Efficient Algorithm for Test Vector Decompression Using an Embedded Processor

Kamran Saleem and Nur A. Touba

Computer Engineering Research Center
Department of Electrical and Computer Engineering
University of Texas, Austin, TX 78712
{ksaleem@utexas.edu, touba@ece.utexas.edu}

Abstract

A new algorithm for implementing test vector compression in software is presented. It is based on a compression procedure recently invented by Melhem, et al., called RDIS (recursively defined invertible set) [Melham 12]. While RDIS was originally proposed for a memory correction application, it is shown here to be very well suited for the problem of compressing test vectors. When the inputs that are unassigned during ATPG are left as don't cares (forming "test cubes"), typically only 1-2% of the remaining bits are care bits. It is shown that the RDIS procedure is very efficient when the number of care bits is small and is able to achieve large amounts of compression. The RDIS procedure works by recursively constructing invertible sets and efficiently encoding the information with row and column counts. The compressed data is stored on the tester. The tester then transfers this data to an embedded processor's memory, and a simple decompression program is executed on the embedded processor which uses this information to reconstruct the original test cubes which can then be used for testing a chip, board, or system. Experimental results are shown which indicate that significant amounts of compression can be achieved independent of the dimensions of the test cube matrix.

1. Introduction

One major challenge for testing complex systems is dealing with the enormous amount of test data required. Large test data volume results in long test application time because the data has to be transferred across the low test data bandwidth link between the tester and the device under test (DUT). Moreover, large amounts of memory on the tester are required to store the test data. One approach for dealing with this problem is to use test data compression techniques to reduce the amount of data that is stored on the tester [Touba 06]. Both the test vectors and the output response can be compressed. Major CAD vendors offer test compression schemes which are implemented in hardware and can be inserted in scan designs. In systems that contain an embedded processor, it is also possible to implement test compression in software. In this case, the tester would transfer test data to the memory of an embedded processor, and a software procedure would be executed on the embedded processor to decompress the test data. Such a scheme can be utilized to decompress test data for testing a system-on-chip design, a board-level design, or a whole system.

A number of techniques for implementing built-in self-test (BIST) in software running on an embedded processor have been proposed (e.g., [Batcher 99], [Chen 01], [Hwang 02], [Kranitis 02], [Zhou 06], and others). These techniques are based on applying pseudo-random patterns to the DUT which has drawbacks including difficulty achieving high fault coverage due to random pattern resistant faults and the need for a BIST ready design (that will not produce any non-deterministic outputs). These issues can be avoided by using test compression instead of BIST which involves decompressing and applying fully deterministic test vectors. Earlier work in implementing test compression in software running on an embedded processor has included using Huffman codes [Jas 02], Matrix codes [Balakrishnan 02], and linear coding [Balakrishnan 03].

This paper presents a new algorithm for implementing test vector compression in software. It is based on a compression procedure recently invented by Melhem, et al., called RDIS (Recursively Defined Invertible Set) [Melhem 12]. While RDIS was originally proposed for a memory correction application, it is shown here to be very well suited for the problem of compressing test vectors. When the inputs that are unassigned during ATPG are left as don't cares (forming "test cubes"), typically only 1-2% of the remaining bits are care bits. It is shown that the RDIS procedure is very efficient when the number of care bits are small and is able to achieve large amounts of compression. The RDIS procedure works by recursively constructing invertible sets and efficiently encoding the information with row and column counts. This is supplemented with pointer breaks to handle alternating loops. The compressed data (consisting of the column and row counts and pointer break addresses) is stored on the tester. The tester then transfers this data to the embedded processor's memory, and a decompression program is executed on the embedded processor which uses this information to reconstruct the original test cubes which can then be used for testing a chip, board, or system. The amount of compression achieved depends on the nature of the test vectors and varies from design to design. Experimental results are shown for different dimension test matrices. The proposed scheme can be used to reduce tester storage requirements and test time without the need for adding extra hardware overhead as the test decompression is all handled in software.

The paper is organized as follows: Sec. 2 describes how the RDIS compression process works to encode a test cube matrix. Sec. 3 describes the decompression process. Sec. 4 explains the need for pointer breaks and gives a greedy

978-1-4799-3005-0/14/$31.00 ©2014 IEEE

algorithm for minimizing the number of pointer breaks. Sec. 5 presents the experimental results, and Sec. 6 is a conclusion.

