Reliability Considerations in Mobile Devices

I. Voyiatzis
Department of Informatics, Technological Educational Institute of Athens, Greece
voyageri@otenet.gr

D. Kavvadias
Department of Mathematics University of Patras Greece
kavadias@ceid.gr

H. Antonopoulou
TEI of Patras & Computer Engineering and Informatics Department, University of Patras, Greece
antonopl@cti.gr

S. Sinitos
Department of Informatics, Technological Educational Institute of Athens, Greece
ssocratis@yahoo.com

ABSTRACT
The problem of reliability in current chips has been the subject of numerous researchers. Mobile devices, commonly used in multimedia communications require low power during both normal operation and testing. In this paper a novel algorithm is presented for embedding test sets containing don't care values into sequences generated by binary counters. Therefore, both test time and power consumed during testing of the chips can be considerably reduced.

Keywords
Low power testing, Mobile device reliability.

1. INTRODUCTION
The reliability problem in current electronic devices is the subject of ongoing research [1]. Mobile devices, apart from the ever-existing need for low power during normal operation suffer from an increasing demand for low power during testing. Various schemes have been proposed in order to provide efficient low-power testing capabilities. One class of proposed solutions, namely Built-In Self Test (BIST) schemes has been shown to be irreplaceable for testing mobile devices; BIST schemes utilize on-chip circuitry to generate test patterns and verify the respective responses of the circuit. Therefore. they drive down the cost of test and boost the quality of the resulting testing scheme possessing various advantages, e.g. low cost, high quality of the delivered IC for both modeled and non-modeled faults and the possibility for performing at-speed test. It is for these (and other) reasons that BIST schemes seem to have been well accepted by the IC design and manufacturing community [2].

The BIST circuitry of a combinational CUT comprises a test generator (TG) that generates the required test patterns (and applies them to the CUT inputs) and a Response Verifier (RV) that compresses the CUT responses and evaluates the compressed output. Pseudorandom BIST schemes utilize sequences generated by modules that can be found on-chip (e.g. counters or accumulators) or can be easily implemented by modifying existing registers (e.g. Linear Feedback Shift Registers, LFSRs [3]). The main advantages of the pseudorandom testing paradigm include the low hardware overhead and the simplicity of the BIST control. Counters have been successfully utilized as BIST pattern generators in the BIST context [5], [8], since these modules commonly exist into current devices.

Among the pseudorandom schemes that have been proposed, the test set embedding paradigm that tries to embed a given test set into the sequence generated by a pseudorandom generator has gained attention by researchers [4], [5], [6], [7]. A test set embedding scheme typically utilizes a test set T and a hardware generator G and tries to embed T into the sequence generated by G, in such a way that the length of the selected subsequence is minimized. The advantage of test set embedding stems from the fact that the pseudorandom generator used has moderate or low hardware overhead.

Firstly, Lempel et al, utilized theory of discrete logarithms to embed test patterns into LFSR-generated sequences [4]. Kagaris and Tragoudas utilized counters, and by using negation and permutation of the counter outputs succeeded to generate complete test sets into the thereof generated sequences within acceptable time limits [5], [8]. Dorsch [6] proposed an algorithm for embedding test patterns, and, as a next step, whole test sets into sequences generated by accumulators. In a recent work [9], an algorithm was presented that, given an n-stage accumulator accumulating a constant pattern B, calculates in O(n) time, the location of any given pattern V into the generated sequence.

In this paper we propose a novel algorithm that, given a test set containing don't care (X) values, calculates a minimum sequence generated by a binary counter that generates all the test patterns of the test set. Therefore, the power generated for the testing of the module is deteriorated.

In order to provide the framework in which the proposed algorithm operates, we shall present the typical procedure one would follow in order to generate a test set for a given circuit. At first, the description of the CUT is entered into an Automatic Test Pattern Generation (ATPG) tool e.g. [12]. The ATPG tool generates test patterns (usually containing don't care values). Next, the test set is compressed, in an attempt to decrease the number of patterns. In effect, some of the 'X' values are replaced with 1's or 0's (some times arbitrarily) without necessarily increasing the fault coverage. The output of the tool is thus a test set containing 0's and 1's.

