Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores

Chung-Ho Chen, *Member, IEEE*, Chih-Kai Wei, Tai-Hua Lu, and Hsun-Wei Gao

***Abstract*—Software-based self-test (SBST) is a promising approach for testing a processor core embedded in a system-on-chip (SoC) system. Test routine development for SBST can be based on information of different abstraction levels. Multilevel abstraction-based SBST develops the test program for a pipeline processor using the information abstracted from its architecture model, register transfer level (RTL) descriptions, and gate-level netlist for different types of processor circuits. The proposed methodology uses gate-level and architecture information to improve coverage for structural faults. This SBST methodology uses an automatic test pattern generation tool to generate the constrained test patterns to effectively test the combinational fundamental intellectual properties used in the processor. The approach refers to the RTL code and processor architecture for the rest of the control and steering logic for test routine development. The effectiveness of this SBST methodology is demonstrated by the achieved fault coverage, test program size, and testing cycle count on a complex pipeline processor core. Comparisons with previous works are also made.**

***Index Terms*—Automatic test pattern generation (ATPG), fault coverage, functional testing, processor testing, scan chain, software-based self-test (SBST).**

I. Introduction

SOFTWARE-BASED self-test (SBST) has become increasingly popular for embedded processor testing [1]. This methodology has many advantages, such as at-speed testing, testing for critical paths [21], no area and performance overhead, and no requirement of using expensive external testers. In SBST, the processor fetches the test program from the memory, executes the instructions, and writes back the results into memory. The results are compared with the correct one to determine whether the processor core is successfully manufactured. The test program and data in a sense are the functional patterns stored in memory. In current system-on-chip (SoC) designs, built-in memory can be used for storing or uploading the test program. Alternatively, a low-cost automatic test equipment (ATE) that provides the external storage and bus interface for accessing the test code and data can jointly support the testing as shown in Fig. 1(a). It is called self test because the processor executes the instructions to test itself despite the help from the low-cost ATE machine.

To improve the speed of processor functional testing, the test routine can be loaded into the processor cache by a special cache loader [12]. The processor executes the test routine at its actual speed. The test results are either written back to the internal memory as in Fig. 1(a) or directly to the external memory in the ATE. In a different way, the test results are compressed and stored in a multiple-input signature register (MISR) as illustrated in Fig. 1(b). Finally, the processor writes back the response out of the MISR to the ATE machine which checks with the correct result during manufacturing testing.

The test program can be developed in different ways, including random test routine [2]–[5], [8] or deterministic test code [1], [6], [10]. In the case of random test code, a mechanism that makes sure continuous and legitimate instruction execution must be provided, either through software instruction generator [12] or built-in hardware [4].

The test routine can be programmed or generated by referring to the original instruction set architecture (ISA) of the processor or by adding special test instructions, hence changing the original ISA to improve the testability of the processor [9]. In test routine development, especially for deterministic methodology, different levels of information of the processor can be used. Using the design information from lower abstraction level, such as the gate-level netlist often results in more efficient and effective testing program since in this case the test program is developed directly against the known structural faults [5].

Test development can also be carried out based on the information of higher abstraction level such as the RTL (register-transfer-level) descriptions [1] or the ISA [2]. In these cases, at the test development stage, the fault list of the processor may be still unknown so the test development deductively targets at the possible structural faults. Due to the high abstract level, a test program only based on the RTL description or ISA is likely difficult to attain a high coverage for the structural faults except for specific functional modules, for instance, a general purpose ALU [10], [21].

In this paper, we present a methodology that uses multiple-level abstractions (MLA)-SBST of embedded processors. The approach explores the SBST development for different types of circuit based on the design information from the processor architecture, register-transfer-level, and gate-level. The idea behind the methodology is to apply the most useful information of a certain level to the different parts of the processor core. The proposed methodology uses gate-level and architecture information to improve coverage for structural faults. Our SBST methodology uses an automatic test pattern generation (ATPG) tool to generate the constrained test patterns to effectively test

Manuscript received July 31, 2006; revised November 20, 2006. This work was supported in part by the National Science Council, Taiwan, R.O.C., under Grant NSC 95-2220-E-006-002 and by the Program for Promoting Academic Excellence of Universities in Taiwan.

The authors are with the Department of Electrical Engineering, Institute of Computer and Communication Engineering, National Cheng-Kung University, Tainan 701, Taiwan, R.O.C. (e-mail: chchen@mail.ncku.edu.tw; peter@casmail.ee.ncku.edu.tw; aaron@casmail.ee.ncku.edu.tw; fiend@casmail.ee.ncku.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2007.893650

1063-8210/$25.00 © 2007 IEEE

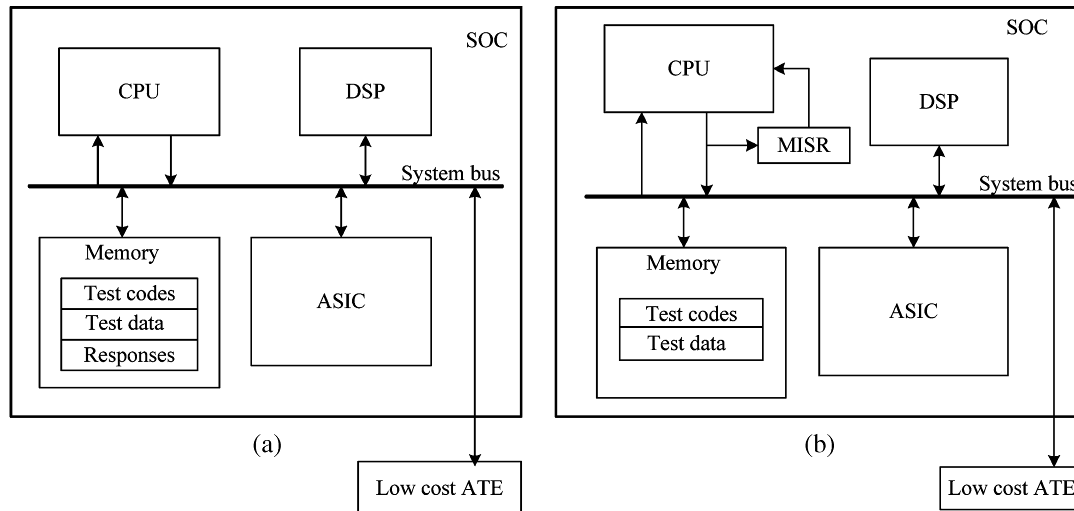


Fig. 1. System environment of SBST for embedded processor cores.

the simple combinational fundamental intellectual properties (IPs) used in the processor. Finally, the approach refers to the register transfer level (RTL) code and processor pipeline architecture for the rest of the control and steering logic in the processor for test routine development.

The effectiveness of this MLA-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 II discusses some related research. Section III presents the proposed methodology. A case study with detailed test sequences is presented in Section IV. Section V shows the experimental results and comparisons with other SBST schemes. Finally, Section VI gives the conclusions of this paper.

