Software-Based Self-Test under Memory, Time and Power Constraints

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Jun Zhou

aus Chengdu/V.R. China

Hauptberichter: **Prof. Dr. rer. nat. H.-J. Wunderlich**
Mitberichter: **Prof. Dr. rer. nat. S. Hellebrand**

Tag der mündlichen Prüfung: 12.10.2009

Institut für Technische Informatik der Universität Stuttgart

2009

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Stuttgart, den 26.11.2009 Jun Zhou

Acknowledgements

The years towards the PhD at Institut für Technische Informatik (ITI), Univerisität Stuttgart, are undoubtedly one of the most wonderful experiences and the best memories in my life. There are many people, who have contributed to this thesis in one way or another. First and foremost, I would like to thank my first advisor Professor Hans-Joachim Wunderlich for the plenty of freedom given to me and excellent guidelines to progress my work towards the correct direction and present the results in the optimal way. The special thanks should go to my second advisor Professor Sybille Hellebrand from Universität Paderbonn. It is not exaggerated at all that her constructive and invaluable suggestion shapes this thesis in the right form. I am always much impressed by her thoroughness in the work and the amiability.

I would like to thank Deutsche Forschungsgemeinschaft (DFG) which supports the topic "Leistungs- und Energiebeschränkung im Selbsttest" in the program "Grundlagen und Verfahren verlustarmer Informationsverarbeitung (VIVA)" under the project number Wu 245 / 2-1.

I have met many wonderful people during the years at ITI. Many thanks go to Melanie Elm, who helps me to refine the abstract in German. Karin Angela, Valentin Gherman, Nicoleta Pricopi, Talal Arnaout, Yuyi Tang, Erika Wegscheider, Alexandra Wiedmann, Günter Bartsch, Michael Imhof, Michael Kochte, Christian Zöllin, Tobias Bergmann, Stefan Holst, Abdul-Wahid Hakmi, I am grateful for the comradeship and the helps.

Last but not least, I would like to thank my family for the endless love, understandings and encouragement that accompany me through difficulties all along.

Abstract

The involvement of embedded systems in people's everyday lives becomes more apparent and indispensable than ever before. From the automotive industry to the household appliances, from multimedia areas to the communication sectors so on and so forth, in short, embedded systems can easily be found everywhere. This is mainly due to the rapid development of the very deep sub-micron (DSM) technology that enables to integrate millions or billions of transistors into a single chip and at the same time the continuous reduction in manufacturing cost. As a major component of an embedded system, the microprocessor plays the central role in information processing and fulfills a few specialized tasks. In contrast to general-purpose processors such as personal computers, microprocessors dedicated for embedded applications feature themselves in several aspects.

First, due to the function specialties for the target application domains, production of microprocessors is usually with small-volumes, which means the price is one of the most critical factors to determine the success of the products. For this reason, effective means to reduce the cost should be carefully taken into account throughout the complete development and manufacturing processes. Second, system resources, such as energy supply for battery-operated devices and on-chip memories, of embedded applications are often limited. Thus, optimization regarding hardware as well as software is supposed to be carefully examined so as to meet the stringent system requirements. Third, in-field periodic test is necessary to ensure sufficient reliability for microprocessors used in safety-critical scenarios. In this way, faulty modules are detected in time and the system can then take effective actions, e.g. replacement with redundant components, to prevent further consequences. Fourth, the emergence of configurable cores and application-specific instruction-set microprocessors (ASIPs) in recent years challenges the conventional ways of test. Just like much freedom left in system design through parameterization and flexible architectures, it is applausive to make tests adaptive as well in order to minimize engineering efforts and ensure the effectiveness.

In tradition, the manufacturing test of microprocessors that aims at structural faults is based on automatic test equipment (ATE). However, with increase of core complexity and operating frequencies, this external method exhibits its inefficiencies concerning the cost and test quality, in some cases even test inaccuracy. As a result, self-test has been proposed as an alternative and studied for years. The hardware-based schemes [EiLi83][BMS87] [ABF90][BGK89] add extra logic into the chip under test, which generates random patterns on the fly for the test purpose. Furthermore, as to random-pattern-resistant circuits, weighted random patterns [BGK89][Waic89][Wun90] are introduced to improve fault coverage within acceptable time.

Nevertheless, the hardware-based self-test strategy is not preferable for microprocessors. First, design modification to incorporate test-dedicated logic is undesirable, since their structures are already well optimized according to design specifications. Second, random patterns are inappropriate for microprocessors. Even though fault coverage can be improved with weighted patterns, high power consumption during structural test exists still as another severe problem, which has profound influences regarding cost, system reliability, performance, and so on.

The problems are alleviated with software-based self-test (SBST), which uses the microprocessor's own instruction set to detect structural faults. Literatures in this area [HWH96][TuAb97][ZhPa98][ChDe00][Chen03][Kran03a][Corn01][Corn03][LKC00b][Sing 03] cover various aspects regarding test generation for different fault types and processor architectures, and diagnosis. The advantages of the SBST scheme are multifold, such as low cost due to dispense of expensive ATEs, no additional test hardware required, low power consumption and at-speed testing due to its functional feature, reusability of test programs both for manufacturing and in-field periodic test purposes.

Our work also looks into the SBST area and contributes mainly to the following aspects:

1. Automatic test program generation. Our method achieves high structural fault coverage with use of deterministic test pattern, and automation of the test program generation process is realized on the basis of templates, which map the low level patterns into corresponding instruction sequences for the test purpose.

2. Test-oriented program optimization for memory, test time and power. We propose a method which reduces the memory usage and shortens the application time of test programs. Meanwhile, it is able to "optimize" power consumption. Here, by optimization, we refer not only to power reduction but also, if desired, "maximization" that is typically concerned during stress-test. The optimization does not incur the loss of fault coverage.

3. Test improvement through instruction extension. We exhibit the way to improve test quality, namely the fault coverage for the hard-to-test components, through instruction extension. The idea itself, namely the instruction-level Design-for-Test (DFT), conforms to the ASIP, and therefore the process for introducing test instructions can be greatly eased when the proposed method is combined together with the ASIP framework.

We use three kinds of microprocessors as case studies, covering the architectures from the non-pipelined, the pipelined to the configurable core. Our experimental results indicate that the schemes for test program generation achieve high structural fault coverage of the microprocessors, and our optimization approaches are effective without any impact on test quality. We demonstrate the feasibility to test user-defined logic, commonly in the System-on-a-Chip (SOC), with the SBST strategy that uses a microprocessor as the test generator and evaluator. In addition, we illustrate the process to improve test quality by instruction extension for hard-to-test components. As our study implies, the combination of the instruction-level DFT and ASIPs profits these two research areas at the same time. The tedious process of instruction-level DFT is to identify the candidates for instruction extension and alter accordingly the compiler to recognize the new instructions. This process can be speedup by the ASIP research on automatic design space exploration and tool-chain generation. On the other hand, it is also significant for ASIPs to extend the optimization goals for test quality aside from performance and power consumption.

Zusammenfassung

Eingebettete Systeme werden in zunehmendem Maße im alltäglichen, menschlichen Leben eingesetzt. Vom Automobil bis hin zu Haushalts- und Unterhaltungsgeräte, von Multimedia- bis zu Kommunikations-Anwendungen findet das eingebettete System überall Einsatz. Die tief eingebettete Halbleitertechnologie (Deep Sub-Micron DSM), welche die Integration von Millionen und Milliarden von Transistoren ermöglicht und gleichermaßen immer kostengünstiger wird, treibt diesen Trend voran. Der Kern eines eingebetteten Systems, der Mikroprozessor, spielt eine zentrale Rolle im gesamten System. Er bearbeitet den Informationsfluss und führt die wesentliche Aufgabe des Systems durch. Verglichen mit dem Prozessor eines herkömmlichen Rechnersystems wird der Mikroprozessor gezielt für eine spezifische, eingebettete Anwendung entworfen. Der Entwurf eines solchen Prozessors und Systems muss folgende Punkte berücksichtigen:

1. weil die Funktion eines eingebetteten Mikroprozessors sehr an den Anwendungsbereich angepasst ist, wird er normalerweise in kleinen Stückzahlen produziert. Der Preis ist daher einer der kritischen Faktoren, welche den Erfolg des Produktes direkt beeinflussen. Aus diesem Grund muss der gesamte Entwicklungs- und Produktions- Prozess effizient sein, um die Kosten gering zu halten.

2. Ein eingebettetes System verfügt normalerweise über sehr eingeschränkte Ressourcen wie z.B. schlechte, mobile Stromversorgung, geringer Speicherplatz und auch geringe Rechenkapazität. Das bedeutet, sowohl die Hardware als auch die Software müssen für die Verwendung möglichst weniger Ressourcen entwickelt bzw. optimiert werden.

3. In manchen sicherheitskritischen Anwendungsszenarien ist oft ein periodischer Test zur rechtzeitigen Erkennung der fehlerhaften Module notwendig. Solch ein Test kann die

Zuverlässigkeit eines Systems bestimmen und die Verlässlichkeit für eine bestimmte Zeit vorhersagen. Im System kann dann die entsprechende Maßnahme getroffen werden, um weitere, negative Konsequenzen zu vermeiden. Ein Bespiel für eine solche Maßnahme ist die Ersetzung des fehlerhaften Moduls durch redundante Logik.

4. Die Weiterentwicklung von konfigurierbaren Cores und Application-specific Instruction-set Processors (ASIPs) wird immer mehr intensiviert. Sie ist motiviert von den sehr strikten Anforderungen an eingebettete Systeme wie z. B. kleine Code-Größe, extrem niedriger Stromverbrauch, optimal kurze Befehlsrechenzeit, usw. Um solchen Anforderungen gerecht zu werden, bleiben im System Entwurf verschiedene Möglichkeiten: Relativ viele Module im System können durch Parametrisierung vom Designer speziell für das Zielsystem angepasst werden; der Designer kann eigne Logik, die speziell für eine Anwendung optimiert ist, definieren, um die Durchführung eines besonderen Stückchen Codes (s. g. hot-spot code) zu beschleunigen und Vieles mehr. Diese benötigte Systemflexibilität ist nur schwer vereinbar mit den konventionellen Entwurfsmethoden und Teststrategien.

Der traditionelle Test des Mikroprozessors, der auf strukturelle Fehler abzielt, läuft auf einer Automatischen Test Anlage (ATE, automatic test equipment). Bei erhöhter Komplexität und Systemfrequenz des Designs, zeigen sich allerdings die Nachteile dieser Methode deutlich: z.B. hohe Kosten und ein manchmal uneffizienter und unpräziser Test. Aus diesem Grund wurde der Selbst-Test als eine Alternative vorgeschlagen und seit einigen Jahren erforscht. Das auf Hardware basierende Schema [EiLi83][BMS87][ABF90][BGK89] fügt extra Logik zum Chip hinzu, damit der Chip die Tests seiner selbst mit zufälligen Testmustern durchführen kann. Weil nicht alle Schaltungen mit zufälligen Testmustern vollständig testbar sind, wurden gewichtete Testmuster zur Verbesserung der Fehlerabdeckung in vertretbarer Zeit vorgestellt [BGK89][Waic89][Wun90].

