Functional Verification of the Equator MAP1000 Microprocessor

Jian Shen*, Jacob Abraham
Computer Engineering Research Center
The University of Texas at Austin
Austin, Texas

Dave Baker, Tony Hurson, Martin Kinkade,
Gregorio Gervasio, Chen-chau Chu, Guanghui Hu
Equator Technologies Inc.
Austin, Texas

Abstract

The Advanced VLIW architecture of the Equator MAP1000 processor has many features that present significant verification challenges. We describe a functional verification methodology to address this complexity. In particular, we present an efficient method to generate directed assembly tests and a novel technique using the processor itself to control self-tests and check the results at speed using native instructions only. We also describe the use of emulation in both pre-silicon and post-silicon verification stages.

1 Introduction

The complexity of modern microprocessors has grown dramatically in recent years making design verification a huge bottleneck for large chip designs. In many companies, verification efforts consume most of the design resources and there are more verification engineers than designers, making verification the real limiter of time to market [1]. Simulation-based verification is the primary means for functional verification. The ever increasing line densities and operating frequencies, together with increasing amount of interacting functional modules on a chip have created new and more complicated problems. Post-silicon at-speed verification has become crucial to exposing failures. Recent publications show that pseudo-random test generation techniques have been the backbone of the verification effort in the industry [1–9]. All of these recent work dealt with superscalar processors. We will present our functional verification methodology and results, particularly at-speed verification, for the Equator VLIW media processor.

Equator Technology Inc. was founded in 1990 by the original members of Multiflow, one of the pioneers in VLIW architecture. Equator has been developing a multimedia processor coded MAP1000 that handles video, audio, imaging and communications. The MAP1000 processor is supported by advanced compiler techniques that allow program developers to do their work in C and still get fast performance. The Advanced VLIW architecture of the processor has many features that present significant verification challenges. These include a large amount of functional units, a large and comprehensive instruction set, a complicated multi-ported data cache, a multi-ported DMA mechanism, and complicated bus/IO interface units with support for various clock domains.

There are two salient features of the MAP1000. First, the MAP1000 has much more complex cache control and DMA mechanisms, compared with the superscalar processors. Since there are multiple operations per instruction, the memory system must support multiple functional units. In addition to pseudo-random tests, directed tests targeting the memory system microarchitecture are essential. Due to the VLIW architecture, much of the complicated mechanism for hazard detection/scheduling is moved from hardware to software (compiler). Thus, the instruction scheduling block is not the threshhold for the MAP1000, in contrast to superscalar processors. Second, the MAP1000 has strong software support. The MAP1000 processor is supported by advanced compiler techniques and application software development efforts. The efforts directly contribute to the functional validation and performance verification. In this paper, we will address these issues and describe our unique methodology.

In Section 2, we give an overview of the verification methodology. In Section 3, we present the technique of directed test generation. In Section 4 we introduce the technique for at-speed post-silicon verification. The results are presented in Section 5, followed by the conclusions in Section 6.

2 Functional Verification Methodology

The term "functional verification methodology" used in this paper refers to an integrated set of techniques in a unified process to produce an error-free microprocessor design. Figure 1 shows the overview of the methodology.

* Jian Shen was supported in part by an internship at Equator Technologies Inc. and in part by the Texas Technology Development & Transfer Program under Project 003658-433 at the University of Texas at Austin.

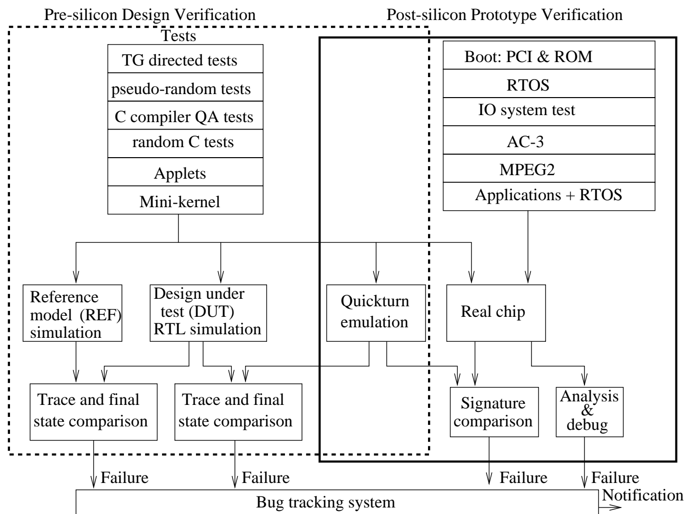

Figure 1. Functional verification flow.