The above-mentioned procedure is viewed differently from the BIST designer viewpoint. BIST engineers, especially those targeting the problem of embedding test patterns, firstly simulate

MOBIMEDIA 2007, August 27-29, Nafpaktos, Greece

DOI 10.4108/ICST.MOBIMEDIA2007.1890

pseudorandom patterns in portions of a predetermined number e.g., 32K patterns. When the simulation comes to a point where a few portions (e.g. 3 consecutive applications of 32K patterns) do not detect any new faults, the remaining faults are labeled as 'non-randomly testable' or so-called 'hard-to-detect' faults [8]. It is for these faults that test patterns (containing don't cares) are generated and test embedding is applied. The rationale of this approach is that if hard-to-detect faults are detected with a pseudorandom sequence of adequate length, then pseudo-randomly testable faults will be also detected. Experimental results indicate that this approach gives fruitful results, e.g. in [4], [5], [6], [8].

The exploitation of don't care values drives down the number of required patterns as is shown experimentally in the sequel.

In this paper we present a novel algorithm for Embedding Test sets containing don't care values (TEX) into counter-generated sequences that, given a test set containing vectors with don't care bits, calculates an optimal subsequence of the sequence generated by a binary counter that contains the required test set.

The paper is organized as follows. In Section 2 some definitions and Lemmas are provided, that are utilized in Section 3 where the proposed algorithm is presented, exemplified and analyzed; in Section 4 experimental results of the application of the proposed algorithm into some test sets are presented. We conclude the paper in Section 5.

2. DEFINITIONS, NOTATIONS AND LEMMAS

We denote by L(V) the location of an n-bit pattern V in the sequence generated by an n-stage counter starting from the all-zero value, i.e. the number of cycles that the counter needs to operate in order to generate V. For example, for n=3, the location of V=6 in the generated sequence is L(6) = 6. Furthermore, the *distance* of two patterns V1 and V2 in the sequence generated by a counter is the difference L(V1) - L(V2).

Definition 1: Consider an n-bit vector V containing k don't care bits, 0<=k<=n. The **expanded list** of V is a list of 2^k vectors, each one of consisting only of '0' and '1', in such a way that all 2^k combinations of '0' and '1' appear into all positions held by 'X' in the vector V.

From Definition 1, if V has k don't care bits, then there are exactly 2^k vectors in its expanded list. For example, let n = 5 and V=01X1X. Then k = 2 and the expanded list of V consists of the $2^k = 4$ binary vectors: {01010, 01011, 01110, 01111}.

Definition 2: Consider a test set T comprising of t test vectors containing don't care values and the t expanded lists L_1, L_2, …, L_t of the t test vectors V_1, V_2, …, V_t respectively. Let T_i be a vector belonging to one (or more) of the expanded lists of V_i, 1≤i≤t. The including list of a vector Ti in the test set is a list of the including lists this vector belongs to.

For example, let us consider the test set consisting of 9 5-bit test vectors, given in Figure 1(a). The expanded lists of the test vectors of the test set are presented in Figure 1(b). The decimal notation of each vector is presented in Figure 1(c) for reference purposes. Then, the including list of V=3 is {L7, L9}, while the including list of V=19 is {L3, L7, L9}. In Figure 2 the including lists of all the test vectors of the test set presented in Figure 1 are presented.

V1 = 101XX
V2 = X10XX
V3 = 100XX
V4 = 0111X
V5 = X10X0
V6 = X00X0
V7 = XX011
V8 = XX101
V9 = X00X1

(a)

L1 = {10100, 10101, 10110, 10111}
L2 = {01000, 01001, 01010, 01011, 11000, 11001, 11010, 11011}
L3 = {10000, 10001, 10010, 10011}
L4 = {01110, 01111}
L5 = {01000, 01010, 11000, 11010}
L6 = {00000, 00010, 10000, 10010}
L7 = {00011, 01011, 10011, 11011}
L8 = {00101, 01101, 10101, 11101}
L9 = {00001, 00011, 10001, 10011}

(b)

L1 = {20, 21, 22, 23}
L2 = {8, 9, 10, 11, 24, 25, 26, 27}
L3 = {16, 17, 18, 19}
L4 = {14, 15}
L5 = {8, 10, 24, 26}
L6 = {0, 2, 16, 18}
L7 = {3, 11, 19, 27}
L8 = {5, 13, 21, 29}
L9 = {1, 3, 17, 19}

(c)

Figure 1. (a) Test set for the c17 benchmark (b) corresponding expanded lists (c) decimal notation

V	Including list of V	V	Including list of V
0	{L6}		
1	{L9}	17	{L3, L9}
2	{L6}	18	{L3, L6}
3	{L7, L9}	19	{L3, L7, L9}
5	{L8}	20	{L1}
8	{L2, L5}	21	{L1, L8}
9	{L2}	22	{L1}
10	{L2, L5}	23	{L1}
11	{L2, L7}	24	{L2, L5}
13	{L8}	25	{L2}
14	{L4}	26	{L2, L5}
15	{L4}	27	{L2, L7}
16	{L3, L6}	29	{L8}

Figure 2. Including lists of the vectors of the expanded lists of Figure 1(b)

Definition 3: Consider a test set T with t vectors T_i, 1≤i≤t, containing don't care values. The covering list of the test set T is a list, each element of which is a vector belonging to one (or more) of the expanding lists of the vectors T_i that belong to T, (in ascending order) together with the including list of each test vector.

