Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation

Ho Fai Ko and Nicola Nicolici
Department of Electrical and Computer Engineering
McMaster University, Hamilton, ON L8S 4K1, Canada
Email: henryko@grads.ece.mcmaster.ca, nicola@ece.mcmaster.ca

Abstract

Embedded logic analysis has emerged as a powerful technique for identifying functional bugs during post-silicon validation, as it enables at-speed acquisition of data from the circuit nodes in real-time. Nonetheless, the amount of data that is observed is limited by the capacity of the on-chip trace buffers. This paper introduces an automated method for improving the utilization of the on-chip storage, by identifying a small set of trace signals from which a large number of states can be restored using a compute-efficient algorithm. This enlarged set of data can then be used to aid the search of functional bugs in the fabricated circuit.

1. Introduction

In the implementation flow for very-large scale integrated (VLSI) circuits (summarized in Figure 1), *pre-silicon verification* techniques, such as simulation or formal methods, assist designers in eliminating functional errors (or bugs) in a circuit before it is manufactured to ensure that the implemented design satisfies the specification. In a subsequent step, manufacturing test helps to detect physical defects (e.g., shorts or opens) prior to delivering the packaged circuits to end-users. As state-of-the-art process technologies enable system-on-a-chip (SOC) designs with multi-million transistors, the time required to extensively simulate or formally verify large circuits and complex interfaces becomes unbearable. Due to this increased design complexity and the inadequate accuracy in modeling integrated circuits (ICs), pre-silicon design verification techniques are insufficient to guarantee error-free first silicon. Given the escalating mask costs, it is imperative to identify the escaped bugs as soon as the first silicon is available [14]. Semiconductor manufacturers currently rely mainly on custom in-house methods for debugging in silicon. Nevertheless the continuous growth in the design size, the use of embedded cores from third parties, and the complex interactions between these cores drive a shift toward scalable *post-silicon validation* techniques. This explains why structured post-silicon validation techniques have experienced an increased adoption over the past few years [1, 2, 12].

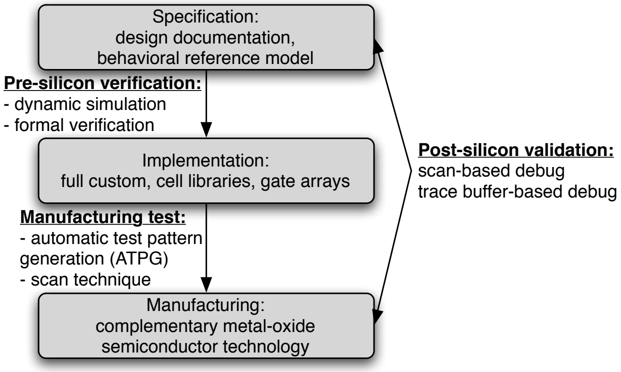


Figure 1. Design flow for VLSI circuits

The main challenge in post-silicon validation (or silicon debug) is observability of internal signals. Physical probing techniques have been widely used for IC failure analysis [13]. Nonetheless, the decreasing feature sizes, flip-chip technologies and the growing complexity of SOCs make data acquisition using physical probing cumbersome, unless they are complemented by design for debug (DFD) techniques. Embedded logic analysis [3] is a DFD technique used for improving observability of internal signals in a design by acquiring data on-chip in trace buffers. The sampled data is then transferred through low bandwidth device pins, so that post-processing algorithms can be applied and help identify design errors. The limited observability of the internal signals may lengthen the debug process and it motivates our work. Instead of introducing yet another technique for increasing observability by acquiring more data on-chip, our objective is to better utilize the limited storage space provided by the DFD hardware. By consciously selecting the trace signals when designing the DFD hardware, we show that one will be able to restore a significant amount of missing data from the internal state elements that are not traced during post-silicon validation. For instance, it is a

978-3-9810801-3-1/DATE08 © 2008 EDAA

common practice for microprocessor designers to *manually identify* which signals are to be captured (e.g., pipe control signals), such that additional state information (e.g., values in pipe data registers) can be reconstructed off-chip [5].

