A Software-Based Test Methodology for Direct-Mapped Data Cache

Yi-Cheng Lin, Yi-Ying Tsai, Kuen-Jong Lee, Cheng-Wei Yen, and Chung-Ho Chen
Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C.
yorkroy@casmail.ee.ncku.edu.tw, kjlee@mail.ncku.edu.tw, chchen@mail.ncku.edu.tw

Abstract

We present a software-based test methodology that utilizes an on-chip processor to perform test procedures for direct-mapped data cache. The cache system under test is divided into two major groups, namely the memory modules and the logic modules. For the memory modules which include the tag memory, the data memory, and the physical address tag memory, systematic procedures to transform a widely-used March algorithm into various executable instruction sequences are developed. For the logic modules, extensive analysis on the functions as well as the structures (architecture, RTL, and gate-level) of these modules is carried out and effective test instruction sequences based on the analysis are derived. A 100% fault coverage for six conventional RAM fault models and 99.13% test efficiency for single stuck-at fault model are obtained on a real 32-bit RISC processor. These results validate the viability and effectiveness of the proposed methodology for data-cache testing.

1. Introduction

Cache memories have played an important role in high performance computing by bridging the speed gap between the processor and the main memory. With the increase in processor clock rate, testing issues of the cache memories have become quite crucial. Conventional design-for-testability (DFT) techniques such as scan-cell insertion or built-in self-test (BIST) [1][2] usually create adverse effects on the area, timing, and power consumption of the cache system and have gradually become the bottleneck of large system designs [3]. Applying a software-based test scheme helps remove these impediments and can greatly improve the cache performance.

In this paper, we propose a high-performance software-based test methodology to support the testing of a direct-mapped data cache. The cache system is divided into memory modules and logic modules for test sequence development. For the memory modules, which include a tag RAM, a data RAM, and a physical address tag RAM, the widely used March C- algorithm [4] is adopted, which is transformed into various pseudo instruction sequences according to the distinct features of these memory modules. For the logic components, we analyze both the functionality and the structure of each target component and derive the memory data state that will demonstrate different behavior should the component contain faults. During test application time, these data states are checked by the developed test program right after the memory states are set up. Therefore, if any data inconsistency occurs, the corresponding component can be identified as faulty immediately.

The proposed methodology is applied to a direct-mapped data cache embedded in an Linux-verified ARM-compatible processor and results in 100% fault coverage for six conventional RAM fault models and 98.03% fault coverage or 99.13% test efficiency for the collapsed single stuck-at faults of the logic modules.

The rest of this paper is organized as follows. Section 2 discusses previous work related to this paper. Section 3 introduces the architecture of the target cache. Section 4 describes the pseudo instruction notation and the development flow of test sequences. Section 5 details the test generation procedures and lists the pseudo instructions for all modules under test. Two sample ARM assembly code segments are also given. Experimental results are presented and discussed in Section 6. Finally, Section 7 concludes our work.

2. Previous Work

Utilizing a processor as an internal tester to test on-chip caches first appeared in [5] when van de Goor et al. introduced functional testing for microprocessors and proposed the first functional test methodology for on-chip memories. In their work, the data cache was divided into three classes: the fetch/store class, the memory class, and the look-up class. Specific functional fault models are proposed according to the functionality of the three classes and test sequences for each class are developed based on the proposed fault

1081-7735/08 $25.00 © 2008 IEEE
DOI 10.1109/ATS.2008.60

IEEE computer society

models. Sosnowski extended the functional test methodology to various cache memory organizations including separated instruction and data cache, unified cache, and multi-level cache [6]. To reduce test time, a more efficient methodology to transform the March algorithm into pseudo-instruction sequences for tag memory testing of a direct-mapped data cache was proposed in [7]. It can be seen that previous test methods for cache memory mainly focus on developing test sequences for functional faults and no fault coverage is calculated. In this paper, we present a component-oriented test development methodology that takes into account not only the functional information, but also the architecture, RTL, and netlist information of a data cache. In addition, we apply the methodology to a real-life direct-mapped cache system in an ARM-compatible processor and demonstrate the effectiveness of the proposed method through actual fault coverage and test efficiency measurement.

3. The Direct-Mapped Cache Model

In this section we present our experimental platform, a direct-mapped data cache embedded in a five-stage pipelined processor. The processor supports ARMv4 instruction set [8] and memory management unit (MMU) for virtual memory capability. Through co-processor instructions, the processor can enable or disable the virtually-addressed cache system and perform privileged operations, including clearing or invalidating a specific cache line. The cache supports programmable write-through or write-back policy for write hit, and write-allocate or write-around policy for write miss. Fig. 1 shows the block diagram of the data cache system.

