MarciaTesta: An Automatic Generator of Test Programs for Microprocessors' Data Caches

Authors: Di Carlo S., Gambardella G., Indaco M., Rolfo D., Prinetto P.,

Published in the Proceedings of the IEEE 20th Asian Test Symposium (ATS), 20-23 Nov. 2011, New Delhi, IN.

DOI: 10.1109/ATS.2011.78

mechanisms. The paper is organized as follows: Section II describes the SBST methodology to adapt march test for cache memories. Section III describes the MarciaTesta tool, exploring its main features, input/outputs, and internal architecture. Section IV shows experimental results gathered targeting the *Xilinx Microblaze* and the *Altera Nios II* microprocessors, respectively. Section V concludes the paper.

II. SBST Methodology

This section shortly overviews the SBST methodology proposed in [11] and exploited in this paper.

Traditional March tests must be properly translated before they can be applied to a cache memory. The overall translation process aims at defining basic cache march test operations that take into account the peculiar way in which cache memory cells can be accessed. The following basic operation are defined:

- $w(\alpha t, DB)$: represents a write operation of a cache line. DB is the data background pattern (for each DB a complemented DB must be defined) written in the data array section of the cache. α identifies the set in which the line must be written and t identifies the tag written in the directory array of cache.
- $r(\alpha t, DB)$: represents a Read & Verify operation. The cache line placed in the set α and identified by the tag t is read and compared with the expected DB.
- $r(\alpha t)$: similar to the Read & Verify operation but the read value is not verified.

We will not discuss the complex addressing order translations presented in [11] since, in its current implementation, our generation tools deals with direct-mapped data cache memories whose cache lines are directly addressable.

According to the internal cache organization, the test approaches for data and directory array are different. In data array testing, DB patterns are crucial while actual tag values are not relevant. In this context, tags are used to correctly address the cache and the only requirement is the number of different tags. For directory array, tags became the actual test patterns. In addition, read and write operations can only be performed in an indirect way exploiting the data array. Taking into account these issues, the read verification can only be performed by detecting cache-miss events. This in turns requires creating an inconsistency between the content of the cache and that of the main memory, using different approaches based on the adopted write policy. The write-through policy requires to disable the cache in order to write a different value in the main memory, while for write-back cache policy the following rules are enough to create the miss condition:

- any write operation must be followed by another one with complemented DB;
- each Read & Verify operation must contain the value of the expected DB stored in the cache memory;
- an initialization march element must be added.

III. MarciaTesta tool

MarciaTesta tool is a general purpose tool able to generate the assembler algorithm (*ASM*) that implements a chosen March Test for a target processor data cache.
In the sequel, we focus first on a tool overview, then we present its basic principles of operations. The section concludes with a detailed analysis of the tool's internal architecture.

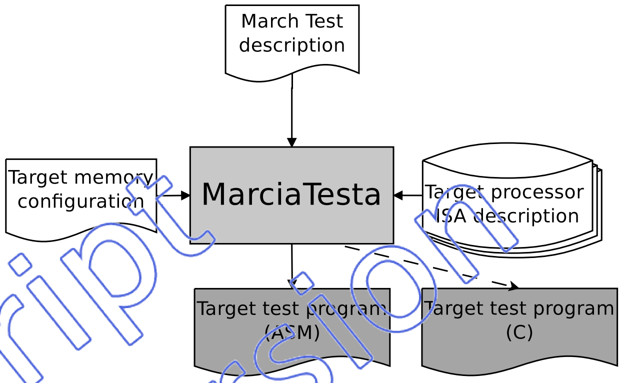
March Test description
Target memory configuration
MarciaTesta
Target processor ISA description
Target test program (ASM)
Target test program (C)

Fig. 1. MarciaTesta tool.

