Software-Based Self-Testing of Processors Using Expanded Instructions

Ying Zhang1,2  Huawei Li1  Xiaowei Li1

1 Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences  Beijing  100190
2 Graduate School of Chinese Academy of Sciences, Beijing, China, 100049
{zhangying, lihuawei, lxw}@ict.ac.cn

Abstract—In this paper, an automatic test instruction generation (ATIG) technique using expanded instructions is presented for software-based self-testing (SBST) of processors. First, mappings between expanded instructions and signals are obtained through data mining, and they are used to impose value ranges of expanded instructions on component signals and generate instruction-level constraints. Second, virtual circuits are established based on the instruction-level constraints, and test patterns are generated for the constrained components. Third, test patterns are translated into test instructions according to the values of controlling signals and constraints for their mappings to instructions, and an SBST program is produced after assembling the test instructions. Experimental results on the Parwan processor show that the proposed ATIG technique can achieve 94.8% stuck-at fault coverage, which is close to that of the full-scan test generation method. In addition, it can cut down 57% test volume of the previous random pattern generation based SBST technique, while the test time reduces to one thirteenth of the previous SBST technique.

Keywords— Software-Based Self-Test; Automatic Test Instruction Generation; Instruction-Level Constraint; On-Line Test

I. INTRODUCTION

Due to the increasing frequency gap between high-performance processors and ATE, it is difficult to implement full-scan test at-speed, while lack of at-speed test would greatly degrade the test quality of deep sub-micron chips [1]. Alternatively, build-in self-test (BIST) has been widely used to support at-speed test [2], but it would bring large area overhead, performance degradation and excessive power dissipation [3]. Subsequently, software-based self-test (SBST) [3] has emerged as an effective method for manufacturing and online testing of processors, which has achieved satisfactory fault coverage on a series of processors. Because usually much manual work is involved in the process of SBST program generation, certain methods to assist or automate this process are desirable.

The early functional test methods for processors can be regarded as functional SBST methods [4]. The first functional test method for processors was proposed in [5] [6], which considers functional fault models instead of structural fault models. Then an automatic test generation method was presented for functional testing of gate-level components [7], which extracts functional constraints with FALCON [8], models these constraints as virtual circuits on component ports, and applies automatic test pattern generation (ATPG) on the constrained components. Later, a structural SBST method was presented in [3] to testing structural faults on processors, which applies instructions to generate random test patterns on-chip, called random pattern based SBST (RSBST) in this work. SBST is now referred to structural SBST methods if not specifically addressed. It provides an effective solution for at-speed testing structural faults on processors, in which test programs are loaded into the on-chip memory at low speed, while the actual tests are generated and applied at-speed inside the chip. Besides, SBST can also be used to test a core-based system or to diagnose faults in a processor [9]. A scalable SBST [10] method was presented, which applies statistical regression to obtain mapping between settable fields on instruction templates and major component signals, extracts instruction-level constraints for ATPG, and finally achieves high fault coverage on a commercial processor.

Some researchers took advantage of inherent regularity of functional components, and delicately designed deterministic tests at register-transfer level (RTL) [11]. Such methods can achieve over 90% fault coverage on different processors and reduce test cost significantly [12], even on the full-pipeline processor [13] and multiprocessors [14]. MLA-SBST [15] [16] uses the information abstracted from different abstract levels, such as gate-level, RTL and architecture-level, and improves fault coverage apparently. Hybrid-SBST [4] combines deterministic test, verification-based self-test and directed random test pattern generation (RTPG) [17] together for testing processors, which provides a very effective method for testing hard-to-detect faults.

Because it is difficult to automate test generation on RTL or Hybrid models, we target instruction-aware automatic test generation at gate-level for SBST. The proposed automatic test instruction generation (ATIG) technique has the following contributions.

I. It applies expanded instructions for extracting instruction-level constraints, which covers all of the signals on component under test. Meanwhile, a decision-tree algorithm is used to obtain the mappings of controlling signals to instructions, which is more suitable for discrete and two-value signals in comparison with statistical regression methods.
It extracts controlling signals from component signals, translates test patterns into test instructions based on these signals, and produces the SBST program automatically.

This paper is organized as follows. Section II covers necessary background for the example processor structure and the classification method. The proposed ATIG is described in Section III in detail. Experimental results are given in Section IV. Finally we conclude this paper in Section V.

