A Hybrid Software-Based Self-Testing methodology for Embedded Processor

Tai-Hua Lu
Department of Electrical Engineering,
National Cheng Kung University,
Tainan, Taiwan
aaron@casmail.ee.ncku.edu.tw

Chung-Ho Chen
Department of Electrical Engineering,
National Cheng Kung University,
Tainan, Taiwan
chchen@mail.ncku.edu.tw

Kuen-Jong Lee
Department of Electrical Engineering,
National Cheng Kung University,
Tainan, Taiwan
kjlee@mail.ncku.edu.tw

ABSTRACT

Software-based self-test (SBST) is emerging as a promising technology for enabling at-speed testing of high-speed embedded processors testing in an SoC system. For SBST, test routine development or generation can base on deterministic and random methodology. The deterministic test methodology develops the test program for a pipeline processor using the information abstracted from its architecture model, RTL descriptions, and gate-level net-list for different types of processor circuits. The random test methodology tries to make the pseudo-exhaustive testing possible using random instructions or patterns. The proposed methodology improves coverage for structural faults using both deterministic and random development of the test code. Not only can the deterministic test program test lots of faults using very small code size, but also the random test program can help detect some of the faults that the deterministic test program is difficult to test. We demonstrated the feasibility of the proposed methodology by the achieved fault coverage, test program size, and testing cycle count on a complex pipeline processor core. Comparisons with previous work are also made. Experimental results show its potential as an effective method for practical use.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture - Embedded processor testing

General Terms

Measurement, Design, and Experimentation.

Keywords

Embedded processor testing, fault coverage, functional testing, software-based self-test.

1. INTRODUCTION

With rapidly advanced semiconductor manufacturing technology, more and more embedded processors are integrated in an SOC. However, the poor controllability and observability of processors make the testing of an SOC system more and more difficult. Traditional scan-based test architectures are applied to embedded processors to achieve high fault coverage, in which scan registers are inserted to the designed circuits to improve the controllability and observability. However, these test methods usually cause area overhead, and performance degradation [1]. Therefore, software-based self-test (SBST) becomes increasingly popular for embedded processor testing [2].

In general, test generation of the software-based methods can be classified into two categories: deterministic development and random generations. Since early 1980's, several graph-theory based test generation algorithms for microprocessors have been proposed which have established the framework of functional testing using deterministic patterns [3-6]. Much work related to deterministic test generation has been presented thereafter. A deterministic methodology to test the data path modules of a processor is developed in [7]. It can use the same test program and operands to test any architecture of adders and multipliers. In [8] Shen and Abraham describe a language that represents every instruction in an instruction set to simplify the test generation procedure. In [9][10], criterions to classify instructions according to the functionalities of components in the processor are provided. In [11], a genetic algorithm-based system is used to generate test programs for microprocessor cores. Recently, several new methodologies were developed to extract functional constraints for modules of a processor from HDL description. Commercial ATPG tools can then be used to generate test patterns of the modules with the extracted constraints. The patterns are then translated into instructions [12]–[15]. For the random pattern generation, several methodologies have been developed [16][17][18]. In [16], a LFSR is used to generate random numbers as random instructions, and a filter is designed to replace each illegal instruction with a legal one. A randomizer in [17] fills a randomized instruction in memory after the microprocessor fetches the instruction previously stored at the same address. Therefore, the microprocessor will get a different instruction at the next time from the identical memory address. In [18], a test generation tool, FRITS, for Intel Pentium 4 and Itanium family was introduced.

In this paper we present a software-based self-testing methodology using deterministic and random development of the test code for embedded processors. The deterministic test methodology explores the different types of circuit based on the design information from the processor architecture, register-transfer-level, and gate-level for SBST development and uses the information to improve coverage for structural faults. Our deterministic test methodology uses an ATPG tool to generate the constrained test patterns for testing the simple combinational fundamental IPs used in the processor. The deterministic

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.
Copyright 2008 ACM 978-1-59593-753-7/08/0003…$5.00.

approach refers to the RTL code for the rest of the control and steering logic in the processor for test routine development. The pseudo-random test methodology is carefully developed and tries to make the pseudo-exhaustive testing possible. The concept of basic block is used to form the random test programs. Basic blocks provide reasonable instruction sequences like the normal user programs that perform: load to register file from memory, arithmetic or logical operations, result store of the operations, and control transfers. We use pseudo-random test program to detect the faults that the deterministic test programs are unable to test. Finally, we combine deterministic test program with pseudo-random test program to test embedded processors.