Nichtsdestotrotz ist die auf Hardware basierende Test-Strategie nicht die Bevorzugte für Mikroprozessoren. Dies liegt zum einen daran, dass die Struktur des Mikroprozessors schon nach der Design-Spezifikation optimiert werden kann. Daher fügt sich die Design-Änderung, die die nur für Test gebrauchte Logik einbaut, nicht gut in den Entwurfsprozess und -ablauf. Zum anderen verursacht die Anwendung struktureller Testmuster im Testmodus um ein Vielfaches mehr Knoten-Transitionen als im funktionalen Modus auftreten. Das System braucht daher während des Tests deutlich mehr Strom, was zu Unzuverlässigkeit und

niedriger Systemleistung führen kann. Um solche Probleme zu überwinden bzw. zu umgehen, wird auf dem Gebiet des Software basierten Selbst-Tests (SBST) geforscht. Diese Methode ermöglicht den Test anhand eines Testprogramms, welches vom Mikroprozessor mit der eigenen, zu testenden Logik ausgeführt wird. Zwar nutzt man sowohl für den SBST als auch für den funktionalen Test Softwareprogramme, dennoch unterscheiden sich die beiden Methoden ganz wesentlich. Die Programme für funktionalen Test werden generiert, um die Richtigkeit der geforderten Funktionalität zu überprüfen. Die interne Struktur der Schaltung ist transparent für die Test-Programme. Im Gegensatz dazu, da das Ziel von SBST die Entdeckung von strukturellen Fehler ist, muss die Erzeugung des Programms für SBST die Struktur der Schaltung im vollen Umfang berücksichtigen, um eine hohe Fehlerabdeckung zu gewährleisten. Die Vorteile von SBST bestehen aus Folgendem: niedrige Kosten dank der Ablösung der teueren ATEs, niedriger Strom-Verbrauch und hohe Leistung dank des funktionalen Charakters und die Wiederverwendbarkeit der Test Programme für sowohl den Produktionstest als auch den periodischen Test im Feld. In letzter Zeit wurde viel Literatur zum SBST veröffentlicht [HWH96] [TuAb97] [ZhPa98] [ChDe00] [Chen03] [Kran03a] [Corn01] [Corn03] [LKC00b] [Sing03]. Es wurden verschiedene Themen in diesem Bereich abgedeckt. Ein Beispiel ist die Test-Erzeugung für unterschiedliche Fehlertypen und Architekturen eines Prozessors. Ein anderes Beispiel ist die Ausdehnung des Gebiets SBST auf die Diagnose.

Auch die hier präsentierte Arbeit befindet sich im Themenbereich des SBST. Mit folgenden Aspekten beschäftigt sie sich besonders:

1) Automatische Testprogramm-Erzeugung mit hoher Test-Qualität und minimalem Aufwand. Mit dem Einsatz deterministischer Testmuster erzielt die Methode eine hohe strukturelle Fehler-Abdeckung. Die Automatisierung der Testprogrammerzeugung ist realisiert durch Instanzierung der Testvorlagen, die die Testmuster auf der Komponentenebene auf Prozessorbefehlssequenzen abbilden.

2) Testorientierte Programmoptimierung der Verlustleistung, des Speicherbedarfs, und der Testlaufzeit. Das Verfahren ist zugeschnitten auf den Test von Mikroprozessoren und optimiert die vorstehenden Parameter ohne einen Verlust an Fehlerabdeckung. Es wird eine Methode vorgestellt, die den Speicher-Bedarf reduziert und die Laufzeit der Test-Ausführung verkürzt. Gleichzeitig kann diese Methode genutzt werden, um den

Leistungsverbrauch zu „optimieren“, wobei Optimierung nicht nur bedeutet, den Leistungsbedarf niedrig zu halten, sondern auch den Leistungsverbrauch maximieren zu können, falls dies für den Test erwünscht ist. Ein typisches Szenario für diese Maximierung des Leistungsverbrauchs ist der „Stress-Test“.

3) Test-Optimierung durch Erweiterung des Befehlssatzes. Die Test-Qualität ist durch die Fehler-Abdeckung für die Komponenten, die schwer testbar sind, gegeben. Befehlssatzerweiterungen können diese verbessern. Die Herangehensweise an diese Erweiterung ist gut vereinbar mit der Entwicklung von ASIPs. Daher kann die Bearbeitung der Test-Befehle viel einfacher werden, wenn die vorgestellte Methode mit einem ASIP Framework kombiniert wird.

Es wird eine Praxisstudie der vorgeschlagenen Methode mit drei verschiedenen Prozessor Typen durchgeführt und die Ergebnisse werden entsprechend diskutiert. Die Typen decken die Architekturen von Non-Pipelined, Pipelined bis hin zum konfigurierbaren Core ab. Die Ergebnisse der Experimente zeigen, dass das Schema des Testprogramms generell hohe strukturelle Fehler-Abdeckung in Mikroprozessoren erzielen kann. Die Programmoptimierung verringert die Leistungsaufnahme ohne die Fehlerüberdeckung zu verringern. Außerdem haben wir nachgewiesen, dass auch benutzer-definierte Logik mit SBST getestet werden kann. Unsere Experimente zeigen ebenfalls, dass das Hinzufügen neuer Befehle die Test-Qualität für solche Komponenten, die nur schlecht testbar sind, deutlich verbessert. Des weiteren ist zu erwarten, dass durch die Fortschritte in der Forschung zur Toolchain Automatisierung und Entwurfsraum-Erkundung für ASIPs die Generierung von Software basierten Befehlsebenen-Tests weiter vereinfacht werden kann. Dies liegt zum einen daran, dass mit entsprechender Werkzeugunterstützung Befehle, die die Testqualität verbessern leichter identifiziert werden können und zum anderen eine Auswertung des Tests leichter wird. Auf der anderen Seite kann der Entwurf zum Test (DfT) auf Befehlsebene selbst zur Erweiterung des Gebiets der ASIP Forschung beitragen, da bisher nur Verlustleistungs- und Performanzaspekte hier berücksichtigt werden. Aus diesen Gründen ist sinnvoll, diese beiden Forschungsgebiete in Zukunft nicht mehr getrennt voneinander zu betrachten.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AHB	Advanced high-performance bus, a bus protocol in AMBA specification
ALU	Arithmetic and Logic Units
AMBA	Advanced Microprocessor Bus Architecture
APB	Advanced peripheral bus, a bus protocol in AMBA specification
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BCU	BIST Control Unit
BDD	Binary Decision Diagrams
BIST	Built-In Self-Test
CAD	Computer-Aided Design
CUT	Circuit Under Test
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CWP	Current Window Pointer
DFT	Design For Test
DG	Dependency Graph
DSM	Deep Sub-Micron
DSP	Digital Signal Processing
DSU	Debug Support Unit
FPGA	Field Programmable Gate Array
FPU	Floating Point Units
FSM	Finite Status Machine
HBST	Hardware-Based Self-Test
IC	Integrated Circuit

IP	Intellectual Property
ISA	Instruction Set Architecture
LBIST	Logic Built-In Self-Test
LFSR	Linear Feedback Shift Register
MATS	Modified Algorithmic Test Sequence
MBIST	Memory Built-In Self-Test
MIPS	Originally acronym for Microprocessor without Interlocked Pipeline Stages, a RISC ISA
MMU	Memory Management Unit
MUT	Module Under Test
PCI	Peripheral Component Interconnect
PSR	Processor Status Register
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
RPR	Random Pattern Resistance
RTL	Register Transfer Level
SBST	Software-Based Self-Test
SOC	System On a Chip
SPARC	Scalable processor architecture, a RISC microprocessor instruction set architecture originally design by Sun Microsystems
TPG	Test Pattern Generator
TRE	Test Response Evaluator
UART	Universal Asynchronous Receiver Transmitter
VCC	Virtual Constraint Circuit
VCD	Value Change Dump
VHDL	VHSIC(Very High Speed Integrated Circuits) Hardware Description Language
VLIW	Very Long Instruction Word
WSA	Weighted Switching Activity

Chapter 1 Introduction

1.1 Motivation and Goal of the Thesis

Nowadays, microelectronic devices are ubiquitously used in human lives, e.g. aeronautic and astronautic technologies, automotive industries, health and medical communities, telecommunication sectors and entertainment, either for personal or business usage purposes. Therefore, system reliability is a concern both for commonplace utilizations and safety-critical applications, as malfunction may lead to use inconveniences or even catastrophic consequences. Test is an indispensable step towards development and fabrication of reliable systems, consisting of a series of systematic processes to find out faults in the products. Under a concerned fault model, test patterns (termed also as test stimuli) are designed in a way to result in distinguishable outputs between faulty and fault-free circuits. During application of the test set, responses of a circuit under test (CUT) are evaluated against the expected values from good-circuit simulation. The circuit is found to be faulty if it fails at least one of the tests. In general, test is not only significant to ensure high product quality, but also likely to help improve the overall manufacturing processes.

The use of deep submicron (DSM) process technologies in the semiconductor industry presents enormous challenges in design verification and validation for complex digital systems, and, at the same time, its influences on test have also received wide recognitions. We enumerate below a few stringent difficulties that system designers and testers are both confronting.

1. Complexity. The continuous shrinking geometries make it now possible to integrate millions, or even billions, of transistors in a single chip. Besides, the advent of SOCs makes module-level or even chip-level integration a reality, that is, entire systems can be built on the basis of pre-designed and pre-verified components or chips. Such development paradigm allows short time-to-market and reduced cost due to reuse of existing intellectual properties (IPs) [Wun98]. Massive integration scales, various integration forms, and emergence of heterogeneous systems, etc. all increase complexity, making verification and validation of these systems a challenging task.

2. Testability. First, from the standpoint of the system design, implementation details should be hidden away as many as possible. However, such information is essential to derive effective test set, particularly for structural test. These contradictory orientations account for current complex systems with rather low testability. Second, accompanying with the increase of complexity, test data grow significantly, and cause memory and test time critical concerns. Techniques to optimize test data while maintaining test quality are therefore of great significance for testers. Third, with process technologies continuously moving down to the nano-meter range, the need for at-speed testing to detect timing-related faults such as delay faults becomes apparent, as these kinds of faults manifest themselves only when the design is running at its normal clock frequency. Finally, early as in 2006, the International Technology Roadmap for Semiconductors (ITRS) indicated that over-testing, where a chip fails merely due to non-functional conditions related with test, is one of sources for potential yield losses [ITRS06]. Although a wide range of fault types and tests may incur over-testing, avoidance in scan-based delay test is the focus of attentions.

3. Energy and power consumption. The rise in operating frequencies and integration scales makes power consumption a primary factor, just as the other two namely the silicon area and performance, during circuit development. High power consumption is related to disadvantageous influences like overheating, inferior performance and reduced reliability. When electronic devices operate on batteries, e.g. mobile computing and multimedia applications, with limited energy resources, energy consumption becomes also a crucial issue concerning battery life time. An even more severe situation is confronted during structural test, as power dissipation is observed several times higher than in the normal mode. As many as possible internal nodes make transitions due to use of patterns with

rather low correlations, while power-efficient systems are usually designed with cautions to limit the number of switching activities at a time. Thus, it is important to optimize energy and power consumption especially in the context of test.

In practice, test of integrated circuits (IC) can be performed externally, which often relies on automatic test equipment (ATE) to apply pre-designed test patterns and compare against the expected responses in storage. The major problems for the ATE-based external test are relevant with test economy and quality. As the ITRS pointed out, the price for an ATE is mainly determined by the base cost, varying from $30K to $550k, and the incremental cost related with the number of channels and pins [ITRS03]. Even though the base cost is expected to decrease over time in a slight manner, the other part of cost, growing dramatically upon the scale of the CUT, may finally make the overall test cost unaffordable for complex devices. This is even worsen for testing a circuit with integration of heterogonous structures, e.g. analog or radio frequency (RF) blocks, where large volume of test data and long test time are necessary. Besides of the economic issue, test quality is another critical difficulty for ATE-based test solutions. With the narrowed gap in frequencies between ATEs and high-performance products under test, detection of speed-related faults is prone to errors, likely leading to yield losses. For these reasons, in contrast to external test, self-test methodologies are studied to alleviate the mentioned problems.

Self-test equips a CUT with the ability to perform test on its own through addition of extra logic. Being an effective DFT technique, logic built-in self-test (BIST) is a widespread scheme with various advantages like reduced test cost and increase fault coverage. In general, a self-testable circuit is composed of a test pattern generator (TPG), a test response evaluator (TRE) and a BIST control unit (BCU) [Wun98]. There are two implementation schemes for logic BIST, test-per-scan or test-per-clock. As to the former, all sequential elements are organized into one or several scan chains, while for the latter special registers are able to work as TPGs and TREs at specific clock cycles. Among numerous publications, [EiLi83][BMS87][ABF90] are commonly accepted as standard architectures, which apply pseudo-random patterns generated by a linear feedback shift register (LFSR) to scan chains and have merits of low implementation efforts and minimal area as well as performance overheads. However, not all circuits are random pattern testable, for example microprocessors due to random pattern resistance (RPR). For this kind of circuits, logic BIST often fails to

reach sufficient fault coverage within acceptable test time. Weighted random patterns are hence proposed to circumvent the drawback [BGK89][Waic89][Wun90].