II. Previous Work

SBST of processor cores has received great attention and interest recently [1]–[7], [10], [11]. The approach in [5] uses pseudorandom patterns to test a processor component by component. In the test preparation step, test patterns for a component are developed using random number generator under constraints imposed by the instructions. The consideration of constraints is to make sure that the generated patterns are realizable by the processor instructions or system constraint [12]. In the case of random patterns, the patterns are encapsulated into signatures. In the self-testing step, the generated patterns are applied to the considered components by a software tester which is an on-chip test generation program that emulates a pseudorandom pattern generator and expands the signatures into test patterns. Then the test application program applies these patterns to the considered components. The authors used an 8-bit Parwan processor core which consists of 888 equivalent NAND gates and 53 flip-flops as the target processor. The work has achieved 91.42% in processor fault coverage.

In [2], test instructions and their operands are generated randomly using probability distribution functions and a genetic algorithm, respectively. The genetic algorithm is used to maximize the number of detected faults by selecting the proper operands for the instructions in the macro. If new faults are detected, the test macro is added to the final test program. If not, the probability of being selected again of this macro is decreased. The authors evaluated their work on an 8-bit 8051 microcontroller which has a gate-count of about 6000. The fault coverage attained is 85.19%.

In [1], Kranitis *et al.* analyzed the RTL descriptions of a processor and then chose instructions and operands to form the test routines determinedly 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. The test routine development methodology is basically a divide-and-conquer approach relying on processor ISA and its RT-level descriptions. The work has reported 95.3% in fault coverage for a three-stage simple pipeline MIPS processor core. For a more complicated five-stage pipeline implementation of the same instruction set, the achieved fault coverage drops to 92.6%. Deterministic SBST has also been used in testing the specific functional units of a processor such as the ALU, multiplier-accumulator, and shifter [10]. Automation in SBST has been the focus in [7] for automatic synthesis of test programs for processor modules as well as for automated constraint extraction from an RTL module in [19]. Apart from SBST for processor cores, the full scan approach has been used in testing a fully synthesizable processor core [13]. The downside of the scan-based approach comes from the processor area overheads and possible performance degradation.

III. Proposed Multiple-Level Abstraction-Based SBST Methodology

The idea behind our proposed processor SBST methodology is to exercise the most cost-effective test method on the selected processor part. The processor core is decomposed into the following parts for deterministic test routine development:

- ISA registers;
- fundamental IP units, which are the simple ALUs inside the processor core;

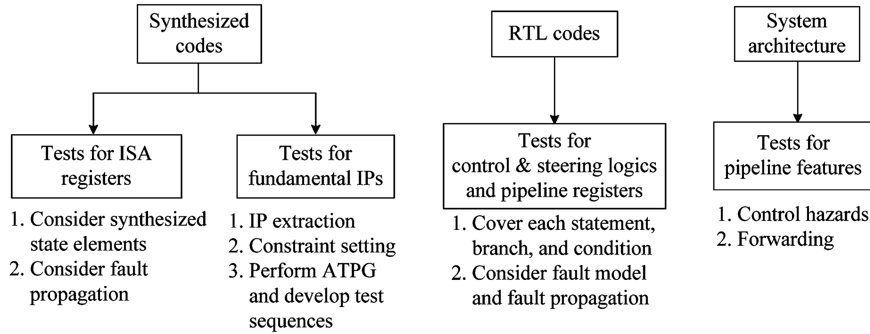


Fig. 2. Proposed MLA-SBST methodology for processor core testing.

TABLE I
LIST OF INTERFACES OF A SINGLE D FLIP-FLOP

pin	Description
D	Data input.
CK	Clock input.
R	Reset input; logic 1: activate reset.
EN	Write enable; logic 1: enable writing.
Q	Data output.

- control, steering logic, and pipeline registers;
- pipeline-related control logic, such as load-use hazards, control hazards, and result forwarding mechanism, including interrupt supporting logic.

The classification is based on the mechanism used to apply on test routine development. The MLA-SBST methodology for processor cores consists of the following two stages of work.

- Stage 1: Processor part classification based on multiple-level abstraction. The first stage is classification of the previously mentioned processor parts based on the multiple-level abstraction model that includes processor ISA, pipeline architecture, and synthesized gate-level processor core.
- Stage 2: Test routine development. The second stage includes the test code development for each individual part, which can be conducted in parallel.

We explain the detail of our multiple-level abstraction-based SBST strategy in the following subsections.

A. *Stage 1: Processor Part Classification Based on Multiple-Level Abstraction*

Before test code development, our SBST methodology requires classification of the processor components into the four functional groups as previously mentioned. Fig. 2 illustrates the concept of the proposed test routine development methodology.

1) ISA Registers: In a soft processor core, all the declared registers in RTL description are clocked and edge-triggered. Typically, they are synthesized into clocked D flip-flops with the input/output (I/O) pins listed in Table I. Different technology libraries and different synthesis parameters may introduce different state elements, but they are normally of little variation.

The MLA-based strategy derives the test sequences for programmer-visible registers according to the information in the gate-level and the processor pipeline architecture. Through the information from these two levels, effective test routines can be developed easily for regular state structures without the tedious manual efforts. After synthesis, the programmer visible registers are mapped into D flip-flops. If the test sequences are developed at the RT-level (register transfer level), some of the faults on the D flip-flops, such as the faults on the write enable pins, are invisible at 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.

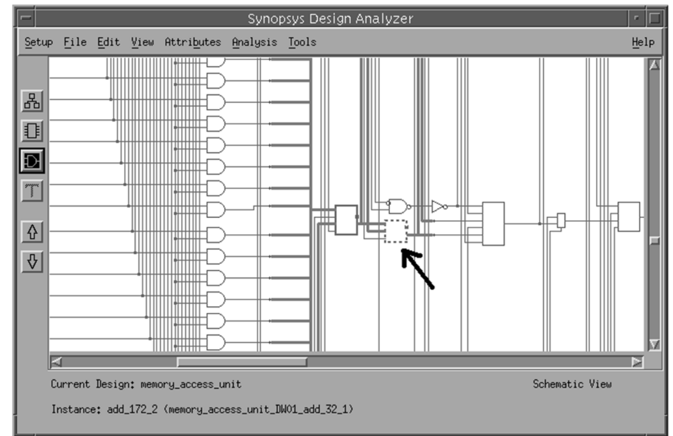


Fig. 3. Fundamental IP (a 32-bit adder) in the netlist.

2) Fundamental IP: The fundamental IP refers to the combinational logic introduced by the synthesis tool, such as adders and multipliers invoked by the related RTL statements. For example, the synthesis tool will insert an adder into the design to realize the addition function in the RTL code of the processor. Fig. 3 illustrates the netlist of the memory access unit which includes a 32-bit adder. This adder's operation and interfaces, both of which are simple enough for an ATPG tool to produce effective constrained test patterns, are shown in Fig. 4.