2. RDIS Compression Algorithm

The proposed methodology is based on a new compression algorithm proposed in [Melhem 12] called RDIS. RDIS works on a given matrix of bits each of which can have three states: *1*, *0*, and *X* (don't care). It can be applied directly on test cubes which typically have only 1-2% of specified bits with the rest being don't cares. For test cubes, the matrix can be formed with each row corresponding to a test cube. The goal of RDIS is to have the bits divided into two disjoint sets *S* and *C* containing specified bits having value *1* and specified bits having value *0,* respectively, with the don't care bits, *X*'s, assigned to each set as needed. These sets are encoded in a clever way by column and row counts which indicate how many times a column/row needs to be inverted during decompression. These column and row counts are all that need to be stored. Using the column and row counts, the test cubes can be regenerated with a simple software decompression program. The size of the column and row counts is much smaller than storing the full test cubes themselves thereby providing significant compression of the test data.

The RDIS algorithm constructs the *S* and *C* sets as the union of progressively smaller subsets. The algorithm first constructs an initial set S_1 of all rows and columns containing *1*'s. All the other rows and columns which contain *0's*, but not *1*'s, form the initial set C_1. While C_1 doesn't contain any *1*'s, S_1 still has *0*'s in addition to *1*'s. Since C_1 contains purely *0*'s and the matrix is initialized to *0* during decompression, there is no need to ever invert these rows/columns, so their count is left at *0*. These rows/columns need no further processing and can be treated as don't cares henceforth by the algorithm. On the other hand, S_1 contains both *1*'s and *0*'s, so it needs to be further processed, so the counters for the rows and columns contained in S_1 are all incremented by 1 and their values are inverted (*0*'s become *1*'s and *1*'s become *0*'s). The algorithm then recursively repeats the same process treating S_1 as the matrix to be processed. All the rows/columns in C_1 are effectively "extracted" from the initial matrix leaving only the rows/columns in S_1 to be further processed.

In the second iteration of the algorithm, all the rows and columns (in S_1 after inversion) that contain *0*'s, but not *1*'s form the set C_2, and all the remaining rows and columns form the set S_2. C_2 is extracted, and only the counters for the rows/columns in S_2 are incremented. The algorithm then recursively repeats this same process on S_2. Each time the algorithm iterates, additional rows/columns are extracted until ultimately only one type of specified bit is left (i.e., either all *1*'s or all *0*'s). At this point, all specified bits have been separated out. The sets *S* and *C* can then be calculated as the unions of the invertible sets extracted at each step:

$S = C_2 \cup C_4 \cup C_6 \ldots$

$C = C_1 \cup C_3 \cup C_5 \ldots$

In the odd steps, *0*'s in the original matrix are extracted out, and in the even steps, *1*'s in the original matrix are extracted out (since an odd number of inversions have been applied to the matrix). The final row and column counter values are all that are needed to decompress the test cubes. This will be discussed further in the next section.

To illustrate the RDIS encoding procedure, a small example is shown in Figs. 1-6. Fig. 1 shows the initial matrix with the *1*'s shown with dark circles and *0*'s shown with light circles. Locations without circles are don't cares. In Fig. 2, the row/column counters containing *1*'s are incremented. This causes set C_1 to be extracted in the form of rows and columns whose counters are 0. The specified bits that lie on those rows/columns don't contribute to the counter values and hence can be removed. As can be seen, five columns and no rows are included in C_1. Those are the only rows/columns that have no *1*'s in them.