Despite successful application of the hardware-based self-test (HBST) scheme to random circuitries, its use for microprocessor test is not preferable. First, microprocessor structures are already well optimized according to specifications and therefore sensitive to any design modifications. Thus changes in structure to incorporate dedicated-test logic are undesirable for microprocessors. Second, poor fault coverage is reached with (pseudo) random patterns for microprocessor test. Even though test quality can be improved with weighted random patterns, power consumption for the structural test in non-system clock cycles is expected to be much higher and risking the system with possible malfunctions, degraded reliability and potential circuit damages.

As an alternative to the HBST, the recent emerging technique of software-based self-test (SBST) promises an attractive and non-intrusive testing solution for programmable cores. Unlike the HBST, SBST implements tests, targeting at manufacturing faults, as software programs. In this way, it is no more required for extra test hardware. Moreover, at-speed testing is supported in nature and due to its functional mode for test application, this method results in low power consumption during test.

In the past few years, SBST becomes a hot topic and a large number of methods and techniques, e.g. [ShAb98][ChDe01][ChDe02][CRRD03][Kran03][Kran05][Psar06][Corn01] [CRS03][ZhPa98][BaPa99][Sing05], have been proposed, addressing aspects with respect to fault detection under considerations of various fault models, automatic test program generation, different architectures for processor or digital signal processing (DSP) cores, diagnosis and application in SOC environments.

Apart from the research in microprocessor testing, there are two more topics that catch up our attentions. The one is on application-specific instruction set processors (ASIPs), opening up a new horizon for processor designs. The background for the newly emerging technique is to optimize embedded application of video/audio processing and encrypt/decrypt algorithms under various constraints, such as area, speed, power and cost. The tradition improvement way for improvement is either to map those critical segments into dedicated hardware like application-specific integrated circuits (ASICs) or to use of general-purpose processors. As

indicated by the relevant studies in this area, ASIPs exhibit a better compromise between flexibility and efficiency in terms of speed and power savings when compared to the hardware solution of ASICs and general-purpose processors.

Design activities of ASIPs involve analysis of the target application for constraint-critical sections e.g. execution hot spots, configuration of existing blocks, addition of application-specific structures through instruction extension, and generation of relevant tool chains to enable user-defined logic fully aware in the design flow. The major research efforts [JaVe00] [Henk03][LDM05] are centralized on performance improvement through instruction extension, while test-related issues, such as automatic test generation and so on, are unfortunately overlooked. If not properly dealt with, it may offset winnings gained from the automated design flow and retard time-to-market. Thus it is apparently significant to initiate a research topic on automatic test generation for this new design paradigm.

With the increasing importance of software played in modern systems, the study of power consumed in software becomes a hot topic. The concentrated aspects are 1) power consumption modeling at the instruction level, e.g. [Tiwa96a][ChGa99][Sami00][Lee01], and 2) software optimization for low power based on compiler techniques, such as power-efficient code generation, register name assignment and instruction scheduling [STD94][Tiwa96b] [Tomi98][Shiu01][ChCh01][PeOr03][Lee03][Pari04].

Software-based self-test operates in the functional mode of processors so that the requirement on peak power consumption, i.e. the maximal power consumed at a clock cycle, is still satisfied. Nevertheless, average power consumption during test can surpass the corresponding limitation, causing problems related with overheating, degraded performances and system reliability. Furthermore, as SBST targets embedded applications where systems may run within a very tight energy budgets, e.g. battery-operated portable devices, it is necessary as well to minimize energy consumption during test. At this point, all existing software optimization techniques are not adequate for the SBST, since their precondition is to ensure the identical program semantics during optimization. In contrast to this, the major focus of SBSTs is the test quality, or more precisely the fault coverage level. For this reason, an optimization strategy is acceptable as long as no sacrifice of fault coverage will be seen, even if the program semantics may thereby be changed. Such different vision brings about degrees of freedom during software optimization.

In the scope of this thesis, we identify test time, memory footprint and power consumption are three factors pertinent to SBST practices in embedded systems and propose accordingly a novel methodology for automatic test program generation with advantages of high fault coverage, reduced test length and power consumption. The optimization algorithm for low power is suitable for SBST which does not incur any loss of test quality. For the first time, we consider application of SBST to extensible core designs and tackle issues concerning automated test generation together with test improvement through instruction extensions.

We summarize the major contributions of our work as follows:

1. Automatic generation of test programs based on template instantiation. Templates are atomic instructions with parameterized fields to realize module-level tests. Through step-wise specification of template variables depending on deterministic patterns for the target modules, we generate the test programs at the process level. According to our experiments, high module-level as well as process fault coverage is achieved in this way;

2. Test-oriented optimization for low power, memory and time. The advantages of the proposed techniques reduces average power consumption through test-appropriated instruction rescheduling, shortens requirements for memory and test time through pattern compaction. Another appealing side-effect with test data reduction is related with minimized energy consumption during test.

3. Test generation under the ASIP framework and improvement of the test quality through instruction extension. For the first time, we combine the two research areas, namely ASIPs and SBST, and exhibit the potential benefits for the both sides. Meanwhile, test within SOC environment for user-defined logic is also tackled in our work.

1.2 Outline

The rest of this dissertation is organized into the following chapters:

Chapter 2 starts with the principle of software based self test and presents the road-map to extend this method to SOC environment where programmable cores are in charge of test

generation, application and evaluation for other non-programmable chips. Next, state-of-the-art techniques in microprocessor testing using software-based approaches are surveyed, from the early use for system verification to the recent structural testing.

Chapter 3 describes power evaluation and optimization techniques. Preliminaries with respect to modeling various sources of power dissipation are firstly provided. Afterwards, in line with a similar structure, techniques on power estimation and optimization are classified from both hardware and software perspectives and discussed respectively.

The SBST method that considers memory, time and power constraints is proposed in Chapter 4. We first give an overview of the proposed approach, and then detail the power-efficient algorithm for test program generation, which basically contains steps of test set compaction, construction of atomic instruction sequences (defined as templates) for pattern application, and template instantiation. The novel method adequate for the SBST is introduced, which minimize software power consumption without any fault coverage loss.

The application of the SBST to the research domain of ASIPs is outlined in Chapter 5. To generate tests for the base core which presents in all ASIP instances, the methods introduced in Chapter 4 are adequate, whereas for the other components such as configurable, optional or user-defined logic, test generation can be automated through the parameterization technique. In addition, based on the work [LaCh01], we extend the conventional ASIP framework so that the SBST quality can be improved through addition of new test instructions for the formerly hard-to-test modules.

Chapter 6 presents in details our experiments on three kinds of processors: a non-pipelined RISC core developed for academic usage, a MIPS-compatible core called Plasma [OPEN] with pipelines, and the configurable and user-extensible core of Leon [Leon]. The focuses of interests for our case studies are to 1) confirm the effectiveness of the schemes proposed in Chapter 4 on non-pipelined as well as pipelined cores; 2) discuss the way to test user-defined logic in a configurable core and 3) improve the test quality through instruction extension.

We summarize in Chapter 7 the whole work and point out promising topics for the future researches.

Chapter 2 State-of-the-art in Software-based Self-test

2.1 Basic Concepts of SBST

A processor core at the micro-architectural level composes modules in the data-path, the control logic, caches and memories, and bus systems. The blocks in the data-path are usually the arithmetic and logic units (ALUs), multiplexers, register files, multipliers, floating point units (FPUs), and perform computations with provided operating data. Components such as pipelines, hazard detection, data forwarding, and branch prediction form the control logic and control behaviors of the entire system. Instructions and data can be stored in memories and caches either jointly or separately. The former organization corresponds to the classical Von Neumann architecture and the latter, namely separation of instruction and data storages, complies with the Harvard architecture which is often used in digital processing processors (DSPs) and small micro-controllers due to its accessing parallelism. Details regarding hardware implementation are properly hidden away from programmers with the use of instruction set architectures (ISAs). On one side, as instructions are processed inside the core, in nature they are ideal candidates to test the concerned modules. On the other side, due to the high abstraction level of ISAs, it is of great help with the knowledge of the target processor structure to derive efficient tests based on instructions, in particular when manufacturing test is considered.

Software-based self-test (SBST) utilizes processor's programmability and on-chip resources to eliminate the need of specialized hardware for test pattern generation, application and evaluation, as commonly required in hardware-based schemes, e.g. logic built-in self-test (LBIST). The SBST targets faults in the central processing unit (CPU), while faults in memories or caches can be effectively tested with the other techniques, e.g. memory built-in

self-test (MBIST). The principle of the SBST scheme is shown in Figure 2-1, which mainly concerns two phases, namely test generation and test application.

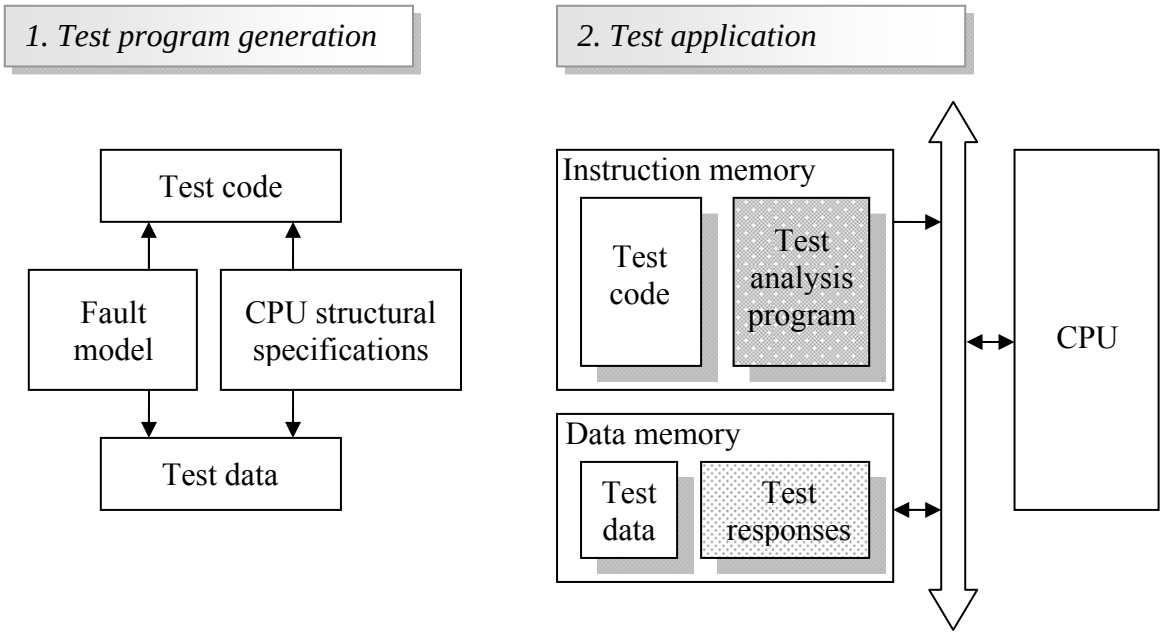


Figure 2-1: Principles of the SBST Scheme

The primary goal in the first phase is to determine the operations (test code) and operands (test data), called together as test programs, to detect the target faults as many as possible. In order to derive effective tests, the structural specification of the CPU, e.g. the architectural or the register transfer level (RTL) description, is also taken into account during program generation. Test program generation is only performed once and the resultant test routines can be reused throughout the whole life cycles of the CPU.

Before test application, test programs have to be equipped onto the chip with appropriate means. As for the manufacturing test, an external tester can be utilized to upload the generated programs. However, since test is in fact applied later on by the processor itself, issues regarding frequencies and bandwidths of the external tester are no more critical. Thus a low-end automatic test equipment (ATE) can sufficiently fulfill the task, which as a result reduces greatly the test cost. Or, as an alternative, test programs can permanently be written into the read-only-memory (ROM), and available all the time when the processor is deployed in its target system. This facet is quite significant, since it makes the SBST an ideal option for in-field periodic test used to detect intermittent faults common for complex systems [GPZ04]. The right part of Figure 2-1 outlines the possible layout of test programs in the memory

system for processors with the Harvard architecture. Then during test application, the testee, namely the CPU, fetches the code and data, and perform corresponding operations. Through instruction execution, the pre-defined test patterns are transported to the concerned modules and relevant test responses are stored back to the memory. When test is done, the responses can either be unloaded and evaluated off-line with the help of low-cost ATEs, or if a test analysis program is available, the evaluation process can also be performed on-line.