In a processor core, the fundamental IP mainly comes from two components: the ALU and the memory access unit. The fundamental IP units used in the ALU are for the implementation of data processing instructions, such as add, subtract, and multiply. The fundamental IP units used in the memory access module are for the manipulation of memory addresses. Pipelining is another source of using fundamental IP. In a pipelined architecture, the address of current executed instruction and the current PC value often have a difference such as 8, 12, or 16, depending on the

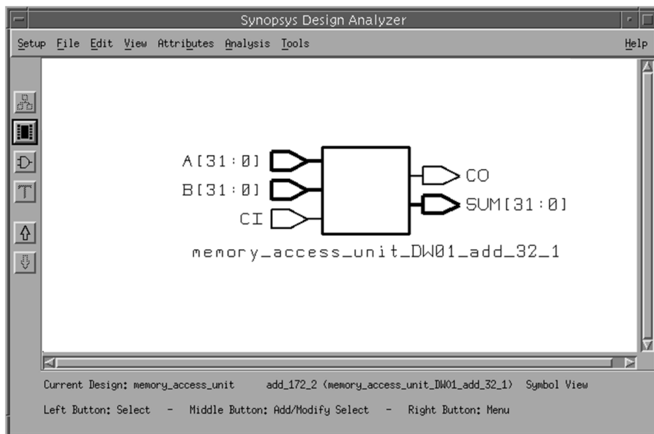


Fig. 4. Interfaces of the fundamental IP in Fig. 3.

pipeline depth. If an interrupt or a function call happens, the processor needs to backup the return address, using a subtractor to recover the actual instruction address from the PC. In a complex pipeline processor with various interrupt and exception modes, there may exist many fundamental IP units to handle address backup.

For these simple fundamental IPs, which are found to occupy a large portion of the processor, 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 as in other complicated processor modules.

3) Control, Steering Logic, and Pipeline Registers: Apart from the ISA registers and fundamental IPs, the remaining parts of the processor components to be considered is the control and steering logic, such as the distributed logic gates, multiplexers, and pipeline registers. For the two former elements, they are difficult to be identified with the structural information from the gate-level netlist. Pipeline registers are not ISA 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 and pipeline architecture.

4) Pipeline-Related Control Logic: Pipeline architecture is widely used in processor designs to enhance the performance. Pipeline architecture not only introduces complex pipeline control logic but also complicates the exception processing unit. To test additional stuck-at faults related to pipeline control, information of pipeline architecture can be explored. 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.

B. Stage 2: Test Routine Development

In this section, we address the process of test routine development for each of the major processor parts.

1) ISA Register: In a synthesized processor core, the ISA registers are mapped into edge-triggered D flip-flops which have a regular structure. Hence, the test routine development can focus on the structural faults of the D flip-flops' I/O interface with the consideration of the processor pipeline architecture, i.e., the pipeline forwarding structure. This part of test routine development is simple and effective.

2) Fundamental IP: To extract a fundamental IP, the IP (with no other logic) is copied directly from the netlist of the selected module and then the ATPG tool is used to generate the test patterns for the IP. We feed one IP to the ATPG tool at a time. At this moment, we have full controllability and observability on the IP's inputs and outputs. However, this is not the case when the IP is embedded in the processor. The input space of each fundamental IP is restricted by the ISA and the architecture of the processor. Hence, we must set constraints for the ATPG process based on the functionality of the IP in the processor core. The constrained ATPG is performed, which generates the dedicated test patterns for each fundamental IP. Then, the test routine that reproduces the test patterns for the inputs of each IP inside the processor can be developed. The fundamental IP is further classified into data processing IP, data memory address manipulation IP, and instruction memory address manipulation IP.

a) Constraint setting: There are two kinds of constraints: mandatory constraints and optional constraints. Mandatory constraints are used to prevent the generation of illegal test patterns for the IP that is embedded in the processor core. For example, consider a subtractor IP that calculates the start address for the multiple load/store instructions (ldm/stm) with a certain addressing mode. One of its data inputs (A) represents the 32-bit base address coming from a general purpose register specified in the instruction while the other data inputs (B) is a 5-bit wide number representing the number of memory accesses in the instruction. The start address is calculated as $\mathrm{A} - \{\mathrm{B}, 2'\mathrm{b}00\}$ (the access is word aligned). The legitimate number of memory accesses in the multiple load/store instruction is from 1 to 16. Thus, the constrained test pattern required for input B only consists of the bit patterns of 1 to 16, excluding the rest of the 5-bit patterns. Another commonly found mandatory constraint is setting the least significant two bits of a word-address test pattern to zero.

The other types of constraints are referred to as the optional constraints that help reduce test cost, but they are optional. Consider the subtractor for the multiple load/store instruction as an example again. The input A of the subtractor represents the base address for accessing a word, and we need to restrict the least significant two bits to 00 for mandatory constraint to ensure word alignment of the memory accesses. Under this constraint, the ATPG tool will generate patterns for input A ranged from 0 to fffffffc(h). This implies that the system needs to prepare 4 GB of memory for the processor to perform memory accesses to test the subtractor. This is not practical. However, if we restrict the most significant 12 bits to only two values: 000(h) and fff(h); the system needs only 2 MB of the memory: 1 MB at the bottom and 1 MB at the top of the 4-GB address space.

Setting these stricter optional constraints may cause fault coverage loss and augmentation of the pattern numbers generated by the ATPG. However, our experimental results show that if

the constraints are not made drastically strict, the fault coverage and the number of test patterns will stay at the same level or not even change at all.

b) Test routine development for data processing IP: The data processing IPs come from the processor ALU for the implementation of data processing instructions. To invoke them, the corresponding data processing instructions are the only choices. In a typical RISC processor, the inputs of the IPs are connected to the general purpose registers specified by the corresponding instruction or to the decoder or instruction register output for an immediate operand coded in the instruction itself. Usually, a 32-bit pattern cannot be coded as the immediate operand with only one instruction. Thus, the test program only requires using register operands. In this case, the test patterns for a data processing IP can be placed into a register via the memory load instructions. Then, the corresponding instructions are executed to invoke the IP and the results are sent to registers, which are then stored into memory for observation. These kinds of test sequences can be coded as a compact loop and the test program size can be reduced.

c) Test routine development for data memory address manipulation IP: To test the IP for data address generation, the data memory access instructions are used. The test patterns for these IPs come from the general purpose registers and the address offsets of the instructions. The instruction ISA of the tested processor provides the information. When testing an address-generation IP, the test responses (data access addresses) will be automatically propagated to the primary outputs. Generally, load and store instructions share the same adders/subtractors for address calculation. Thus, it is preferred that load instructions are used instead of store instructions because a store instruction can modify the content of target address which may contain data or instructions for future use.

d) Test routine development for instruction memory address manipulation IP: An instruction address generation IP is mainly used for the calculation of return address of branch-and-link instructions, exceptions, and interrupts. The IP is invoked when such special events occur. In general, one of their inputs is the PC value, and the other is a constant. Consequently, they can be seen as an IP with a single input. As a result, the generated patterns must be propagated to this type of IP via the program counter. This means that the test routine must invoke the event at the addresses indicated by the test patterns generated from the ATPG.

As for the branch-and-link instructions or internal exceptions raised by instructions, this is not a problem. For these cases, we just need to place the PC-control instructions at the indicated addresses, ensuring that the addresses are not occupied by other instructions or data. The address overlapping problem can be solved by using additional constraints to avoid the used addresses, but this may affect the fault coverage and the number of generated patterns. To resolve this issue, we propose the use of self-modifying code, a much more elegant and effective approach. For example, if a software-interrupt instruction and a branch-and-link instruction must be placed at the same address, after the software-interrupt instruction is executed, the instruction on this address can be changed to the branch-and-link by the test program itself. This is done in the following way. First, a simple loader replaces the old test program with the new test program of the IP for instruction address manipulation. The loader moves the new test program from the data area to the program area where the generated patterns must be propagated to this type of IP via the program counter. The second step is simply jumping to and executing the new test program. As for the external interrupt, the interrupt signal is scheduled to occur from outside at appropriate time by matching the interrupt event with the desired PC value.