2.1 Verification Environment

2.1.1 Models

The simulation based verification environment mainly consists of a register transfer level (RTL) design of the MAP1000 and a reference model. Both the design and reference model include the memory. This allows both models to execute programs as the real chip does in a system. Later in the verification stage, the RTL design is synthesized into a gate-level design - a model on an emulation system.

The reference model (REF) needs to be fast and correct. Prior to the development of the RTL model (DUT), one instruction set level reference model written in C and a more accurate version had been extensively used for architecture-level analysis. As the development of RTL continued, a cycle accurate reference model was developed. Although the cycle accurate REF models the pipeline stages, caches and some of the other internal memories, it represents a higher abstraction level than the RTL and is able to execute at a speed over 100 times faster the RTL model.

There are three common checking methods for simulation: assertion checkers inside the RTL model, self-checking tests, and comparisons between the RTL model and the reference model [2]. These assertion checkers can only be applied to some events. The method of self-checking tests is not fine grained and disturbs the code sequence with a large amount of self-checking instructions. Since we have a reference machine with enough details, we relied on the trace comparison between the design and simulation model for the majority of our effort. Only when targeting a block for which the REF was not accurate enough, did we resort to self-checking tests on the RTL.

2.1.2 Bug Tracking System

In the verification database, in addition to the diagnostic test tree, there is a bug tracking system with a Web interface. A reported bug has several attributes which can be queried. The *source* is the location or nature of the bug, such as the integer unit, the floating point unit, or occasionally the assembler. The *status* can indicate “new”, “fixed”, “work in process”, “rejected” or “completed”. The *owner* is the person responsible for fixing the bug. The *cause* is the means by which it is detected, such as directed test, random tests, emulation, etc. The bug statistics are periodically posted on the Web. Whenever the status of a bug changes, or a regression result becomes available, the system automatically notifies corresponding designers and verification engineers.

2.1.3 Queueing System

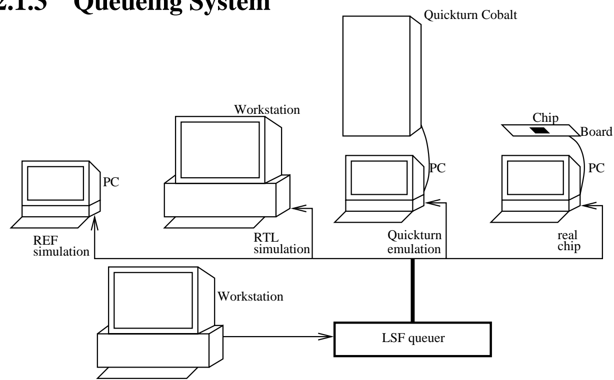

Figure 2. Queueing for all verification jobs.

To develop the leading edge processor MAP1000, sharing resources and equitable distribution of the processing load is not only a need, but also a critical requirement for being productive. A job queueing system should offer large job throughput, reliability, fault tolerance and flexibility in terms of type and number of the jobs the system is able to handle. We chose LSF from Platform Computing Corporation for this purpose. With the assistance of LSF, thousands of jobs were run daily, from either individual designers or regressions. We had dozens of queues with different priorities and number of jobs allowed. For example, there are regression queues: regression-daytime, regression-night, 24-hour-random, etc. Although regressions and interactive debugging took a lot of processing power, every design engineer had ready access to sufficient computing resources.

We have a unified queueing system, as shown in Figure 2. The RTL simulations were queued to run on workstations and REF simulations were sent to PCs. The Quickturn Cobalt system and the real chip each connected to a PC were eventually added under LSF control. This unified queueing system increased productivity significantly and allowed the groups in Campbell, Austin, Seattle and Irvin to work on different tasks “concurrently”, fully utilizing the expensive resources.

