A Hybrid Self-Testing Methodology of Processor Cores

Tai-Hua Lu, Chung-Ho Chen, and Kuen-Jong Lee
Department of Electrical Engineering and
Institute of Computer and Communication Engineering, National Cheng Kung University
Tainan, Taiwan
aaron @casmail.ee.ncku.edu.tw, {chchen,kjlee}@mail.ncku.edu.tw

Abstract—**Software-based self-test (SBST) is a promising new technology for at-speed testing of embedded processors in SoC systems. This paper introduces an effective and efficient new SBST methodology that uses information abstracted from the processor instruction set architecture (ISA), pipeline architecture model, RTL descriptions, and gate-level net-list for test program development of different types of the processor circuitry. This paper demonstrates the feasibility of the proposed methodology by the achieved fault coverage 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.**

I. INTRODUCTION

With the evolution of SoC technology, more and more IP cores can be integrated into a single chip. As a result, more and more embedded processors are used for either general or special purpose. Although the processors are widely used in various SoC applications, the poor controllability and observability of the embedded processors result in the hurdle of testing. Traditional test methods mainly insert scan chain into the circuits. However, this kind of approaches usually causes the overhead of area and cost, and may seriously degrade the performance [1]. Therefore, software-based self-test (SBST) methodologies become 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]. This approach can use the same test program and operands to test any architecture of adders and multipliers. In [8] Shen and Abraham described a language to represent every instruction in an instruction set so as to simplify the test generation procedure. In [9], [10], criterions to classify instructions according to the functionalities of components in a 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 the functional constraints of processor modules from HDL description of the processor. Commercial ATPG tools can then be used to generate test patterns of the modules with the extracted constraints. The test patterns are then translated into instructions [12]–[15]. For random pattern generation, several methodologies have been developed [16], [17], [18]. In [16], a LFSR is used to generate random numbers to represent random instructions, and a filter is designed to replace an 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 that uses a deterministic programming procedure and a random program generation method for embedded processor testing. The deterministic test methodology explores the different types of circuitry based on the design information from the processor architecture, register-transfer-level, and gate-level for program development and uses the multiple-level information to improve coverage for structural faults. The pseudo-random test methodology is carefully developed and tries to make the pseudo-exhaustive testing possible. We use the pseudo-random test program to detect some of the faults that the deterministic test programs are unable to test. Finally, we combine the deterministic test program with the pseudo-random test program to improve processor fault coverage. 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 presents the proposed methodology. Section 3 shows the experimental results and comparisons with other SBST-based schemes. Finally, Section 4 gives the conclusions of this paper.

II. THE SBST METHODOLOGY

A. Overview of the proposed SBST

The test code development flow consists of three phases of work as illustrated in Figure 1. The first two phases which can

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

Authorized licensed use limited to: National Cheng Kung University. Downloaded on January 14, 2009 at 02:33 from IEEE Xplore. Restrictions apply.

be conducted in parallel simply involve a deterministic code development and a random code development, respectively. The third phase is the integration of both the test codes.

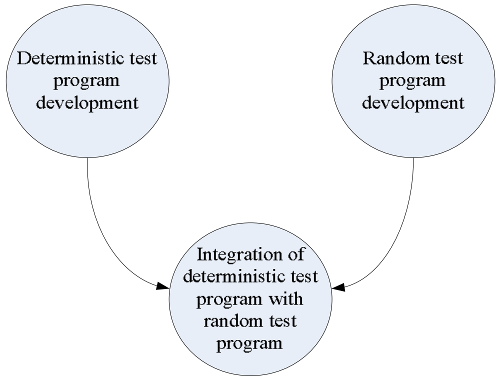
Deterministic test program development
Random test program development
Integration of deterministic test program with random test program

Figure 1. Outline of the proposed SBST methodology.