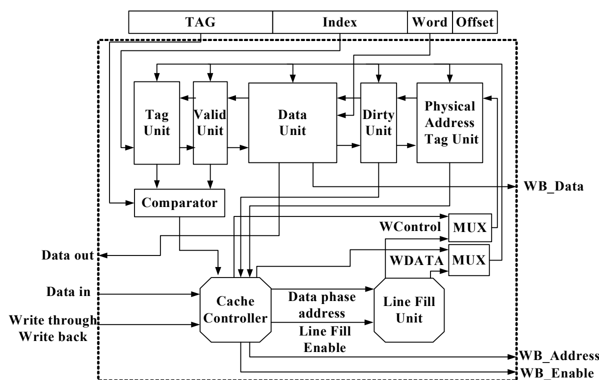

Figure 1. The target direct-mapped data cache

This data cache contains 64 lines of tag and data memory and each data line consists of 8 words. Note that there is a physical address (PA) tag associated with each cache line [8]. The PA tag caches the physical page number of each cache line so that a replaced dirty cache line can be written back to the main memory using the PA tag for physical address without an expensive page table look-up. As can be seen, this data cache platform represents the most comprehensive design of a state-of-the-art data cache system and thus the proposed method can be easily extended to other simpler variants.

4. Software-Based Test Sequence Generation

To apply the software-based methodology, our first step is to classify the cache system into structural components so that a divide-and-conquer test code development strategy can be used. After classification, the development flow in Fig. 2 of the proposed software-based test methodology is applied. For memory arrays, we employ the March C- algorithm directly [4], which is a well-known and effective testing technique for RAM faults. For logic components, we make use of both functional and structural information to develop the test sequences. Then, the generated test code is fed to two fault simulators to examine the test efficiency and fault coverage. This process will be applied to all modules under test.

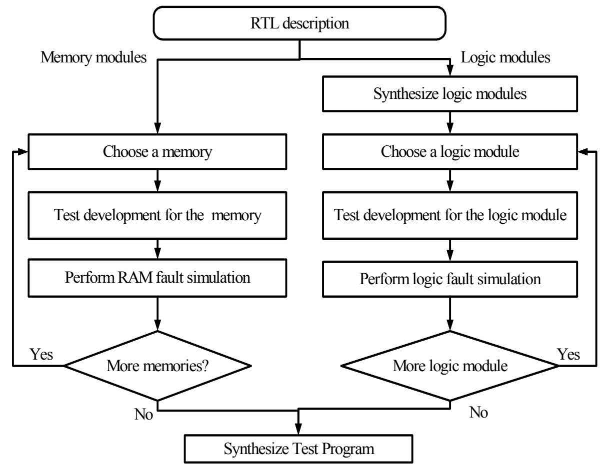

Figure 2. Development flow of the proposed software-based cache testing methodology

In the following, we introduce the notations of the pseudo instructions that are used to represent the developed test sequences, followed by the test sequence development method.

4.1. Pseudo Test Instructions

The pseudo instruction notation helps improve the portability of the developed test sequences. Since each pseudo instruction can be easily translated into one or more processor-specific instructions, migration of the

developed test sequences to a target processor instruction set architecture (ISA) can be done easily through simple translation.

Fig. 3 shows the address decoding scheme of a typical direct-mapped cache. The entire address is divided into three parts, namely tag, index, and word (concatenated with an offset). We use T, i, x to denote the contents of the tag, the index, and the word fields, respectively, and the triple [T,i,x] to denote the address. The definitions of all pseudo instructions are given below.

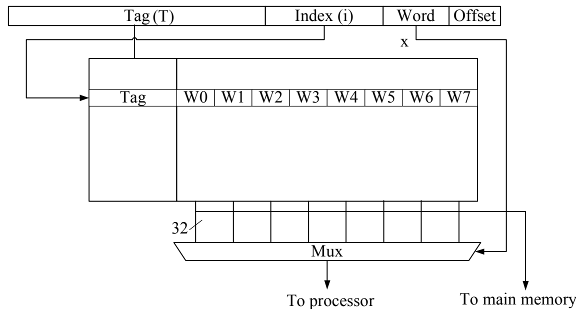

Figure 3. Address decoding of a direct-mapped cache.