3) Test Routine Development for Control, Steering Logic, and Pipeline Registers: To develop test routines for control, steering logic, and pipeline registers, we refer to the RTL descriptions of the processor core with emphasis on preventing fault masking. Taking a multiplexer for example, for the pursuit of high RTL code coverage, the instruction sequence only makes sure in selecting every input of the multiplexer. However, for testing, selecting every input is not enough. The test sequence must also differentiate every input value to prevent masking of faults on the select signals due to identical input values.

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.

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. The rationale behind the test program is to test if the flush logic successfully nullifies the instructions in these pipeline stages. As for the forwarding mechanism, the idea is simply to test if the forwarding does occur from behind the pipeline stages.

IV. Case Study: Test Routine Development for a Linux-Verified Processor Core

A. Processor Model

We have realized a processor core that implements the compatible ARMv4 instruction set to demonstrate the proposed methodology [14]. Despite its RISC ISA, this processor has resembling complex instruction set computer (CISC) features, such as the multiple load and store instructions that manipulate more than one memory address and register by just one instruction. In addition to the complex instruction set, the processor is implemented in a classic five-stage pipeline that supports the mechanisms for resolving branch hazard, load-use hazards, and pipeline forwarding [15]. Fig. 5 depicts the processor core architecture. Including the caches, translation lookaside buffers (TLBs), and memory management unit (MMU) system,

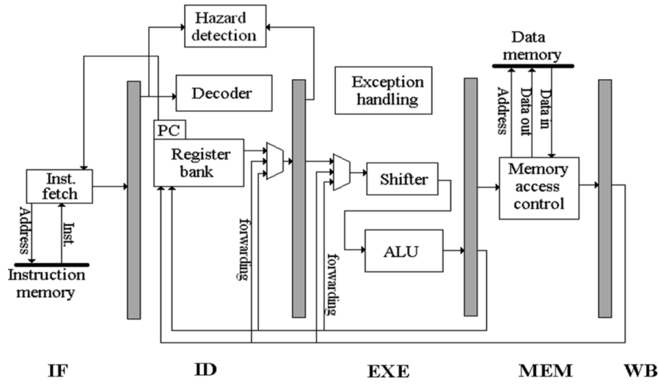


Fig. 5. Architecture of the target processor.

the processor core was verified in a field-programmable gate array (FPGA) board that successfully runs the Linux operating system. This shows that the full functionality of the processor core can be included in the test routine development. Different from many previous works that use a much simpler processor, the processor we use in this paper incorporates the complete functionality for operating system support in addition to its pipeline mechanisms for hazard detection and operand forwarding.

B. ISA Registers

In this paper, the fault model assumed is single stuck-at-fault. Thus, a 1-bit register (or a D flip-flop) with the gate-level structure shown in Table I will have ten faults. To test a general purpose register, say r1, it is intuitive to write the following instruction sequence:

```
mov r1, #0x 0      ;write all 0's to r1
str r1, [address]  ;read r1 and store it for observation
mvn r1, #0x 0      ;write all 1's to r1
str r1, [address]  ;read r1 and store it for observation.
```

The register r1 is a 32-bit register composed of 32 D flip-flops, and, according to the previous D flip-flop model, it has 320 probable single stuck-at faults. In the previous sequence, values of all 1's and all 0's are written into r1, respectively. After the write to the register, a store instruction reads the same register and propagates the value to the primary outputs (data output bus). Unfortunately, this instruction sequence cannot detect any faults of r1 because the sequence does not actually access r1. The store instruction receives its operand from the forwarding path but not the physical register r1.

To resolve this problem, the pipeline architecture of the processor must be taken into account, which requires the insertion of no operations (nops) between the move and store instructions. The function of nops is to wait for the written value to actually go into the register. Hence, the useful test segment is shown as follows:

mov r1, #0x 0
$\text{nop}_{\mathbf{1}}$
...
$\text{nop}_{\mathbf{n}}$
str r1, [address]

where the number $\mathbf{n}$ represents the number of pipeline stages between register reading and write-back.

After the insertion of nops, the faults on D, Q, and CK of every bit of the register can be detected. The reason that faults on the clock signals can be detected is that the register is edge-triggered. If the CKs are stuck at any level, the contents of r1 will not be able to be updated. Additionally, R's s-a-1 (reset signal) and EN's s-a-0 (register write enable) are also detected. This is because the s-a-1 faults of reset will set the register value to 0 and EN's s-a-0 faults will cause the register value unable to be changed; effects of these faults can be observed via the store instruction.

To test R's s-a-0 faults (reset signal), the method is to store the register's value immediately after system reset. If the register had been successfully reset, the value of 0 would be propagated to the data output bus. If not, an unpredictable value will be propagated and the fault effect of R's s-a-0 can be observed on the data output bus. To detect EN's s-a-1 faults (register write enable) needs a special treatment. The netlist of the processor core shows that all general purpose registers have the same input data source except r14. This means that a value written to a particular register, say r1, will also be transmitted to all other general purpose registers. Whether a register receives this value or not is decided by the write enable (EN) signals. The following sequence can be used to test this mechanism:

```
mvn r1, #0x 0      ;write all 1's to r1
mov r5, #0x 0      ;write all 0's to r5
...
...
str r1, [address]  ;read r1 and store it for observation.
```

If the ENs of r1 are stuck at 1, that is, always write-enable, the value of r1 will be changed and the fault effect can be observed on the data output bus after storing r1. Using the previous techniques, all general purpose registers that share the same input data source can be fully tested in a pipeline processor with forwarding path.

C. Test Routine Development for Fundamental IP

The test pattern for testing each fundamental IP can be generated through the constrained ATPG process.

1) Test Routine Development for Data Processing IP: As an example, we show the sequence of instructions for the testing of a combined adder-subtractor in the ALU that realizes related data-processing instructions. This fundamental IP has three inputs, two 32-bit data inputs, and a control input deciding the operation mode (add or subtract). For this IP, the ATPG tool generates 27 patterns each of which is 65-bit wide $(32 + 32 + 1)$. The two 32-bit patterns represent the two data inputs which will be placed in the memory while the control bit is used to decide the type of operation (add or sub). The ATPG process for this fundamental IP does not require any constraint setting.