The SBST methodology is demonstrated by the achieved fault coverage, test program size, and testing cycle count on a Linux-verified pipeline processor core that incorporates pipeline hazard control and operand forwarding. Comparisons with previous work are also presented. The rest of this paper is organized as follows. Section 2 discusses related researches. Section 3 presents the proposed methodology. Section 4 shows the experimental results and comparisons with other SBST-based schemes. Finally, Section 5 gives the conclusions of this paper.

2. PREVIOUS WORK

Software-based testing was proposed by S. M. Thatte and J. A. Abraham first [3]. They proposed the graph-theory based test generation algorithm for microprocessors in 1980. The test method is functional testing without any knowledge of the processor structure. However, low fault coverage is the main issue of using software-based testing [2-11][16-22]. In [2], the authors analyze the RTL descriptions of a processor, and then choose instructions and operands to form the test routines according to their test library. This approach is carried out at a high abstraction level and gate-level information is not required during test development. There are three phases in their methodology. First, they extract information of the processor. In the second phase, they classify the processor components into three classes, functional components, control components, and hidden components. In the final phase, the most appropriate instructions and operands are selected for the considered components. The authors use two processor cores implementing the MIPS instruction set to evaluate their proposed methodology. Their work report fault coverage of 95.3% for a 3-stage pipelined implementation in a synthesis case resulting in a gate-count of 26080. For a more complicated implementation of the same instruction set that has a 5-stage pipeline, a fault coverage of 92.6% is achieved in a synthesis case resulting in a gate-count of 37402. Deterministic SBST has also been used in testing the specific functional units of a processor such as the ALU, multiplier-accumulator, and shifter [7].

The approach in [21] uses pseudo-random patterns to test a processor component by component. There are two steps in their self-test scheme, the test preparation step and the self-testing step. In the test preparation step, test patterns for a component are developed using random number generator under constraints imposed by instructions. The consideration of constraints is to make sure that the generated patterns are realizable by processor instructions. Alternatively, the patterns can be generated by ATPG under constraints. In the case of random patterns, these patterns are encapsulated into signatures. In the self-testing step, the generated patterns are applied to the considered components by a software tester. First, the on-chip test generation program (a software LFSR) emulates a pseudo-random pattern generator and expands the signatures into test patterns. Then the test application program applies these patterns to the considered components. Finally, the responses are collected and analyzed by the response analysis program. The authors use an 8-bit Parwan processor core which consists of 888 equivalent NAND gates and 53 flip-flops as the target processor. The experimental result shows that fault coverage of 91.42% can be achieved.

Apart from SBST for processor cores, full scan approach has been used in testing a fully synthesizable processor core [1]. The downside of scan-based approach comes from the processor area overheads and possible performance degradation.

3. THE SBST METHODOLOGY

In this section, we first present the overview of the proposed SBST methodology. Then, we introduce the deterministic approach followed by random test code development. An early version of the deterministic method has appeared in [23]. Figure 1 shows the SBST methodology for processor cores consisting of four stages of work as follows.

- Stage 1: Deterministic test program development. The first stage is to classify the above mentioned processor parts based on the abstraction model that includes processor ISA, pipeline architecture, and synthesized gate-level processor core. Then, construct the deterministic test program by each individual part, which can be conducted in parallel.
- Stage 2: Random test program development. The second stage is to generate a random program using constrained basic block.
- Stage 3: Fault simulation. The third stage is to combine deterministic test program with random test program for fault simulation. If the fault coverage is not accepted, repeat stage 1 or 2.