2.2 Test Case Sources

A test case is a diagnostic test program to be run on the RTL model of the processor to check a particular area of functionality. According to a report at Hewlett-Packard [3], running test code from as many sources as possible was of high value for microprocessor verification. Each test effort has its own focus and unique value, but each also has its blind spots. A large overlap in coverage of different tests proved to be an invaluable safety net against the limitations and blind spots of individual test generators. To address the verification complexity of the processor, we relied on test case sources including directed tests, pseudo-random tests, chip level tests converted from block level test vectors, C compiler quality assurance (QA) tests, random C tests, mini-kernel and real media applets (small application programs). The random C tests were generated by a modified compiler which inserted random harmless codes into a C source program to stress the hardware, adding more CPU and memory traffic loads.

2.3 Verification Process

There were two main stages of functional verification: the pre-silicon design stage and the post-silicon prototyping stage.

2.3.1 Pre-silicon Design Verification

At the beginning of the pre-silicon design verification stage, we mainly relied on directed assembly tests targeting each area of the functionality. In Section 3, we describe our efficient technique to generate directed tests. We were able to complete the directed tests for the core in a period of six weeks. These thousands of tests became the first part of the regression suite. Tests for the other blocks of the chip were also generated using this technique. Every morning the designated designers and members of the verification team were notified about the status of the nightly regression. The detailed information included the number of pass/fail tests, categories of failures (DUT fails or runaways, REF fails or runaways, final states mismatch, traces mismatch).

Compared to superscalar processors, VLIW processors place more programming constraints on assembly programs. When most of the directed tests passed and the RTL was stabilized, the pseudo-random assembly test generator also gathered enough knowledge about the processor microarchitecture. We then used the pseudo-random tests, compiler QA C tests, and chip level tests converted from the block level. Special dedicated queues for random assembly tests and random C tests issued batch jobs every 30 minutes. A failing test would be copied into a repository and reported to the bug tracking system.

The verification and design teams jointly created test plans for each block of the processor. Directed tests were used to complete the micro-architectural level test plan. Finally, we used the commercial tool Vericov to measure the verification coverage in terms of RTL sequential code block covered, under the assumption that untested blocks potentially have bugs. While high block coverage doesn't guarantee the completeness of the test suite, it is mandatory. Vericov showed most of the area on which this technique focused to be over 90% covered (sequential block coverage). Most of the sequential RTL code blocks not covered were those error checkers which were not supposed to be asserted in a normal case.

As the bug curve became flat, we started a major effort to set up the Quickturn Cobalt emulator for the synthesized design. Since Equator was one of the first companies to load such a complex system on a Quickturn Cobalt box and the project had a very tight schedule, this was a challenging goal for the MAP1000. Engineers from different groups such as verification, design, system engineering, OS, and Quickturn field engineers joined the effort on a 24-hour basis. The Quickturn model became fully functional in an impressively short period of time, and was plugged into a system to demonstrate media applications. Thus the functional Quickturn model not only was an important milestone for the functional verification, but also served for software prototyping and demonstration for investors and customers. In addition, dozens of problems not found on the RTL level simulation, such as combinational loops and unconnected ports, were found much earlier than tape release.

2.3.2 Post-silicon Prototype Verification

The techniques in the pre-silicon design verification were adequate to find nearly all of the bugs in the processor and produced a working first pass silicon useful for prototyping. However, there are three major reasons for post-silicon verification. First, there might be design bugs that elude the comparison between the processor design and the reference model, due to bugs either in the RTL simulator or/and the reference model. Second, by attempting to run at high speed, the chip will expose subtle bugs that only show up at high speed or bugs preventing the chip from operating at the target speed. Third, some bugs need a large number of cycles or interaction with other devices to be exposed.

However, it is much harder to find the internal signals in a chip. There are much fewer capabilities in post-silicon debugging, aggravated by the difficulties in pass/fail decision and the generation of a large number of tests for the fast real chip. We applied the regression test suite to the chip on a tester and compared the response with the pre-captured pin-output. Also, we developed a native mode self-checking technique to generate self-checking tests. This will be discussed in detail in Section 4, and elsewhere.

A major effort during post-silicon verification was running real media applications using the chip on a board. The in-house "pciplay" utility provided the debug environment for both Quickturn and the chip. Through the PCI bus, we could effectively use a PC to access the internal memories

of the chip or the Quickturn model on the board. We ran a sequence of tests on the chip. First, directed IO tests. Second, AC-97 audio, modem interface, NTSC in/out, transport channel, SVGA out, USB host. Third, OS boot. There are two ways to boot the chip, one may either boot the chip stand-alone using a flash ROM, or boot the chip through PCI. We successfully booted the chip both ways and ran the Real Time Operating System (RTOS). Last, media applications: stand-alone of MPEG, AC-3 video and audio decompression, and both video and audio integrated with RTOS.