Motivated by the lack of information in this area in the public domain, *the aim of this paper is to provide a first step to develop the understanding on how computer-aided design (CAD) technology can aid structured post-silicon validation at the core level*. By proposing automated algorithms for identifying trace signals and restoring state information, we aim to develop a structured method for debugging cores in future SOCs. Note that the proposed method can be integrated together with system level debug techniques such as [15]. When an error among cores is detected at the system level, the proposed method can be used to identify the source of the error by analyzing individual cores in an SOC. Also, the proposed method complements existing data acquisition techniques, because the more signals are traced, the more data can then be reconstructed by our algorithms. It is also important to note that the proposed technique focuses only on post-silicon validation for *functional errors*.

2. Post-Silicon Validation Techniques

To address the problem of limited observability in post-silicon validation, a number of ad-hoc DFD solutions for improving data acquisition in microprocessors were proposed [14]. These solutions can be divided into two main categories: *scan-based* and *trace buffer-based* techniques.

The scan-based technique utilizes internal scan chains to capture and off-load the internal states in a design when a specific trigger event occurs. The captured data can then be analyzed using post-processing algorithms such as failure propagation tracing [7] to identify the failing state elements. However, because captured data is shifted out of the chip through the scan chains, the circuit has to stop and then resume its execution. This prevents the designer from acquiring data in real-time. Since functional bugs can appear in circuit states that may be exercised thousands of clock cycles apart [10], it is therefore desirable to maintain circuit execution during scan dumps. This can be achieved by double buffering the scan elements, which will obviously lead to a substantial area penalty [11]. Even if this penalty would be acceptable, data sampling in consecutive clock cycles would still not be possible, which is, however, an essential requirement for identifying timing-related problems in a design during post-silicon validation.

In order to facilitate real-time data acquisition during post-silicon validation, the *trace buffer-based* technique is employed. Using this approach, data is sampled at-speed through embedded logic analyzers, and then stored in trace buffers such as embedded memories on-chip. The sampled data is subsequently transferred off the chip via a low bandwidth interface for post-processing [4, 16]. In this case, the amount of data that can be acquired is limited by the trace buffer *depth*, which limits the number of samples to be stored, and its *width*, which limits the number of trace signals sampled in each clock cycle. Despite the recent advancement in the design of embedded logic analyzers (e.g., [3]), the reluctance to invest additional area for large trace buffers only for the purpose of post-silicon validation limits the amount of available data that can be acquired on-chip. This indirectly translates into a more time-consuming process for identifying the design errors. Thus, it is desirable to find a way to identify the trace signals in a design, such that the acquired data can be used to reconstruct as much missing data (for other internal signals) as possible. This should be done in such manner that any post-processing algorithm can search for design bugs using the enlarged set of data. To the best of authors' knowledge, the only solution discussed in the public domain is [9]. However, based on the description from [9], their algorithm restores data only in the combinational logic nodes of the circuit.

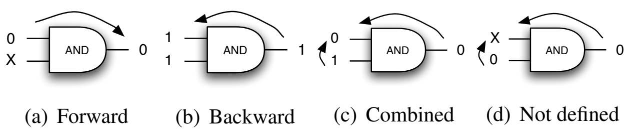
Unlike any prior works, this paper proposes a solution to reconstruct data for sequential circuits across multiple time frames by introducing a compute-efficient state restoration algorithm. In addition, by coupling with automatic trace signal identification for random logic, we show that a significant amount of missing data can be restored for the circuit during post-silicon validation.

3. State Restoration

Throughout this paper we will use the simple circuit shown in Figure 3 to illustrate the key points of our solution. Note, although it is unlikely that one would need our method to debug such a simple circuit in a practical environment, we have decided to use it to demonstrate that our algorithms based on circuit analysis can indeed automatically identify the same trace signals as what an experienced designer would select manually.