A. Input Data

The tool gets the following input data (Fig. 1):

(i) Target memory configuration
It is the input file that describes the architecture of the target data cache, according to the following parameters:

- *Write policy*: the write policy of the cache (*Write Through* or *Write Back*.
- *Word size*: the data-width of the target system.
- *#Offset (O)*: The number of less significant bits of the memory address (O) that identify a specific word into the cache line (Fig. 2):

$$O = \lceil \log_2(nW) \rceil$$

  where nW identifies the number of words per cache line.
- *#Index (I)*: the set of bits of the memory address that identifies the cache line where the desired information can be stored (Fig. 2):

$$I = \lceil \log_2(nC) \rceil$$

  where nC identifies the number of cache lines.
- *#Tag (T)*: the number of most significant bits of the memory address that identify the content of the directory array used to tag the cached information (see Fig. 2):

$$T = N - I - O$$

  where N is the memory address-width.
  Resorting to the values of the above presented parameters, one can easily calculate the memory's and cache's dimensions:

$$Cache\_Dim = 2^{I+O}$$

$$Mem\_Dim = 2^{T+I+O}$$

- *Base address (BA)*: the address serving as a reference point (*"base"*) for other addresses in data array testing. An important condition on this parameter is:

$$BA + 2 * (T + I + O) \in \{CacheableMemory\}$$

(ii) March Test description
It includes:

- The desired March Test to be implemented
- The *Addressing Order (AO)*, i.e., the exact sequence in which the cache lines must be addressed during the ascending (*'U'*) and descending (*'D'*) addressing orders. Such a facility is of primary importance to define custom addressing sequences required to detect dynamic memory faults [7].
- The *Data Background (DB) List*, i.e., the set of background patterns required for the implementation of the data and directory array test.

(iii) Target processor ISA description
It is a library that contains the minimum set of supported target microprocessors instructions, grouped in Meta-ISAs [16], useful to synthesize march test, suitable for data cache memories, in assembly code.

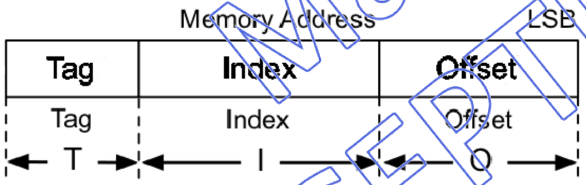

Fig. 2. Memory address logical division.

B. Output Data

The tool generates the following output data:

- *Target test program (C)*
It is an optional output useful for test engineer for emulating test execution. It represents the C/C++ intermediate implementation of the March Test generated according to the *SBST methodology*.
- *Target test program (ASM)*
It is the generated assembly code for testing the data cache.

C. Internal architecture

The assembly test program generation relies on two main translation steps (Fig. 3). The former one, implemented by the *YAUF2C* module, transforms the input March Test into a suitable test for data cache, exploiting a given *SBST methodology*. In the generated output, the test is expressed in terms of a set of C-based macros.

The latter step translates such a C program into the right assembly instructions of the target microprocessor.

In the sequel we briefly outline the main features of the two translation steps.

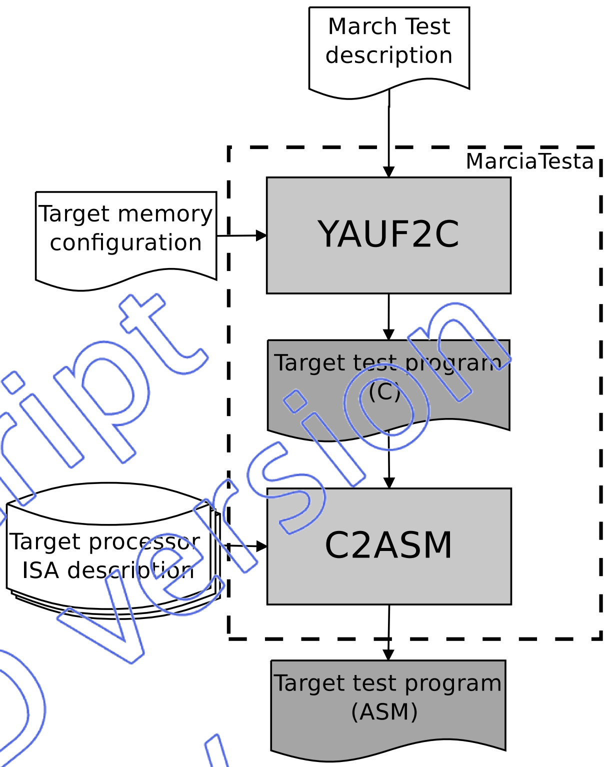
March Test description
MarciaTesta
Target memory configuration
YAUF2C
Target test program (C)
Target processor ISA description
C2ASM
Target test program (ASM)

Fig. 3. Flow diagram of the tool.