We explain the detail of our SBST strategy in the following subsections.

3.1 Stage 1: Deterministic test routine development

Before deterministic test code development, we classify the processor components into the four functional groups as illustrated in Section 3. The concepts of the deterministic test routine development methodology are as follows.

- Synthesized codes
  - i. Tests for programmer-visible registers
    1. Consider synthesized state elements.
    2. Consider fault propagation.
  - ii. Tests for Fundamental IPs
    1. IP extraction
    2. Constraint setting
    3. Perform ATPG and develop test sequences.
- RTL codes
  - i. Tests for control and steering logic and pipeline registers
    1. Cover each statement, branch, and condition.
    2. Consider fault model and fault propagation.
- System architecture

i. Tests for pipeline features
    1. Control hazards.
    2. Forwarding.

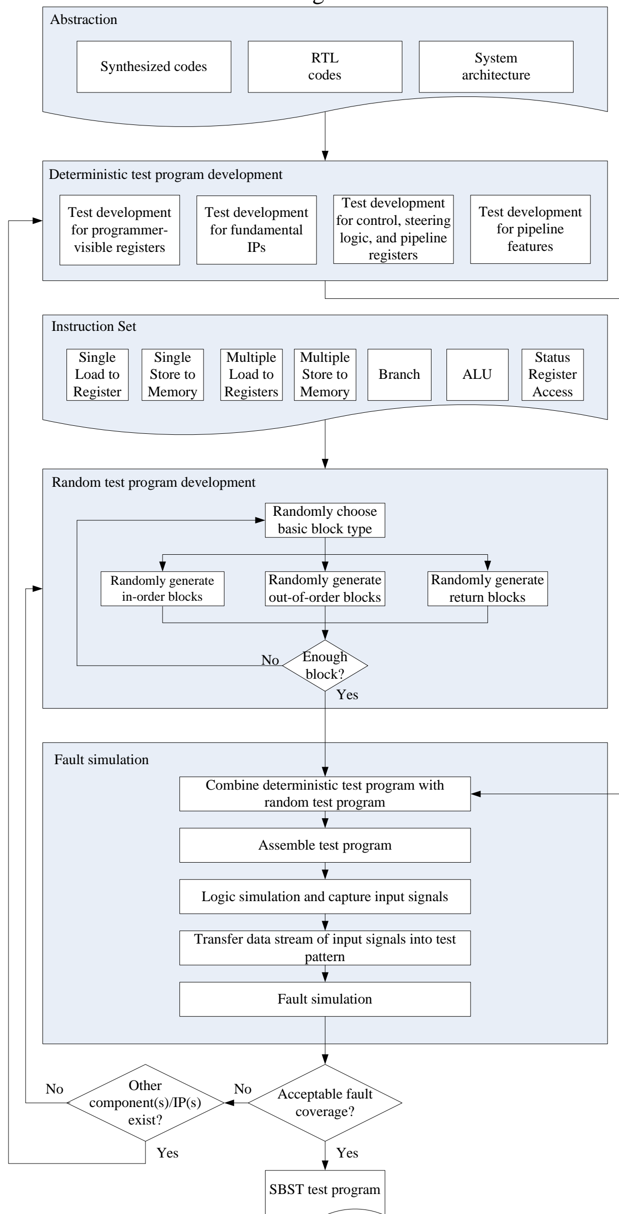

Figure 1. Overview of the proposed software-based self-test methodology

3.1.1 Test routine development for programmer-visible registers

We derive the test sequences for programmer-visible registers according to the information in the gate-level and the processor pipeline architecture. When using the information from these two levels, the deterministic test routines development can be easier without the tedious manual efforts. The programmer visible registers are typically mapped into D flip-flops after synthesis. Some of the faults on the D flip-flops, such as the faults on the write enable pins, are invisible in this level and thus are probably ignored and untested by the test program based on the high RT-level. In addition, information of architectural level such as the pipeline forwarding depth between the EXE stage and the WB stage must also be used in order to develop useful test sequences for the general purpose registers.