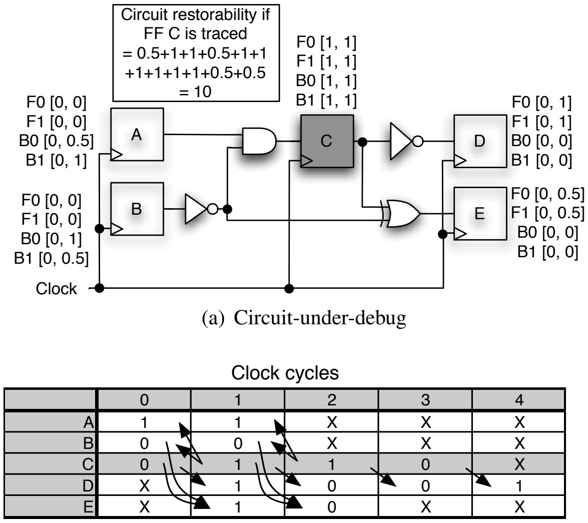
3.1 Principal Operations for State Restoration

Our proposed algorithm relies on applying two *principal operations* to forward propagate and backward justify known values among circuit nodes to reconstruct the missing data. Thus, we label them *forward* and *backward* operations, and they are illustrated in Figure 2. One may argue that the application of the principal operations is similar to the technique used in automatic test pattern generation (ATPG) algorithms to forward propagate and backward justify test patterns for manufacturing test on a circuit netlist. However, unlike ATPG, the state restoration algorithm only needs to check if data can be reconstructed at a circuit node



(a) Forward (b) Backward (c) Combined (d) Not defined

Figure 2. Principal operations



(a) Circuit-under-debug

	0	1	2	3	4
A	1	1	X	X	X
B	0	0	X	X	X
C	0	1	1	0	X
D	X	1	0	0	1
E	X	1	0	X	X

(b) Restored data in sequential elements

Figure 3. Example of state restoration

and no branching and backtracking will be done (if unsuccessful, undefined values will be shown in the simulator as illustrated in Figure 2(d)). After state restoration, the designer can then use the expanded set of data to verify the behavior of the circuit against its specification. In this work, we assume that a gate level netlist is available while debugging a circuit and the trace signals are flip-flops. One may argue that the set of expanded data obtained from the netlist does not reflect the actual circuit responses from the chip. However, as we only focus on the identification of functional bugs, it is assumed that the behavior of the manufactured circuit matches the behavior of the circuit netlist since it has already passed the manufacturing test. This assumption is validated from how state restoration has been employed successfully for microprocessor systems [5], where control data is acquired on-chip, and reconstruction of states in the data path is done off-chip.

Figure 3 shows an example that demonstrates how the principal operations are applied to a simple circuit that has both combinational and sequential elements. While Figure 3(a) shows the simple circuit with five flip-flops (FFs) (F and B values will be explained in Section 4), Figure 3(b) gives the data in the state elements after the restoration algorithm is applied. In this example, only FF C is sampled during clock cycles $0-4$. It should be noted that the X in the table refers to values that cannot be restored using only the available sampled data. By forward propagating and backward justifying known data between gates in the circuit, data can be restored for other state elements one clock cycle at a time. When comparing the amount of data that is available before and after state restoration, a *restoration ratio* of $14/4 = 3.5X$ can be achieved for the elaborated example. One drawback of the proposed algorithm is that the amount of data that can be restored depends on the initial set of sampled data. For instance, if only FF E is sampled in Figure 3(a), no new data can be reconstructed for any state elements in the circuit.

Logic value	Two bit code
0	00
1	11
undefined	01, 10

Table 1. Two bit codes for data representation

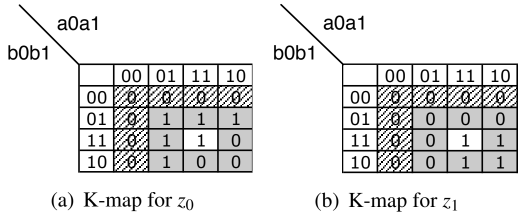
When a circuit netlist is translated into a graph, where the nodes in the graph represent logic gates, state elements, primary inputs and outputs and directed edges represent signal dependencies, the *state restoration algorithm* will apply the principal operations to each node repeatedly until no more data can be reconstructed for all signals from the given subset of data. Thus, the computation time for the state restoration process is directly proportional to the amount of nodes presented in the circuit graph. In addition, when restoring state data for a circuit across a large number of clock cycles, the CPU run time for reconstructing the missing data will be affected, since the principal operations that are introduced so far can just restore state data for one clock cycle at a time.