The test routine shown as follows consists of the loop for the sub instruction. The operands (test patterns) for the IP are first loaded from the memory to the registers r1 and r2, respectively. Then, the principal instruction, sub, is executed to activate the fundamental IP. Next, the store instructions are exe-

cuted to propagate the result to the primary outputs for observation. Note that some of the faults of this fundamental IP can be observed only on the carry-out pin, so the current processor status register (CPSR) is stored for observation.

```
  mov r5, #0x 2100     ;starting address for response store
  mov r6, #0           ;loop-index preparation
  mov r0, #0x 2400     ;starting address for pattern fetch
atpg_label_1
  ldr r1, [r0], #4     ;load test pattern, after loading,
                       r0 = r0+4
  ldr r2, [r0], #4     ;load test pattern
  subs r3, r1, r2      ;test pattern application (subtract)
  str r3, [r5], #4     ;store for result(r3) observation
  mrs r9, cpsr         ;move cpsr to r9
  str r9, [r5], #4     ;store for result(r9) observation
  add r6, r6, #1       ;increment of the loop index
  cmp r6, #13          ;compare r6 with 13(total 13 iterations)
  bne atpg_label1      ;branch if not equal.
```

2) Test Routine Development for Data Memory Address Manipulation IP: We choose the adder that calculates the data address of the multiple-load instruction (ldmda) as the example to illustrate the test routine development through constrained ATPG. This adder has a 32-bit input that receives a value from a general purpose register specified by the ldmda instruction as the base address and another input represents the number of memory accesses of the multiple-load instruction (from 1 to 16). The first constraint, which is a mandatory constraint, is to limit the least significant two bits of the generated patterns to 00 to ensure word alignment of memory accesses. Besides, an optional constraint can be set to limit the most significant 12 bits of the test pattern to zero, reducing the memory blocks needed.

Another place for setting an optional constraint is the input that represents the access number of the multiple load instruction. For instance, the maximum access number can be limited to two instead of 16 specified by the ISA. In this way, the number of the test cycles can be reduced. By exploring the fault coverage attained from the constrained ATPG, a satisfying access number can be determined easily.

As a result, the ATPG tool generates 23 patterns for this fundamental IP. Before placed in the memory, these patterns are sorted according to the indicated access number. The final instruction sequence is as follows. When the ldmda instruction is executed, both the bit pattern in r1 and the access number will be fed to this adder. Then, the calculation result, i.e., the data address, is propagated to the primary outputs (data address pins) where the result can be observed.

```
  mov r6, #0              ;loop-index preparation
  mov r0, #0x 00240000;starting address for pattern fetch
atpg_label_1              ;loop for the 2-access ldmda
  ldr r1, [r0], #4        ;load the base address into r1
  ldmda r1, {r3, r4}      ;activate the targeted adder
                          ;load data at addresses r1 & r1 + 4
                          ;from memory to r3 and r4
                          sequentially
  add r6, r6, #1          ;increment of the loop index
  cmp r6, #20             ;total 20 iterations for 2-access
  bne atpg_label_1        ;branch if not equal


  mov r6, #0              ;loop-index preparation
atpg_label_2              ;loop for the 1-access ldmda
  ldr r1, [r0], #4        ;load the base address into r1
  ldmda r1, {r3}          ;activate the targeted adder
  add r6, r6, #1          ;increment of the loop index
  cmp r6, #3              ;total 3 iterations for 1-access
  bne atpg_label_2        ;branch if not equal.
```

3) Test Routine Development for Instruction Memory Address Manipulation IP: We demonstrate the test routine development strategy for instruction memory address manipulation IPs with the following example. Inside the processor, a subtractor is used to calculate the return address of a software interrupt (SWI) which is activated by the SWI instruction. When an SWI is raised, the address of the next instruction is stored internally for the use of future return. In many RISC processors, the stored address is the PC value at this moment subtracted by a constant, 0xC in our example. Thus, the subtractor IP has a 32-bit input that is fed with the value of the PC and another input is simply the fixed value. For this fundamental IP, the ATPG tool generates 15 test patterns which represent the PC values.

Applying the mandatory constraint that limits the least significant 2 bits of the generated patterns to 0 ensures the word alignment of instructions. Additionally, an optional constraint is set to limit the most significant 12 bits of the generated patterns to zero. Therefore, the SWI instructions will be centralized at some memory blocks. On the other hand, if these constraints are not set, the SWI instructions will scatter over the 4-GB memory space, making it not practical for program execution.

In order to set the PC to the desired value, the test routine should execute instructions at the corresponding address, and consequently by placing the SWI instructions at addresses indicated by the generated patterns (there is a fixed offset between the addresses of SWI instructions and the generated pattern values). The following sequence is one of the 15 instruction blocks that invoke the instruction memory address manipulation IP:

```
@01F2C5BE                    ;starting address determined by ATPG
  EF123456 //swi 0x 123456   ;activate a software-interrupt
  E4964004 //ldr r4, [r6], #4 ;load next address from
                             memory
  E1A0F004 //mov pc, r4      ;jump to next SWI block.
```

To observe the fault effect of this fundamental IP, the value of r14_svc (a general purpose register where the calculated return address is deposited) is written to the memory in the SWI handling routine (not shown previously). After the return from the SWI handling routine, the following load instruction loads the address of the next SWI instruction from the memory (the 15 addresses of SWI instructions are placed in the memory before test program execution) and this value is written to the PC (a jump operation) for the next execution of the SWI instruction.

D. Test Routine Development for Pipeline-Related Control Logic

To test a pipeline processor effectively, we explore the pipeline architecture of the target processor for test program development.

1) Test Routine Development for Control Hazard Detection Logic: In a classic five-stage pipelined processor with always not-taken for branch prediction, when a branch is taken, the IF and ID stage must be flushed for correct program execution. This means that the decoding of the instruction following the branch must be disabled, and the fetching of the next instruction must also be disabled, too. To functionally test these mechanisms, we can insert another control-transfer instruction following the branch. For example, note the following:

```
b branch_target    ;branch
swi 0x 123456      ;raise an exception
....
```

In the viewpoint of normal program execution, the instruction following the branch is meaningless. However, from the viewpoint of functional testing, it can check the correctness of the pipeline control hazard mechanism. The effect of the SWI instruction is easily observable, if it is decoded, a control transfer (to the exception vector) will occur. Thus, we can observe the fault effects on the instruction address bus. Alternatively, other types of instructions can be used to test this mechanism with the arrangement for fault observation as shown in the following:

```
b branch_target    ;branch
mvn r1, r1         ;invert r1's value
...
branch_target
str r1, [address]  ;store r1
....
```

In this example, if the ID stage flush mechanism does not function correctly, the value in r1 will be inverted, and the store instruction at branch target will propagate the fault effect to the primary output. Note that flushing incorrect fetched instruction in the pipeline is not centric to a particular implementation. It is a common practice in pipeline implementation when dealing with branch hazards. More details on this issue can be found in [15].

The method of testing the IF flush mechanism is similar. Normally, when a branch occurs, the IF stage will be disabled and a bubble instruction is inserted into the pipeline. If this is not the case, an instruction will be fetched and cause unpredictable results after decoding. We can append an instruction with well observable effects, such as software interrupt or branch, after the previously mentioned ID-flush-testing instruction to test this mechanism.