In comparison with the conventional ATE-based tests and the hardware-based self-test scheme, the SBST exhibits its multiple benefits as follows:

- *Low Design-for-Test (DFT) efforts.* Realized in processor instructions and reusing on-chip resources, the SBST scheme dispenses with design modification to incorporate extra test logic as usually required for hardware-based test methods, therefore avoiding area overheads and performance degradation to the entire system.

- *Low test cost*. Test is fully performed on its own of the processor, thus eliminating the requirement for expensive ATEs and reducing significantly the test cost. For this reason, the SBST is particularly useful for low cost applications and low volume production flows [GPZ04].

- *Low power consumption*. Unlike the high power consumption commonly observed during hardware-based self-test due to its non-functional attribute, the SBST is in nature a low power test solution, and alleviates some severe consequences related with high power dissipation, such as performance and reliability degradation.

- *At-speed testing*. With the continuous shrinking of integrated circuit (IC) technologies, at-speed testing, which runs test at the actual speed of the CUT, becomes prevalent [Saxe03]. At this point, the SBST is one of the candidates for at-speed test due to its functional mode for test application.

- *Reusability and flexibility*. Once test programs are synthesized, they can be utilized throughout the whole system life cycles, e.g. for manufacturing test or in-field test. Furthermore, because of programmability of processors, it is easy to extend the SBST for another fault model or for diagnosis purposes.

At the beginning, the SBST was primarily used in the functional test to validate the specified functionalities, for example whether multiplications or other computations correctly work. Some later studies indicated the usage of these test programs, which are centralized on design validation, for the manufacturing test as well. However, since no structural information is taken into account during program generation, the resultant fault coverage is apparently far beyond satisfaction. The recent SBST work focuses on a broad scope, covering topics related with 1) automatic test program generation under considerations of processor architectures and fault models, and 2) test improvements. In the following sections, we survey the major techniques in these two directions.

2.2 SBST for Functional Test

Microprocessors are practically tested in industry based on heuristic techniques, namely to test each valid instruction with a few "typical" operands so as to detect malfunctions in the system mode. However, such approach is infeasible for processors with complex structures and instruction sets. In the late 1970's and early 1980's, there were lots of studies, e.g. [Cric79][ThAb80][Bell82][BrAb84][ShSu88][LiHo88][TAS89][GoVe92][KaRa00], aiming at detection of functional faults with instructions as few as possible. Depending on whether functional fault models are referred for test generation, these literatures can be classified into two categories accordingly [ShAb98].

Crichton proposed a test strategy for microprocessors which generated respectively test programs for the two split internal logic of data blocks and control logic, and demonstrated its application on a SAB 8080 A microprocessor [Cric79]. Viewed as an effective means to ease definition of test vectors and resulted in reduction of test time, logic separation of a microprocessor core introduced in the work is not always practical, especially with the increased complexity. Bellon et al. [Bell82] outlined a test environment, which automatically generated modular test programs on the basis of the microprocessor behavioral description including the instruction set, memory elements and operation sets, and the interfacing signals. The test environment enumerated all valid instruction instances without any reference to the processor implementation, which was likely to come up with resultant programs that required long test time. Instead of utilizing the complete instruction set for functional testing, Talkhan et al. [TAS89] proposed a mapping method to identify a subset of instructions that covered all

the control states and exercised all the functional modules, and then use these instructions to construct test programs. It may be straightforward to build up a mapping matrix for a processor with a limited number of instructions and functional components, as shown with the practice in this work on a TMS32010 processor. Again, it is difficult to scale this approach to deal with modern designs of more realistic sizes.

Another mainstream in functional testing started with conceptual division and abstraction of microprocessor operations, and built a model, called functional fault model, to describe possible faulty behaviors for each operation type. The representative work in this direction was introduced by Abraham, Thatte and Brahme in the early 80's [ThAb80] [BrAb84], which developed a comprehensive model for the instruction execution process. Due to its generality and independence of processor implementation details, it was widely cited since then. Shen et al. [ShSu88] emphasized the fault model related to the control logic. With the guide of the built control fault model, this work derived tests for the kernel instructions related to register writes and reads, and then used them in testing of remaining instructions. Lin et al. adopted the Turing machine model to improve the fault model with respect to functions for register decoding, data registers and I/O interfaces [LiHo88]. Application to the Intel 8086 processor showed a better fault coverage with reduced algorithm complexity. Extending the model with modern functionality such as cache and memory management unit (MMU), van de Goor et at. [GoVe92] exhibited functional test generation for the Intel 8085 processor with the major focuses on components of internal registers, instruction and data caches as well as memory units. The work of Kannah et al. [KaRa00] looked into reduction of test sizes, since the size of test sets is an important factor which determines both the storage requirement and test application time. Combining the idea of fault-grading, the modified test generation algorithms in line with the functional fault model resulted in 21% and 31% reduction of test sets for Intel 8086 and Motorola 68000 processors respectively.

The functional testing is mainly used to validate correctness of complex microprocessor designs. As it is independent of implementation details and the model considered is quite uniform, test generation can be automated relatively with ease. However, the relation between design validation and manufacturing test was pointed out in [AFK88] and [MAH96], which showed that manufacturing test pattern generation can be used for design verification while design verification is helpful in finding better manufacturing tests. Nevertheless, in general, since little on the specific structure of the processor under test is considered during the

generation process, the resultant functional test programs usually turn out to reach quite limited levels for manufacturing test. This is just the concern of the recent studies in SBST, which improve the fault coverage of test programs by taking into account the processor structure from the start of program generation. In the next sections, we will present some basic concepts concerning structural fault modeling, and then discuss the state-of-the-art in the area of the so-called "structural" SBST.

2.3 SBST for Structural Test

Numerous works in recent years has broadened visions of the SBST approaches. We categorize these studies into the following directions.

2.3.1 Test Program Generation under Consideration of Fault Models

Physical defects of integrated circuits (IC) due to fabrication failures are modelled at the logic and behavioural levels as "faults". Such abstraction results in multiple benefits: first, the efforts for defect analysis are alleviated to a large extend, since several defects may be represented as a unique fault due to the same failure behaviour. Second, the logic fault model remains effective irrespective of any technology changes. Third, tests derived based on the fault model are useful to detect physical faults [ABF90][MoZo00]. It is an extensive research aspect in SBST, which concentrates on generation of effective test programs for various fault types.

2.3.1.1 Program Generation for Stuck-At Faults

Proposed by Eldred in 1959 [Eldr59], the stuck-at fault model is one of the most classical ones, which represents a line in a circuit with the logic value fixed to '0' or '1'. Figure 2-2 illustrates the two possible physical defects, namely a short to the power shown on the left and a short to the ground on the right, causing a permanent logic value at line A. With the development of complementary metal oxide semiconductor (CMOS) technologies there are numbers of defects types which can not be expressed with the stuck-at fault model. Nevertheless, it is still the most common model considered in structural test due to its simplicity, abstraction and, more importantly, its correlation with detection of other fault types. For example, Patel claimed that, if each stuck-at fault gets detected 5 times, then the

probability for detection of bridge faults due to falsely connected wires can be as high as 99.9% [Pate98].

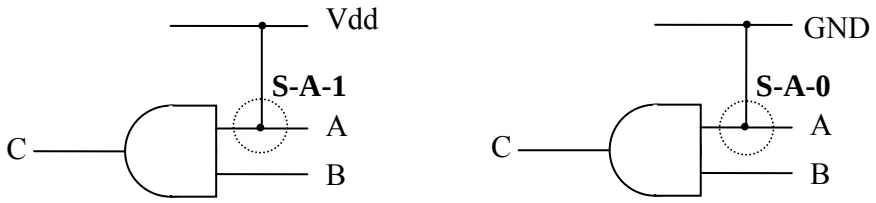


Figure 2-2: Stuck-at Faults

The early literature aiming at stuck-at faults in microprocessor designs was introduced by Karpovsky et al. in 1984 [KaMe84]. Test programs were generated to implement particular algorithms for the aimed components, for an instance, the so -called “linear checks method” [Karp81] for the ALU. A SBST technique based on emulated LFSR schemes was proposed and analyzed by Hellebrand et al. [HWH96]. Hatayama et al. generated test programs for embedded functional blocks (the ALU and the multiplier unit) with translation of temporal-constrained test patterns into corresponding instructions [Hata97]. The similar idea was extended by Tupuri [TuAb97], which took into account not only the temporal but also the spatial constraints. The obstacle for the constrained test program generation is how to efficiently extract the correct constraints with acceptable engineering efforts. Zhao outlined a method to generate self-test programs for testing DSP cores [ZhPa98] with use of metrics regarding testability and structure coverage (the percentage of RTL components exercised by instructions). With the specification of the target instruction set, the process to generate test programs was automated through instruction enumeration and randomization [ShAb98]. However, insufficient fault coverage and long test time remain as the major drawbacks of the approach. Dedicated on-chip hardware was explored to randomize generation of test instructions by Batcher [BaPa99].

2.3.1.2 Program Generation for Delay Faults

Delay faults manifest themselves as violations of the system timing specification rather than logic failures. Usually, detection of delay faults requires application of two-pattern tests at the system clock, one pattern for setting up the initial status of the CUT and the other for fault sensitization and propagation till to the circuit outputs. Delay faults are commonly classified

into three groups, namely the transition, the gate delay and the path delay fault model [MaAg97].

Transition faults include the nodes either slow-to-rise or slow-to-fall. Recent studies [KrCh98][BuAg00] indicate a tight relation between tests for transition faults and stuck-at faults. That is, test for transition faults is easily generated with slight modifications of the stuck-at test set, and at the meanwhile, high stuck-at fault coverage usually implies high transition fault coverage as well. *Gate delay faults* quantitate the timing discrepancy of each gate with assignment of a numerical value [CIR87]. The number of transition faults or gate delay faults grows linearly with the complexity of the CUT [Majh96]. To the contrary, *path delay faults* are the cumulative effect of gate delays from the input to the output, and can be exponential to the size of a CUT.

The relevant SBST schemes are mainly proposed for detection of path delay faults. Lai pointed out structurally testable paths of a microprocessor may no longer be testable with instructions, and delay faults at those functionally untestable paths do not need to be tested as long as they do not cause the path delay to exceed twice the clock period. Based on path identification, a scheme to generate test programs targeting only at delay faults at functionally testable paths is proposed [LKC00a][LKC00b]. Singh et al. [Sing03][Sing05a] presented an approach to identify the functionally testable paths, which uses the instruction execution graph (IE-Graph) and the finite state machine to extract constraints imposed on the data-path and the control unit. Bernardi et al. introduced a way to automate the test program generation for path delay faults. The evolutionary algorithm is explored to refine the quality of the resultant test program, while an evaluation framework is built up to acquire the metrics used by the evolutionary algorithm [Bern07].

2.3.1.3 Program Generation for Crosstalk Faults

The deep submicron (DSM) technology makes circuits susceptible to interconnect noises known as crosstalk faults. Several sources from manufacturing variations such as capacitive crosstalk, power supply noise and leakage noise account for the system perturbation [Zach03]. In general, crosstalk faults can be viewed as a special case of delay faults. Based on analysis of four error cases, Cuviello et al. [Cuvi99] modeled crosstalk faults as 1) positive glitch, 2) negative glitch, 3) rising delay and 4) falling delay. The line that the error effect takes place is

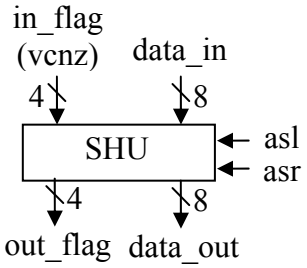
termed as “victim”, while other interconnects that result in the failure are “aggressors”. Erroneous glitches are observed if the victim remains a stable logic value while the aggressors make transitions. Rising delay or falling delay is resulted from the opposite signal switches between the victim and aggressors.