II. PRELIMINARIES

A. Processor Structure and Expanded Instructions

This paper uses the processor Parwan [18] as a bench to illustrate the proposed ATIG technique. As shown in Figure 1, it contains eight components, while ALU, the shifter SHU, PC, and the controlling unit CTRL are four main blocks. Because CTRL is mainly affected by the instruction types, all types of instructions can be applied in an SBST program for testing CTRL. So we only target SBST program generation for the former three units. In Parwan, there are four flag signals, such as zero and negative, and 17 types of instructions, while six of them are single-byte instructions.

2. When records of \( t_i \) relate to several values on the target attribute, the attribute is built as a leaf node.
3. The function is ended till all condition attributes are covered or all records relate to the same value.

Finally, a decision tree is obtained, which covers all rules between condition and target attributes. If a decision tree is built up for every signal, the mappings between instructions and signals are gained. An open tool of data mining, Weka [20], is used to accomplish this process.

III. AUTOMATIC TEST INSTRUCTION GENERATION

In this work, ATIG is used to automatically generate test programs for SBST, as if ATPG provides patterns for full-scan testing. The framework of ATIG is shown in Figure 2. First, mappings between expanded instructions and signals are obtained through data mining, and they are used to impose value ranges of expanded instructions on component signals and generate instruction-level constraints. Second, virtual circuits are established based on the instruction-level constraints, and test patterns are generated for the constrained components. Third, test patterns are translated into test instructions according to the values of controlling signals and constraints for their mappings to instructions, and an SBST program is produced after assembling the test instructions. In the following sub-sections, ATIG is implemented on Parwan, and every step is described in detail.

A. Constraint Abstraction on Parwan

In [7-8], functional constraints on registers, such as instruction structure and address range, are not considered during constraint abstraction. An instruction-level constraint extraction method was brought in [10], which obtains mappings between instructions and component signals. This method only focuses on the operating code and the operands of instructions. Several other signals are not considered, such as flag and controlling signals. Moreover, it adopts statistical regression, which is not suitable when the target attribute is discrete [19]. For example, when statistical regression is used to obtain mappings between settable fields and component signals under Boolean function, it faces a great problem [10]. Instead of applying constrained ATPG, a simulation based test generation technique was presented in [2] to establish learning models between component ports and processor ports, apply ATPG on the component directly, verify the test patterns through the learning models, and finally map them as processor input signals. However, it is not suitable to some components that only relate to inner information of the circuit, such as PC.
In this work, an expanded-instruction based method is proposed for instruction-level constraint abstraction, which obtains maps bit by bit through building a decision tree. Compared with statistical regression, building decision tree is suitable for both Boolean function and arithmetic function. In addition, the proposed expanded instruction-based method can cover all component signals, and is also suitable for components only relating to inner information of the circuit.

First, a training program, which is possibly a compacted verification program and can achieve high functional coverage on the targeted components, is implemented on both the signal simulator (such as Modelsim) and the instruction simulator, as shown in Figure 3. Second, the instruction simulator is used to record all processor information when running the program and producing the expanded instructions, while the signal simulator provides related information on component signals. Subsequently, critical segments of simulation results are combined together as a training list. For example, considering ALU, Modelsim can report a list of ALU signals during instruction operation, and the list is connected with expanded instructions, finally the items for the last state of every instruction are extracted as a training list. Third, the training list is sorted according to instruction types. For each type of instructions, mappings between component signals and expanded instructions are obtained through building decision trees. However, the decision tree may have over-fitting problems [19], and produce redundant rules (i.e., mappings) for the training list. So it is necessary to validate its rules. Fourth, test programs with random instructions are established to validate the rules for each instruction one by one. Once the rules are not satisfied, its test program is added into the training list, and the algorithm returns to the third step. Finally, a valid mapping will be obtained for each type of instructions.