2) Test Routine Development for Pipeline Forwarding Mechanism: In a pipeline processor, the forwarding mechanism is often implemented to improve the performance of the processor. Intuitively, to test the forwarding logic functionally, the following instruction sequence can be used:

```
mov r0, #0x ff
add r2, r0, r1     ;forward to r0
str r2, [address]  ;response observation.
```

In this example, the result of the mov instruction is forwarded to the add instruction; this sequence will activate the forwarding mechanism. However, considering the processor pipeline architecture, we can further raise the fault coverage by adding the following two instructions before the previous sequence:

```
mov r0, #0x 12
ldr r0, [address_1] ;load a value other than 0x 12 and 0x ff
mov r0, #0x ff
add r2, r0, r1      ;forward to r0
str r2, [address_2] ;observation.
```

A forwarding path is typically implemented by a register-index comparator and an operand multiplexer. In a classic five-stage pipeline, the multiplexer has three data inputs, input from the EXE stage, input from the MEM stage, and input from the physical general purpose register. As mentioned previously, when testing a multiplexer, we must differentiate its inputs to prevent fault masking. If the three inputs of the forwarding multiplexer are all the same, we will not be able to observe the faults on the select signals, and thus the faults in the register-index comparator.

In the previous sequence, the first three instructions contribute to the three inputs, respectively, (three possible sources of r0 for the add instruction), and their results arrive at the same time due to the pipeline architecture, and thus the three inputs are differentiated. The first two instructions seem meaningless because the value of r0 will be overwritten immediately, but in views of testing they can help detecting more faults.

A pipeline processor may use a different implementation model such as three-stage or seven-stage, or even more. Without the loss of generality, the rationales presented above can be extended in the in-order execution models of different pipeline depth.

V. Experimental Results

We use ModelSim from Mentor Graphics, Design Analyzer from Synposys for logic simulation and logic synthesis. TurboScan from SynTest [16] is used for ATPG and fault simulation [17].

A. Fault Simulation

Fig. 6 depicts the flow of fault simulation which is performed. We first prepare the memory image file that contains the instructions and data required for test program execution. The test program has two parts: scattered instructions and sequentially executed instructions. The sequences for the testing of the fundamental IPs for return address backup belong to the former, and the other instructions belong to the latter. The sequentially executed instructions and the ATPG generated data can be placed at

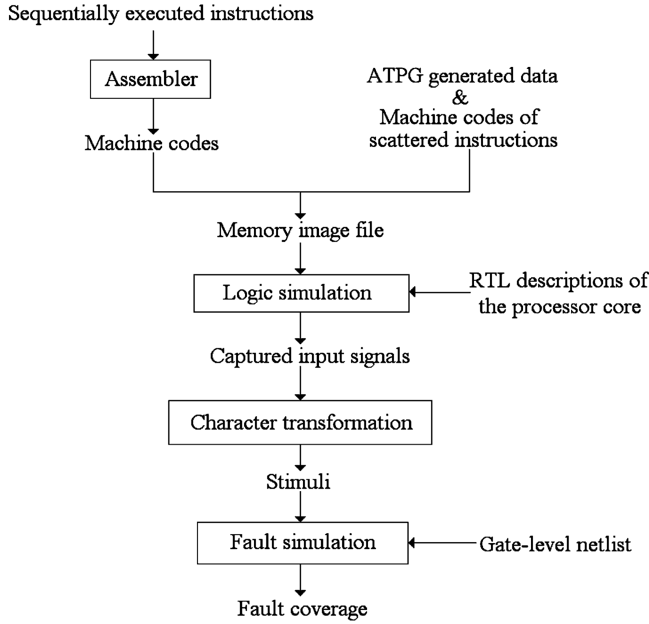


Fig. 6. Flow of fault simulation.

a desired place in the memory; however, the scattered instructions must be placed at the addresses indicated by the ATPG generated patterns.

After the preparation of the memory image file, we perform the logic simulation using ModelSim to acquire the 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 implement Verilog code 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 capture code in the test bench is shown as follows:

```
always @(posedge CLK or negedge CLK) begin
  $fwrite(fp3, "%d ", half_cycles);
  $fwrite(fp3, "%h ", uEASY.uARM9.uARM9TDMI.ID);
  $fwrite(fp3, "%h ", uEASY.uARM9.uARM9TDMI.DDIN);
  $fwrite(fp3, "%h ",
        uEASY.uARM9.uARM9TDMI.CHSDE);
  $fwrite(fp3, "%h ",
        uEASY.uARM9.uARM9TDMI.CHSEX);
  $fwrite(fp3, "%h ", uEASY.uARM9.uARM9TDMI.CPDIN);
  $fwrite(fp3, "%b ", uEASY.uARM9.uARM9TDMI.GCLK);
  $fwrite(fp3, "%b ", uEASY.uARM9.uARM9TDMI.rst_n);
  $fwrite(fp3, "%b ", uEASY.uARM9.uARM9TDMI.nWAIT);
  $fwrite(fp3, "%b ",
        uEASY.uARM9.uARM9TDMI.IABORT);
  $fwrite(fp3, "%b ",
        uEASY.uARM9.uARM9TDMI.DABORT);
  $fwrite(fp3, "%b ",
        uEASY.uARM9.uARM9TDMI.BIGEND);
  $fwrite(fp3, "%b ",
        uEASY.uARM9.uARM9TDMI.HIVECS);
  $fwrite(fp3, "%b ", uEASY.uARM9.uARM9TDMI.nfiq);
  $fwrite(fp3, "%b ", uEASY.uARM9.uARM9TDMI.nirq);
  $fwrite(fp3, "%b\n",
        uEASY.uARM9.uARM9TDMI.CPEN);
End.
```

In the previous code, fp3 is the file pointer that represents the file where the input signals are stored, and the last part of each line is the signal (interface) name of the processor core. We capture input signals twice in a clock cycle. The recorded data do not conform to the TurboScan's input format, so a simple character transformation is required. The experimental processor core has 110 input pins, so after the 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 transformed stimuli and the gate-level netlist of the processor as inputs and get the fault coverage, which is based on the structural single stuck-at fault model.

B. *Synthesis Case 1: TSMC 0.35 μm, 30 MHz*

In this case, we have synthesized the target processor using the TSMC 0.35-μm library. The synthesized processor has a gate count of 45 046 (two-input NAND-gate equivalence) and operation speed of 30 MHz. We develop the test program according to our proposed methodology and produce a test program of 1747 instructions (6988 bytes). It takes only 18 675 (with cache) or 24 073 (without cache) cycles to execute this test program. The size of the stored patterns generated by the ATPG tool is 2120 bytes. A total of 9108 bytes of data (instructions + patterns) must be placed into the memory before self-testing.

The resultant fault coverage of each part as well as the processor is listed in Table II. For comparison, the fault coverage of both the full processor and the functional modules from full scan chain is also presented. The MLA-SBST methodology has achieved 93.74% in processor coverage. This result indicates that the proposed pure software-based self test methodology has shown promising test quality for the complex pipeline processor. The difference of the fault coverage between applying the full scan chain and the MLA-SBST methodology is about 4%–5%.

The fault coverage for each group of the pipeline registers, the register file, and the status registers for this synthesis case is listed in Table III. As expected, the fault coverage for the ISA registers has achieved the best coverage among the registers.