In a synthesized processor core, the programmer-visible registers in RTL descriptions are synthesized into clocked D flip-flops. Hence, it is simple and effective to develop the test routine which focuses on the structural faults of the D flip-flops' input/output interface with the consideration of the processor pipeline architecture, i.e., the pipeline forwarding structure.

3.1.2 Test routine development for Fundamental IPs

Among these combinational logics, some of them are the fundamental intellectual properties (IPs) introduced by the synthesis tool. The fundamental IP refers to small components such as adders, multipliers introduced by related RTL statements. For example, if we write an addition statement in the hardware description of a processor, the synthesis tool will insert an adder into the design to realize the addition function. In a processor core, the fundamental IPs mainly come from two components, the ALU and the memory access unit. For these simple fundamental IPs, the effective strategy for test routine development is to rely on the ATPG tool for the generation of efficient test patterns with only simple constraints. The test routine for a fundamental IP can be easily designed without struggling in the complex constraint setting process.

After the fundamental IPs are extracted from the processor core, the next step is to perform constrained ATPG and generate the dedicated test patterns for each fundamental IP. We use ATPG just for fundamental IPs that are much smaller and less complicated than components or modules. A component or module may include control logics, multiplexers, or registers and has heterogeneous inputs. These features will make constraint setting and pattern application more complicated. On the other hand, a fundamental IP is simply a small computational component, such as an adder or subtractor, without any control logic or memory elements. So the constraint setting is relatively easy or even not required at all.

3.1.3 Test routine development for control, steering logic, and pipeline registers

Apart from the programmer-visible registers and fundamental IPs, the control and steering logic, including the distributed logic gates, multiplexers, and pipeline registers, are considered. For the distributed logic gates and multiplexers, they are difficult to be identified with the structural information from the gate-level net-list. Pipeline registers are not programmer accessible. To test these units, we explore the instruction sequences similar to the programs for verification, which attains very high code coverage of the RTL descriptions with the consideration of fault observations.

We refer to the RTL descriptions of the processor core to develop test routines for control, steering logic, and pipeline registers. For example, when we pursue code coverage for a multiplexer, we just need to write an instruction sequence that will select every input of the multiplexer. But, in the case of testing, besides selecting every input, we must differentiate every input value to prevent masking of faults on the select signals by identical input values. And we must use store or other instructions to propagate the test responses to primary outputs.

The methodology for testing the pipeline registers is similar, but the difficulty of testing them well becomes higher. This is because there is no instruction that can explicitly access them as well as there is no way to definitely freeze their values. To test a specific pipeline register, we refer to the ISA specification and RTL descriptions to select the proper instructions. In general, the faults related to D, Q, and CK of all pipeline registers are easy to be detected while R's s-a-0 and EN's s-a-1 are the faults that appear to be functionally undetectable.

3.1.4 Test routine development for pipeline-related control logic

Pipeline architecture is widely used in processor designs to enhance the performance. Information of pipeline architecture can be explored to test additional stuck-at faults related to pipeline control. When the processor encounters a branch instruction, exception, or external interrupt, its execution flow will be altered. In a pipelined processor, the mechanism to handle these operations is known as the branch hazard detection or control hazard detection. Another commonly found pipeline control logic is the forwarding logic which bypasses the result to the execution unit before the destination register is updated.

To test the pipeline control logic, the methodology relies on the exploration of the pipeline architecture of the target processor. Specifically, the test routine examines the single stuck-at faults from the pipeline control hazard logic and the forwarding logic. In a classic five-stage pipelined processor with always not-taken for branch prediction, when a branch occurs, the IF and ID pipeline stage must be flushed for correct program execution. It means that the decoding of the instruction following the branch must be disabled, and the fetching of the next instruction must be disabled, too. We can insert another control-transfer instruction following the branch to functionally test these mechanisms. As for the forwarding mechanism, the idea is simply to test if the forwarding does occur from behind the pipeline stages.

3.2 Stage 2: Random test program development

We develop a random test program generation methodology for fault coverage improvement of the test program. In this section, we will illustrate the structure and hardware support for the random test program.