YAUF2C
The *YAUF2C* module is a C program that translates the input march test (MT) into an intermediate behavioral implementation, resorting to the set of macro-operation summarized in Table I).

TABLE I
C-like MT MACRO-OPERATION

Name	Meaning
Write_memory_cell(*word*, *address*)	Write in memory the *word* at *address*
Read_memory_cell(*address*)	Write the cache line corresponding to *address*
Read_and_verify_for_Data(*address*)	Read the data corresponding to *address* from cache and verify the correctness of it
Read_and_verify_for_Directory (*DB_TAG*,*DB_OK*)	Read the data corresponding to *DB_TAG+addressing_order* from cache and verify the equality with *DB_OK*
Invalidate_cache_line(*index*)	Invalidate the cache line corresponding to *index*
Enable_cache	Enable read and write from the cache
Disable_cache	Disable read and write from the cache

To better understand the translation process implemented by *YAUF2C*, in the sequel we provide three examples of

March Test translation targeting the data and directory arrays of no write-allocate caches, considering both write-through and write-back policies.
The cache write operation is implemented by means of a read operation that fills a given cache line assuming that the cache is initially empty. To guarantee this condition, all cache lines must be initially invalidated. Table II shows the translation of a test for the data array. The *Init Memory* operation is required to initialize the memory with the correct DB patterns. The other two operations show how to translate Write and Read operations, respectively. The second one is an upward $w0$, implemented by a sequence of invalidate operations on each cache line, followed by read memory statements at the addresses corresponding to the ascending sequence defined by the *addressing order* content.
The third operation is a downward $r1$ implemented by a sequence of Read & Verify operations at the addresses corresponding to the descending sequence defined by the *addressing order* content.

TABLE II
TRANSLATION EXAMPLE FOR DATA ARRAY TEST

MT		***Target test program C***

Init Memory	⇒	for(i=0;i<pow(2,I+O);i++)
		Write_memory_cell(DB, BaseAddress+i)
		for(i=pow(2,I+O);i<pow(2,I+O+1);i++)
		Write_memory_cell($\overline{DB}$, BaseAddress+i)

U(w0)	⇒	for(i=0;i<pow(2,I);i++)
		Invalidate_cache_line(i)
		for(i=0;i<pow(2,I);i++)
		Read_memory_cell
		(BaseAddress+pow(2,I+O)+Addressing_Order[i])

D($r1$)	⇒	for(i=pow(2,I)-1;i>=0;i- -) Read_and_verify_for_Data (BaseAddress+Addressing_Order[i])

Table III and IV show a translation example of the directory array test for write-through and write-back policies, respectively.
The first row writes the data background pattern at the addresses that have the T MSB equals to the desired *DB_TAG*, in order to achieve the goal to write the correct *DB* in the directory array when we will read at that address, knowing the corresponding pattern in the data array. In this example, the only difference between the two types of cache is in the mismatch creation between cache and memory during write operations [11]. In the write-back cache such a difference is created using DB_zero and DB_one, while in the write-through cache it is created using *Enable_Cache* and *Disable_Cache*.

The second row presents the steps required to write in the array the desired data and to be able to read it indirectly. The first step invalidates each line of the cache, avoiding undesired cache hit during the write operation. Then the test, in the write-through case (Table III), reads the cache at the correct address (*DB_TAG* one), that corresponds to a *DB_TAG* write in directory array [11]. After the data cache disabling, the algorithm writes, at the same address in memory, the complemented DB for the data array, creating a cache incoherence; the data cache is then enabled again. A Read & Verify operation in directory array can thus be implemented as a read from data cache, verifying that the read data equals the one written when the cache was enabled. If the read data is the second one (written when the cache was disabled), an error occurred on the directory array.
In the write-back case (Table IV), the cache incoherence is obtained by writing in the data array the complemented value of the previous written one, thus avoiding to use the disable and enable cache operations [11]. Resorting to this kind of write operation, the Read & Verify operation for the directory array can be implemented by a read from the data array, verifying that the data it is equal to the last written, DB_one for $r1$ and DB_zero for a $r0$. If the read data is equal to $\overline{DB\_one}$ or $\overline{DB\_zero}$, an error occurred in the directory array