The first type of signals are combined together, and used as a guide for both extracting constraints and creating test instructions, called controlling signals (CSs) in this work. When extracting constraints, the space of every signal $S_i$, $Space_{cs}$, can be described as the combining value range of the signal $S_i$ under every instruction $instr$ in the controlling instruction set $I_c$ [3]. For CSs, their fixed values for all instruction types establish their space. In other words, a set of values on CSs usually indicate the possible instruction types. For other signals, the type of instructions, $CS_{instr}$, should be considered, while the value range of the signal is altering with the value range of related instruction bits $R(I_t)$ through mappings. As shown in equation 1, their signal space is the combination of the value range related on instruction bit $I_t$ for each type of instructions.

$$Space_{si} = \bigcup_{instr \in I_t} (CS_{instr} \land mapping(R(I_t)))$$  \hspace{1cm} (1)

In conclusion, the simulation based constraint abstraction is very suitable for processors. Because most component signals are discrete and two-value, it is very effective to obtain mappings through building decision trees. Meanwhile, expanded instructions can cover all component signals. Then value ranges of instructions are imposed on signals through equation (1), and instruction-level constraints are obtained automatically.

### B. Test Pattern Generation on Constrained Components

**Figure 4** Three Types of Signals in Expanded Instructions

![Figure 4](image)

**Figure 5** ATPG on Constrained Component
To generate test patterns for both combinational and sequential units, virtual circuits are established with the signal space obtained above, and that help the implementation of constrained ATPG. On combinational circuits, all points in the signal space with value 1 on a constrained signal are extracted, and then they are combined as a conjunctive normal form. After simplifying the form, a logic circuit is established. For example, \((S_{17}, S_{18}, S_{19})\) are CSs of the ALU component, whose signal space has 6 points as shown in Figure 5(a). When designing a virtual circuit for signal \(S_{19}\), three points, \(\{001, 101, 111\}\), are extracted. After simplification, the value of signal \(S_{19}\) would be determined by \(S_{17} i \& S_{19} i \mid \neg S_{17} i \& S_{18} i \& S_{19} i\) while \(S_{17} i, S_{18} i, S_{19} i\) are new introduced primary input signals. The virtual circuits for CSs are shown in Figure 5(b), which make the test patterns executable under the functional model. Finally, the virtual circuits are connected to the component under test in Figure 5(c), and ATPG is run directly on the extended combinational units.

On sequential circuits, such as PC unit, sequential patterns are generated with a sequential ATPG. Specifically, there are three CSs on PC unit (reset signal is ignored), whose space is \(\{100, 010, 011\}\). In the signal space, “100” is related to single-byte instructions and instruction “jsr”, “010” is related to branch instructions, and “011” is related to “jmp”. Because “jmp” can load both page and offset address, it covers the effect of branch instructions on PC unit. Besides, single-byte instructions solely make current address plus 1 on PC unit just like “jsr”. Therefore, only “jmp” and single-byte instructions are used to test PC unit, and the controlling signals are set to either “100” or “011”. After inserting proper virtual circuits, test patterns for the PC unit are obtained with a sequential ATPG.

C. Implementation of ATIG

ATIG implementation includes translating test patterns into expanded instructions with the guide of CSs and assembling these instructions as an executable program. Compared with the work in [22] that also translates test patterns into instructions, the proposed method can automate this process with the CSs and mappings rules obtained above. Moreover, it can deal with sequential patterns used for sequential units.

### Figure 6

First, CSs extracted from a test pattern indicate the instruction type, while other signals are translated into instructions according to the mappings under this instruction type. As shown in Figure 6, the CSs (“101”) solely indicate instruction “add”, and other signals are mapping into related instruction bits. However, there are two extra cases. In the first case, when the CSs relate to several instructions, it is required to generate related instructions and operate them one by one. Once an output result is fit to the predicted result, the matched instruction is chosen to implement the test pattern. For example, the CSs of test pattern T2 for ALU are “110”, which relate to three instructions “asr”, “asl”, and “sta”. Because only the result of “sta” (0x01) is fit to the predicted one, it is chosen to implement the pattern T2. In the second case, the CSs relate to most of instructions, and the other signals map to the output. For example, test pattern T3 for SHU has CSs of “000”, which indicates all instructions except of “asl” and “asr”, and the other signals map to the result or flag out. This implies that the unit solely transmits its input signals to outputs. Therefore, it only needs an observable instruction as the next instruction, which makes result and flag out equal to its OPB and flag in, such as I3’ in Figure 6.

