A Technique for High Ratio LZW Compression

Michael J. Knieser, Indiana University Purdue University Indianapolis, mknieser@iupui.edu
Francis G. Wolff, Case Western Reserve University, fxw12@po.cwru.edu
Chris A. Papachristou, Case Western Reserve University, cap2@po.cwru.edu
Daniel J. Weyer, Cisco Systems, dweyer@cisco.com
David R. McIntyre, Cleveland State University, mcintyre@cis.csuohio.edu

Abstract

Reduction of both the test suite size and the download time of test vectors is important in today's System-On-a-Chip designs. In this paper, a method for compressing the scan test patterns using the LZW algorithm is presented. This method leverages the large number of "Don't-Cares" in test vectors in order to improve the compression ratio significantly. The hardware decompression architecture presented here uses existing on-chip embedded memories. Tests using the ISCAS89 and the ITC99 benchmarks show that this method achieves high compression ratios.

1. Introduction

Testing Systems-on-a-Chip (SoC) devices or embedded Intellectual Property (IP) is dominated by two major factors: time and size [1]. The time is a function of the Automated Test Equipment (ATE) tester clock rate and the number of applied test patterns. Unfortunately, the clock rate is proportional to the price of the ATE. The number of test patterns is a function of the test insertion algorithm and the number of parallel test pattern scan chains. These lead to the size of the test patterns required for proper verification of the custom silicon device. This volume also impacts the ATE memory size and it's own price proportionally. Built-In Self Test (BIST) addresses the issue of test set volume by embedding test vectors on-chip. However, this reduction of test set volume by using BIST [2] may work well for embedded memory cores but, is not efficient for custom IP cores. Applying scan test vectors to chips is still the most preferred method.

1.1. Related Work

In order to apply test vectors to a SoC, data compression methods have been proposed which focus on efficiently transferring (i.e. downloading) test vectors from the ATE to the SoC. This can be broken down further into two cases: methods that require that the scan chain to have a particular architecture or physical layout [12, 19, 20], and those that are independent of the scan architecture. Some of these methods in the independent case can be classified into their analogous classical software data compression counterparts, some examples of which are: Run-Length-Encoding [10, 11], Huffman or statistical based coding [15, 16], and LZ77 [3, 8].

Many of these compression methods require careful assignment of "Don't-Care" bits with in test vectors to achieve practical compression ratios. The number of these bits is very high [9, 17, 18]. How these "Don't-Care" bits are assigned, given a compression scheme, is very critical. These "Don't-Care" bits need to be assigned in a way that favors the compression algorithm. For example, a Run-Length-Coding compression scheme may find that assigning the "Don't-Care" bits to form the longest string of 1's or 0's is best. Although a "Don't-Care" assignment may be best for a given compression scheme, the goal is still maximum compression of a test vector suite.

1.2. Our Work

In this paper, a method for compressing the scan test patterns using LZW [4, 21-23] that does not require the scan chain to have a particular architecture or layout is presented. This method leverages the large number of "Don't-Cares" in test vectors in order to improve the compression ratio significantly. An efficient hardware decompression architecture is also presented using existing in-chip embedded memories. In order to reduce chip area overhead, existing BIST-based embedded cores can be reused.

Some work has already been done in LZW architectures [24], but this implementation uses a complex memory structure and has high decode times for subpattern look-ups. Our method overcomes this limitation by bounding the maximum dictionary pattern size to the width of embedded memory word of the decompressor.

Section 2, gives a brief overview. In Section 3, the LZW compression method is described with a brief example. In Sections 4 and 5, decompression, and its implementation, is shown. Results on the ISCAS89 [5] and the ITC99 [6, 7] benchmarks show that this method achieves a high compression ratio in Section 6. Finally, Section 7 concludes the paper.