3.2. Exploiting Bitwise Parallelism for Speeding Up the State Restoration Algorithm

To be applicable to large circuits, it is essential for the state restoration algorithm to be compute-efficient. This is because the designer may need to test the circuit with different stimuli during the debug process. To reduce computation time for the state restoration algorithm, we explore the fact that when performing the principal operations on a node, the results are independent of each other for each data point in different clock cycles. For example, in order to restore data for a circuit for 5 clock cycles, one can iteratively apply the principal operations to each node in each clock cycle. However, this can also be achieved by having 5 copies of the circuit graph, each copy containing the corresponding data for the specific clock cycle, and applying the principal operations in the different graphs simultaneously. Nevertheless, having duplicate copies of the circuit graph requires a prohibitively large amount of memory during program execution for large circuits. As a result, we derive new logic operations for the primitive gates such that the principal operations can be applied concurrently at a node across multiple clock cycles, *without duplicating the circuit graph during state restoration.*

We exploit the integer data type in ANSI C on a 32-bit platform to enhance the performance of our algorithm by storing data for 32 consecutive clock cycles in two integers (8 bytes) for each node. For example, to represent the data $[0,1,1,0,X]$ for clock cycles 0-4 for FF C in Figure 3(a), using the two-bit codes in Table 1, we can store the data for FF C using two integer variables as follows:

$$int0 = 0,1,1,0,1,\ldots,1$$
$$int1 = 0,1,1,0,0,\ldots,0$$



b0b1 \ a0a1	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	0	1	1	0
10	0	1	0	0

(a) K-map for z_0

b0b1 \ a0a1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	1
10	0	0	1	1

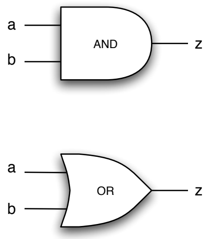
(b) K-map for z_1

Figure 4. Derivation of forward equations for the *AND* gate

In the above equations, the first 5 bits of the two variables store the data for clock cycle 0-4 for FF *C*, and the remaining 27 bits store the code for undefined data. By working with two integer variables, the algorithm can restore data for 32 consecutive clock cycles at a time using a sequence of logic equations based on the bitwise operations provided by ANSI C for each of the primitive gates. For each principal operation, two different equations (one for each integer) will be developed in such way that the number of 2-operand bitwise operations are minimized. Although the formalism of multi-valued logic and input/output encoding from logic synthesis can be used to derive these systems of equations, the following discussion relies on the *illustrative advantage* of the Karnaugh map (K-map) representation.

Figures 4(a) and 4(b) show the K-maps for deriving the logic equations for the forward operation at the output z of an *AND* gate. Note that the inputs of the *AND* gate are labeled a and b, and since two bits are needed for data representation, the variables for the logic equations are labeled a_0, a_1, b_0, b_1 for the inputs, and z_0, z_1 for the output. From the principal operations in Figure 2, we know that when any input of an *AND* gate is 0, the output should also be a 0. This is why the entries are set to 0 on the first rows and the left-most columns of the K-maps for z_0 and z_1 in Figures 4(a) and 4(b). The remaining parts of the K-maps are also filled according to the principal operations. Note that the shaded regions of the K-maps represent inconclusive values due to the insufficient data from the other ports. In these regions, the values of z_0 and z_1 can be filled consciously in such way that the resulting code is 01 or 10 to represent undefined values, and at the same time, the number of bitwise operations is minimized. The K-maps for deriving the logic equations for the backward operation can also be constructed using the same principle. Note that for backward operation, four logic equations (two equations for each input port) will be derived. However, the variables in the logic equations among input ports of a gate will be exchanged. In this example for an *AND* gate, the equations for forward and backward operations become:

$$\begin{array}{ll} z_0 = a_1b_1 + \overline{a_0}a_1b_0 + a_0\overline{b_0}b_1 & z_1 = a_0b_0 \\ a_0 = z_0z_1 + b_0(z_0+z_1) & a_1 = z_0z_1 + \overline{b_0} + \overline{z_0}\,\overline{z_1}\,\overline{b_1} \\ b_0 = z_0z_1 + a_0(z_0+z_1) & b_1 = z_0z_1 + \overline{a_0} + \overline{z_0}\,\overline{z_1}\,\overline{a_1} \end{array}$$



$$\begin{array}{rcl} F0(z) & = & \max\{F0(a), F0(b)\} \\ F1(z) & = & (F1(a) + F1(b))/2 \\ B0(a) & = & (\max\{B0(z)\} + F1(b))/2 \\ B1(a) & = & \max\{B1(z)\} \end{array}$$

$$\begin{array}{rcl} F0(z) & = & (F0(a) + F0(b))/2 \\ F1(z) & = & \max\{F1(a), F1(b)\} \\ B0(a) & = & \max\{B0(z)\} \\ B1(a) & = & (\max\{B1(z)\} + F0(b))/2 \end{array}$$