- **R ([T,i,x], D)** is a load instruction. The address [T,i,x] is used to load data D from the cache and a read hit is expected.
- **R ([T,i,x], D)** is similar to **R([T,i,x],D)** except that a cache miss is expected. If this instruction is executed as expected, the cache line indexed by i is replaced with tag T along with data D from the main memory.
- **W ([T,i,x], D)** denotes a store instruction that expects a cache hit. The data D are written into the addressed cache line after the execution of this instruction.
- **W ([T,i,x], D)** denotes a store instruction that expects a cache miss. If this instruction is executed as expected, the data D are written to the main memory for write-around policy or written into the cache line after the line is fetched from the memory for write-allocate policy.
- **RM ([T,i,x], D)** is only applicable when the data cache is disabled. The data are read directly from the main memory of address [T,i,x].
- **WM ([T,i,x], D)** is only applicable when the data cache is disabled. The data D are written to the main memory directly.
- **INVi** indicates an operation to invalidate row *i* of the data cache. With this instruction, the valid bit corresponding to the *i*-th cache line will be cleared.
- **CLEARi** indicates a dump operation of the *i*-th cache line, i.e., the entire data line will be written back to the main memory and the dirty bit will be cleared.
- **EC** and **DC** denote the operation(s) to enable and disable the data cache respectively. Note that the cache contents remain unchanged after **DC** instruction.

4.2. Pseudo Test Sequences

The test sequence development for each module can be divided into four phases: initialization, data setting, function execution, and result checking. These 4 phases are described below using the valid unit as an example wherever appropriate:

- **Initialization phase:** The program initializes the processor state and cache system. Using ARM processors as an example, the initialization program includes co-processor instructions that enable the MMU, code that fills in the appropriate page tables, and code that sets the proper cache policy.
- **Data setting phase:** The instruction sequence creates data inconsistency between the test unit and the main memory by intentionally making the cached data or states different from their counterpart in the main memory. For example, if the valid bit of the i-th cache line is tested against the stuck-at-1 fault, we can use **R ([T,i,x], D)** to load the data **D** into the cache. Then the **INVi** instruction is used to clear the valid bit. Next we issue **DC** to disable the cache. Finally, the **WM ([T,i,x], !D)** instruction writes a complement data **!D** to the address **[T,i,x]** in the main memory. In this case, the cache should has data **D** while **!D** are in the main memory of the same address.
- **Function execution phase:** The instruction sequence triggers the target component. Continuing our example, the cache is activated again with the **EC** instruction and then an **R ([T,i,x], !D)** instruction is used to trigger the updating of the i-th valid bit with different loaded data **!D** from memory.
- **Result checking phase:** The instruction sequence checks the result(s) of the previous phase against the expected one(s). In our example, if the loaded data are **D** instead of **!D**, then we can tell that the data are loaded from the cache instead of the main memory because the execution of **R ([T,i,x], !D)** results in a cache hit, i.e., becomes an **R ([T,i,x], D)** operation. Therefore, the **INVi** instruction did not take effect and we can conclude that there might be a stuck-at-1 fault on that valid bit.

Many components of a direct-mapped data cache have similar features as the valid bit and therefore our methodology is quite realistic and effective for these components. For other components such as multiplexers or comparators that are not easy to detect using only functional patterns but have explicit input

controllability, a structural analysis is performed to collect specific input vectors for good structural fault coverage. These vectors are then transformed into pseudo instruction sequences and integrated into the final test software. In the next section we detail the test procedures for each component in the designated cache system.

5. Testing a Direct-Mapped Data Cache

The target data cache is classified into memory modules which include data RAM, tag RAM and PA tag RAM, along with logic modules including multiplexer, dirty unit, valid unit, tag address comparator, and cache controller. Since a detailed explanation of the test development flow has been depicted in Section 4.2 using a valid bit as an example, in the following we focus on highlighting the key features of each module and developing test sequences targeting these features.

5.1. March Test Sequence development for memory modules

The memory modules are all word-oriented and implemented with SRAM. To detect the various word-oriented RAM faults including stuck-at-fault (SAF), transition fault (TF), address decoder fault (AF), state coupling fault (CFst), inversion coupling fault (CFin), and idempotent coupling fault (CFid), we use the data background described in [6] for the designated March C- algorithm. The data background sets for different word widths are generated for the data memory, tag memory, and PA tag memory respectively during the data setting phase as detailed below.