Second, the expanded instructions of both the combinational and the sequential units are assembled as an executable program. For combinational units, OPB and flag in cannot be loaded from memory directly, so that a general former subroutine is used to initialize these segments. As shown in Figure 7, instruction “addi” is used as an initial instruction, whose two operands (input1 and input2) and Sum are calculated according to equation (2)–(4), while input1 is loaded in with instruction “ldai”. For example, to initialize OPB and flag in as 0x31 and “0100” (carry_in=1), the subroutine should load in 0xFF, and then add 0x32. Next, a general latter subroutine is used to output test responds on result and flag out. Because “sta” would not change flag out (the same to “cla”), it is used to output result directly. A 16-branch program, comprised of four types of branch instructions, is used to cover 16 cases of flag out. Once some bits of flag out have value 1, the related bits on AC unit are set as 1 through instruction “add”. For example, when flag out is “1001”, the value on AC unit is set as 0x09 through “add x9”. In this way, flag out can be observed together without repeating the test instruction.

Subroutine former

```
ldai pinput1
addi pinput2
```

Subroutine latter

```
sta out1
cla
brav if1
brac if01
braz if001
bran if0001
jmp if0000
label if1001
add x9
```

Figure 7 ATIG for Combinational Unit

\[
\begin{align*}
\text{input1} &= 255 \quad \overline{\text{flag in}} = 0 \quad \text{carry in} = 1 \\
\text{input2} &= \frac{\text{Sum}}{4} \quad \overline{\text{flag in}} = 0 \quad \text{carry in} = 0
\end{align*}
\]

(2)
For a sequential unit like PC unit, sequential patterns are translated into an instruction sequence based on CSs and the mappings, and then the sequence is assembled as an executable program. First of all, an indirect method [3] is used to observe faults on address bus. It defines a flag that is equal to the offset address of instruction “jmp”, such as SF3B, and it is stored into memory once “jmp” happens. If a fault happens on the address, the flag would not be stored on the right position, and the fault is detected indirectly. Specifically, an observing template is added before every “jmp”, which stores the former flag for observing and loads another flag for next instruction. In the following the instruction “jmp” is connected to its next instruction through a connecting template as shown in Figure 8.

In this way, an executable program for testing PC unit is obtained. Finally, the programs for combinational and sequential units are connected together, and the final SBST program is assembled well.

The fault coverage and test overhead of the proposed SBST method, ATIG, are analyzed in this section, while RSBST method [3, 15] and the full-scan ATPG method are used for comparison. The modified Parwan mentioned in [3], which has no bidirectional I/O pins and no tristate buffers, is used as the benchmark, and the experimental flows for SBST program evaluation are shown in Figure 9.

### IV. EXPERIMENTAL RESULTS

The fault coverage and test overhead of the proposed SBST method, ATIG, are analyzed in this section, while RSBST method [3, 15] and the full-scan ATPG method are used for comparison. The modified Parwan mentioned in [3], which has no bidirectional I/O pins and no tristate buffers, is used as the benchmark, and the experimental flows for SBST program evaluation are shown in Figure 9.

#### A. Fault Coverage of ATIG

The fault coverage on every component of Parwan is shown in Table 1 for all the three method. ATIG achieves 94.8% fault coverage, which is very close to that of full-scan test. On the data path unit, ATIG detects almost all structural testable faults. Note that due to DFT, the fault number of the full-scan test is larger than that of the SBST methods, which may result in more non-detected (ND) faults. On the controlling unit, the fault coverage of ATIG is nearly 11% smaller than that of the full-scan method. Our further analysis shows all ND faults of full-scan test and ATIG are non-observed (NO) faults, though ND faults should include non-controlled (NC) and non-observed (NO) faults. It means that although all faults on the controlling unit are sensitized using ATIG, the faults cannot be propagated as conveniently as in the full-scan test. It results in loss of fault coverage. In fact, most CSs from the controlling unit are not output directly, and they are easily blocked by other signals or blocks. The more complex the processor is, the more serious the situation would be. Therefore, it is necessary to consider the observation of CSs just like other signals when designing the SBST program.

Compared with RSBST, ATIG can achieve higher fault coverage on almost every component. On the data path unit, test patterns generated by ATPG are more efficient than random patterns, so that ATIG covers the testable faults more effectively. When the ND faults of RSBST are analyzed deeply, 8 NC faults are found out on the data path unit, while half of them are on PC. It means several faults are not activated by RSBST. For the controlling unit, ATIG has enhanced fault coverage apparently, more than 5% in comparison with RSBST [23]. That is because the test programs obtained by RSBST in [23] are lack of several types of instructions, such as “jsr”.