The deterministic approach consists of two steps: classification of processor parts and test program development for each part. To improve the effectiveness of the test code, the processor core is classified into the following four groups for test code development: (1) ISA registers, (2) fundamental IPs which are the simple ALUs inside the processor, (3) control, steering logic, and pipeline registers, and (4) pipeline mechanisms such as load-use hazard, control hazards, forwarding control, including interrupt supporting logic. Test program development for each group is based on the information abstracted from the processor's architecture model, RTL descriptions, and gate-level net-list. This deterministic approach is very effective in detecting the processor faults of which the random test code may fall short. On the other hand, the random test program is used to remedy the weakness of the deterministic code by boosting the processor fault coverage mainly on the control and glue logic as well as some corner cases that are neglected by the mindset in deterministic programming. The proposed random test methodology uses a program entity called constrained basic block as the building module of the random test code.

B. *Deterministic test routine development*

Before deterministic test code development, we classify the processor components into the four functional groups as illustrated in Section 2. The concept of the deterministic test routine development methodology as follows.

- Use synthesized processor codes for developing:
  - i. Tests of programmer-visible registers
    1. Consider synthesized state elements.
    2. Consider fault propagation.
  - ii. Tests of Fundamental IPs
    1. IP extraction
    2. Constraint setting
    3. Perform ATPG and develop test sequences.
- Use processor RTL codes for developing:
  - i. Tests of control and steering logics and pipeline registers
    1. Cover each statement, branch, and condition.
    2. Consider fault model and fault propagation.
- Use abstracted information from processor system architecture for developing:
  - i. Tests of pipeline features
    1. Control hazards.
    2. Forwarding.

1) Test routine development for programmer-visible registers

In a synthesized processor core, the programmer-visible registers in RTL descriptions are typically 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.

2) Test routine development for Fundamental IPs

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 the fundamental IPs that are much smaller and less complicated than other processor components or modules. A component or module may include control logic, 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) 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 can attain very high code coverage of the RTL descriptions with the consideration of fault observations.

4) Test routine development for pipeline-related control logic

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. This means that the decoding process of the instruction following the branch and next instruction fetching must both be disabled. We can insert another control-transfer instruction following the branch to functionally test these mechanisms. As for the forwarding

Authorized licensed use limited to: National Cheng Kung University. Downloaded on January 14, 2009 at 02:33 from IEEE Xplore. Restrictions apply.

mechanism, the idea is simply to test if the forwarding does occur from behind the pipeline stages.

C. Random test program development

We design a random test program development method to improve fault coverage for the previous test program. In this section, we illustrate the structure and hardware support of the random test program.

1) Random test program

We use the concept of basic block formed by instructions to generate the random test program. The advantage of using the 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 should be present in a basic block, and how the instruction sequences are ordered. To build the basic blocks, we classify the instruction set into several categories, including load/stores, ALUs, branches, and status register accesses and each basic block contains every classification of the instructions. In our method, there are eleven instructions in a basic block. Two types of the basic block are used for switching the processor mode. The first one is a general block, and the return block is the second. For lots of simple microprocessors, the general block is enough because they do not include different processor modes for instruction execution. For a multiple mode processor, a return block is used for the purpose of returning from the privilege mode to the user mode.

