using dynamic un-testability debug in ATPG," VLSI Test Symposium (VTS), 2011 IEEE 29th , vol., no., pp.291-296