The extraordinary achievements of the software groups should be mentioned. All of the application and operating system software had been verified on the reference model, and no major software bugs were found during post-silicon verification.

3 Directed Test Generation

Publications in recent years show that more companies are exploiting pseudo-random test generators as test case sources [1–9]. However, most of these test generators were developed over years based on several progressive architectures in a company. They have high development cost, for example, the one at IBM cost $3,000,000 [4]. Since the MAP1000 is the first processor at Equator, and due to its VLIW architecture, we decided to rely mainly on the focused or directed tests. However, manually generating the tests for the complex processor with a large instruction set and a large number of functional units could be very time consuming. To meet the verification challenge, we developed an efficient test creation method, with the type of test called TG. Native assembly languages and Perl language are combined to provide full control and efficiency during test generation.

3.1 Operand Selection

Most of the directed tests target a group of instructions, or an area of the functionality. To test each instruction comprehensively using a compact test, operands must be selected systematically. We studied the instruction set, and designed a set of operands. For each immediate field in the instruction set, we chose a set of immediate numbers, then categorized and listed them in a library.

For example, for the 8-bit immediate operands, we chose the Reed-Muller code: $\{11111111, 11110000, 11001100, ..., 00001111\}$. Similarly, for the general register operands, we also constructed their corresponding array of patterns, representing the general and special values. For floating point operations, we added floating point special numbers such as *NaN, Normalized number, Infinity, Zero*, etc. In a *TG* test, if the routine for the value patterns has an index parameter, the indexed array element will appear in the compiled assembly test. Otherwise, a random pattern fitting the instruction field requirement is generated. By pseudo-enumerating the combinations of the short operand arrays, we verified individual instructions comprehensively.

3.2 Instruction Set Categorization

Similarly to the immediate operands, we also listed all the instructions in the library categorized by their functionality and instruction syntax. For example, the integer arithmetic operation *op_iarith* array contains the following instructions: add.32, adds2.32, adds4.32, adds8.32, etc.

3.3 Macro Construction

The standard assembly program preamble and postamble were written as subroutines into the library. Similarly, basic operations such as initializing the data TLB and comparing two lists of registers followed by a branch, were also written as macros in the library. Higher and higher levels of subroutines were gradually constructed. As a result, writing a test program testing an area of functionality was almost as efficient as writing a C program, but with direct control on instruction sequence, operands, address, etc.

The MAP1000 has a complicated multi-ported data cache. To validate the data cache controller, a suite of micro-architecture level *TG* tests was written manually. It would be unlikely for a simple pseudo-random generator to cover all the corner cases, and it would be too inefficient to write all the cases manually.

3.4 *TG* Tests

The TG tests are highly readable and compact. Figure 3 shows an example of a manually created TG test and the compiled native assembly test. In addition to manually writing the tests, stand-alone programs or scripts were used to generate such tests. For example, to test the DMA engines between multiple memories, we progressively built subroutines at higher and higher levels. Then a test generator was able to create hundreds of tests automatically exercising dozens of parameters comprehensively. This approach has proven to be much more efficient than writing the assembly or C tests manually, and offered more control and ease than a dedicated test generator for the processor like the one in [4].

```
#———TG test ———————————————————#Extended assembly test——
for($i=0; $i < 64; $i++){                        LDI(r0, 0xffffffff)
$j = $i% 12;                                     LDI(r1, 0xffff0000)
;LDI(r$i, $tglib::imm32[$j])                     LDI(r2, 0xff00ff00)
}                                                ...
for($i=0; $i < $nr; $i++){                       LDI(r63, 0x33333333)
$j = ($i+32)% 64;                                instr cl0 add r0, r0, r32;
$op = int(rand(10));                             instr cl0 sub r1, r1, r33;
;instr cl0 $tglib::alu[$op] r$i, r$i, r$j;       instr cl0 add r2, r2, r34;
}                                                ...
                                                 instr cl0 xor r63, r63, r31;
```

Figure 3. A sample of the *TG* test and compiled assembly test.