2. Overview

Figure 1 and Figure 2 show the LZW-based test vector compression architecture. Figure 1 shows how the LZW compressed vectors are developed. The embedded core

1530-1591/03 $17.00 © 2003 IEEE

test engineer generates his test vectors and then those results are passed on to the compression tool. The compression tool assigns the "Don't-Cares" and compresses the test vectors for the production tester. Figure 2 shows the embedded core, including a LZW decompressor, and how it interfaces to a tester. The compressed test vector file for the embedded core tester is then loaded. Since the tester is limited in speed, an external fast clock is provided to the embedded core under test to run the LZW decompression engine. Before the embedded core is fabricated, the production scan test chains, the production embedded core tester interface, any board level testing interfaces, memory testing logic and the LZW decompressor are all instantiated.

The LZW decompressor's input consists of a compressed scan input stream and the internal clock. This internal clock is used to drive the state machine of the LZW algorithm. It also drives the core's scan clock. It is assumed that this internal clock is faster than the tester's clock.

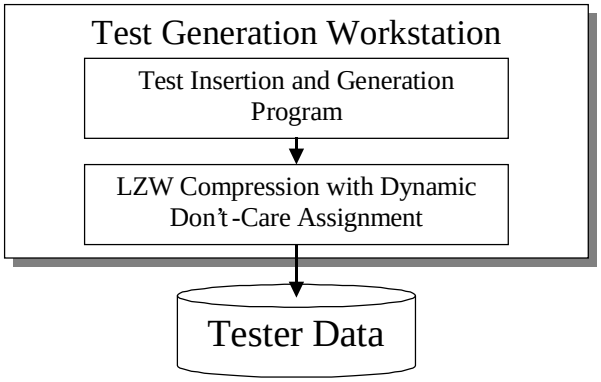

Figure 1. Test Generation Architecture

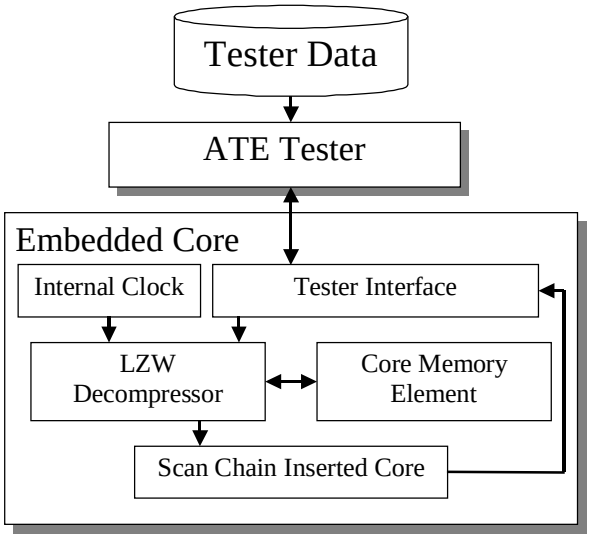

Figure 2. Test Application Architecture

3. LZW Compression

LZW compression is a dictionary lookup-based algorithm. Two important features are that the dictionary is built dynamically and the dictionary is included within the compressed message. This scheme works for the same algorithm is used for both compression and decompression. The algorithm requires some configuration, thus a configurator block is required. The LZW configurator allows for the selection of the LZW dictionary size as well as the LZW character size. Any configuration options for compression are made through this block and configuration is required prior to the start of sending any compressed data.

Given an LZW dictionary, there are three basic memory elements for the LZW algorithm to function. The first memory element is called the "buffer" which consists of 'C_E' bits. 'C_E' represents the number of encoded or compressed character bits. The second memory element is called the "input" which is 'C_D', the number of decoded or uncompressed character bits. The third memory element is called the "output" which it is 'C_E' in size. For illustration purposes, an example of a 1-bit message character is used to explain the operation of the LZW algorithm as shown in Figure 3. Figure 3a shows that the first message character is stored in "Buffer" to initialize this memory element. The left-most character of the "Uncompressed Input" is the contents of the "Input" memory element. In Figure 3b, the "Buffer" and "Input" pair are checked for existence in the LZW dictionary.