For example, the covering list for the test set under consideration is presented in Figure 3.

L = { 0 {L6}
1 {L9}
2 {L6}
3 {L7, L9}
5 {L8}
8 {L2, L5}
9 {L2}
10 {L2, L5}
11 {L2, L7}
13 {L8}
14 {L4}
15 {L4}
16 {L3, L6}
17 {L3, L9}
18 {L3, L6}
19 {L3, L7, L9}
20 {L1}
21 {L1, L8}
22 {L1}
23 {L1}
24 {L2, L5}
25 {L2}
26 {L2, L5}
27 {L2, L7}
29 {L8}
}

Figure 3. Covering List for the test set of Figure 1

In the sequel we will say that the pair of vectors V_1, V_2, $V_1<V_2$ ***covers*** list L1 if L1 belongs in the including list of any vector V_i for $V_1 \leq V_i \leq V_2$.

In the sequel, we shall say that a pair of vectors V1, V2, covers the test set T if it covers the lists of all vectors of T. For example, the pair of vectors (14, 25) covers the test set T, since it covers all lists L1 ..L9 of the test set.

Definition 4: We will say that a pair of vectors (V1, V2) *minimally covers* the test set T if it covers the test and the difference V2-V1 is the minimum among the pairs that cover the test set.

For example, the pair of vectors (15,24) minimally covers the test set under consideration. Indeed, it is easy to see that it covers the test set T. Furthermore, the distance 24-15=9 is the minimum among the pairs that cover the test set.

Lemma 1: If the pair of vectors (V1, V2) covers the test set, then all pairs (V1, V3) for V3 > V2 cover the test set.

Proof: Since all lists are covered by V1, V2, and V3>V2, all lists are covered by (V1, V3) **Q.E.D**.

Lemma 2: Let V_t be the last (larger in value) vector in the expanded lists of the test set and V1 a vector in the expanded list. Then, if the pair (V1, Vt) does not cover the test set, no pair (V2, V3) for $V1 \leq V2 \leq V3$ covers the test set.

Proof: If (V_1, V_t) does not cover the test set, then there is (at least) one list Li that is not covered by the pair (V1, Vt). This list cannot be covered by the pair (V2, V3), since V2≥V1. Q.E.D.

For example, in Table I, the pair (16,29) does not cover the test set. Therefore, none of the pairs whose first vector is V2, with V2 >=16 can cover the test set. These "hopeless" pairs are presented in the Table 1.

3. THE PROPOSED ALGORITHM

Given the definitions of the previous Section, the proposed TEX algorithm operates in 3 steps as outlined below:

Algorithm TEX (T, t, n)
//T is a test set, containing t test vectors Ti, ($1 \leq i \leq t$). Each vector in the test set contains n-tuples of '1', '0' and 'X'.

Step 1. Construct the expanded list L_i for every test vector T_i in T.

Step 2. Construct CL, the covering list of the test set.

Step 3. For all pairs of numbers (Vi, Vj) in CL, calculate the lists covered by the pair (Vi, Vj). Calculate the distance of all pairs covering the test set T. Among the pairs that cover the test set, the pair having the smallest distance is chosen as the best solution for the considered generator.

The following Observations can be used to drastically reduce the time required by the TEX algorithm. For a justification of Observation 1, see Lemma 1; for a justification of Observation 2, see Lemma 2.

Observation 1: If pair (Vi, Vj) generates all lists, there is no point in examining pairs (V_i, V_k) for k>j, since the distance will be greater.

For example, in Table I, the pair (13, 24) covers the test set; thus, there is no reason to examine pairs (13, 25), (13, 26) e.tc., since the respective distance (D) will be certainly greater. Taking into account Observation 1, the number of tried pairs in this example reduces by 71 (grey shaded vectors in Table I).

Observation 2: Let Vn be the greatest vector that belongs to the expanded lists of all vectors of the test set. If the pair (Vi, V_N) does not generate the test set, no pair (V_j, V_k) for $V_k >= V_j >= V_i$ will cover the test set.

For example, in Table I, since the pair (16, 29) (last try having 16 as the first vector) does not generate all lists, there is no point in trying pairs whose first vector is 17, 18, …. With Observation 2, the number of tried pairs is further reduced by another 78 pairs.