TABLE III
TRANSLATION EXAMPLE DIRECTORY ARRAY TEST OF A WRITE-THROUGH CACHE

MT		***Target test program C***

Init Memory	⇒	for(i=0;i<pow(2,I+O);i++)
		Write_memory_cell(DB, DB_TAG+i)
		for(i=pow(2,I+O);i<pow(2,I+O+1);i++)
		Write_memory_cell(DB, $\overline{DB\_TAG}$+i)

U($w0$)	⇒	for(i=0;i<pow(2,I);i++)
		Invalidate_cache_line(i)
		for(i=0;i<pow(2,I);i++)
		Read_memory_cell
		($\overline{DB\_TAG}$+Addressing_Order[i])
		Disable_Cache
		for(i=pow(2,I+O);i<pow(2,I+O+1);i++)
		Write_memory_cell($\overline{DB}$, $\overline{DB\_TAG}$+i)
		Enable_Cache

D($r1$)	⇒	for(i=pow(2,I)-1;i>=0;i- -) Read_and_verify_for_Directory (DB_TAG,DB)

C2ASM
The second main module of MarciaTesta (Fig. 3) translates the C-like March Test generated by the previous module into an equivalent assembly program exploiting the ISA of the target processor. In order to write a correct *ASM* software, the tool exploits the description of the ISA contained in the *Processor ISA Library* and translates the macro-operations of the *C-like MT* (with related parameters) into a sequence of machine-level instructions.
The *Processor ISA Library* contains a selection of instructions for the implementation of the macro-operations listed in Table I. It is very important to notice that the *for* statements

TABLE IV
TRANSLATION EXAMPLE DIRECTORY ARRAY TEST OF A WRITE-BACK CACHE

MT		*Target test program C*

Init Memory	⇒	DB_zero = DB
		DB_one = DB
		for(i=0;i<pow(2,I+O);i++)
		    Write_memory_cell(DB_one, DB_TAG+i)
		for(i=pow(2,I+O);i<pow(2,I+O+1);i++)
		    Write_memory_cell(DB_zero, $\overline{DB\_TAG}$+i)

$U(w0)$	⇒	for(i=0;i<pow(2,I);i++)
		    Invalidate_cache_line(i)
		DB_zero = $\overline{DB\_zero}$
		for(i=pow(2,I+O);i<pow(2,I+O+1);i++)
		    Write_memory_cell(DB_zero, $\overline{DB\_TAG}$+i)
		for(i=0;i<pow(2,I);i++)
		    Read_memory_cell
		    ($\overline{DB\_TAG}$+Addressing_Order[i])

$D(r1)$	⇒	for(i=pow(2,I)-1;i>=0;i- -)
		    Read_and_verify_for_Directory
		    (DB_TAG,DB_one)

implemented in *C-like MT* are unrolled in the assembler program, in order to have a faster test and to avoid problems due to the presence of branch-prediction strategies in the processor.

Table V and VI show a translation example of two macro-operations for the *Nios II* and *MicroBlaze* ASM, respectively. These tables display the first translation step, from macro-operation to Meta-ISAs [16], and the second one from Meta-ISAs to ASM, obtained applying the *Target processor ISA description* of the target processor.

TABLE V
TRANSLATION EXAMPLE OF MACRO-OPERATION FOR MICROBLAZE ASM

Target test program C		*Meta-ISA*		*ASM*
Write_memory_cell (DB, DB_TAG+i)	⇒	Register32write	⇒	imm <DB[31:16]> ori <r_DB>,<r0>,<DB[15:0]>
		MemoryWrite	⇒	sw <r_addr>,<r0>,<r_DB>
Read_and_verify_for_Data (BA+AO[i])	⇒	MemoryRead	⇒	lw <r_read>,<r0>,<r_addr>
		Verify Cache	⇒	cmpu <r_read>,<r_read>,<r_DB> bneqi <r_read>,<ERROR>

TABLE VI
TRANSLATION EXAMPLE OF MACRO-OPERATION FOR NIOSII ASM

Target test program C		*Meta-ISA*		*ASM*
Write_memory_cell (DB, DB_TAG+i)	⇒	Register32write	⇒	movhi <r_DB>,%hi(<DB>) ori <r_DB>,<r_DB>,%lo(<DB>)
		MemoryWrite	⇒	stw <r_addr>,<0>(<r_DB>)
Read_and_verify_for_Data (BA+AO[i])	⇒	MemoryRead	⇒	ldw <r_read>,<0>(<r_addr>)
		Verify Cache	⇒	bne <r_read>,<r_DB>,<ERROR>