If there is no compressed code in the dictionary for the "Buffer, Input" pair, then "Buffer" is assigned to "Output" and "Input" is assigned to "Buffer". For example in Figure 3b, the "0,1" pair is not in the dictionary. Thus, compressed code "2" is assigned to the "0,1" pair, then "Buffer" is assigned to "Output" and "Input" is assigned to "Buffer". The dictionary reference of "2" is used because both "0" and "1" represent uncompressed characters within the compressed output result. In the general case, the first available dictionary entry is one greater than the largest uncompressed representation.

If there is a compressed code in the dictionary, then nothing is assigned to "Output" and the compressed code is assigned to "Buffer". Figure 3e shows that the "0,1" pair is in the dictionary. Therefore, the compressed code "2" is assigned to "Buffer" and to "Output". Finally, the next uncompressed input character is assigned to "Input".

Figure 3f shows an instance where the "Buffer" contains a compressed code and the "Buffer, Input" pair, (2,1), is added to the dictionary. Here, the new dictionary reference "5" represents an uncompressed string of "011" bits. The compression process continues until all of the uncompressed input characters are read. After the last iteration, Figure 3k, the content of "Buffer" is assigned to the "Output" to complete the compression process and generates the resulting compressed output. There is a

limit on the number of physical dictionary elements that can be included so both the compression and the decompression algorithm need to recognize this dictionary limit.

	Compressed Output	Dictionary	Buffer	Uncompressed Input
a)			0	100110101
b)	0	2(0,1)	0	100110101
c)	01	3(1,0)	1	00110101
d)	010	4(0,0)	0	0110101
e)	010		0	110101
f)	0102	5(0,1,1)	2	10101
g)	0102		1	0101
h)	01023	6(1,0,1)	3	101
i)	01023		1	01
j)	01023		3	1
k)	010236		6	

Figure 3. LZW compression table representation.

"Don't-Care" mapping is the key to quality compression results for test vector sets. Many methods were explored for assigning the "Don't-Care" bits. Most of these methods focused on pre-processing the "Don't-Care" bits first and then applying the LZW compression. All of these methods produced 40% to 60% test vector compression. What finally produced the published results was mapping the "Don't-Care" bits of the test patterns, as published in [8]. This mapping idea is a dynamic sliding window approach where "Don't-Care" bits are assigned while the LZW algorithm is processing the uncompressed input bits.

4. LZW Decompression

The LZW compression scheme creates the dictionary when compressing and reconstructs the dictionary when decompressing. The dictionary references are contained within the LZW compressed data stream. Using the compressed results from Figure 3, the operation of the LZW decompression algorithm (as shown in Figure 4) recreates the original input data stream. The decompression algorithm assigns either the "Input" memory element or the "Input" referenced dictionary contents to the "Output" memory element. When the compressed input character represents an uncompressed character, the "Input" is directly assigned to the "Output". When the input is a compressed character, this is a reference to an uncompressed character string. This is to be sent to the "Output" memory element from the memory block.

The decompression process starts with Figure 4a where the "Input" is assigned to "Output". Then, in Figure 4b the "Input" is assigned to the "Buffer" and the next "Input" character is sent to the output. Next, the "Buffer, Input" pair is added to the LZW dictionary. Figure 4d shows that when the compressed input character represents a string of uncompressed characters, the dictionary entry "2(0,1)" is accessed. Finally, the resulting uncompressed characters, "0,1", are passed to the output.

When "Input" is a compressed character, as in Figure 4d-f, the new dictionary entry contains the characters represented by "Buffer" and the left-most character represented by "Input".

Figure 4f shows a special case of LZW where the input compressed character is referencing a dictionary entry that has not been created yet. For these cases, the uncompressed character to be sent to "Output" contains the characters represented by the contents of the "Buffer" memory element and the left most character of that "Buffer" memory element.