Therefore, thanks to observations (1) and (2) a total reduction of 71 + 78 = 149 / 325 ≈ 45% is achieved.

In order to perform an analysis of the proposed algorithm, we shall calculate the complexity of each step as follows.

Step 1. Requires the generation of the including list 1 of each vector of the test set. If t is the number of unspecified (don't care) bits of a test vector, the complexity of this step is of the order $O(2^t)$. Therefore, the complexity of the step is of the order

$$\sum_{i=1}^{t} 2^{k_i} \qquad (1)$$

Where k_i is the number of unspecified bits in T_i, 1<=i<=t.

Step 2. The complexity of this step is of the order O(nlogn) required by the insertion sort algorithm to insert the patterns into the sorted list.

Step 3. The complexity of this step is of the order $O(n^2)$, where n is the number of test patterns in the covering list. Since the number of test patterns depends on the number of the unspecified bits in the original test set, the complexity is upper bounded by $O((\sum_{i=1}^{t} 2^{k_i})^2)$. The upper bound lies on the fact that some (if not many) vectors are contained in the lists of more than one vector, therefore the number of patterns in this step is (much) less than the one calculated by (1).

4. SIMULATIONS AND EXPERIMENTS

In order to investigate the applicability of the proposed algorithm, we have applied it into randomly-generated test sets for various values of n, the number of bits and for various values of the number of X, the don't care values. The utilized test sets are presented in the Appendix.

In Table 2 we present the results of the conducted experiments. Initially we generated randomly test patterns containing unspecified bits of various values of n and computed the length of the sequence required by a binary counter to generate the test patterns. Then we filled the unspecified values with 0 and 1, in three ways: all X-values were substituted by 0 (column denoted 'All-0') or 1(column denoted 'All-1'), or the unspecified bits were filled randomly (column denoted 'Random'). We calculated the length of the sequence required to generate the resulting test set in each case.

In Table 2 in the first column we present the value of the number of the bits (n). In the second column we present the length of the sequence calculated by the proposed scheme. In the next columns we present the length of the sequence required to generate the patterns resulting after substituting 'X' with either '0' (second column), '1' (third column) or randomly '1' or '0' (fourth column), as well as the increase in the number of required cycles. From Table 2 we can conclude that not only the proposed scheme always results in a shorter test sequence than all the other sequences (as expected), but also this reduction increases as the number of bits increases.

It should be noted that the presented results come as a straightforward application of the proposed scheme into the sequence generated by a simple binary counter. However, the proposed algorithm can be applied to a wide variety of BIST generators, e.g. counter with shifted outputs, accumulators accumulating a constant value, or even the -ubiquitous- LFSRs.

5. CONCLUSIONS

The problem of reliability of current electronic systems has been the subject of various researchers, e.g. [1]. Mobile devices call for an unprecedented need for low power during both normal operation and testing. Reducing the length of the required test patterns reduces both test time and consumed power. Test set embedding schemes [4]-[9] utilize generators that either exist or can be easily implemented on-chip and try to embed complete test sets within a subsequence (as short as possible) of the sequence generated by the utilized hardware. Test sets extracted from well-known vendor tools contain don't-care (i.e. 'X') values which can be set to '0' or '1' in order to drive down the number of test patterns

In this paper, a novel algorithm has been presented that can be utilized in order to embed test sets containing don't care (X) values into sequences generated by binary counters. Conducted experiments reveal that the proposed scheme results in considerably shorter sequences compared to filling the X values with all-'0', all-'1' or random filling. Therefore, both test time and consumed power can be considerably reduced. The proposed scheme may prove useful for the testing of mobile devices.

6. REFERENCES

[1] S. Zezza, M. Grangetto, M. Martina, F. Vacca, G. Masera, "Error Correcting Arithmetic Coding for JPEG 2000: Memory and Performance Analysis", MobiMedia 2006, 2nd International Mobile Multimedia Communications Conference September 18-20, 2006, Alghero, Sardinia, Italy.

[2] Abramovici M., Breuer M., Freidman A., "Digital Systems Testing and Testable Design", Computer science Press, 1990.

[3] Dufaza C., Gambon G., "LFSR-based Deterministic and Pseudorandom Test Pattern Generator Structures", Proc. European Test Conference, pp. 27-34, 1991.

[4] Lempel M., Gupta S., Breuer A., "Test Embedding with Discrete Logarithms", IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no 5, May 1995.

[5] Kagaris D., Tragoudas S., "On the Design of Optimal Counter-based Schemes for test set embedding", IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no.2, February 1999.