#### B. Test Overhead of ATIG

The major overheads of these methods are analyzed in this section, which include test volume, test time and area overhead. First, because all of them have to upload test patterns into the processor, test volume would determine the memory required and data upload time in the external ATE. As shown in Table 2, the application of ATIG requires 189 instructions (517 bytes) with 116 bytes of data. It cut down nearly 57% test volume of RSBST with two programs, “alu-shu” and “pc”, while the compiler “par_asm” reports they has 502 instructions (1067 bytes), and 67 instructions (147 bytes), respectively. Without a complex on-line test pattern generator and with compacted and regular structures, ATIG reduces the instruction count significantly. Compared with full-scan ATPG patterns, it saves about 39% test volume.

![Figure 8 ATIG for Sequential Unit](image)

![Figure 9 Experimental Flow](image)

First, an SBST program is compiled into a binary file, and loaded into the memory with test data. Second, gate-level simulation starts, and a VCDE file is obtained. Third, both the gate-level netlist and the VCDE file are loaded into a commercial ATPG tool, which reports detailed fault coverage on full-chip fault simulation. Because the unused wires are deleted when running fault simulation, the redundant faults are not included in the fault coverage below. In addition, 50% potentially detected (PT) faults are regarded as tested when calculating fault coverage.

### Table 2 Test Overhead

<table>
<thead>
<tr>
<th></th>
<th>Full-Scan</th>
<th>RSBST</th>
<th>ATIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>517</td>
<td>502</td>
<td>189</td>
</tr>
<tr>
<td>Data (Byte)</td>
<td>1067</td>
<td>116</td>
<td>517</td>
</tr>
<tr>
<td>Time (clock)</td>
<td>121105</td>
<td>6957</td>
<td>9413</td>
</tr>
<tr>
<td>Area Overhead</td>
<td>6.1%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Second, ATIG only requires 9413 clock cycles of test time. The test time of ATIG is almost 1/13 of RSBST (Because signal write and read are responded earlier in the experimental system, the test time of RSBST is smaller than that in [3]). There are three major reasons. First of all, the test patterns generated by ATPG are much less than random patterns. Moreover, it only takes 6 or 8 instructions to observe all flag_out bits with the latter subroutine, but RSBST has to observe flag_out bit by bit and even repeat the test instruction. Finally, test patterns of ATIG are not generated on-chip, which also saves much time. For full-scan ATPG method, it takes about 6957 clock cycles to apply patterns by ATE. Third, ATIG can be applied on-line without area overhead, while full-scan test brings in 6.1% extra area. In conclusion, ATIG is a very effective method for testing processors.

V. CONCLUSIONS

This paper shows a preliminary work to produce SBST programs automatically. In this work, instruction-level constraints are obtained from a simulation-based method, and used to establish virtual circuits to constrain components under test. After gate-level ATPG on constrained components, test patterns are translated into test instructions based on controlling signals and their mappings to instructions. Experimental results demonstrate that ATIG is an effective test method with the lowest overhead.

REFERENCES


Table 1 Fault Coverage of Full-Scan ATPG, RSBST and ATIG

<table>
<thead>
<tr>
<th>Components</th>
<th>Full-Scan Test</th>
<th>RSBST</th>
<th>ATIG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DT</td>
<td>ND</td>
<td>FC</td>
</tr>
<tr>
<td>AC</td>
<td>181</td>
<td>1</td>
<td>99.5</td>
</tr>
<tr>
<td>IR</td>
<td>144</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>PC</td>
<td>511</td>
<td>3</td>
<td>99.4</td>
</tr>
<tr>
<td>MAR</td>
<td>216</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>SR</td>
<td>78</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>ALU</td>
<td>670</td>
<td>6</td>
<td>99.1</td>
</tr>
<tr>
<td>SHU</td>
<td>246</td>
<td>2</td>
<td>99.2</td>
</tr>
<tr>
<td>Total</td>
<td>2451</td>
<td>37</td>
<td>98.5</td>
</tr>
<tr>
<td>Controlling Unit</td>
<td>561</td>
<td>29</td>
<td>95.1</td>
</tr>
<tr>
<td>Total</td>
<td>3012</td>
<td>66</td>
<td>97.9</td>
</tr>
</tbody>
</table>