5. Implementation

5.1. Decompressor

	Uncompressed Output	Dictionary	Buffer	Compressed Input
a)	0			010236
b)	01	2(0,1)	0	10236
c)	010	3(1,0)	1	0236
d)	01001	4(0,0)	0	236
e)	0100110	5(0,1,1)	2	36
f)	0100110101	6(1,0,1)	3	6

Figure 4. LZW decompression table representation.

To implement a reasonably performing LZW decompressor, the dictionary memory needs to contain the complete uncompressed character stream for each compressed character. In software LZW algorithm

implementations both a dictionary memory block and a stack memory block are used. The reason being that it minimizes the overall memory requirements. This, however, incurs a performance penalty tradeoff.

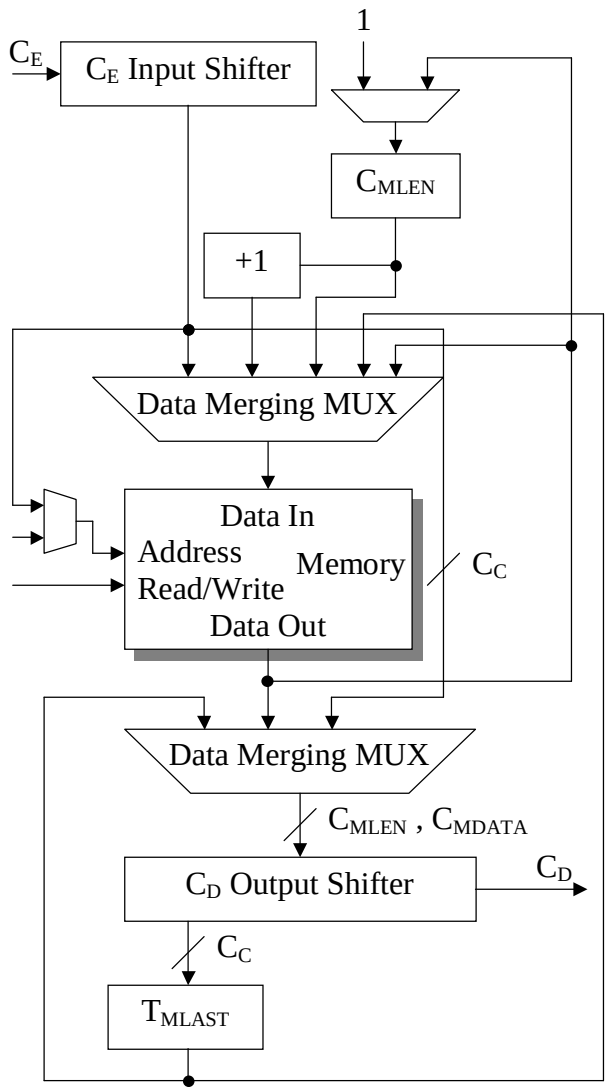

Figure 5. LZW decompression architecture.

Therefore, a performance-driven hardware decompressor implementation was created. Figure 5 shows the high-level hardware architecture for the decompressor. The process starts when 'C_E' is fully loaded in to its input shifter. The finite-state-machine controls the data-merging muxes to either read the dictionary or pass 'C_E' to the 'C_D' output shifter. The dictionary will be read when 'C_E' is a compressed character. The 'C_D' output shifter data-merging mux will be set to pass the memory's dictionary entry to its output shifter. If 'C_E' represents an uncompressed character, then the 'C_D' output shifter data-merging mux will be set to pass 'C_E' to its output shifter. 'C_C' is used to denote a single uncompressed character. After either case, and if there is an available dictionary entry, the memory's data-merging MUX is configured to write to that location. When creating a new dictionary entry, it is implied that a new character is being appended to another character or preexisting string of characters. Thus, a 'C_{MLEN}' incrementor is needed. 'C_{MLAST}' is equivalent to the "Buffer" memory element as discussed when explaining the LZW compression and decompression algorithms. It is also used in the creation of new dictionary entries.