4 Native Mode Self-Test

One major difference between pre-silicon design verification and post-silicon prototype verification is the difficulty of the test pass or failure decision. During pre-silicon design verification the simulation trace of the design is compared with that of a reference model. For a post-silicon test, there is no trace of the test. However, modern processors have large memory modules, register files and powerful ALUs with comprehensive instructions, which can be used to generate and control at-speed self-test and to evaluate the response of the tests at speed [5]. Based on the instruction set and architecture of the processor, we can use a native assembly routine to compute the signature of the processor internal state represented by general registers. The signature computation routine can be stored in the main memory. To avoid interrupting the normal control flow of the test program, the instruction sequence can be scheduled carefully such that the effects of the sequence are spread over many registers. Then, signature compression is done for all the registers, instead of compressing the target registers after each normal operation under test. By systematically incorporating the signature computation routines into the functional test sequence, we can achieve at speed self-test.

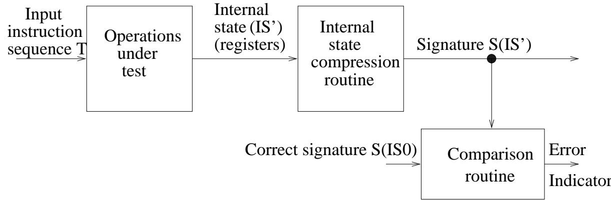

Figure 4. Native mode signature computation.

The compression algorithm is similar to a multiple input signature register (MISR) used in built-in self-test (BIST). We use the ALU shift and exclusive-or operations to implement the multiple input signature compression for registers. We define this process as "native mode signature computation", as shown in Figure 4. Only native instructions of the processor are involved in compressing the responses, and no modification of the processor circuitry is necessary. Thus our method has no area overhead or performance impact, in contrast to conventional built-in test approaches. Since the functional tests are designed to be closed loop tests without the need for a tester, we can run them at native speed. The functional test takes much less time and avoids the cost of expensive test equipment. At the same time, storing the test program in the main memory significantly reduces the memory capacity needed when compared to storing it to the tester memory [6].

We can use prime characteristic polynomials to minimize the probability of fault masking. For a processor with 32-bit registers, if we choose 32-bit parallel load signature compression, the probability of masking is only approximately 2^{-32}. The compressed signature can be compared periodically with a predetermined golden signature to further reduce the probability of masking a fault. We can rerun the test but with a different characteristic polynomial by changing the compression routine. This flexibility is an advantage over using hardware signature analyzers, in addition to the savings in area and performance overhead. Compared with random testing using BIST hardware, our technique is more helpful for failure analysis, since the tests are functional test vectors. To debug a failure and identify the failing units, we can increase the frequency of signature comparison by a divide-and-conquer method. Existing software based self-test techniques as in [7] require a significant amount of memory storage for holding the correct outputs associated with test vectors. Our native mode signature compression technique saves on both memory storage and bandwidth.

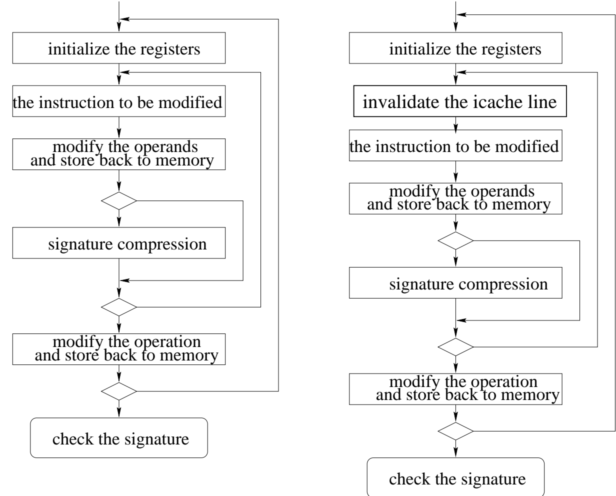

Figure 5. Self-test with self-modifying code.

In addition to monitoring a test, the processor under test can also be used to generate self-test without any BIST hardware logic. A test which contains self-modifying code and nested loops can effectively generate numerous operations at run time. Thus a short test with dozens of instructions may stress a particular group of operations thousands of times at high speed. Consider, for example, a generic assembly instruction: add r1, r2, r3. The opcode for the instructions is $\{00000000000001, 000001, 000010, 000011\}$ with bits[31:18], bits[17:12], bits[11:6] and bits[5:0] corresponding to the operation (add), the destination register (r1), the first and second source registers (r2 and r3), respectively. Increasing or decreasing the operand fields will change the registers to be used in the instruction; increasing or decreasing the operation field will exercise other related instructions, such as add.s32, add.u32, sub, sub.s32, sub.u32. A simple program flow to generate tests and periodically compress the signature is shown in Figure 5(a).