Lai discussed a method to generate test programs for this kind of faults at on-chip buses in a SOC environment [LHC01]. Only the tests that meet the constraints imposed by instructions and the specific bus type can be delivered by the embedded processor in the system mode. Moreover, a cluster and compaction algorithm that makes use of the regularity of the tests to be delivered was outlined so as to generate programs with short test length. Chen detailed a method to map test vectors for crosstalk faults at interconnects into system-level instructions, and a validation framework for evaluation of the proposed method [CBD01].

2.3.2 Test Program Generation under Consideration of Processor Architectures

2.3.2.1 Non-pipelined Simple Cores

Early work demonstrated the effectiveness of the SBST scheme in structural fault detection with non-pipelined simple processor cores. Chen et al. detailed an approach to implement software testers for delivery of structural tests for microprocessor components [ChDe00][ChDe01]. In order to generate instruction deliverable test patterns, the constraints imposed by instructions for the target component are systematically extracted beforehand. Taking the component (SHU) from the PARWAN processor as an example, Chen illustrated the process for constraint extraction based on the instruction set.



a) Block diagram

1) z = !(data_in[7] + …+data_in[0]), and
2) n = data_in[7], and
3) !asl*!asr*(!v*!c + !(v⊕c⊕s)) + (asl*!asr +!asr*asr)*!v*!c = 1, and
4) data_in[7] * !(data_in[6] + … + data_in[0]) = 0

b) Instruction-imposed spatial constraints

Figure 2-3: Constraint extraction for SHU in the PARWAN processor (from [ChDe01])

The block diagram of the SHU is outlined in Figure 2-3 a), and the dependencies among its inputs (in_flag, data_in, asl and asr) are shown in Figure 2-3 b). According to the PARWAN instruction set, the input z indicates whether the value at data_in is zero, and such constraint is modelled with expression 1) in Figure 2-3 b). Likewise, expression 2) models the consistency between n and data_in[7]. Expression 3) presents the following constraints: first, asl and asr cannot both be 1; for arithmetic instructions the relation of $v = c \oplus s$ is valid, whereas for other instructions, v and c equal zero. Finally, data_in can never be 10000000, which is presented in expression 4). The extracted constraints are then given to an ATPG to generate the deterministic test set for the component, and the instructions used to deliver the generated test patterns are then determined. Chen proposed using random patterns to test components with independent inputs, such as the arithmetic and logic unit (ALU).

Aside from the discussion on test generation for components using instructions, another interesting topic regarding observation of status outputs after test application is also presented in [ChDe01]. Although there are no instructions to store directly the status outputs of a component to memory, the image of them can be created instead in the memory with conditional instructions.

Kranitis extended this work with notable reduction of the program size, test data size and execution time by taking into account the RTL structure of the target processor [Kran02a][Kran02b]. Another contribution of Kranitis' work lies in that a systematic and deterministic way is proposed to choose the target components as well as instructions and operands.

Corno adopted the genetic algorithm to automate the process for test program generation [Corn01][Corn02]. A set of macros, namely atomic pieces of instructions for pattern delivery, have to be built in advance. Then the tests are generated with iterative selection of the best macros, evaluated against the achievable fault coverage, from the pre-built library and addition of them into the program. The mechanisms to check the harmful conditions, e.g. infinite loops, are proposed and applied during program generation.

2.3.2.2 Pipelined Simple Cores

A case study on application of SBST for pipelined processors was presented by Hatzimihail et al. [Hatz05]. The SBST methodology presented in [Kran05] for non-pipelined processor was firstly applied to a pipelined processor called miniMIPS CPU [Mini]. The results implied that despite the high fault coverage for the functional components, the coverage of the entire processor remains insufficient. The reason is, the pipeline structure including pipeline registers, hazard detection and forwarding hardware, cannot be efficiently tested using the approaches in 2.3.2.1, and on the other hand, their contributions to the areas as well as faults are not trivial.

Based on analysis of the major difficulties with SBST for the pipelined processor, corresponding solutions were proposed to improve fault coverage in address-related and data-related pipeline components, harzard detection and forwarding unit, and the system co-processor respectively. According to these solutions, to excite faults and propagate them to the primary outputs of the processor, it is necessary to use instructions and operands with special cares. For example, the code shown in Figure 2-4 a) creates an unresolved hazard and forces the pipeline to stall, which in turn propagate faults through all the pipeline stages. Figure 2-4 b) exemplifies the instructions to activate the forwarding unit, which bypasses the value in "$s4" from the MEM stage to the EXE stage. Similar conditions can purposely be created to excite other forwarding paths, e.g. MEM to ID, or EXE to ID. A notable improvement regarding the overall fault coverage for the miniMIPS CPU is achieved with the proposed strategies.

```
lw    $t1,   offset(base)
and   $t2,   $t1,          (any other reg)
sw    $t2,   offset(base)
```

a) Code to enable faults propagated through all pipeline stages

```
lw    $s4,   offset($t0)
add   $s1,   $s4,          $s4
sw    $s4,   offset($t0)
```

b) Code to activate the forwarding path from MEM to EXE

Figure 2-4: Exemplary code to detect faults in pipeline-related logic (from [Hatz05])

They later introduced a method [Psar06] which incorporates into basic test programs with code sections dedicated to exercise specific functionalities of relevant units, and exhibited its application in periodic on-line testing of intermittent faults of pipelined processors [Kran06].

2.3.2.3 Superscalar Cores

A preliminary study on the test issues regarding SBST for the superscalar cores are presented by Singh et al. in [Sing05b] and [Sing05c]. A superscalar processor is able to execute several instructions simultaneously with incorporated multiple functional units, and moreover, in a out of order manner, that is to say, the order of instruction execution are no longer determined by the program but the processor scheduler on the fly. All of these architectural features challenge the SBST, as they can not guarantee that the desired test patterns are in deed applied to the target functional components and likewise the determined orders are obeyed.

The basic idea proposed by Singh is to concurrently test the multiple functional components in a way that the processor scheduler will schedule instructions in the same order as that required by predefined test sets. An example to explain this idea is given in [Sing05c]: for a superscalar processor with 4 instruction-wide fetch and 2 ALUs, testing of each ALU requires the test pattern delivered with instructions of ADD and SUB in the following sequence:

```
ADD   R1,   R2,   R3
SUB   R5,   R6,   R7
```

However, applying these two instructions alone do not deliver the desired test pattern, as the scheduler will allocate them to the two ALUs and thus faults are not activated. To overcome this problem, Singh suggested using the code below:

```
ADD   R1,   R2,   R3
ADD   R21,  R2,   R3
SUB   R5,   R6,   R7
SUB   R25,  R6,   R7
```

In this way, the desired pattern is applied to the ALUs regardless of the order determined by the processor scheduler. Although the work itself is preliminary, the topic regarding SBST for superscalar processors and the idea are definitely worthwhile.

2.3.2.4 System-on-a-Chip (SOC)

In its latest report [ITRS06], ITRS predicts a steady increase of reused designs and exponential growth in the entire logic size of SOCs for the next 10 years. This gives us a hint that the current obstacles in SOC testing will yet be in existence, unless better ways are found to counteract. Among them, test cost is of first importance. Heterogeneity and complexity due to integration of digital, analog, or radio-frequency Intellectual Property (IP) blocks, and reduced accessibility to internal devices, etc., all of these features apparently make high-end ATE-based test an extremely expensive and sometimes even insufficient solution.

In structure, a typical SOC usually contains at least one programmable core like a microprocessor, a few other logic modules that may be non-programmable but optimized to achieve high performance for specific applications, memory blocks and the bus systems. For this case, only slight extensions are necessary to make SBST schemes applicable for the complex SOC systems, where microprocessors are responsible to deliver pre-designed test patterns to the interconnected blocks and analyze corresponding responses. Some research efforts [ZDR00][TAS00][Krst02a][Krst02b][TBG05]are already spent in this aspects.

The critical challenges in SOC testing and potential solutions were analysed by Zorian et al. [ZDR00]. This work explicitly pointed out the use of SBST in SOC environment, although applications were still limited to embedded processor cores. In [TAS00], Tupuri et al. extended their early scheme [TuAb97] to tackle issues of functional test generation for peripheral blocks in a SOC system. Targeting one module at a time, this approach hierarchically extracted its spatial constraints as virtual logic and later adopted a commercial sequential automatic test pattern generation (ATPG) tool to generate test patterns for this transformed block. And final functional test programs were obtained through translating the constrained ATPG patterns at module-level into system-level instructions. However the coverage level is directly linked to capacity of sequential ATPG, which may be insufficient particularly for complex designs with deep sequential depth and large numbers of internal states. A novel approach was introduced by Krstic et al [Krst02a][Krst02b], which

implemented functional self-testing for the entire SOC design. Under the assumption that programmable cores (microprocessors or DSPs) are firstly self-tested by running test programs, the scheme continues with testing of other non-programmable blocks including on-chip busses and heterogeneous components, where programmable cores are used as pattern generator and response analyzer. In [TBG05], the authors presented a framework to fully support the software-based testing of SOC designs. Unlike Krstic's work, the need for an external tester is eliminated with availability of the on-chip general-purpose microprocessor, dedicated random access memory (RAM) for storage of IP-specific test programs with instructions to be executed by a test coprocessor.

2.3.3 Strategies for Test Program Generation

Test program generation is essential to the SBST methodology, which aims to achieve fault coverage as high as possible with minimal manual work involved. As a result, there are a lot of research efforts focusing on this aspect.

A scalable SBST method for program generation was proposed by Chen et al. in [Chen03]. The novel concepts demonstrated in this paper concern the two aspects, i.e. 1) test templates and 2) renovating the idea of virtual constraint circuits (VCCs), firstly introduced in [TuAb97], to ease the process for module-level test generation. Templates consist of an instruction sequence necessary to deliver test patterns to the module under test (MUT) and capture the responses. Classified into single-instruction templates and multi-instruction templates, they are used as basic building blocks for constructing effective self-test programs. Exemplary template structures are illustrated in Figure 2-5. Bracket-enclosed symbols are settable fields and lines in bold represent the key instructions used to test the target module.

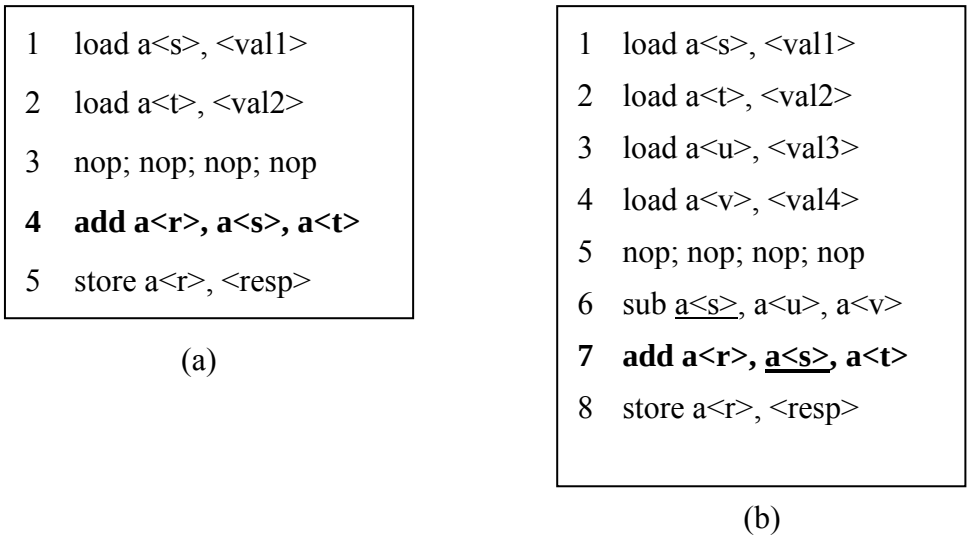


Figure 2-5: Single-instruction vs. multi-instruction templates (from [Chen03])

Functional tests are generated with assignment of random values to settable fields of templates. Then, these tests are simulated to study the relationships between the settable fields and the MUT inputs and outputs with utilization of regression analysis. Such relationships, termed as VCCs, represent the instruction-imposed constraints and at the same time serve as the mapping function to translate the module-level test patterns into instructions. Constrained ATPG is finally performed and the resultant test patterns are then translated into instructions. The proposed method was applied to an industrial processor "Xtensa" and achieved an attractive result. Due to inherent incompleteness of simulation, the extracted VCCs may not be accurate. For example, the observation that an input is always '1' during simulation does not necessarily mean the existence of the constraint that this port should be '1', since it may be the case that during simulation none of the tests assign the value '0' at that input. Such inaccurate VCCs may possibly degrade the achievable fault coverage level. In addition to regression analysis mentioned here, [Wen05] and [WWC06] establish mapping functions with use of other algorithm, e.g. Boolean- and arithmetic mappings, for control logic and the data-path units.