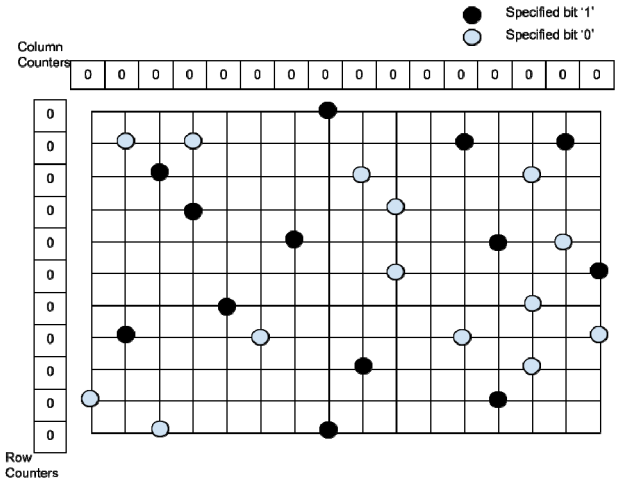

Figure 1. Initial Spread of Specified bits in Matrix *M*

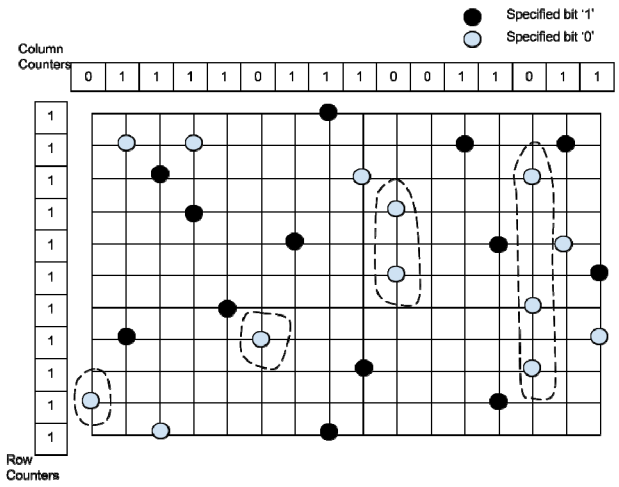

Figure 2. Columns corresponding to the dotted sets form C_1

Fig. 3 represents the matrix with its specified bits inverted. Again the row/column counters containing *1*'s are incremented. Set C_2 is extracted in the form of rows and columns with counter value 1 (that were not incremented in this step). Five columns and six rows are included in C_2. This set contains only the original *1*'s hence it forms part of *S*

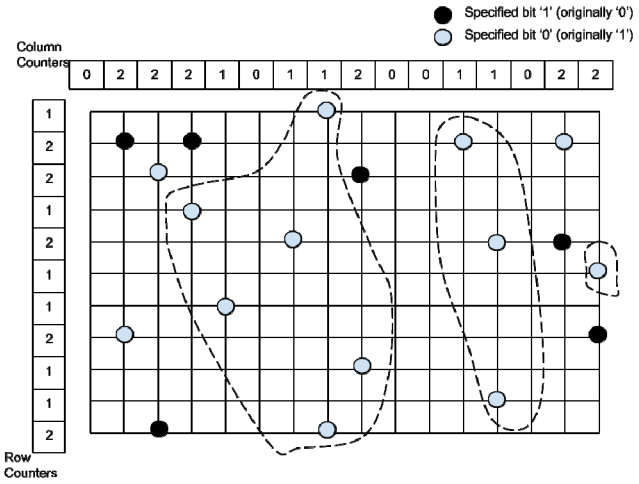

Figure 3. Rows/Columns corresponding to C_2

In the next iteration of the algorithm, C_3 with three columns and two rows is extracted out as shown in Fig. 4. Further iterations of the algorithm result in sets C_4 and C_5 as illustrated by the dotted sets of specified bits corresponding to those rows/columns in Fig. 5 and Fig. 6. The final counter values shown in Fig. 6 are all that are needed to regenerate the matrix as will be explained in the next section.