Figure 5. Restorability calculation

Using these logic equations to restore data in 32 clock cycles, the total number of bitwise operations for the *AND* gate is 10 for one forward operation, and 26 for one backward operation. The equations for other primitive gates can be obtained using similar concepts from the above elaborated example. It should also be emphasized that large digital circuits often involve more complex logic gates, or logic gates with higher fan-in. These complex gates can either be decomposed into a hierarchy of the primitive gates, or additional equations specific to their behavior can be generated.

4. Trace Signals Identification

As shown in the previous section, the amount of data that can be restored by the state restoration algorithm depends on an initial set of data acquired from the DFD hardware.

If a trace signal has large input and output logic cones, the probability of restoring data for other signals through forward and backward operations on this signal will be higher. Moreover, as the probabilities of restoring either 0 or 1 in different logic gates are different, we consider the restorability of 0 and 1 for each logic gate separately. The equations in Figure 5 capture these parameters for calculating the *restorability* of a node in a circuit. We define the *forward restorability* to be the probability of restoring data of a node through forward propagation, while *backward restorability* represents the probability of restoring data from backward justification. When a node can be fully restored through forward (backward) operations, the forward (backward) restorability will be 1.

After defining the equations for calculating restorability of a node, Algorithm 1 can be used to identify the desired trace signals. The algorithm uses a breadth-first-search approach to calculate restorability values for all the nodes. The calculation starts by first computing the forward restorability of all the child nodes of the first node in the search list. It then works out the backward restorability of all the parent nodes of the same node. In the case when sequential loops are found in a circuit, the algorithm will iterate the forward and backward calculations for the nodes in the loop. This is because the state restoration algorithm may be able to restore data for multiple clock cycles by iterating in the loop. In order to limit computation time for the trace signal identification algorithm, a user-defined parameter called *Threshold* is employed. This threshold is used to check the newly computed values against the ones from the previous iteration. It is obvious that the lower the threshold, the more effort the algorithm will spend on calculating the restorability of each signal in the circuit. Figure 3(a) can be used to explain the greedy nature of the algorithm when trying to select the first trace signal. In the figure, the *F0*, *F1*, *B0* and *B1* values represent the forward and backward restorability of each flip-flop respectively for two iterations when assuming FF *C* is selected as trace signal. Note that for the sake of clarity, only the restorability values of the flip-flops are shown, but in fact, the restorability for the logic gates between flip-flops are also calculated.

In the first iteration, the backward restorability *B0* and *B1* of FF *A* are calculated to be 0.5 and 1 respectively according to the equations shown in Figure 5 for the AND gate, since the backward restorability values of FF*C* is set to 1 initially. The restorability of other nodes are calculated in the same manner, and then summed together to give the restorability of the circuit if FF *C* is selected as traced signal. To decide which node to select as the trace signal, the algorithm will calculate the circuit restorability for when each node is selected, it will then choose the node that produces the highest circuit restorability as the trace signal. To select the targeted number of trace signals, Algorithm 1 incrementally calculates circuit restorability to select one signal at a time in a greedy manner. Using the circuit in Figure 3(a) as an example when choosing two signals, and assuming signal FF *C* is chosen in this iteration, Algorithm 1 will then select the second trace signal by trying to select signals in this sequence: FF $C \bigcap$ FF A, FF $C \bigcap$ FF B, ..., until all other signals are selected together with FF *C* as trace signals. It will then choose the signal pair that will produce the highest restorability values across the circuit as trace signals. This gradual approach for trace signals selection follows the same philosophy on how new states are chosen to be probed during microprocessor debug, when signals are also selected incrementally to determine what additional data can be gathered from the microprocessor.