5.2. Embedded Memory

The LZW decompressor requires a memory block from the embedded core to minimize its area overhead. The memory requirement is known prior to the completion of the embedded core and before integration techniques are started. The memory utilized for the LZW decompressor architecture is shown in Figure 6. The size of the dictionary is 'N'. Each memory location contains two data elements: the number of uncompressed characters 'C_{MLEN}' and the uncompressed characters themselves with a length of 'C_{MDATA}'.

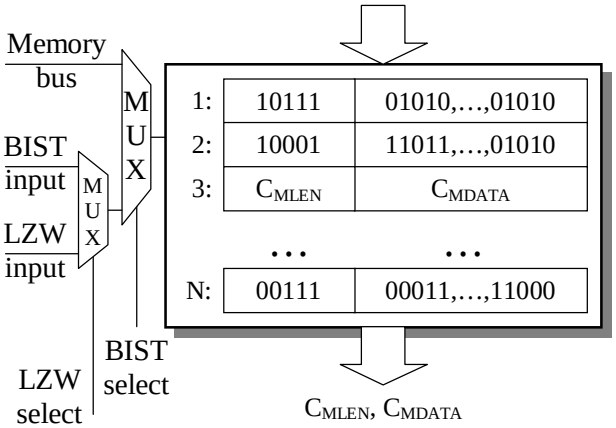

Figure 6. LZW decompression memory utilization of the core memory blocks.

Since it is desirable to reuse any memory elements from the circuit under test, it is possible that it can be integrated in the same fashion as memory BIST testing. Figure 6 shows how the control signals for the memory devices can be added in order to not impede normal circuit operation. Normal memory BIST adds muxes in front of the memory control signals. Another mux can be added on to the memory BIST side to enable LZW decompression. As for the addition loads on the outputs, a single buffer can be added to isolate the memory BIST and LZW decompression output loads. This would minimize the impact of the production test circuitry.

6. Results

A software program was developed for the LZW compressor. The ISCAS89 benchmarks were easily applied as the input to this tool; however, the ITC99 benchmarks required test insertion and test pattern generation prior to compression. To accomplish this Synopsys dc_shell, DFT-compiler and TetraMAX tools were used.

With the LZW compression and an efficient technique for assigning the "Don't -Care" bits, we first generated compression results that allowed comparison to other documented compression techniques. Table 1 shows comparison results for the LZW, LZ77 [8] and RLE [11] compression techniques.

Test	Compression Ratios		
	LZW	LZ77	RLE
s13207f	81.69%	81.45%	81.30%
s15850f	76.26%	61.90%	65.83%
s38417f	70.60%	61.56%	60.55%
s38584f	75.14%	59.97%	61.13%
s9234f	70.67%	37.66%	44.96%

Table 1. Compression Comparison Results

The LZW compression result generated above used a 64-bit dictionary entry and a 7-bit character representation. s13207f, s15850f and s9234f all have a 1024 dictionary size while s38417f and s38584f have a 2048 dictionary size. All of these tests are based on a single scan chain input and output.

After attaining quality compression results, the architecture required to implement the LZW decompressor became the focus. Using the architecture as described, Table 2 shows attainable performance results.

Test	Dict. Size	Decompress Clock		
		4x	8x	10x
s13207f	1024x64	56.19%	67.69%	70.85%
s15850f	1024x64	51.27%	62.79%	65.71%
s38417f	2048x64	43.81%	55.46%	57.99%
s38584f	2048x64	49.34%	60.83%	63.80%
s9234f	1024x64	45.75%	57.34%	59.97%

Table 2. Download Performance Improvement Results and Memory Sizes

Even with a clock four times faster than the tester clock rate, performance improvements of about only 50% were attainable. With a ten times faster clock relative to the tester clock rate, the performance is a 10% lesser difference from the compression rate. The performance improvement can match the compression rate; however, the dictionary size must be increased to accommodate.