C. *Synthesis Case 2: TSMC 0.35 μm, 50 MHz*

In the second case, we have synthesized the target processor using the TSMC 0.35-μm library but with different parameters for a higher speed. The synthesized processor has a gate count of 63281 (two-input NAND-gate equivalence) and an operation speed of 50 MHz. We redo the ATPG parts of the test program development for the fundamental IPs while the remained parts are the same as the first synthesis case. The change in the test program size and the execution time is quite small. The fault coverage of each processor part and comparisons are shown in Table IV. It is noted that the test program developed for synthesis case 1 has shown only a small degradation (0.04%) in

TABLE II
BREAKDOWN OF FAULT COVERAGE FOR THE PROCESSOR IN SYNTHESIS CASE 1

Component	Number of faults	SBST-MLA	Full scan chain
Register file & access control	47394	93.15%	96.55%
ALU	43292	97.73%	98.04%
Shifter	3598	98.75%	99.92%
Memory access unit	12176	92.94%	98.16%
Instruction fetch unit	2176	80.97%	93.02%
Decoder	2878	87.04%	96.00%
Status registers & access control	5932	83.45%	95.98%
Coprocessor access unit	1154	86.74%	94.12%
Exception handling unit	308	86.69%	85.31%
Other	9936	90.19%	95.88%
ARM9-v4 compatible processor (MLA-SBST)	128,844	93.74%	
ARM9-v4 compatible processor (full scan chain)	137,258		97.88%

TABLE III
FAULT COVERAGE OF VARIOUS REGISTERS IN SYNTHESIS CASE 1

Register (not including access control logic)	Number of faults	Fault coverage
Pipeline register IF/ID	320	90.31%
Pipeline register ID/EXE	3844	86.99%
Pipeline register EXE/MEM	2700	87.81%
Pipeline register MEM/WB	580	88.60%
Register file	9952	97.13%
Status registers	1920	99.69%

fault coverage when directly used in the processor of this synthesis case. The difference comes from the fundamental IPs in the memory access unit.

D. Result Discussions

The fault coverage of the register bank cannot reach near 100% coverage because the write-enable pins of the r14s (6 r14s in different modes) are hard to test functionally. In the ARM architecture, the r14 of each processor mode is used to store the return address of branch-and-link or exceptions, so it has a data input source not shared by the other registers. The method of testing the write-enable pins used for the general purpose registers is not applicable to this irregularity of structure. With a more regular register file structure such as the MIPS processor, we expect the proposed test methodology can achieve a higher coverage than those obtained from the ARM-compatible processor.

Among the undetected faults, some faults are functionally untestable, for example, bit 4 of the status registers (M4 of the processor mode) are always 1 in any circumstance, so the stuck-at-1 faults on the data interfaces of the D flip-flops representing these bits cannot be detected functionally. However, this stuck-at-1 fault will not affect the normal operation of the processor. Hence, for the 6 status registers of the processor, the fault coverage can be seen as 100% with our methodology. Due to the limitation of the simulation tool, a quantitative analysis of fault coverage based on functionally untestable faults is not available.

The fault coverage of the pipeline registers ranges from 87% to 90%. In general, faults related to the data or clock interfaces of the D flip-flops composing the pipeline registers are easy to detect. The undetected faults of the pipeline registers are mainly the faults related to the write-enable and reset pins which are hard to test or even functionally undetectable. To improve fault coverage of the pipeline registers, two possible methods can be explored: 1) providing special instructions to access the pipeline registers and 2) using randomly generated test code. Integrating randomly generated test code is also useful in improving the coverage for control logic such as the decoder.

There are about 17 fundamental IPs in the processor core of synthesis case 1. The number of generated test patterns for each IP is small, generally ranging from 20 to 40 patterns, but in some special cases the number may be up to 50. The test sequences for most of the fundamental IPs can be coded as a compact loop. The instruction number of each loop ranges from 10 to 20, and the iteration number depends on the number of the generated patterns. The fault coverage of each fundamental IP ranges from 90% to 100%; the undetected faults are generally caused by the constraints or the redundant faults in the fundamental IP itself.

1) Application and Limitation: The effectiveness of a deterministic SBST test program highly depends on the knowledge about the processor's ISA, architecture, and RTL implementation. The proposed method for the testing of pipeline mechanisms such as control hazard detection logic and forwarding logic can be applied to a processor of the similar pipeline structure. For a processor with more complex features such as superscalar capability or out-of-order execution, it is beyond this study.

Due to the nature of deterministic programming, the test development requires manual efforts; however, through multiple abstractions, we can take advantage of the modern synthesis and fault simulation tool to improve the efficiency of test development. The method of using constrained ATPG for fundamental IP units can also be used to other processor cores that are generated through a synthesis tool. To extract the constraint for a fundamental IP, the test code developer just needs to find out the functionality of the selected IP for constraint setting and relies on the ATPG tool to get satisfied results. This is not an automated solution so far; however, it shows the feasibility that a testing and synthesis integrated tool might eventually exploit the architecture feature and generate the recommended constraints for the testing of a fundamental IP.

TABLE IV
BREAKDOWN OF FAULT COVERAGE FOR THE PROCESSOR IN SYNTHESIS CASE 2

Component	Number of faults	F.C. after re-doing IP	F.C.before re-doing IP	Full scan chain
Register file & access control	56406	93.68%	93.68%	97.04%
ALU	45244	97.35%	97.35%	98.60%
Shifter	7272	97.21%	97.21%	99.20%
Memory access unit	14398	92.22%	91.64%	97.89%
Instruction fetch unit	2446	81.40%	81.40%	93.67%
Decoder	3110	86.60%	86.60%	96.01%
Status registers & access control	5864	83.51%	83.51%	95.94%
Coprocessor access unit	1138	86.56%	86.56%	95.64%
Exception handling unit	298	89.26%	89.26%	85.03%
Other	16582	90.13%	90.12%	95.96%
ARM9-v4 compatible processor (MLA-SBST)	152,758	93.62%	93.58%	
ARM9-v4 compatible processor (full scan chain)	161,172			98.15%

TABLE V
COMPARISONS OF VARIOUS SBST WORK AND FULL SCAN CHAIN

	CPU	Methodology style	Gate count/fault number	Execution cycle	Program size	F.C. %
[1]	32-bit MIPS	Deterministic	37,402 /N.A.	10,061	1,728 w.	92.60
[2]	8-bit 8051	Random	6,000 /N.A.	N.A.	624 inst.	85.19
[3]	8-bit 8051	Random	12,000/ 28,792	N.A.	N.A.	90.77
[4]	16-bit DLX	Random + DFT hardware	27,860/43,927	50,000	166 inst. seeds.	92.50
				220,000	166 inst. seeds.	94.80
[5]	8-bit PARWAN	Random	888 gates + 53 FFs/N.A.	137,649	1,129 (bytes)	91.42
[6]	8-bit PARWAN	Deterministic	1,300/N.A.	16,572	885 bytes	91.10
[7]	EX1 module of Xtensa processor	SBST	N.A./ 24,962	27,248	7602 inst./ 20373 (bytes)	95.20*
MLA-SBST	ARM9-v4 compatible processor	Deterministic	45,046/128, 844 (0.35 um/ 30MHz)	18,675/ 24,073 (cache disabled)	1747 inst./ 530 data (words)	93.74
MLA-SBST	ARM9-v4 compatible processor	Deterministic	63,281/152,758 (0.35 um/ 50MHz)	18,937/ 23,809 (cache disabled)	1743 inst./ 552 data (words)	93.62
	ARM9-v4 compatible processor	Full scan chain	49,961/137,258 (0.35 um/ 30MHz)	N.A.	N.A.	97.88
	ARM9-v4 compatible processor	Full scan chain	68,194/161,172 (0.35 um/ 50MHz)	N.A.	N.A.	98.15

*Fault coverage based on functional testable faults.