Kranitis et al. presented a divide-and-conquer approach to generate test programs [Kran03a][Kran03b][Kran05]. The entire process starts with information extraction based on the instruction set architecture and the RTL structural specification of the target processor. The purpose of this step is to identify micro-operations that each component performs. These operations are later used as "basic test instructions". Then, components are prioritized into categories with respect to their fault contributions and instruction accessibility. Enclosing the

identified “test instructions”, self-test routines are finally generated for the components selected according to their priorities. Corno et al. utilized the genetic-algorithm to automate the program generation process [Corn03][Corn04].

2.3.4 Test Program Generation for Diagnosis

Diagnosis involves fault detection and localization. Some recent literatures explore the diagnosis capability of SBST schemes. The initial work in this direction was presented by Chen et al. [ChDe02]. In order to achieve a high diagnostic resolution, special principles for program generation are introduced, that is, each program is designed to cover as few faults as possible while the union of all programs should cover as many faults as possible. Thus, it is necessary to 1) limit the variety as well as number of instructions used in a test program, and 2) create multiple copies of the same test programs but vary them to observe test responses on different outputs of the target module. After applying diagnostic test programs, the results are analyzed with use of a diagnostic tree.

Bernardi et al. introduced an approach for diagnostic test program generation [Bern06], which consists of three phases. In the first step called “sporing”, an initial test program is partitioned into a large set of small programs and the next selecting step “sifting” is activated to filter them with respect to their diagnostic capabilities. During the last step, diagnostic capability of the resultant test set is improved with resort of an evolutionary approach.

2.3.5 Test Improvement: Instruction-Level DFT Scheme

Lai proposed a novel method to add DFT mechanisms in the form of new instructions with the purpose to improve 1) fault coverage for hard-to-test modules and 2) shorten sizes of test programs [LaCh01]. With analysis of the synthesized test programs, the modules with low fault coverage are identified. Later, instructions to transfer data between the hard-to-test modules and general-purpose registers are implemented such that their testability is improved. The program sizes are reduced through substitution of frequently repeated code segments with new instructions that occupy smaller memory footprints. To minimize area overhead, the existing hardware is supposed to be reused during instruction implementation.

At its core, the idea itself is quite close to the emerging technology in processor designs, i.e. application-specific instruction-set processors (ASIPs). However, for the work by Lai [LaCh01], test improvement is the primary motivation to extend the instruction set, whereas ASIPs concentrate on other issues such as performance or power consumption. We identify it as an interesting and promising research to combine the instruction-level DFT and ASIPs. In Chapter 5, we will cover this aspect in more details.

2.4 Conclusion

The principle of the SBST methodology is presented in this chapter, which has a variety of advantages like low DFT efforts, low test cost, low power consumption, at-speed testing, reusability and flexibility. A comprehensive survey on SBST methods used for functional testing as well as structural testing is presented in this chapter.

With application of the SBST methodology in the embedded microprocessors, issues regarding energy and power consumption, memory requirement and test application time become apparently important. However, till now this aspect is yet lack of exploration, and as a result, this motivates our work here. We present in Chapter 4 the novel method to generate test programs that lead to high fault coverage with simultaneously optimized average power consumption, memory and test time [ZhWu05][ZhWu06].

Chapter 3 State-of-the-art in Low Energy and Power Techniques

3.1 Motivation for Low Energy and Low Power Designs

Today's chips feature in high density and high operating frequencies and all of these characteristics give rise of power consumption for these modern systems. Just as the other two criteria namely performance and silicon area, power dissipation is now the third basic concern for the designer as well as the test due to its fundamental consequences to the entire system. First, high average power consumption usually can be translated into a high temperature level, which may further deteriorate the system performance and reliability [Smal94][PeVa97]. Second, to avoid the problems due to overheating, special packaging and cooling technologies have to be adopted, thus increasing the cost. Furthermore, as to portable devices, energy/power consumption is an issue directly related with battery life time. The battery technology is evolved at a slower pace compared with the IC industry [RaPe96][NeMe97]. Thus high power consumption may mean to the autonomy systems reduced battery life times and usage inconveniences.

The excessive power dissipation is observed during test, which is several times higher than that in the normal mode [Zori93]. One explanation for this is that test patterns may cause node transitions as many as possible, whereas a power saving system mode only activates a few modules at the same time. Meanwhile, the normal mode ensures relatively high correlation among internal functional signals as opposed to the pseudo-random feature during test. BIST, for example, is usually executed at system speed and its execution typically results in considerably high circuit activity. If BIST is activated when the device is fully packaged, the power consumption may overpass the package limits and lead to malfunction or even circuit

destruction. This problem is more critical when testing complex SOC designs, since many cores are tested in parallel within the same BIST session [Zori93]. For this reason, it is particularly significant to minimize power and/or energy consumption during test.

In the next section, we start with discussion of the power dissipation sources in CMOS circuits, and present the corresponding models for them. The three basic metrics for low power design and test, i.e. energy consumption, peak and average power consumption, are introduced as well in this part. Then, we survey the techniques for power estimation at different abstraction levels, and emphasize the technique most relevant to our work, that is, to estimate power consumption at the instruction level. Over the years, a lot of methodologies are proposed for low power systems, and we give a general overview about these techniques. Afterwards, we detail the methods for low power testing that are related with our work. Moreover, as ideas from the research area about software power optimization are also considered in this work, it is also necessary for us to outline the state-of-the-art accordingly. A short conclusion is presented in the end.

3.2 Energy and Power Modeling

3.2.1 Sources of Power Consumption

The primary questions concerned here are, what are the sources of power consumption in CMOS circuits and how they contribute to the overall consumption. In general a circuit consumes power when it operates, and during its idle status when no operation is performed at all it draws as well power from the supply. The former corresponds to *dynamic power consumption*, which basically stems from switching of internal circuit nodes. The latter, termed as *static power consumption*, is sensitive to the applied transistor technology. In the past, dynamic power was the dominant factor, which accounts for 80% of total dissipation in a circuit implemented with technologies up to 0.35µm [SPG02]. However, this is just being changed as the technology scales down. Static power consumption already shows its increasing role in total dissipation.

Equation 3.1 suggests that both dynamic and static power can be further specified.

$$P_{\text{total}} = P_{\text{dynamic}} + P_{\text{static}}$$
$$= (P_{\text{capacitance-current}} + P_{\text{short-circuit}}) + (P_{\text{leakage}} + P_{\text{standby}}) \quad \text{Equation 3.1}$$

Two sources contribute to dynamic power consumption. Logic transitions at outputs of a node correspond to the process of charging or discharging of corresponding capacitances, and thus power, expressed with $P_{\text{capacitance-current}}$ in the equation, is consumed. '0'-to-'1' transitions charge a node i with the energy equal to $C_{\text{i}} \cdot V_{\text{dd}}^2$ drawn from the power supply, where C_{i} is the output capacitance of node i and V_{dd} is the supply voltage. Half of this energy is stored in the output capacitor, while the other half is dissipated through the parasitic capacitor and interconnects. And during the discharging process when the node experiences a '1'-to-'0' transition at its output, the stored energy will be dissipated. Thus, if node i makes in total the number of N_{sw} switching activities within the time span of T (correspondingly, the frequency is f_{clk}), we can compute $P_{\text{capacitance-current}}$ with Equation 3.2. The *transition density*, product of N_{sw} and f_{clk}, defines the "average switching rate". The efficiency of simulation-based power estimation can be drastically improved when transition densities instead of actual input transitions are propagated over the circuit [Najm91].

$$P_{\text{capacitance-current}} = \frac{1}{2} \cdot C_i \cdot V_{dd}^2 \cdot N_{sw} \cdot \frac{1}{T} = \frac{1}{2} \cdot C_i \cdot V_{dd}^2 \cdot N_{sw} \cdot f_{clk} \quad \text{Equation 3.2}$$

The other dynamic power consumption, $P_{\text{short-circuit}}$, also takes place during node transition. Its physical explanation is, there exists a very small time interval for an occurrence of transitions that both pMOS and nMOS transistors are ON. This means a short circuit current from the supply to the ground. Power consumption of this kind is proportional to the output capacitance, transistor size, input/output transition times [Pedr96].

As mentioned already, the applied technology is influential over static power in the forms of leakage power and standby power. P_{leakage} is caused by leakage currents of two types [ChBr95]: reverse-bias diode leakage on the transistor drains and sub-threshold leakage through an turned-off channel. Meanwhile, the actual operations of CMOS circuits, particularly related with degraded voltage levels feeding into static complementary gates as

well as utilization of pseudo-nMOS logic families, incur power dissipated even in the steady state [SPG02], represented as P_{standby} in the equation.

With move to the deep sub-micron technology, static power, especially leakage power, is becoming the dominant fraction of the total power consumption and therefore starting to receive more and more research efforts. However, leakage power consumption can be reduced through appropriate circuit design techniques. As to dynamic power consumption, it is related with circuit activities. When combined with the research on the SBST methods, it is depedent on the instructions executed. Therefore, our work tackles dynamic power consumption in particularly during test, and accordingly the later sections of the chapter will present perspectives in this regard.

3.2.2 Basic Metrics

There are a couple of metrics commonly considered in low-power designs and test, which reflect energy and power-related characteristics of a digital circuit.

- *Energy* (E_{total}), corresponds to the total energy drawn from the power supply within a particular time span. E_{total} has a special meaning for battery-operated systems as well as autonomous tests because of its relevance to the battery life time.

- *Average power* ($P_{average}$), is the quotient of E_{total} and the time span. Generally, by power estimation, people refer to the problem of estimating $P_{average}$ of a digital circuit. It is directly related to the issues of chip heating, temperature and reliability.

- *Peak power* (P_{peak}), is the worse case instantaneous power, namely the maximal power consumed in the circuit at a clock cycle. This factor is especially significant in the test context due to excessive power dissipation. Peak power violation is severe to the system, as it may lead to excessive power and ground noise, erroneously change the logic states, introduce additional delays, IR-drop and crosstalk, cause test fails and thus result in yield loss [Saxe03].

In particular, as to the test area, it is an overriding concern to measure these metrics for a specific test set. On the basis of power modeling presented in the previous section, we can delineate equations commonly referred in the test field.

As we already know that average energy consumed at a node i (E_i) over a time period is: $E_i = \frac{1}{2} \cdot C_i \cdot V_{dd}^2 \cdot N_i$, where C_i, V_{dd}^2 and N_i correspond to the equivalent output capacitance, the power supply voltage and the number of transitions at node i during the period. Nodes connected to more than one gate have higher parasitic capacitance, and consequently we assume C_i proportional to the fanout F_i of the node [WaRo96]. Thus, assuming C_0 is the minimum parasitic capacitance of the circuit, the energy E_i during a period is given by $E_i = \frac{1}{2} \cdot C_0 \cdot V_{dd}^2 \cdot F_i \cdot N_i$. The value N_i can be obtained through logic simulation and F_i depends on the circuit's topology. The product of N_i and F_i, termed as *weighted switching activity* (WSA) of node i, is often used to represent the supply current demands and power dissipation of the relevant gate during test. Summing the WSA of all the gates in the circuit gives the entire circuit's WSA.

Now, we can extend the above formulation to estimate energy consumption for a test set TS consisting of input vectors V with a length of L, that is, $TS = \{V_k \mid k \in [1, L]\}$. The energy consumed by a pair of successive patterns $(V_{k-1},\ V_k)$ is now calculated by:

$$E(V_{k-1},\ V_k) = \frac{1}{2} \cdot C_0 \cdot V_{dd}^2 \cdot \sum_i N_i(V_{k-1},\ V_k) \cdot F_i \qquad \text{Equation 3.3}$$

Where $N_i(V_{k-1},\ V_k)$ represents the number of switching activities provoked by $(V_{k-1},\ V_k)$ at node i. The equation implies that i should range all the internal node of the circuit.