[6] Dorsch R., Wunderlich H., "Accumulator-Based Deterministic BIST", International Test Conference, pp. 412-421, 1998.

[7] Boubezari S., Kaminska B., "A Deterministic BIST Generator Based on Cellular Automata Structures", IEEE Trans. Computers, vol. 44, no 6, June 1995.

[8] Kagaris D., Tragoudas S. Majumdar A., "On the use of Counters for reproducing Deterministic Test Sets", IEEE Transactions on Computers, vol. 45, no. 12, December 1996.

[9] Voyiatzis I., "Test vector embedding into Accumulator generated sequences: a linear time solution", IEEE Transactions on Computers, April 2005.

[10] D. Coppersmith, "Fast Evaluation of Logarithms in Fields of Characteristic Two", IEEE Trans. Inf. Theory, July 1984, pp. 587-594.

[11] Pohligg St., Hellman M., "An Improved Algorithm for computing Logarithms over GF(p) and its Cryptographic Significance", IEEE Trans. Inf. Theory, Jan 1978, pp. 106-110.

[12] TestGen, version TG3.0.2 User Guide, Synopsys Inc., 1999.

Table 1. Hopeless vectors for the example test set

17,17											
17,18	18,18										
17,19	18,19	19,19									
17,20	18,20	19,20	20,20								
17,21	18,21	19,21	20,21	21,21							
17,22	18,22	19,22	20,22	21,22	22,22						
17,23	18,23	19,23	20,23	21,23	22,23	23,23					
17,24	18,24	19,24	20,24	21,24	22,24	23,24	24,24				
17,25	18,25	19,25	20,25	21,25	22,25	23,25	24,25	25,25			
17,26	18,26	19,26	20,26	21,26	22,26	23,26	24,26	25,26	26,26		
17,27	18,27	19,27	20,27	21,27	22,27	23,27	24,27	25,27	26,27	27,27	
17,29	18,29	19,29	20,29	21,29	22,29	23,29	24,29	25,29	26,29	27,29	29,29

Table 2. Considered test sets and results

			All-0		All-1		Random	
	Test set	**Initial #cycles**	**#Cycles**	**Increase**	**#Cycles**	**Increase**	**#Cycles**	**Increase**
n=10	#1	642	931	**45%**	704	**10%**	753	**17%**
	#2	332	824	**148%**	511	**54%**	442	**33%**
	#3	459	666	**45%**	638	**39%**	563	**23%**
n=12	#4	1834	1982	**8%**	2552	**39%**	2378	**30%**
	#5	677	2358	**248%**	1708	**152%**	1893	**180%**
	#6	1987	3079	**55%**	2609	**31%**	2705	**36%**
n=15	#7	15893	24635	**55%**	20867	**31%**	29223	**84%**
	#8	10774	24394	**126%**	17864	**66%**	25942	**141%**
	#9	3222	22599	**601%**	10688	**232%**	23073	**616%**
			Average	**148%**		**73%**		**129%**

Appendix: Test sets utilized in our experiments

#test set	test vectors	#test set	test vectors	#test set	test vectors
#1	0100xxxx1x	#4	011x01001x1x	#7	1xx0111x0x1x010
	xx11x11x0x		1x1x11xx0xx0		1xx01x1x0xx011x
	1xxx0xx111		1xxx0x1x11xx		0x001x000x10xx1
	xxxxx11110		011x11xx101x		11x01x0010x110x
	0xx1x11x1x		0101x11x1x10		1xx001x111x11xx
	1111xxxxx1		1101xx1x10x0		0x11x11x1xx0x11
#2	xxxx01001x	#5	100101001x1x	#8	1xx0111x0x1x010
	xx1xx1100x		1x1x1x1x0xx0		xxx0xx1x0xx0x11
	1xxx0x1111		1xxx100111xx		0xx01x000x10xx1
	x11xxx1110		x111xx11101x		11x0xx0010x1101
	0xx1x11x1x		x001x11x1x10		1xxx01x111x11xx
	1101xx1x10		100xxx1x10x0		0x11x11x1xx0x11
#3	x11x01001x	#6	1xx0111x0x1x	#9	x0x0111x0x1x010
	xx1x11xx0x		1xx01x1x0xx0		xx10x01x0x10x11
	1xxx0x1x11		0x001x000x10		x0x01x0001x0x11
	011x11xx10		11x01x0010x1		1x00x10x10x1101
	0101x11x1x		1xx001x111x1		xxxx01x111111xx
	1101xx1x10		0x11x11x1xx0		1x11x11x1xx0x11