3.2.1 Random test program

Deterministic test program written by hands can test lots of faults using very small size of memory space. It is, however, impossible to test some faults no matter how good the deterministic program is. For example, the 12 bits close to MSB of the 32-bit program counter will never be touched if the memory size of manual program is allocated in 0 to 1 mega-byte. Additionally, deterministic test program is unable to enumerate all the combinations of instructions. Random instructions can test microprocessors as random patterns do. They contribute the fault coverage that a deterministic test program cannot achieve and take much less test generation time than a deterministic methodology. Random test program can contain pseudo-exhaustive combinations of operation codes, addressing modes, and branch targets. Instead of fully random instruction sequence, we use the concept of basic block formed by instructions to generate the random test program.

The advantage of using basic blocks to form the test program is that we can define several constraints in the basic blocks. In our method, we define what kind of instructions can be present in a basic block, and how the instruction sequences are ordered.

Table 1. Categories of ARMv4 instruction set.

Classification	Description
Single Load to Register	Load a value from memory, and write it into only one register.
Single Store to Memory	Store a value from only one register to memory.
Multiple Load to Registers	Load a block of data from memory, and write them into several registers.
Multiple Store to Memory	Store a group of register contents to memory.
Branch	Change the program counter.
ALU	Arithmetic and logical operations (Some have the effect of branch).
Status Register Access	Read or write the processor status registers.

3.2.1.1 The structure of basic block

To build the basic blocks, we classify the instruction set into several categories first. For example, for the ARMv4 instruction set, it has seven categories as Table 1 shows.

Each basic block contains every classification of instructions. In our method, there are eleven instructions in a basic block. Two types of basic block are used for the switching of processor mode. The first one is a general block, and the second is return block. For lots of simple microprocessor, using general block is enough because they do not have different processor modes. A return block is used for the purpose of returning from privilege mode to user mode.

Figure 2 and Figure 3 show the instruction sequence of basic blocks. We make some true data dependencies as shown by those curved arrows to test forwarding mechanism. There are also two types of general block. The difference between them is the order of the instruction sequence. The first one is the in-order basic block as Figure 2 shows, and the other is the out-of-order basic block. The sequences of instructions in an out-of-order basic block are changed randomly, but the basic blocks still contain every type of instructions. This is just like we shuffle the in-order basic blocks to form the out-of-order ones. The out-of-order basic blocks give random instruction sequence that is almost impossibly given in normal programs. But they will test some corner cases and detect the difficult-to-test faults.

3.2.2 Test Shell

While the random test program runs, the address of the instruction and data access are unpredicted. If we do not use any mechanism to handle this, the processor will fetch unknown instructions and access unknown data because it sends the addresses to places where no test program exists. So, we develop a hardware device to solve this problem.

Figure 4 shows the block diagram of the test system. The block on the left side is the processor core. The test program is placed in the memory on the right side.

The test shell is placed between the processor core and system bus, just like a bus interface. If the processor core needs a bus interface

to connect to the system bus, the test shell will be placed between the processor core and the bus interface. The functionality of the test shell is – to correct the illegal instruction and data addresses sent from the processor to legal ones. This functionality is only enabled when the processor executes the random test program. When executing normal user programs, generating the legal address is the responsibility of the programmers.

We also use the test shell to control exception signals and control test modes for test procedure in our test method. We use undefined instructions to control our test shell to trigger the exception signals.

1. Load to R0.
2. Multiple Load use R0 as base register.
3. ALU operation.
4. ALU operation.
5. Store the result of instruction 4.
6. Store the result of instruction 4.
7. Move CPSR or SPSR to one of R0 to R14.
8. Multiple Store contents of R0 to R14.
9. Multiple Store (randomly select registers to store).
10. No operation
11. Branch or Branch and link

Figure 2. Instruction sequence of a general block.