Accordingly, the total energy consumption of the test set with the length of L is:

$$E_{total} = \frac{1}{2} \cdot C_0 \cdot V_{dd}^2 \cdot \sum_{k=2}^{L} \sum_i N_i(V_{k-1},\ V_k) \cdot F_i \qquad \text{Equation 3.4}$$

If T is the clock period, the average power consumed by those L test vectors is:

$$P_{average} = \frac{E_{total}}{L \cdot T}$$ Equation 3.5

The instantaneous power $P_{inst}(V_{k-1},V_k)$ incurred by (V_{k-1}, V_k) is given by: $P_{inst}(V_{k-1}, V_k) = \frac{E(V_{k-1}, V_k)}{T}$. Therefore, the peak power consumption, namely the maximal instantaneous power, can be specified as:

$$P_{peak} = MAX[P_{inst}(V_{k-1}, V_k)]_{k=2}^{L} = MAX[\frac{E(V_{k-1}, V_k)}{T}]_{k=2}^{L}$$ Equation 3.6

3.3 Power Estimation

3.3.1 Overview of Power Estimation Techniques

In general, there are numbers of factors that challenge the accuracy of power estimation [Pedr96]. First, complicated signal dependencies exist extensively in ICs, and as a result, even for small circuits, it may be infeasible to estimate power dissipation based on exhaustive simulation under consideration of all correlations [MMP94]. Second, due to different timing properties of gates and paths inside ICs, nodes may experience several transitions before settling to their steady values. These extra transitions, known as glitches or hazards, sometimes can contribute 9% - 38% of the total power dissipation [BFR94] and however, it is extremely difficult to precisely calculate this kind of power consumption. Third, the logic style used to implement the circuit, e.g. static or dynamic CMOS, also influences power consumption [Pedr96].

An intuitive way for power estimation is to simulate the circuit, and monitor directly the current drawn, or as an alternative, signal switching activities. Its major attractiveness lies in accuracy and generality, as it is applicable to any circuit regardless of design styles, technologies and so on. However, the major pitfall for it is its strong input dependence [Najm94]. In practice, power estimation for a circuit may be started without complete

availability of the end environment where it is embedded. In this case, it is difficult to obtain specific input signal information. Moreover, as numerous input data have to be simulated, it is practically not feasible for large circuits due to computational expensiveness.

Basically, the power estimation techniques fall into two categories: the simulation-based and the probability-based. Figure 3-1 [Najm94] illustrates the flows of these two methodologies. As to simulation-based approaches, a large set of input patterns are applied to the circuit and the corresponding current waveforms are monitored and averaged to derive the power consumption. Alternatively, one can move the "average" step before simulation, and determine the probability of an input signal to make a transition. With such information, only a single run with probabilistic analysis tool is necessary.

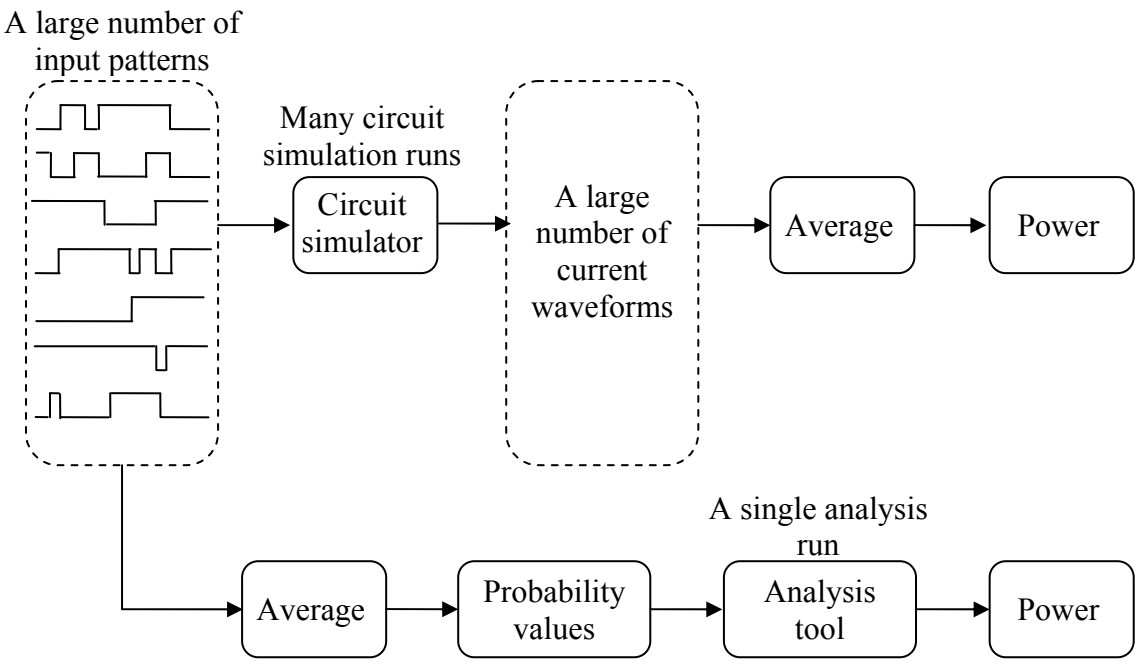


Figure 3-1: Two Power Estimation Flows (from [Najm94])

There are a lot of publications in the area of simulation-based power estimation that focus on reduction of the number of input vectors [Burc93][Ding98][MMP99]. In [Burc93], Burch et al. based the power estimation on Monte-Carlo simulation, where random patterns are applied at the circuit inputs. During simulation, the switching activity of each node is monitored and checked against a stopping criterion to determine whether its stable value is reached. The simulation ends and derives average power dissipation when the stopping criterion for all nodes is met. The major shortcomings of this approach, as pointed in [Ding98], are concerning the aspects of 1) the large set of input vectors and thereby the long simulation time

to extract reliable statistics; 2) estimation inaccuracy due to the un-captured spatiotemporal correlations and 3) unsuitability for circuits with non-normal sample distribution. In order to significantly reduce the number of simulated vectors while satisfying the specified confidence level, [Ding98] presented an estimation method that is based on the stratified sampling technique to derive appropriate samples for simulation. Another possibility to reduce simulation efforts is vector compaction. The basic idea behind is to generate a new input sequence out of the given one, which has a shorter length and offers a good approximation of the original power consumption. To ensure estimation accuracy, correlations should be preserved. Marculescu et al. proposed in [MMP99] a vector compaction methodology that relies on Markov chains to capture complex correlations.

For probability-based estimation, [Ciri87] propagated the given signal probabilities at the primary inputs inside of a circuit to obtain the probabilities at every node. Under assumptions of the zero-delay model and temporal as well as spatial independencies, Binary Decision Diagrams (BDDs) can be used to propagate the signal probabilities. A realistic timing model is considered in [TPD93]. In addition to techniques for combinational circuits, power estimation for sequential logic was also tackled in [Tsui95].

3.3.2 Related Technique: Power Estimation at Instruction Level

It may not often be possible to estimate power consumption of software using measurement tools that are applicable with availability of layout details of a processor, since such knowledge is usually not provided with software designers. Even when it is available, the low level measurement tool can not be yet desirable, and sometimes impractical, due to its long simulation time and computational expensiveness. In this regard, a more appropriate alternative can be to build up and validate an instruction-level power model for the target processor. Thus, given the model and the assembly program, software designers can quickly verify whether the required power constraints are met or not on the basis of the corresponding power prediction, and even use the information in design space exploration towards power minimization.

The studies in this direction were firstly initiated in the 1990's. Among them the technique presented by Tiwari et al. [TMW94][Tiwa96a][Tiwa96b] was recognized as the milestone work and cited by the later literatures, e.g. [Schu02][SiCh02]. According to their formulation,

energy consumption of an assembly instruction is basically associated with two components: the base cost ($Base_i$) and the inter-effect cost ($IE_{i,j}$). The first factor represents the basic energy needed for executing the specific instruction. There may be variations in base cost of a same instruction family due to difference in operands and addressing modes used. For example, an "add" instruction with immediates may yield basic energy consumption different from the one with register-related operands. However, the variation range in this regard is found to be small, less than 5% [TMW94], and average base cost values are used to limit the discrepancy level. It is observed that energy consumption of a pair of two different instructions is greater than the average of their base costs. The reason behind is the circuit state overheads due to instruction context changes, which is correspondingly modelled by the second factor ($IE_{i,j}$). At last, during real program execution, effects such as cache misses and pipeline stalls may take place, which consume as well energy and therefore should also be considered in the overall model. Putting all the factors together, Tiwari proposed the equation below to predict energy consumption of an assembly program P [Tiwa96a]:

$$E_P = \sum_i (Base_i \times N_i) + \sum_{i,j} (IE_{i,j} \times M_{i,j}) + \sum_k E_k \qquad \text{Equation 3.7}$$

N_i and $M_{i,j}$ correspond to the numbers of occurrences of instruction i and instruction pair (i, j) in the program respectively. E_k tackles the energy consumption yet unconsidered for the other two factors, just as mentioned before the effects of stalls and cache misses for instances. In order to create a reference table with all the information regarding instruction base costs and inter-effect costs, Tiwari set up a physical measurement environment. To ensure stable readings from the ammeter, instruction instances, either of a single instruction or a pair of different instructions, were implement with special care and put into infinite loops. The overall technique was then validated with various programmable cores, and proved its close correspondence between the estimated and measured costs.

[Schu02] adopted the same idea and concretely built up the instruction level power model for the SPARC v8 compatible Leon processor that has variety of real-world applications. Instead of physical measurement, hardware simulation at the circuit level, e.g. RTL, was used, and energy consumption was correspondingly approximated by the number of signal switching activities monitored during simulation. For the validation purpose, the model was used to

evaluate energy consumption for four benchmark programs and the discrepancy in terms of total transitions was found to be less than 20%.

[SiCh02] presented an energy estimation methodology with accuracy of 3% for a set of benchmark programs evaluated on the StrongARM SA-1100 and Hitachi SH-4 microprocessors, and a technique to fine analyze software energy consumption into switching and leakage categories.

3.4 Power Optimization

In this section, we start with discussion of the freedom for power minimization and present afterwards in brief an overview of existing optimization techniques. As it is an emphasis of our work to deal particularly with power consumption during test, we then illustrate the approaches proposed in the area of low power testing. Besides, with the increasing roll of software in embedded systems, it is an intensified topic to reduce power consumed by software and we elaborate the methods relevant to our work here. Note that, for the first time, we take explicitly into account power consumption for software during self test and propose the approach that optimizes power dissipation without penalty of fault coverage.

3.4.1 Optimization Degrees

As indicated in Equation 3.2, physical capacitances, the power supply voltage, node switching activities and the clock frequency are the basic factors to be tuned towards low power designs, and in fact, a power reduction technique often tackles some of these factors at the same time.

Linear to the power consumption, the physical capacitances of a circuit can be split into two aspects: capacitances of nodes and interconnects, whose values are related with the used technology library and detailed information about the place and route of the circuit. Thus, it is possible to reduce power dissipation through optimization of wire lengths, transistor or gate sizes, and logic sharing [Pedr96].

The voltage of power supply is the dominant factor, as it is quadric to the dynamic power dissipation. However, we do not have the complete freedom to scale this parameter, since degradation of the voltage level leads to drastical increase of delays and thus impact the entire

system speed. To avoid this negative influence, voltage reduction can commonly be achieved through exploitation of processing parallelism at the cost of hardware overhead.

Minimization of switching activities, or sometimes weighted switching activities (WSA), also reduces power consumption. In constrast to the previous two alternatives, it depends mainly on the design techniques [SPG02], and thus the expense for the new low-power technology library or increase of area overhead is no more needed. The design techniques can be, for example, circuit partition, path balancing and signal reordering during logic synthesis.

3.4.2 General Overview of Optimization Techniques

Over the years, a variety of methods have been proposed to minimize power consumption at all levels of the system design. In general, the higher the abstraction level is aimed, the larger the power saving is expected to obtain. In this section, we present a brief overview of the optimization approaches in a hierarchical way.