Many more of the ISCAS89 and ITC99 test benches were used to verify algorithm effectiveness. Table 3 shows selected results from both test bench suites. For each test bench the percentage of "Don't -Care" bits within each test is noted. In general, the amount of compression is proportional to the "Don't -Care" data ratio. Also in Table 3, the uncompressed test bench size and the resulting LZW dictionary size is reported. Upon examining these results, it was observed that the growth of the dictionary size is a factor of powers of 2 as the test size grows larger.

Test	Don't Cares	Orig. Size	Comp-ression	Dict. Size
s13207f	93.15%	165200	81.69%	1024
s15850f	83.56%	76986	76.26%	1024
s35932f	35.30%	28208	72.65%	128
s38417f	68.08%	164736	70.60%	2048
s38584f	82.28%	199104	75.14%	2048
s5378f	72.62%	23754	59.00%	1024
s9234f	73.00%	39273	70.67%	1024
itc b04	87.34%	46980	80.86%	512
itc b05	97.95%	128554	86.76%	256
itc b07	82.14%	19209	80.27%	512
itc b12	92.01%	152750	83.19%	1024
itc b13	90.06%	23986	84.20%	512

Table 3. ISCAS89 and ITC99 Benchmark Results

Test	LZW Character Size in Bits (C_C)			
	1	4	7	10
s13207f	75.21%	80.10%	79.50%	0.00%
s15850f	59.98%	74.57%	74.78%	0.01%
s38417f	50.58%	61.85%	65.54%	0.00%
s38584f	52.31%	61.50%	64.08%	0.00%
s9234f	54.17%	67.84%	69.44%	0.00%

Table 4. Compression versus LZW Character Size

Test	Dictionary Entry Size in Bits (C_{MDATA})			
	60	123	250	505
s13207f	79.50%	88.02%	91.56%	92.53%
s15850f	74.79%	80.89%	81.06%	81.06%
s38417f	65.54%	66.47%	66.47%	66.47%
s38584f	64.08%	65.26%	65.26%	65.26%
s9234f	69.44%	73.54%	73.88%	73.88%

Table 5. Compression versus Entry Size

Table 4 shows the effect of LZW character size given compression. The data was generated with N = 1024 and C_{MDATA} = 63. The results show that the "Don't -Care" assignment improves as the character size increases. At about a 10-bit character size with a dictionary of size N = 1024, there are no more compress codes available. Thus poor compression results are observed.

Table 5 shows the effect of the LZW dictionary entry size given compression. The data was generated with N = 1024 and C_C = 7. The results show that the larger the dictionary entry, the higher the compression.

Test	Longest String	C_{MDATA}		
		60	123	505
s13207f	483	69.30%	77.99%	82.33%
s15850f	126	64.60%	70.63%	70.73%
s38417f	91	55.38%	56.25%	56.25%
s38584f	91	54.07%	55.11%	55.11%
s9234f	189	59.34%	63.34%	63.63%

Table 6. Performance versus entry size.

Table 6 shows the effect of the same tests as in Table 5 except performance improvement metrics were generated with a 10x internal clock relative to the ATE tester clock. However, Table 6 has the longest string column, which shows why both the compression and the performance increase and then level out when the dictionary entry increases. Given the LZW compression, each test bench generated a longest uncompressed string representation. Until the dictionary entry size is large enough to incorporate the longest string, sub optimal compression and performance is experienced. However, the memory for any given embedded core may be the limiting factor. For example, if s13207f is an embedded core and optimal compression was desired, the following parameters of N = 1024, C_C = 7 and C_{MDATA} >= 483 are needed. This causes a 1024 by 490-bit memory requirement.

7. Conclusion

A technique for high compression ratio was described which exploits the high number of "Don't -Care" bits that occur in test sets. This technique reduced the test set size in terms of number of bits. A fast hardware decompressor is also necessary and thus designed to reduce the test set download time. In addition, reusing BIST–based embedded memory cores or by making this hardware compression engine part of normal operation, can further reduce or eliminate the chip area overhead. Engineering tradeoffs such as the dictionary entry size, uncompressed character size and longest support compressed string are required to optimize performance and minimize area. The benchmark results show that good results can be achieved by selecting a reasonable dictionary size, dictionary entry width and character width.