1. Load to R0.
2. Multiple Load use R0 as base register.
3. ALU operation.
4. Store the result of instruction 3.
5. Move some values to CPSR or SPSR
6. Move CPSR or SPSR to one of R0 to R14.
7. Multiple Store contents of R0 to R14.
8. Multiple Store (randomly select registers to store).
9. Move an immediate value "16" to R0.
10. Move R0 to SPSR[7:0].
11. Return to User Mode.

Figure 3. Instruction sequence of a return block.

3.3 Stage 3: Fault simulation

We here describe fault simulation which is depicted in Figure 1. We first prepare the memory image file that contains instructions and data required for test program execution. Our test program has two parts, deterministic test program and random test program. After the preparation of the memory image file, we perform logic simulation using ModelSim to acquire stimuli for fault simulation. During the logic simulation, the instructions and data in the memory image file will be fetched into the processor core according to the execution of the test program. We write codes in the test bench of the logic simulation to capture all input signals during the logic simulation, and these captured signals will become the stimuli fed to the fault simulation tool. The recorded data do not conform to the TurboScan's input format, so some simple character transformation is required. Our processor core has 110 input pins, so after transformation the input stimulus for a time frame (a half cycle) will has 110 characters representing the 110-bit input signals. Finally, we perform fault simulation using the previously transformed stimuli and the gate-level net-list of the processor as inputs and get the fault coverage.

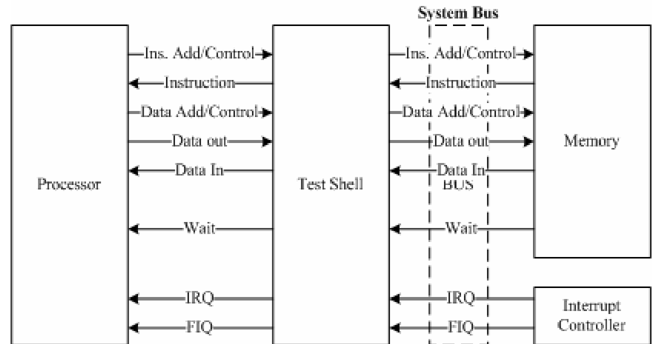

Figure 4. Block diagram of test system.

4. EXPERIMENTAL RESULTS

We have realized a processor core that implements the compatible ARMv4 instruction set to demonstrate the proposed methodology [24][25]. The ARMv4 instruction set is a RISC one, but it has several CISC features, such as the multiple load or store instructions that manipulate more than one memory address and registers by just one instruction. In addition to the complex instruction set, the processor includes five-stage pipeline that supports the mechanisms for resolving branch hazard, load-use hazards, and pipeline forwarding [26]. This processor also supports the caches, TLBs, and MMU system. Besides, the processor core was verified in an FPGA board that successfully runs the Linux operating system. Different from many previous works that use a much simpler processor [2][9][17-21], the processor we use in this paper incorporates the complete functionality for operating system support.

We have synthesized the target processor using the TSMC 0.35 um library. The synthesized processor has a gate count of 45046 (2-input NAND-gate equivalence) and operation speed of 30 MHz. The deterministic test program has a 1747-instruction test sequence (6988 bytes). And the size of the test data is 2356 bytes (2120 bytes of them are patterns generated by the ATPG tool, and the other 236 bytes are hand-made patterns, page tables, and so on). Thus, a deterministic test program (instructions + data) of 9108 bytes must be loaded into the memory by the external tester before self-testing. And it takes only 18675 (with cache) or 24073 (without cache) cycles to execute this test program. We also generate different random test programs in the forms of basic blocks and combine these random test programs with deterministic test program.

Table 3 lists the resultant fault coverage of each part as well as the processor. The pure deterministic test program and a pure random test program of 1000 basic blocks (1100 instructions) can attain fault coverage 93.84% and 90.43% respectively, but this still is not enough for practical use. Unfortunately, it seems that there is a bottleneck to further raise fault coverage only using deterministic and random methods. However, when we combine the deterministic test program and the random test program which has 5000 basic blocks, the fault coverage achieves 97.30%.