- **Data RAM:** The '**w0**' operation of the March C- algorithm can be transformed into an **R([T,i,x], D)** instruction with **D** being zero. The access of a subsequent **R([T,i,x], D)** instruction after the previous read miss will hit the cache and complete the '**r0**' operation. The interleaved write-and-check process of the March algorithm can be carried out by issuing consecutive **R** and **R** instruction pairs with ascending or descending ***i***. For the n-th pattern **DBn** in the data background set, the test sequence shown in Table 1 is applied for the traversal of the data RAM.

Table 1. The March sequence for data RAM.

Operation	Instruction sequence
wDB	DC; WM([T,i,x],DBn) EC; R([T,i,x],DBn)
rDB	R([T,i,x],data)

- **Tag RAM:** The transformed March algorithm is similar to the one used in data RAM except that the pre-generated data background pattern **DBn** is applied to tag (**T)** instead of a data line. To check if the tag **DBn** is correctly written into the tag array, complement data **!D** are written to address **[DBn,i,x]** in the main memory to set up the inconsistency. If the tag RAM at the ***i***-th line is faulty, then the **R([DBn,i,x],D)** instruction will fail, i.e., miss in cache, and return **!D** instead of **D**. Table 2 shows the March test sequence for the tag array.

Table 2. The March sequence for tag RAM.

Operation	Instruction sequence
wDB	DC; WM([DBn,i,x],D) EC; R([DBn,i,x],D)
rDB	DC; WM([DBn,i,x],!D) EC; R([DBn,i,x],D)

- **PA tag RAM:** The test patterns of the PA tag RAM are the physical page numbers. Therefore, extra steps must be taken in the initialization phase to properly set up the contents of the page table. For each virtual address tag **Tn**, a corresponding physical page number **DBn** is allocated in the page table. Table 3 shows the test sequence for the PA tag RAM. First, **W([Tn,i,x])** writes data D into cache entry (write-allocate) and **DBn** into PA tag. Then the complement data **!D** are written to the main memory location **[Tn,i,x]** by **WM([Tn,i,x],!D)**. After **CLEARi** writes back data **D** from cache to memory location **[Tn,i,x]**, the **R([Tn,i,x],D)** instruction will return **!D** if the PA tag memory is faulty.

Table 3. The March sequence for PA tag RAM.

Operation	Instruction sequence
wDB	EC; W ([Tn,i,x],D) DC; WM([Tn,i,x],!D)
rDB	EC; CLEARi DC; RM([Tn,i,x],D)

5.2. Test Sequence development for logic modules

The logic modules of the target data cache include several multiplexers, a dirty unit, a valid unit, a tag address comparator, and a cache controller. Next we describe the test procedures for each logic module.

- **Multiplexer**: The test vectors for an N-to-1 multiplexer can be efficiently derived using the scheme in [9]. Referring to Fig. 3, we can see that the inputs to the output multiplexer of the data RAM come from the entire data contents (W0 to W7) of the i-th data line and the word address x. The input vectors W0 to W7 can be loaded into data RAM using **R([T,i,xn], Wn)**, where **xn** denotes the offset of the n-th vector word **Wn**.

Then **R([T,i,xn], Wn)** can be issued to observe the test outputs.

- **Dirty unit:** For the x-th word in the i-th data line, the corresponding dirty bit can be tested by checking if the data D are correctly written to the main memory after a replacement occurs. The instruction sequence "**R([T,i,x], D), W([T,i,x], D)**" loads data D into the cache line and asserts the dirty bit. After the sequence "**DC, WM([T,i,x], !D), EC**" creates the inconsistency between the cache and memory, a **CLEAR*i*** or **R([T',i,x], !D)** can be used to trigger the replacement for observation.
- **Tag address comparator:** To obtain efficient vectors for the comparator, the netlist of the module is fed to the ATPG tool to generate the input pairs (A,B). The input **An** of the n-th test pattern can be loaded into tag RAM by issuing **R([An,i,x], D)** and then the comparison can be triggered by **R([Bn,i,x], !D)** for a cache miss or **R([An,i,x], D)** for a cache hit. The result of comparison can be observed from the acquired data **D**.
- **Cache controller:** The functions and related test sequences of the cache controller are listed in Table 4 without detailed explanation due to space limitation.

Table 4. Software-based test sequences for the cache controller.