IV. EXPERIMENTAL RESULTS

The MarciaTesta tool has been used to generate the test for the data cache memories of two different microprocessors: *Microblaze* [17] and *Nios II* [18].
Microblaze is a soft core processor designed for *Xilinx* FPGAs, with a Harvard memory architecture, a RISC-like instruction set and a data cache with write-through policy. Nios II is a 32-bit RISC embedded-processor designed for the *Altera* FPGAs with a data cache with write-back policy. The actual boards on which we deployed the automatically generated tests are a *Virtex4 ML-403 Embedded Platform* [19] and a *Nios Development Board Cyclone II Edition* [20], respectively.
Both processors have been implemented with $2kB$ instruction and data cache with four words per cache line, $128kB$ of on-chip memory. This means that $T = 6$, $I = 9$ and $O = 2$ (see Fig. 2).
Table VII shows some data related to the automatic generation of the assembly programs to implement 11 different March Tests. For each test, the table lists the number of assembly

TABLE VII
EXPERIMENTAL RESULTS ON NIOS II AND MICROBLAZE PROCESSORS

March test	*Test Length(Xn)*	*Number of ASM rows*	
		NiosII	*MicroBlaze*
MATS+ [21]	5n	50,289	42,079
March C- [22]	10n	99,453	87,147
March U [23]	13n	128,121	111,719
PMOVI [24]	13n	130,169	117,863
March LR [25]	14n	138,365	121,963
March SR [26]	14n	140,413	128,107
March B [27]	17n	164,985	140,391
March MSS [28]	18n	177,277	156,779
March SS [29]	22n	218,237	197,739
March G [30]	23n	226,433	201,839
Abraham, Thatte [31]	30n	296,109	267,419

instructions of the test program generated by MarciaTesta for both the data and the directory array test.
One can notice that there is no correlation between the complexity of the original march test and the number of instructions of the test program. This can be easily understood by focusing on the write and read operations. While these are usually considered as atomic in ”traditional” March test, i.e., in March tests for ”normal” memories, when targeting cache memories they must be implemented resorting to a sequence of instructions.
In addiction, the *MicroBlaze* test requires less assembly instructions than the *Nios II*. This is due to the reduced number of operations required to create the cache incoherence during the write operation in the directory array test of the write-through cache.

V. CONCLUSION

In this paper we presented MarciaTesta, an EDA tool for the automatic generation of assembly cache test program. Starting

from a formal definition of March Test, MarciaTesta generates assembly test code for the target processor.
The cross-architecture feature of the tool, i.e., its capability of generating assembly code for several architectures, relies on the availability of a set of libraries describing the ISA of the target processors.
The correctness of the tool has been proved by a novel validation and verification approach based on the generation of log files at three different abstraction levels, and namely at the March Test level, at the C implementation level and, eventually at the machine Instruction level. The correctness of the generated code has been checked on several target architectures and in the implementation of several test algorithms on each of them.
Next releases of MarciaTesta will support from the one hand set-associative cache memories and, from the other hand, instruction caches.

Acknowledgment

The authors would like to express their sincere thanks to the whole design team of Ansaldo STS SpA for their helpful hints and guidelines.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[2] E. J. Marinissen, B. Prince, D. Keitel-Schulz, and Y. Zorian, "Challenges in embedded memory design and test," in *Proc. Design, Automation and Test in Europe*, pp. 722–727, 2005.

[3] "International technology roadmap for semiconductors." http://www.itrs.net, 2010.

[4] S. Kornachuk, L. McNaughton, R. Gibbins, and B. Nadeau-Dostie, "A high speed embedded cache design with non intrusive bist," in *Proc. Records of the IEEE International Memory Technology, Design and Testing Workshop*, pp. 40–45, 1994.

[5] J. Bralich and J. Fleischman, "Design of cache test hardware on the hp pa8500," *IEEE Design & Test of Computers*, vol. 15, no. 3, pp. 58–63, 1998.

[6] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, 2005.

[7] S. Di Carlo and P. Prinetto, *Models in Hardware Testing*, ch. Models in Memory Testing. Springer, 2010.