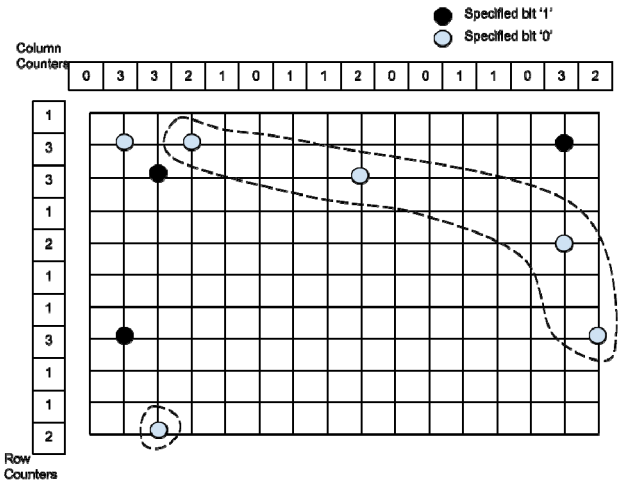

Figure 4. Rows/Columns corresponding to C_3

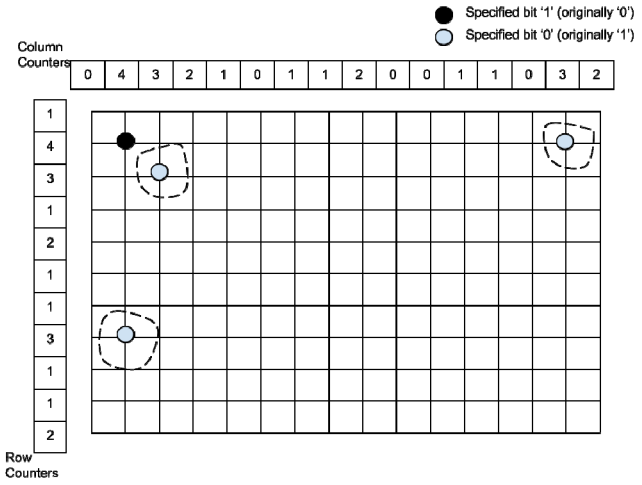

Figure 5. Rows/Columns corresponding to C_4

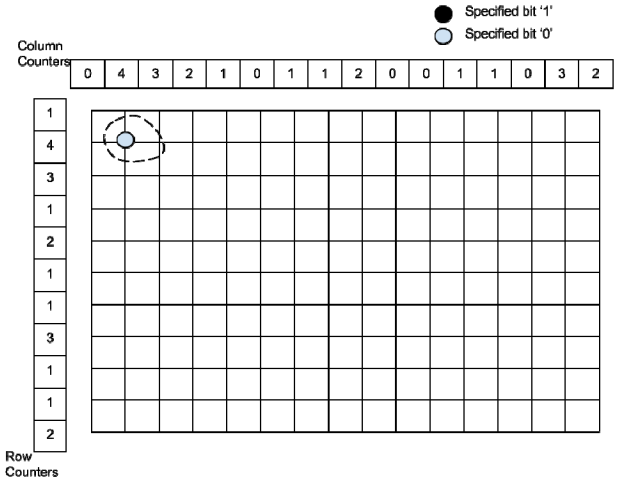

Figure 6. Rows/Columns corresponding to C_5

3. Decompression

As described in the previous section, at each step *i* of the algorithm, the counters for all rows/columns in the set S_i are incremented. So the final counter value for each row/column indicates how many steps passed before it was "extracted" out.

If the algorithm ends in *k* steps, the maximum counter value will be *k*. Thus for *m* rows and *n* columns, the total column metadata will be $(m+n) \times (\text{bits required for each counter}) = (m+n) \times \lceil \log_2 k \rceil$. This is the amount of data that needs to be stored after compressing the test cubes.

The specified bits are inverted at each step until the point of extraction. Starting from a default value of *0*, the decompression mechanism can determine the correct value of any specified bit if it has the information about the number of times that bit has been inverted. It can be observed that all the *1's* are extracted in even steps (C_2, C_4, C_6 , ..) when they don't affect the counter values. This implies that at least one of the

counters corresponding to the *1* was not incremented in the even step and hence it must be odd. The other counter value may or may not be incremented in the even step depending on the presence of a *0* in that row/column. Thus any specified bit is a *1* if the lower of its row/column counter is odd, otherwise it is a *0*.

Thus the steps involved in decompression are first to transfer the row and column metadata from the tester to the embedded processor. Then for each bit of the test cube matrix, its corresponding row and column counter values are compared. If the lower of the row or column counter is odd the bit is set to *1* otherwise it is set to *0*. This process is repeated until the whole test cube matrix is decompressed. This can be done with a very simple software procedure. Once the test cube matrix is decompressed, the test can be applied to test on-chip logic, board-level test, or system-level tests.