Because pseudo-random instruction blocks can detect hard-to-test faults of the deterministic program, the test program which combines the deterministic part and pseudo-random instruction block achieves 97.30% fault coverage. The total fault coverage improvement is 3.46%. For the status registers and access control category, the improvement is much greater, a fault coverage gain of 12.39%.

Table 3. Breakdown of fault coverage

Component	# faults	Det.. (%)	Ran 1000 (%)	Det.+ Ran 1000 (%)	Det.+ Ran 5000 (%)
Register file & access control	47394	93.15	88.86	97.05	97.58
ALU	43292	97.73	95.75	98.61	98.73
Shifter	3598	98.75	99.17	99.81	99.86
Memory access unit	12176	92.94	88.82	95.33	95.95
Instruction fetch unit	2176	80.97	83.92	83.96	84.42
Decoder	2878	87.04	88.18	91.97	92.42
Status registers & access control	5932	83.45	77.49	93.98	95.84
Coprocessor access unit	1154	86.74	61.44	89.34	89.69
Exception handling unit	308	86.69	76.30	87.66	87.66
Other	9936	90.19	87.06	95.00	96.72
Whole processor	128844	93.84	90.43	96.76	97.30

Table 4 summarizes the comparisons of the SBST methodology with other works. Only our target processor core possesses the full functionality of pipeline hazard control, forwarding mechanism, and modern operating system supports. The table is clearly shown from the statistics of gate count used and the number of faults present. The achieved fault coverage of full scan chain is 97.88%. But the full-scan test method is of high area overhead and has a longer test application time. The SBST methodology has achieved 97.30% in processor coverage. The SBST test method not only is of very low area overhead but also has at-speed test application time. The difference of the fault coverage between applying the full scan chain and the SBST methodology is 0.58%. These results indicate that the proposed SBST methodology can be an option sufficient for practical use.

5. CONCLUSIONS

We have presented a software-based self-test methodology and demonstrated on a complex pipeline processor. The proposed methodology produces a test program using both deterministic and random development of test codes. The deterministic test program which refers to the design information of processor architecture, RT-level, and gate-level for different types of the processor components. The deterministic test routine development methodology applies the most useful information of a certain level to the different parts of the processor core. The pseudo-random test program tries to make the pseudo-exhaustive testing possible. They also can detect the faults that the deterministic test programs are unable to detect. The concept of basic block is used to form the random test programs. Experimental results show that this concept can help the deterministic test program to get higher fault coverage. We have demonstrated that the proposed SBST methodology has attained 97.30% (Deterministic + 5000 Random basic blocks) in processor fault coverage, 0.58% less than the expensive full scan approach. The result indicates that the proposed SBST methodology is an option for practical use. The experiments were performed on a pipeline processor which has full functionality of pipeline hazard control, forwarding mechanism, and complete ISA supports for modern operating systems.

Table 4. Comparisons of various SBST work.

	CPU	Methodology style	Gate count/fault number	Program size	F.C. %
[2]	32-bit MIPS	Det.	37,402 /N.A.	1,728 (words)	92.60
[9]	8-bit PARWAN	Det.	1,300/N.A.	885 bytes	91.10
[17]	16-bit DLX	Random + DFT hardware	27,860/43,927	166 inst. seeds.	94.80
[19]	8-bit 8051	Random	6,000 /N.A.	624 inst.	85.19
[20]	8-bit 8051	Random	12,000/ 28,792	N.A.	90.77
[21]	8-bit PARWAN	Random	888 gates + 53 FFs/N.A.	1,129 (bytes)	91.42
SBST	ARM9-v4 compatible processor	Det.	45,046/128, 844	1747 inst./ 530 data (words)	93.84
SBST	ARM9-v4 compatible processor	Random 1000	45,046/128, 844	11012 inst.	90.43
SBST	ARM9-v4 compatible processor	Det.+ Random 5000	45,046/128, 844	56759 inst./ 530 data (words	97.30
SBST	ARM9-v4 compatible processor	Full scan chain	49,961/137,258	N.A.	97.88

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] A. Burdass, G. Campbell, R. Grisenthwaite et. al., “Embedded Test and Debug of Full Custom and Synthesisable Microprocessor Cores,” *Proceedings of the IEEE European Test Workshop*, 2000, pp.17-22.