Our observations from the synthesis result of the processor core motivate the development of the constrained-ATPG based technique because the simple fundamental IP units the synthesis tool inserts have occupied a good portion of the processor core and contributed about 35% of the faults in the fault list. Because of the simplicity of the fundamental IP in its usage, this eases the work for constraint extraction once the functionality of the IP is identified, and thus allows the code development to take the advantage of the ATPG technique for high fault coverage. However, for a less known IP or complicated module, constraints can be quite complex and constraint extraction can become a key challenge [7], [18]–[20].

For the hard-to-test steering logic, such as the multiplexers, we not only explore the RTL code but also the effect of fault masking to improve test result. However, in general deterministic SBST such as ours has limitation on the testing of control logic; one way to improve the fault coverage for this part is further applying randomly generated test code.

E. Comparisons With Other Works

Table V summarizes the comparisons of the MLA-SBST methodology with other works. It is worth of pointing out that among the processors examined in [1]–[6], only our target processor core possesses the full functionality of pipeline hazard control, forwarding mechanism, and modern operating system supports.

In [7], not all the experimental results of the processor are available; only the fault coverage based on functional testable faults for a large logic module, called EX1, which was extracted from the Xtensa processor is shown. Assessing the scale and complexity of the processor core tested, the proposed MLA-SBST methodology clearly presents itself to be a promising solution for processor functional testing.

VI. Conclusion

We have presented a MLA-based SBST methodology and demonstrated its application on a complex pipeline processor with different gate-level implementations. The proposed methodology explores the design information of processor architecture, RT-level, and gate-level for different types of the processor components. The test routine development methodology applies the most useful information of a certain level to the different parts of the processor core. Exploiting processor architecture improves the selection and coding efficiency in developing the test routine. Using gate-level information has shown to provide good results in structural fault coverage, especially for regular state elements. To test fundamental IPs effectively, we present an ATPG-based scheme for the generation of the constrained test patterns. The methodology aims to develop simple yet high fault-coverage test routines for the fundamental IPs with a systematic approach. For processor components that tend to be structurally unmanageable, we resort to the information from the RT-level; however, randomly generated test code can be useful to improve the test quality of this part. We have demonstrated that the proposed SBST methodology has attained 93.74% in processor fault coverage, about 4%–5% 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.

Acknowledgment

The authors would like to thank the reviewers for their many helpful suggestions and Prof. K. J. Lee for his useful discussions.

References

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

[2] F. Corno, M. Reorda, G. Squillero, and M. Violante, "On the test of microprocessor IP cores," in *Proc. Design Autom. Test Eur.*, 2001, pp. 209–213.

[3] F. Corno, G. Cumani, M. Reorda, and G. Squillero, "Fully automatic test program generation for microprocessor cores," in *Proc. Design Autom. Test Eur.*, 2003, pp. 1006–1011.

[4] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. 17th IEEE VLSI Test Symp.*, 1999, pp. 34–40.

[5] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, Mar. 2001.

[6] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective software self-test methodology for processor cores," in *Proc. Design Autom. Test Eur.*, 2002, pp. 592–597.

[7] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. 40th Design Autom. Conf.*, 2003, pp. 548–553.

[8] C. H.-P. Wen, L.-C. Wang, K.-T. Cheng, W.-T. Liu, and J.-J. Chen, "Simulation-based target test generation techniques for improving the robustness of a software-based-self-test methodology," in *Proc. IEEE Int. Test Conf. (ITC)*, 2005, pp. 936–945.

[9] W.-C. Lai and K.-T. Cheng, "Instruction-level DFT for testing processor and IP cores in system-on-a-chip," in *Proc. DAC*, 2001, pp. 59–64.

[10] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *Proc. Design Autom. Test Eur.*, 2001, pp. 92–96.

[11] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 88–99, Jan. 2005.

[12] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS-A microprocessor functional BIST method," in *Proc. Int. Test Conf.*, 2002, pp. 590–598.

[13] A. Burdass, G. Campbell, and R. Grisenthwaite *et al.*, "Embedded test and debug of full custom and synthesisable microprocessor cores," in *Proc. IEEE Eur. Test Workshop*, 2000, pp. 17–22.

[14] *"ARM Architecture Reference Manual,"* ARM Corp., (2000). [Online]. Available: http://www.arm.com, ARM DDI 0100E

[15] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2005.

[16] *"TurboScan Reference Manual,"* Syntest Corp., 2002 [Online]. Available: http://www.syntest.com/, version 2.6

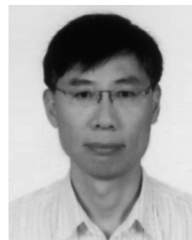
[17] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing*. Norwell, MA: Kluwer Academic, 2000.

[18] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," in *Proc. Int. Test Conf.*, 1997, pp. 743–752.

[19] R. S. Tupuri, A. Krishnamachary, and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," in *Proc. Design Autom. Conf.*, 1999, pp. 647–652.

[20] V. M. Vedula and J. A. Abraham, "A novel methodology for hierarchical test generation using functional constraint composition," in *Proc. High-Level Design Validation Test Workshop*, 2000, pp. 9–14.

[21] C. H.-P. Wen, L.-C. Wang, K.-T. Cheng, K. Yang, W.-T. Liu, and J.-J. Chen, "On a software-based self-test methodology and its application," in *Proc. 23rd IEEE VLSI Test Symp.*, 2005, pp. 107–113.



Chung-Ho Chen (M'06) received the M.S. degree in electrical engineering from the University of Missouri, Rolla, in 1989, and the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 1993.

He is currently a Professor with the Department of Electrical Engineering, National Cheng-Kung University, Tainan, Taiwan, R.O.C. His research interests include advanced computer architecture, SoC testing, video technology, and network storages. He co-holds one U.S. patent on a multicomputer cluster-based processing system and one R.O.C. patent on a multiple-protocol storage structure.

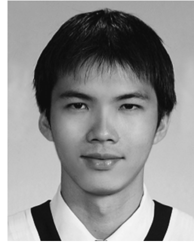
Dr. Chen was the Technical Program Chair of the 2002 VLSI Design/CAD Symposium held in Taiwan, R.O.C.

Chih-Kai Wei received the B.S. degree in electrical engineering from the National Cheng-Kung University, Tainan, Taiwan, R.O.C., in 2004, and the M.S. degree from the Institute of Computer and Communication Engineering, National Cheng-Kung University, in 2006.

His research interests include computer architecture and testing for embedded microprocessors.

Tai-Hua Lu received the B.S. degree in electronic engineering from the National Yunlin University of Science and Technology, Yunlin, Taiwan, in 2000. He is currently pursuing the Ph.D. degree in electrical engineering from National Cheng-Kung University, Tainan, Taiwan.

His primary research interests include computer architecture, SOC design, and processor testing.



Hsun-Wei Gao received the B.S. degree in engineering science and the M.S. degree in computer and communication engineering from the National Cheng-Kung University, Tainan, Taiwan, R.O.C., in 2004 and 2006, respectively.

His research interests include operating systems and processor design.