Tested function	Test sequences
Enable/ disable	DC, WM([T,i,x],D), EC, R([T,i,x],D), DC, WM([T,i,x], !D), EC, R([T,i,x],D)
Invalidate	DC, WM([T,i,x],D), EC,R([T,i,x],D), DC, WM([T,i,x], !D), EC, INVi, R([T,i,x], !D)
Clear	DC, WM([T,i,x],D), EC, R([T,i,x],D), DC, WM([T,i,x], !D), EC, CLEARi, DC, RM([T,i,x],D)
Write-back policy	DC, WM([T,i,x],D), EC, R([T,i,x],D), DC, WM([T,i,x], !D), EC, R([T'i,x],D), DC, RM([T,i,x],D)
Write-through policy	EC, W([T,i,x],D), W([T'i,x],!D), DC, RM([T,i,x],!D), EC, R([T,i,x],!D)
Write-allocate policy	DC, WM([T,i,x], !D), EC, R([T,i,x], !D), DC, WM([T'i,x], !D), WM([T,i,x],D), EC, W([T'i,x],D), DC, RM([T'5p ,i,x], !D) , EC, R([T'i,x],D)
Write-around policy	DC, WM([T,i,x],D), EC, R([T,i,x],D), DC, WM([T,i,x],D), EC, W([T'i,x], !D), DC, RM([T'i,x],!D) , EC, R([T,i,x],D)

5.3. Example of assembly code for direct-mapped data cache

We use two examples to illustrate the relation between the pseudo instructions and the actual processor assembly code. Fig. 4 shows the assembly codes for March element "↑**wDB**" of data RAM and those for the "**Enable/Disable**" operation of the controller. In the program, R0 is used for cache enable/disable, R1 for address **[T,i,x]**, R2 for **D**, R3 for **!D**, R4 for entry size (64 in our cache), R5 for line size (8 in our cache), and R6 for read value. **EC** and **DC** are cache enable and disable macros.

For **wDB**, lines 01~ 06 set the required register; lines 08~09 (**DC**, **WM[T,i,x]**, **DBn**) set **D** in memory; lines 10~11 (**EC**, **R [T,i,x]**, **DBn**) use a cache miss to write **D** into data memory; lines 12~16 check two loop parameters to see if all data locations are processed. For Enable/Disable, the test sequences are similar to wDB except that the results are stored to memory for observation in lines 14~15.

```
; ↑wDB
01:  MOV R1, #DataBase ;[T,i,x]
02:  MOV R2, #0         ;DBn
03:  MVN R3, #0         ;!DBn
04:  MOV R4, #DCacheEntry
05:m_wDB_entry_1
06:  MOV R5, #LineSize
07:m_wDB_line_1
08:  DC  R0
09:  STR  R2, [R1]
10:  EC  R0
11:  LDR  R6, [R1], #WordInc
12:  SUBS  R5, R5, #0x1
13:  BNE  m_wDB_line_1
14:  MOV R5, #LineSize
15:  SUBS  R4, R4, #0x1
16:  BNE  m_wDB_entry_1
```

(a) wDB test sequence of data RAM

```
; Enable/Disable
01:  MOV R1, #DataBase ;[T,i,x]
02:  MOV R2, #0         ;D
03:  MVN R3, #0         ;!D
04:  DC  R0
05:  STR  R2, [R1]
06:  EC  R0
07:  LDR  R6, [R1]
08:  DC  R0
09:  STR  R3, [R1]
10:  EC  R0
11:  LDR  R6, [R1]
12:  CMP  R2, R6
13:  BLNE  error
14:  DIS_DC  R0
15:  STR     R6, [R1]
```

(b) Enable/disable test sequence of controller

Figure 4. Assembly code of wDB and Enable/Disable.

6. Experimental Results

To evaluate the efficiency of our methodology, we use ARM Development Suite (ADS) v1.2 [10] to assemble our test code, Cadence Verilog-XL [11] to run the logic simulation and capture stilumi for test analysis, Design Compiler [12] to synthesize our Linux-verified ARM-compatible processor [13] along with the target direct-mapped cache, RAMSES simulator [14] to run the RAM fault simulation, and Syntest Turboscan [15] to run the logic fault simulation.

6.1. Test Results for Memory Modules

For memory module testing, the processor performs March operation on memories just like an internal memory BIST tester. We apply our transformation methodology to March C- algorithm which can effectively detect all SAF, TF, AF, CFst, CFin, and CFid faults. The RAM fault coverage reported from RAMSES shows that the proposed software-based

approach has achieved the same fault coverage, i.e., 100 %, as the original March C- algorithm.