[2] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, “Software-Based Self-Testing of Embedded Processor,”

IEEE Transactions on Computers, vol. 54, no.4, April 2005, pp.461-475.

[3] S. M. Thatte and Jacob A. Abraham, "Test Generation of Microprocessors," *IEEE Trans. Computers*, vol.33, no.6, June, 1980, pp.429-441.

[4] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Transactions on Computers*, Vol.C-33, No.6, June 1984, pp.475-485.

[5] C.-S. Lin and H.-F. Ho, "Automatic Functional Test Program Generation for Microprocessors," *Design Automation Conference*, June, 1988, pp.605-608.

[6] A. J. van de Goor and Th. J. W. Verhallen, "Functional Testing of Current Microprocessors (applied to the Intel i860)," *International Test Conference*, September, 1992, pp.684-695.

[7] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic Software-Based Self-Testing of Embedded Processor Cores," *Design Automation and Test in Europe*, 2001.

[8] J. Shen and J. A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation," *International Test Conference*, October, 1998, pp.990-999.

[9] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective Software Self-Test Methodology for Processor Cores," *Design Automation and Test in Europe*, 2002.

[10] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Instruction-Based Self-Testing of Processor Cores," *In Proc. VLSI Test Symposium*, 2002, pp.223-228.

[11] F. Cormo, M. S. Reorda, G. Squillero and M. Violante, "A Genetic Algorithm-based System for Generating Test Programs for Microprocessor IP Cores," *International Conference on Tools with Artificial Intelligence*, 2000, pp.195-198.

[12] R. S. Tupuri and J. A. Abraham, "A Novel Hierarchical Test Generation Method for Processors," *In Proc. International Conference on VLSI Design*, January, 1997, pp.540-541.

[13] R. S. Tupuri and J. A. Abraham, "A Novel Functional Test Generation Method for Processors using Commercial ATPG," *International Test Conference*, November, 1997, pp.743-752.

[14] R. S. Tupuri, A. Krishnamachary and J. A. Abraham, "Test Generation for Gigahertz Processors Using an Automatic Functional Constraint Extractor," *Design Automation Conference*, 1999, pp.647-652.

[15] V. M. Vedula and J. A. Abraham, "A Novel Methodology for Hierarchical Test Generation using Functional Constraint Composition," *High-Level Design Validation and Test Workshop*, 2000, pp.9-14.

[16] H.-P. Klug, "Microprocessor Testing by Instruction Sequences Derived from Random Patterns," *International Test Conference* (September 1988), pp.73-80.

[17] K. Batcher and C. Papachristou, "Instruction Randomization Self Test for Processor Cores," *the 17th IEEE VLSI Test Symposium,* 1999, pp.34-40.

[18] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS-A Microprocessor Functional BIST Method," *Proceeding of International Test Conference*, 2002, pp.590-598.

[19] F. Corno, M. Reorda, G. Squillero, and M. Violante, "On the Test of Microprocessor IP Cores," *Design Automation and Test in Europe*, 2001.

[20] F. Corno, G. Cumani, M. Reorda, and G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores," *Design Automation and Test in Europe*, 2003.

[21] L. Chen and S. Dey, "Software-Based Self-Testing Methodology for Processor Cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, March, 2001, pp.369-380.

[22] A. Paschalis and D. Gizopoulos, "Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1,January, 2005.

[23] C.-H. Chen, C.-K. Wei, T.-H. Lu, and H.-W. Gao, "Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.15, no.5, May, 2007, pp.505–517.

[24] ARM Corporation, *ARM Architecture Reference Manual*, ARM DDI 0100E, 2000.

[25] ARM Corporation, *ARM922T Technique Reference Manual*, ARM DDI 0184A, 2000.

[26] David A. Patterson and John L. Hennessy, *Computer Organization & Design*, San Francisco, CA: Morgan Kaufmann, 3rd, 2005.