4. Dealing with Pointer Breaks

While it was mentioned that the algorithm terminates when a singleton set (all *1s* or all *0s*) is reached, this may not be possible in all cases. Specifically if the given matrix has a loop of alternating *1s* and *0s*, then the algorithm will get stuck in a set which cannot be reduced further. An example of this is shown in Fig. 7. Bits $M(i_1,j_1)$, $M(i_2,j_1)$, $M(i_2,j_2)$, $M(i_3,j_3)$,……, $M(i_q,j_q)$, $M(i_1,j_q)$ are said to be alternatively-stuck if $M(i_1,j_1)$, $M(i_2,j_2)$, …, $M(i_q,j_q)$ have value *1* while bit $M(i_2,j_1)$, $M(i_3,j_2)$, …., $M(i_1,j_q)$ have value *0* and vice-versa. To remedy the problem, [Melhem 12] proposed using pointer breaks which store the addresses of some bits that need to be loaded directly instead of using the RDIS decompression procedure. Since these bits are loaded directly, they can be considered as don't cares when running the RDIS procedure. By carefully selecting pointer break locations in the test cube matrix to break the alternating loops, all the remaining specified bits in the test cube matrix can be successfully encoded with the RDIS procedure. The alternating loop shown in Fig. 7 can be broken by using a pointer break to directly load one of the 4 specified bits in the loop. The amount of metadata needed to store the addresses of the pointer breaks depends on the size of the given matrix, e.g. for a 40 x 700 matrix, ($\log_2 40 = 6$) + ($\log_2 700 = 10$) = 16 bits will be needed for every pointer break bit. Hence the number of pointer break bits should be minimized as much as possible when breaking all the alternatively-stuck loops in the matrix.

A greedy procedure can be used to minimize the number of pointer breaks. RDIS encoding is first performed to identify which bits cannot be encoded. These bits are considered as candidate locations for inserting pointer breaks. All loops of alternating *0*'s and *1*'s need to be broken, but note that some bit locations lie in the intersection of multiple such loops. The proposed greedy procedure evaluates the candidate locations and identifies the one that will have maximum impact in increasing the number of bits that can be encoded. A pointer break is then inserted at that location, and then the procedure is iteratively repeated. Each time a new pointer break is inserted, it monotonically reduces the number of bits that cannot be encoded. Thus, pointer breaks continue to be inserted in this greedy manner until all remaining bits can be encoded.

The addresses of the pointer breaks are stored as part of the metadata, and the decompression procedure simply loads the values of the pointer break locations directly. All other locations are loaded using the counter values as described in Sec. 3.

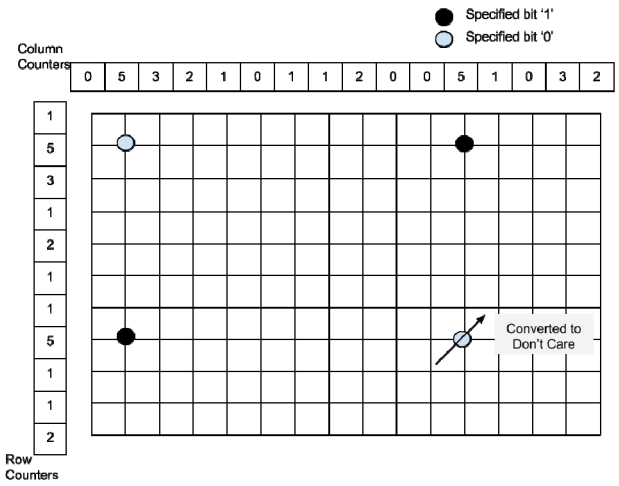

Figure 7. Pointer Break Example

5. Experimental Setup and Results

Experiments were performed to evaluate the effectiveness of the proposed test data compression scheme. Random test cube matrices of different proportions were generated with1% and 2% specified bits to emulate the typical characteristic of test data seen in industrial circuits. For successful compression, the proposed scheme should be able to decompress all the specified bits correctly. In the experiments, it was observed that the need for pointer breaks was common in order to allow all remaining bits to be successfully encoded with the RDIS algorithm.