[8] Y.-C. Lin, Y.-Y. Tsai, K.-J. Lee, C.-W. Yen, and C.-H. Chen, "A software-based test methodology for direct-mapped data cache," in *Proc. 17th Asian Test Symposium (ATS)*, pp. 363–368, 2008.

[9] J. Sosnowski, "In-system testing of cache memories," in *Proc. IEEE International Test Conference (ITC)*, pp. 384–393, 1995.

[10] S. Alpe, S. Di Carlo, P. Prinetto, and A. Savino, "Applying march tests to k-way set-associative cache memories," in *Proc. 13th European Test Symposium (ETS)*, pp. 77–83, 2008.

[11] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 1030–1044, 2011.

[12] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, "A software-based self-test methodology for in-system testing of processor cache tag arrays," in *Proc. IEEE 16th International On-Line Testing Symposium (IOLTS)*, pp. 159–164, 2010.

[13] S. M. Al-Harbi and S. K. Gupta, "A methodology for transforming memory tests for in-system testing of direct mapped cache tags," in *Proc. 16th IEEE VLSI Test Symposium (VTS)*, pp. 394–400, 1998.

[14] W. J. Perez H, J. V. Medina, D. Ravotto, E. Sanchez, and M. S. Reorda, "Software-based self-test strategy for data cache memories embedded in socs," in *Proc. 11th IEEE Workshop Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 1–6, 2008.

[15] J. Sosnowski, "Improving software based self - testing for cache memories," in *Proc. 2nd International Design and Test Workshop (IDT)*, pp. 49–54, 2007.

[16] S. D. Carlo, G. Gambardella, M. Indaco, P. Prinetto, and D. Rolfo, "A unifying formalism to support automated synthesis of sbsts for embedded caches," in *Proc. of the 9th East-West Design & Test Symposium*, pp. 39–42, 2011.

[17] Xilinx, *MicroBlaze Processor Reference Guide*, 2004.

[18] Altera, *Nios II Processor Reference Handbook*, v7.2 ed., 2007.

[19] Xilinx, *ML403 Evaluation Platform*, v2.5 ed., 2006.

[20] Altera, *Nios Development Board Cyclone II Edition Reference Manual*, v1.3 ed., 2007.

[21] I. Mrozek and V. N. Yarmolik, "Mats+ transparent memory test for pattern sensitive fault detection," in *15th International Conference on Mixed Design of Integrated Circuits and Systems, (MIXDES)*, pp. 493–498, 2008.

[22] M. Marinescu, "Simple and efficient algorithms for functional ram testing," in *Proc. of International Test Conference (ITC)*, pp. 236–239, 1982.

[23] A. J. van de Goor and G. G.N., "March u: a test for unlinked memory faults," in *Proc. of IEEE Circuits, Devices and Systems*, pp. 155–160, 1997.

[24] J. H. D. Jonge and A. J. Smeulders, "Moving inversions test pattern is thorough, yet speedy," 1976.

[25] A. van de Goor, G. Gaydadjiev, V. Mikitjuk, and V. Yarmolik, "March lr: a test for realistic linked faults," in *Proc. of 14th VLSI Test Symposium (VTS)*, pp. 272–280, 1996.

[26] S. Hamdioui and A. Van De Goor, "An experimental analysis of spot de fects in srams: realistic fault models and tests," in *Proc. of the 9th Asian Test Symposium (ATS)*, pp. 131–138, 2000.

[27] A. Van De Goor, "Using march tests to test srams," *Design Test of Computers, IEEE*, vol. 10, no. 1, pp. 8–14, 1993.

[28] G. Harutunyan, V. Vardanian, and Y. Zorian, "Minimal march tests for unlinked static faults in random access memories," in *Proc. of 23rd VLSI Test Symposium (VTS)*, pp. 53–59, 2005.

[29] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March ss: A test for all static simple ram faults," in *Proc. of the 2002 IEEE International Workshop on Memory Technology, Design and Testing*, pp. 95–100, 2002.

[30] D. Suk and S. Reddy, "A march test for functional faults in semiconductor random access memories," *IEEE Trans. Comput.*, vol. C-30, no. 12, pp. 982–985, 1981.

[31] R. Nair, S. Thatte, and J. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 572–576, 1978.