One may argue that the proposed restorability equations resemble the SCOAP controllability/observability concept [8], which is used to calculate the controllability/observability of circuit nodes to guide the ATPG process for manufacturing test. However, it should be noted that during state restoration, the combined operation shown in Figure 2(c) can be used to reconstruct missing data in a circuit. As this is not a valid operation during ATPG, the SCOAP measure does not reflect how data is actually restored among circuit nodes. For instance, in the circuit shown in Figure 3(a), the SCOAP measure will identify FF *E* as a hard to control signal due to the presence of the XOR gate. However, tracing only FF *E* will not be able to help restore data for any other signals in the circuit.

Algorithm 1: Algorithm for identifying trace signals

```
Input   : Circuit, TB_width, Threshold
Output  : list of selected trace signals

while $cur\_width < TB\_width$ do
    while not all nodes in Circuit are calculated do
        search_list = Get chosen nodes;
        Set initial values for chosen nodes;
        while search_list is not empty do
            cur_node = first node in search_list;
            foreach (child_node of cur_node) do
                CalculateForwardRestorability(child_node);
                if ($new\_value - old\_value \geq Threshold$)
                then
                    Put child_node at end of search_list
            foreach (parent_node of cur_node) do
                CalculateBackwardRestorability(parent_node);
                if ($new\_value - old\_value \geq Threshold$)
                then
                    Put parent_node at end of search_list
        Sum the restorability of all nodes in the circuit;
    Select the node with highest restorability;
    cur_width++;
Return list of selected trace signals;
```

5. Experimental Results

Experimental studies [2] indicate that trace buffers of size 1k x 8 (i.e., depth of 1024 and width of 8 bits) to 8k x 32 are accepted in industry today, and it is common that the trace buffer is used as a time-shared resource when debugging larger cores in an SOC. Time sharing is also justified by the fact that if only 32 signals are traced, it is difficult to restore values for over 2,000 flip-flops (which normally belong to a logic block of about 50,000 gates). Given this expected core size, we perform our experiments on the ISCAS89 benchmark circuits [6]. Moreover, since the ISCAS89 benchmark circuits are publicly available, we hope that future proposals on this emerging area can benchmark their algorithms against ours. For our experiments, both the state restoration algorithm and the trace signals identification algorithm are implemented using ANSI C and the program is executed on a PC with dual-Xeon processors at 2.4 GHz with 1 GB of RAM. Due to space limitation, we only discuss the results for s38584 and s35932 to illustrate the key benefits of the proposed solution. However, it should be noted that the same findings are also observed in the experimental results for other ISCAS benchmark circuits.

We first discuss the runtime of the algorithms proposed in this paper. Although the trace signal identification algorithm relies on a greedy strategy, it requires a large number of iterations to propagate the metric in Figure 5 for calculating restorability of each node through the circuit. This algorithm is of order O(*knm*), where *k* is the total number of gates, *n* is the number of trace signals and *m* is the trace

Buffer	Buffer	Random		Thres-	Proposed metric	
depth	width	Ratio	Time (s)	hold	Ratio	Time (s)
4096	8	1.70	0.00	0.50	131.88	120.60
				0.10	126.92	125.80
	16	1.91	0.00	0.50	67.17	126.40
				0.10	65.43	164.60
	32	4.98	5.48	0.50	39.84	196.60
				0.10	37.27	255.20
8192	8	1.70	0.00	0.50	132.17	341.00
				0.10	127.20	345.80
	16	1.91	0.00	0.50	67.37	360.80
				0.10	65.57	470.00
	32	5.23	15.76	0.50	39.96	563.80
				0.10	37.36	834.40

Table 2. State restoration results for s38584

Buffer	Buffer	Random		Thres-	Proposed metric	
depth	width	Ratio	Time (s)	hold	Ratio	Time (s)
4096	8	2.25	0.20	0.50	1.92	0.00
				0.10	254.90	52.80
	16	40.01	1.60	0.50	13.35	3.80
				0.10	127.80	52.40
	32	30.42	2.80	0.50	7.68	3.80
				0.10	64.59	50.80
8192	8	2.24	0.32	0.50	1.92	0.00
				0.10	254.85	132.20
	16	40.67	2.84	0.50	13.08	8.40
				0.10	127.77	130.40
	32	30.16	4.84	0.50	7.54	8.60
				0.10	64.58	126.40

Table 3. State restoration results for s35932

buffer depth. Therefore, it can take hours for selecting 32 trace signals. Although this is a large number which obviously needs to be addressed in the future, it should be noted however that this time is invested only once before the circuit is fabricated. What matters is how much time is spent on state restoration after extracting the samples from the circuit-under-debug, since state restoration is run over and over while searching for design errors. The runtime for state restoration, as shown in Tables 2 and 3, is in the range of seconds to minutes.