Experimental results are presented in Table 1. The first column gives the size of the random test cube matrix. Different proportions have been used. The first has 1000 test cubes with 200 bits each. The second has 200 test cubes with 1000 bits each, and the third has 500 test cubes with 500 bits each. The second column shows the percentage of specified bits. The third column is the amount of uncompressed data which is simply the total number of bits in the matrix. The next three columns show the amount of data required for the proposed RDIS compression procedure. The amount of metadata needed for storing the counter values is shown first, followed by the amount of data for the addresses of the pointer breaks, and finally the total data which is the sum of the first two. The last column shows the compression ratio comparing the total compressed data for the proposed method with the uncompressed data.

Table 1. Results for Proposed Test Compression Methodology

Test Cube Matrix	Percent Specified Bits	Uncompressed Data (Number of Bits)	Compressed Data for Proposed RDIS Procedure			Compression Ratio
			Counter Data	Pointer Data	Total Data	
1000 x 200	1%	200,000	5,100	3,306	8,406	23.8 x
	2%		6,150	19,779	25,929	7.7 x
200 x 1000	1%	200,000	5,190	2,793	7,983	25.1 x
	2%		5,360	19,304	24,664	8.1 x
500 x 500	1%	250,000	5,275	4,693	9,968	25.1 x
	2%		6,136	31,293	37,429	6.7 x

As can be seen from the results, significant amounts of compression can be achieved (up to 25.1x) with a very simple software based decompression procedure. The total amount of compression obviously is highly dependent on the percentage of specified bits, but surprisingly, it is relatively independent of the matrix dimensions. This suggests that the methodology is effective regardless of the test cube matrix dimensions.

6. Conclusion

The proposed software-based test compression scheme, based on the RDIS algorithm, is able to achieve significant amounts of compression. It is effective regardless of the dimensions of the test cube matrix. The advantage of this approach is that it requires only a very simple decompression procedure with no hardware support required (assuming an embedded processor is already available as part of the functional design), and yet achieves significant compression which helps to reduce tester equipment costs and reduce test time.

Acknowledgements

This research is supported in part by the National Science Foundation under Grant No. CCF-1217750.

References

[Batcher 99] K. Batcher and C. Papachristou, "Instruction Randomization Self Test for Processor Cores," *Proc. of VLSI Test Symposium*, pp. 34-40, 1999.

[Balakrishnan 02] K.J. Balakrishnan and N.A. Touba, "Matrix Based Deterministic Test Vector Decompression Using an Embedded Processor," *Proc. of Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT)*, pp. 159-165, 2002.

[Balakrishnan 03] K.J. Balakrishnan and N.A. Touba, "Deterministic Test Vector Decompression in Software Using Linear Operations," *Proc. of VLSI Test Symposium*, pp. 225-231, 2003.

[Chen 01] L. Chen and S. Dey, "Software-based Self-Testing Methodology for Processor Cores," *IEEE Trans. on Computer-Aided Design*, Vol. 20, Iss. 3, pp. 369-380, Mar. 2001.

[Hwang 02] S. Hwang and J.A. Abraham, "Optimal BIST Using an Embedded Microprocessor," *Proc. of International Test Conference*, pp. 736-745, 2002.

[Jas 02] A. Jas, and N.A. Touba, "Deterministic Test Vector Compression/Decompression for Systems-on-a-Chip Using an Embedded Processor," *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 18, Issue 4/5, pp. 503-514, Aug. 2002.

[Kranitis 02] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective Software Self-Test Methodology for Processor Cores," *Proc. of Design, Automation, and Test in Europe Conf.*, pp. 592-597, 2002.

[Melhem 12] R. Melhem, R. Maddah, and S. Cho, "RDIS: A Recursively Defined Invertible Set Scheme to Tolerate Multiple Stuck-at Faults in Resistive Memory," *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, 2012.

[Touba 06] N.A. Touba, "Survey of Test Vector Compression Techniques," *IEEE Design & Test Magazine,* Vol. 23, Issue 4, pp. 294-303, Jul. 2006.

[Zhou 06] J. Zhou and H. Wunderlich, "Software-Based Self-Test of Processors under Power Constraints," *Proc. of Design, Automation and Test in Europe (DATE) Conf.*, 2006.