6.2. Test Results for Logic Modules

For logic module testing, the processor executes the test program to trigger and observe faults in the logic circuits. To verify our test program, we synthesize the direct-mapped cache into gate-level netlists using the TSMC 0.18um technology library, and then feed the netlists and input stimuli trace to the fault simulation tool. Our test program can detect 12656 out of 12840 uncollapsed faults (98.57% fault coverage) and 7613 out of 7766 collapsed faults (98.03% fault coverage). Table 5 shows the collapsed fault coverage and test efficiency of each cache logic module. The test efficiency which is obtained by removing the structurally untestable faults from the total number of faults has achieved 99.13%.

Table 5. Collapsed stuck-at fault test results.

Module	Det.	Und.	Unt.	Total	F.C.(%)	T.E.(%)
Controller	525	18	38	581	90.36	96.69
Comparator	142	1	2	145	97.93	99.30
Valid unit	1269	3	2	1274	99.61	99.76
Dirty unit	2528	2	0	2530	99.92	99.92
Multiplexer	2758	22	41	2821	97.77	99.21
Others	391	21	3	415	94.22	94.90
Total	7613	67	86	7766	98.03	99.13

Det. – # of hard detected faults; Und – # of undetected faults; Unt.– # of untestable faults; F.C.– Fault coverage; T.E.– Test efficiency.

6.3. Statistics of Test Program

Table 6 shows the test program size, storage used, and execution time. The total program size is less than 30KB and has an execution time of less than one million cycles with about 150KB memory. This software-based test scheme clearly shows the advantage in testing time and needed program space for current SoC designs.

Table 6. Test program statistic for data cache.

Target component	Program Size(KB)	Memory Usage(KB)	Execution Time(cycles)
Data memory	0.67	8.13	187,453
Tag memory	2.59	11.3	139,219
PA tag memory	4.64	22.4	178,769
Logic modules	21.10	106	490,581
Total	29	147.83	996,002

7. Conclusion

In this paper, we present a high-performance software-based testing methodology to support the testing of a direct-mapped data cache. Unlike previously proposed methods which mainly focus on the cache functionality, we explore additional information from architectural analysis, RTL description, and netlist analysis for effective test program generation. We apply the methodology to a real-life cache design and the results show that the transformed March test program can detect all commonly used RAM faults on all cache memory modules and the derived test program for logic modules can achieve 99.13% test efficiency for collapsed stuck-at faults.

8. Acknowledgement

This work was supported in part by the National Science Council, Taiwan, R.O.C., under NSC 96-2221-E-006-192-MY3 and NSC 95-2220-E-006-005, and by the Program for Promoting Academic Excellence of Universities in Taiwan.

9. References

[1] S. Kornauck, L. McNaughton, R. Gibbins, and B. N.Dostie, “A High Speed Embedded Cache Design with Non-intrusive BIST”, in Proc. Int’l Workshop Memory Technology, Design, and Testing, pp. 40-45, 1994.

[2] J. Bralich and J. Fleischman, “Design of cache test hardware on the HP PA8500”, *IEEE Trans. Design & Test*, vol. 15, issue 3, pp.58-63, July 1998.

[3] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded Processor-Based Self-Test*, Springer press, 2004.

[4] A. J. van de Goor, Testing Semiconductor Memories: Theory and Practice, Wiley Publisher, 1991.

[5] A. J. van de Goor and Th J. W. Verhallen, “Functional Testing of Current Microprocessor (applied to the Intel i860TM)”, in Proc. Int’l Test Conf., pp. 684, 1992.

[6] J. Sosnowski, “In System Testing of Cache Memories”, in Proc. Int’l Test Conf., pp. 384-393, 1995.

[7] 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 VLSI Test Symp., 1998, pp. 394-400.

[8] ARM922T Technical Reference Manual, ARM Corp., Available: http://www.arm.com

[9] S.R. Makar and R.J. McCluskey, “On the Testing of Multiplexer”, in Proc. Int’l Test Conf., pp. 669-679., 1988.

[10] ARM Development Suite v1.2, ARM Corp. http://www.arm.com

[11] Verilog-XL, Cadence Corp., http://www.cadence.com

[12] Design Compiler, Synopsys Corp., http://www.synopsys.com

[13] C. H. Chen, C. K. Wei, T. H Lau, and H. W. Gao, “Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores”, *IEEE Trans. VLSI*, vol. 15, issue 5, pp. 505-517, May 2007.

[14] C.-F. Wu, C.-T. Huang, and C.-W. Wu, “RAMSES: A Fast Memory Fault Simulator”, in Proc. Int’l Symp. Defect and Fault Tolerance in VLSI Systems, pp. 165-173. , 1999.

[15] TurboScan, Syntest Corp., www.syntest.com