If the instruction cache is enabled, the cache invalidation instruction should be included in the program, as shown in Figure 5(b). By modifying the instruction in the memory and invalidating the cached instruction, we verified the instruction cache invalidation operation, and generated tests at run time. We also used the method at the RTL level during pre-silicon verification, and detected design errors in the instruction cache invalidation operation. This bug was fixed prior to tape release.

We applied the method to the MAP1000 prototype processor. One group of tests loaded the test program into a chip internal buffer of 4K bytes. This technique has proved to be highly effective. It enabled the chip to execute ALU operations numerous times running a small program stored in this 4K buffer. The signature thus obtained was compared with that from the Quickturn model. Again, the Quickturn emulation has proved to be of high value, providing the signature for a test in a few minutes, while RTL simulation takes hours.

5 Results

The effective bug detection mechanism is demonstrated on Figure 6. It shows that more than half of the bugs were found using the directed tests. It should be noted that many problems detected when setting up the Cobalt system were not documented here, partially because they were not functional errors of the design. But the Colbalt system did speed up the tape release. The goal of functional first pass silicon has been achieved. The chip was able to execute all the instructions. The chip on a board could boot the RTOS operating system and perform media applications, such as MPEG2 video decompression, AC-3 audio decompression, and both audio and video decompression integrated with the RTOS.

Directed tests	69%
Random assembly tests	16%
Mini kernel	1%
OS boot	1%
DVD applet	2%
3D applet	2%
2D applet	1%
Emulation	1%
Others	7%

Figure 6. Effectiveness of tests.

6 Conclusions

In this paper, we have described the verification of the MAP1000 – a world-class media processor. The directed test generation technique has been proven to be highly efficient and effective. Also, we have presented the post-silicon native mode self-test technique. We used the processor to generate tests at run-time by self-modifying code, and to perform signature compression using native instructions only. The processor compared the signature with the one obtained from emulation. To the best of our knowledge, this is the first time such a technique has been applied to a commercial processor. All the above mentioned techniques have contributed to the success of a first-silicon running processor.

Quickturn emulation was an important verification technique for both the pre-silicon and post-silicon stages. However, setting up the Cobalt system required a significant hardware load.

Acknowledgements

The authors would like to thank all of the different teams who contributed to the successful functional verification of the MAP1000. Special thanks to all of the individuals in the verification group (particularly Andrew Peebles, Anoosh Hosseini, Krist Roginski, Murali Chinnakonda, Steve Dougherty, George Moussa and Tohru Nojiri), the design group (particularly Sarang Paldalkar and Mayur Mehta), the system group, the OS and application groups (particularly Rhadika Thekkath and Wim Colgate).

References

[1] T. B. Alexander, K. A. Dickey, D. N. Goldberg, R. V. La Fetra, J. R. McGee, N. Noordeen, and A. Prakash. Verification, characterization, and debugging of the HP PA 7200 processor. In *Hewlett-Packard Journal*, pages 1–12, February 1996.

[2] M. Kantrowitz and L. M. Noack. I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor. In *Proc. of the Design Automation Conf.*, pages 325–333, June 1996.

[3] S. T. Mangelsdorf, R. P. Gratias, R. M. Blumberg, and R. Bhatia. Functional verification of the HP PA 8000 processor. In *Hewlett-Packard Journal*, pages 1–13, August 1997.

[4] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proc. of the Design Automation Conf.*, pages 279–285, June 1995.

[5] J. Shen and J. A. Abraham. Native Mode Functional Test Generation for Microprocessors with Applications to Self Test and Design Validation. In *Proc. Intl. Test Conf.*, pages 990–999, 1998.

[6] C. Hinchcliff. Simplified Microprocessor Test Generation. In *Proc. Intl. Test Conf.*, pages 176–180, 1982.

[7] A.J. van de Goor and O. Jansen. Self Test for the Intel 8085. In *Microprocessing and Microprogramming*, 29:165–175, 1990.