References

[1] E. H. Volkerink, A. Khoche, J. Rivoir, and K. D. Hilliges. Test economics for multi-site test with modern cost reduction techniques. In Proc. VLSI Test Symp., pages 411-416, 2002.

[2] Y. Zorian, E. J. Marinissen, and S. Dey. Testing embedded-core-based system chips. Computer, 32(6):52-60, 1999.

[3] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Trans. on Information Theory, 23(3):337-343, May 1977.

[4] J. Ziv and A. Lempel. Compression of individual sequences via variable rate coding. IEEE Trans. on Information Theory, 24(5):530-536, 1978.

[5] F. Brglez, D. Bryan, and K. Kozminski. Combinatorial profiles of sequential benchmark circuits. In Proc. Int' l Symp. on Circuits and Systems, pages 1129- 1234, 1989.

[6] L. Basto. First Results of ITC'99 benchmark results. IEEE Design & Test of Computers, 17(3):54-59, July 2000.

[7] S. Davidson. ITC'99 Benchmark Circuits – Preliminary Results. In Int' l Test Conf., pages 1125-1125, 1999.

[8] Wolff, F. G., Papachristou, C., "Multiscan-based Test Compression and Hardware Decomposition Using LZ77," ITC, 2002

[9] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, B. Keller, and B. Koeneman. Opmisr: The foundation for compressed atpg vectors. In Proc. Int' l Test Conf., pages 748-757, 2001.

[10] A. Chandra and K. Chakrabarty. System-on-a-chip test-data compression and decompression architecture based on golomb codes. IEEE Trans. on CAD/ICAS, (3):355-368, March 2001.

[11] A. Chandra and K. Chakrabarty. Reduction of soc test data volume, scan power and testing time using alternating run-length codes. In Proc. Design Automation Conference, pages 673-678, June 2002.

[12] R. Dorsch and H.-J. Wunderlich. Tailoring atpg for embedded testing. In Int' l Test Conf., pages 530-537, 2001.

[13] H. Ichihara, A. Ogawa, T. Inoue, and A. Tamura. Dynamic test compression using statisical coding. In Proc. of Asian Test Symposium, pages 143-148, 2001.

[14] V. Iyengar, K. Chakrabarty, and B. T. Murray. Deterministic built-in pattern generation for sequential circuits. JETTA, 15:97-115, October 1999.

[15] A. Jas, J. Ghosh-Dastidar, and N. A. Touba. Scan vector compression/decompression using statisical coding. In Proc. VLSI Test Symp., pages 114-120, April 1999.

[16] A. Jas and N. Touba. Test vector decompression via cyclical scan chains. In Proc. Int' l Test Conf., pages 458-464, 1998.

[17] S. Kajihara and K. Miyase. On identifying don' t care inputs of test patterns for combinational circuits. In Proc. Int' l Conf. Computer-Aided Design, pages 364- 369, 2001.

[18] A. Khoche, E. H. Volkerink, J. Rivoir, and S. Mitra. Test vector compression using eda-ate synergies. In Proc. VLSI Test Symp., pages 97-102, 2002.

[19] B. Koeneman, C. Barnhart, B. Keller, T. Snethen, O. Farnsworth, and D. Wheater. A smartbist variant with guaranteed encoding. In Proc. Asian Test Conf., pages 325-330, 2001.

[20] J. Rajski. Dft for high-quality low cost manufacturing test. In Asian Test Symp., pages 3-8, 2001.

[21] K. Loudon. Mastering Algorithms with C. O' Reilly, 1999.

[22] M. Nelson and J. L. Gailly. The Data Compression Book. M & T Books, New York, 1996.

[23] T. A. Welch. A technique for high-performance data compression. IEEE Computer, 17(6):8-19, June 1984.

[24] C. Su, C. Yen, J. Yo. Hardware efficient updating technique of LZW CODEC design. Proc. Circuit and Systems Symp., pages 2797-2800, 1997.