Tables 2 and 3 give the results for s38584 and s35932 respectively. The restoration ratios are obtained by comparing the total number of restored values versus the number of values acquired on the trace signals through the on-chip trace buffer. The reported results stand for five different sets of random data and the restored values account also for the primary inputs and outputs of the circuit.

There are several important points to be noted. First, using the proposed metric for trace signal selection can achieve higher restoration ratio than when trace signals are selected randomly. Also, if the threshold for signal selection is decreased, the amount of data that is restored increases for s35932. For s38584, decreasing the threshold does not improve the restoration ratio. This is due to the greedy nature of the signal identification algorithm. Moreover, note that increasing the trace buffer depth for s35932 does not help improve the restoration ratios due to the low sequential depth of the circuit. This is unlike s38584 where the sequential depth is large and it is visible from the results that as the trace buffer depth increases, better ratios are achieved for the same trace buffer widths. This comes at the expense of higher restoration time which is still within an acceptable range of a few minutes. Another factor that significantly contributes to high restoration ratio is the presence of large fan-ins, as in the case of s35932. This is why for this circuit, when using the proposed metric with a threshold of 0.1, state restoration for almost all the flip-flops, inputs and outputs can be achieved.

6. Conclusion

In this paper, we have demostrated how by consciously choosing only a small number of signals to be probed in real-time, the observability of the circuit-under-debug can be improved through automatic state restoration.

References

[1] Special Session: Why Doesn't My System Work? *43rd the IEEE/ACM Design Automation Conference*, 2006.

[2] M. Abramovici. Experience and Opinon (Design for Debug). In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, 2006.

[3] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 7–12, 2006.

[4] Altera Verification Tool. SignalTap II Embedded Logic Analyzer. http://www.altera.com/products/software/products/quartus2 /verification/signaltap2/sig-index.html, 2006.

[5] ARM Limited. Embedded Trace Macrocells. http://www.arm.com/products/solutions/ETM.html, April 2007.

[6] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1929–1934, 1989.

[7] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor Silicon Debug Based on Failure Propagation Tracing. In *Proceedings of the IEEE International Test Conference*, 2005. Paper 12.2.

[8] L. H. Goldstein. Controllability/Observability Analysis of Digital Circuits. *IEEE Transactions on Circuits and Systems*, 26(9):685–693, Sept 1979.

[9] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang. Visibility Enhancement for Silicon Debug. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 13–18, 2006.

[10] D. Josephson. The Manic Depression of Microprocessor Debug. In *Proceedings of the IEEE International Test Conference*, pages 657–663, Oct 2002.

[11] D. Josephson and B. Gottlieb. The Crazy Mixed up World of Silicon Debug. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 665–670, 2004.

[12] A. Khoche and D. Conti. TRP in Action: Embedded Instrumentations in FPGA. In *Proceedings of the 24th IEEE VLSI Test Symposium*, 2006. Session 4C.

[13] N. Nataraj, T. Lundquist, and K. Shah. Fault Localization using Time Resolved Photon Emission and Stil Waveforms. In *Proceedings of the IEEE International Test Conference*, pages 254–263, Sept 2003.

[14] B. Vermeulen and S. K. Goel. Design for Debug: Catching Design Errors in Digital Chips. *IEEE Design and Test of Computers*, 19(3):35–43, May 2002.

[15] B. Vermeulen, K. Goossens, R. v. Steeden, and M. Bennebroek. Communication-Centric SoC Debug Using Transactions. In *Proceedings of the IEEE European Test Symposium*, pages 69–76, 2007.

[16] Xilinx Verification Tool. ChipScope Pro. http://www.xilinx.com/ise/optional_prod/cspro.htm, 2006.