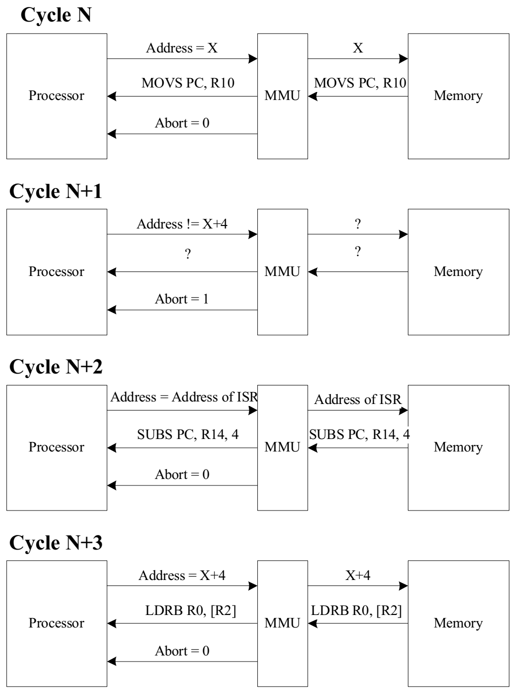

Figure 2. MMU used to correct instruction fetching address

2) Supporiting mechanism for random program execution

As the random test program runs, the address of the instruction fetch and data access are not constrained. If we do not use any mechanism to correct these illegal memory accesses, the processor will fetch unknown instructions and access unknown data because it may send the addresses to the place where no legitimate test program exists. In this paper, we present an MMU (memory management unit) mechanism to perform the address correction function, namely, converting the instruction fetch and the data access addresses from illegal to a legal one. For example, as indicated in Figure 2, when the processor issues a non-sequential instruction fetch address, the MMU will generate an instruction abort to interrupt the processor. However, the ISR (exception handler) for this abort is only one instruction, that is, a return instruction, so the processor directly returns to the next address (PC+4) and continues executing the random test program which is sequentially placed in the memory.

III. EXPERIMENTAL RESULTS

We have realized a processor core that implements the compatible ARMv4 instruction set to demonstrate the proposed methodology [23], [24]. 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. Table 1 lists the resultant fault coverage of each part as well as the processor. Executing the deterministic test program and the random test program of 1000 basic blocks (11000 instructions) individually can attain 93.84% and 89.99% of fault coverage, respectively. There is a bottleneck to further raise the fault coverage only using the deterministic program or the random test program alone. However, when we combine the deterministic test program and the random test program, the fault coverage has achieved 96.29%. This is because the pseudo-random instruction block can detect those hard-to-test faults of the deterministic program. The average fault coverage improvement is 2.45%.

Table 2 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 achieved fault coverage of using the full scan chain is 97.88%. However, the full-scan method contributes area overhead to the processor core and has longer test application time. The SBST methodology has achieved 96.29% in processor coverage. The difference of the fault coverage between applying the full scan chain and the SBST methodology is about 1.59%.

IV. CONCLUSIONS

We have presented a software-based self-test methodology and demonstrated on a complex pipeline processor. The proposed methodology produces the test program using both the deterministic and random test program development methods. Experimental results show that this methodology can help the deterministic test program to get higher fault coverage. We have demonstrated that the proposed SBST methodology has attained 96.29% (Deterministic + 1000 Random basic

Authorized licensed use limited to: National Cheng Kung University. Downloaded on January 14, 2009 at 02:33 from IEEE Xplore. Restrictions apply.

blocks) in processor fault coverage, 1.59% less than the expensive full scan approach. 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 I. BREAKDOWN OF FAULT COVERAGE.

Component	#faults	Det. (%)	Ran1000 (%)	Det.+ Ran1000 (%)
Register file & access control	47394	93.15	88.19	96.32
ALU	43292	97.73	95.74	98.60
Shifter	3598	98.75	99.21	99.86
Memory access unit	12176	92.94	88.55	95.04
Instruction fetch unit	2176	80.97	83.90	83.94
Decoder	2878	87.14	86.55	90.27
Status registers & access control	5932	83.45	75.75	91.87
Coprocessor access unit	1154	86.74	61.16	88.94
Exception handling unit	308	86.69	76.87	88.31
Other	9936	90.19	86.81	94.73
Whole processor	128844	93.84	89.99	96.29

TABLE II. 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.+ Random 1000	45,046/128, 844	56759 inst./ 530 data (words	96.29
SBST	ARM9-v4 compatible processor	Full scan chain	49,961/137,258	N.A.	97.88

REFERENCES

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

[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, pp. 461-475, April 2005.

[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, pp.475-485, June 1984.

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

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

[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, pp.990-999, October 1998.

[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, pp.195-198, 2000.

[12] Raghuram S. Tupuri and Jacob 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, pp.743-752, November 1997.

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

[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, pp.9-14, 2000.

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

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

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

[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, pp. 369-380, March 2001.

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

[24] ARM Corporation, "ARM922T Technique Reference Manual," ARM DDI 0184A, 2000.

Authorized licensed use limited to: National Cheng Kung University. Downloaded on January 14, 2009 at 02:33 from IEEE Xplore. Restrictions apply.