At the physical level, the focus of optimization is usually the load capacitances or switched capacitances. Scaling the capacitances not only affects power consumption but also gate delays, and as a result, it is necessary for an optimization method to take all of them into full consideration. The proposed approaches concern mainly optimization during place and route [VaPe93][ChBr95], transistor sizing [ChBr95][Pedr96] and reordering [SLW95][PeVa97]. Instead of net lengths, [VePe93] used switched capacitances as the objective function during the placement process, and an average power reduction of about 10% was reached compared with the net length optimal solution without increase of delays. [ChBr95] applied a similar idea during routing. The principle of power-optimal place and route is to assign short wires to signals with high switching activities whereas longer wires to those with lower transitions [ChBr95]. [PeVa97] studied the relationships among transistor sizes, power and delays. The basic problem in this regard is to find an appropriate solution for transistor sizes that meets a given delay constraint within a minimal power budget. About 15% - 20% reduction in total power dissipation can be obtained as a result of transistor sizing. Due to existence of functional equivalency among pins of logic gates, the idea of pin reordering is to match the switching-intensive inputs with pins of low capacitances. Heuristics were proposed, e.g. [SLW95]. Experimental results showed that transistor reordering is likely to lead to about 5% power reduction.

At the circuit level, numerous approaches of hardware synthesis and design for low power have been proposed [Alid94][TAM95][ChBr95][PeRa02][BeMi95a][MoDe96][SPG02] [OlKa94][BeMi95b][Tsui98]. Sometime, particular circuit properties can be exploited to decreasing power dissipation. For example, Alidina et al. proposed a technique, known as "pre-computation", to generate the output values one clock cycle before required and use them afterwards to reduce internal switching activities [Alid94]. A similar idea is found in another method called "guarded evaluation" proposed by Tiwari et al. [TAM95], where existing signals, instead of new logic, are utilized to generate the disabling signals to prevent unnecessary transistions. Other special properties, such as input sequences, can also avail for power optimization. Chandrakasan et al. pointed out in [ChBr95] that switching activity can be minimized through reordering the input sequences. When outputs of a module in a circuit are not needed, the clock can therefore be disabled so as to prevent transitions in 1) flip-flops; 2) the fanout gates of flip-flops and 3) the clock tree [PeRa02]. A technique to automatically synthesize low-power finite status machines (FSMs) with gated clocks was presented in [BeMi95a]. Power metrics need to be considered during logic synthesis as well. [MoDe96] introduced a circuit topology with balanced delay paths so as to decrease signal hazards. A partitioning algorithm was outlined in [SPG02] to generate a chain-structured circuit where transition-intensive inputs are injected into the decomposition tree as late as possible. The encoding schemes for FSM's states were proposed in [OlKa94][BeMi95b][Tsui98] to minimize transitions at state lines.

The optimization approaches at the system level are mainly categorized into two groups: 1) dynamic power management and 2) dynamic power scaling. The main idea of the first class is to place the idle modules into a low-power state based on a certain power management policy. As the policy is quite important to the system both in terms of performance and energy savings, lots of research efforts are spent in this regard, e.g. timeout and predictive algorithms proposed by [RaGu00][SCB96], and stochastic model-based approaches [Beni99][Simu01]. For dynamic power scaling, the system voltage is adapted to that actually needed by the running tasks. In this way, power as well as energy consumption is reduced without performance penalty. The difficulty for dynamic power scaling is to predict the future workload as accurately as possible, especially when the workloads differ from one time span to another. [SKL01] and [LeSa00] tackle scaling issues under the real-time scenario, where the information of the running applications is exploited to determine the suitable voltage for

the next clock cycle. The non real-time scheduling is presented in [Weis94] which adjusts the voltage based on regular monitoring of system utilization at a uniform time interval.

3.4.3 Low Power Testing

As shown in [Zori93], power consumption during test (test power) could be twice as much as that during the normal mode. Several reasons can explain this observation. First, to reduce test time, much more modules are activated simultaneously in the test mode than in the system mode. Second, lower correlation is found among test patterns than functional input vectors. Therefore, test power needs to be properly dealt and optimized in order to avoid severe consequences such as system reliability, yield-related problems, or even circuit destruction. Unfortunately, the algorithms and techniques introduced in section 3.4.2 are generally not applicable in the test mode. The primary concern for a low-power testing solution addresses test efficiency and power consumption at the same time. In this section, we present the low power testing methods that are related with our work.

3.4.3.1 Power-Efficient X-Filling

It is quite common that a test vector generated by ATPG is not fully specified, as the binary values assigned to those bits are irrelevant to fault detection, thus called Don't-cares (X's). Moreover, it is observed that the fraction of X's is nearly always a large portion of the total available bits [Wohl03][Hira03]. The conventional ATPG fills X's randomly and the resultant fully-specified pattern is then simulated to evaluate fault detection. From the low-power standpoint, it is not appropriate to randomize the values at X's and accordingly there are studies to specify X's in a power-efficient way [Butl04][Bade06a][Bade06b][Reme06]. In [Butl04], three heuristics for X-filling were presented:

- Adjacent filling also called MT-filling (Minimum Transition Filling): to specify X's with the value of the last encountered care bit;
- 0-filling: to specify X's always with '0'
- 1-filling: to specify X's always with '1'

Assume a test pattern is "0XX1XX0X1XX0XX", according to the above three filling schemes, the resultant vector is:

- 00011100111000 with MT-filling
- 00010000100000 with 0-filling
- 01111101111011 with 1-filling

The evaluation results implied that the MT-filling scheme performs better than the other two in limiting the overall switching activities. In [Bade06a][Bade06b], Badereddine et al. took into account structural properties of given circuits, and utilized such information during X-filling to minimize peak power consumption in scan testing.

Another method, called Preferred Fill, was introduced in [Reme06] to deal with scan test power consumption. The authors discussed separately the X-fillings at m-length primary inputs (PIs) and n-length state signals (S) for two consecutive vectors $V^i = (PI^i_{1..m},\ S^i_{1..n})$ and $V^{i+1} = (PI^{i+1}_{1..m},\ S^{i+1}_{1..n})$. X's at PIs are assigned as follows:

- If $(PI^i_j = X) \wedge (PI^{i+1}_j \neq X)$, then $(PI^i_j \leftarrow PI^{i+1}_j)$, and vice versa;
- If $(PI^i_j = PI^{i+1}_j = X)$, then fill with a random value ('1' or '0')

For filling Xs at state lines, the signal probability (HP) of each scan cell is used to define the corresponding preferred value. For example, if the j^{th} flop-flip is with high probability to take the value '1' than '0', namely HP(1) > HP(0), then the preferred value is 1, otherwise is 0. If HP(0) = HP(1), then no preferred value is available. The process to fill Xs at state lines proceeds in two steps:

Step 1: for $j \in [1,\ n]$, if $(S^i_j = X) \wedge (S^{i+1}_j \neq X)$, then $S^i_j \leftarrow S^{i+1}_j$

Step 2: for all the remaining X's in S^i:

If corresponding preferred values exist, then fill with the preferred values,

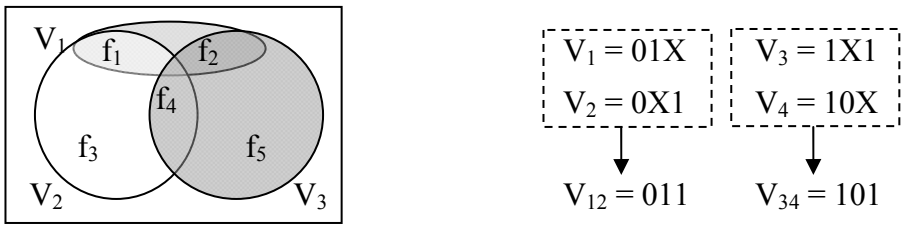
Otherwise, fill with random values

The experiments with use of benchmark circuits as well as industry circuits showed a substantial reduction of both peak power consumption and average power consumption during scan test.

3.4.3.2 Test Data Compaction

The volume of test data is an important factor which affects test application time and the memory requirements, and especially for autonomy systems, it is also related with energy consumption during test. For this reason, compaction algorithms are often explored with the aim to reduce the sizes of test data while maintaining fault coverage, and can be classified into two groups: static compaction or dynamic compaction. The static compaction is known as the post-processing after test generation, whereas a dynamic compaction algorithm is usually incorporated into the test generation stage.

As to static compaction, there are several degrees of freedom left within the test set generated by an ATPG. First, ATPG adds a vector into the test set as soon as new faults are detected. By this means, it may result in a test set with redundancy, as faults detected by early-generated test patterns are likely to be covered as well by later patterns. For example, in Figure 3-2 a), as all faults of the vector V_1 generated before are detectable by the union of V_2 and V_3, V_1 becomes redundant and can be removed to shorten the data volume. Section, the existence of don't-cares in the test set also enables test data compaction. As shown in Figure 3-2 b), with properly filling X's in compatible test patterns, the size of the test set is reduced from the previous 4 to 2.



a) Redundance in test vectors b) Don't-cares in test vectors

Figure 3-2: Possibilities for Data Compaction

Data compaction has not only influences on the test size, but also the average and peak power consumption. In [SOT00], Sankaralingam et al. presented a compaction method, which minimizes power consumption with careful selection of the merging order of test cube pairs during static compaction. A static compaction algorithm is proposed by Corno et al, which utilizes Genetic Algorithms to find the optimal order for the given test sequences such that the test length is minimal [Corn97].

3.4.3.3 Reordering

As power consumption is proportional to signal transitions during test, the basic idea of reordering is to find the appropriate order for test vectors and/or the order for scan cells such that the total switching activities are minimized without degradation of initial fault coverage.

Usually, the ordering problem is formulated as the classical "Traveling Salesman Problem". In [ChDa94], Chakravarty and Dabholkar construct a complete directed diagram where vertexes represent test vectors and edges correspond to transitions activated by the vector pair. A greedy heuristic is later introduced by the authors to find out a Hamiltonian tour with the minimal cost. As the simulation time to derive transitions grows in polynomial with the number of test patterns, this approach may not be applicable for circuits that have large test sets. To overcome this problem, Girard et al. use Hamming distances to evaluate the switching activities invoked by consecutive test patterns [Gira98]. Under consideration of the circuit structural characteristic, the authors reorder the test vectors to reduce transition density at circuit inputs and showed a better result in terms of average and peak power savings [Gira99].

In addition to techniques focusing on test vectors, there are some research efforts on the order of scan cells. Dabholkar et al. introduced two techniques, i.e. the random heuristic and the simulated-annealing algorithm, to order scan cells [Dahb98], which reached around 10% to 25% reduction of power consumption. In [GBT03], Ghosh et al. target minimization of area overhead and power consumption at the same time via scan cell reordering, and tune the trade-off with a parameter λ that specifies the relative importance between these two factors.

3.4.2.4 Other Techniques

Aside from the above-mentioned techniques, there are a variety of publications dedicated to low-power BIST. [Zori93] tackled the test scheduling for a distribute BIST scheme under the given power budgets. Gerstendoerfer and Wunderlich proposed an approach to reduce power consumption during scan test [GeWu97]. As most energy, from 62.84% up to 98.99%, is consumed in the CUT during the shifting phase, dramatic power savings can be obtained by the usage of a modified shift register, which suppresses the switching activities at the outputs. As to the test-per-clock scheme, Girard et al. introduced a low-power method, which activates

one half of the D flip-flops with a modified clocking scheme and results in significant energy and power savings in the CUT as well as the test pattern generator (TPG) due to minimized signal transitions [Gira01].

3.4.4 Software Power Optimization

Hardware-software co-design is a key part in embedded systems, which defines the hardware and software portions in the target system, and considers concurrently their dependencies [Pedr01]. From the standpoint of power minimization for the entire system, it is necessary to optimize power consumed in software as well as in hardware. Thus, in recent years, reduction of software power consumption becomes one of the hot topics in low power designs.

Tiwari showed that the conventional compiler optimization aiming mainly at program speed is also effective for low-energy [Tiwa96b], and most of the existing software techniques for low power are therefore based on compilation with emphases on the following aspects: 1) reducing memory accesses through appropriate register allocation [Tiwa96b]; 2) processor specific optimization (dual memory accessing, instruction packing, operand swapping etc.) [Lee97]; 3) instruction scheduling [STD94][Lee00]; 4) register name adjustment [PeOr03]; 5) instruction encoding [STD94][PeOr04].

Based on analysis of software energy consumption in three processors with different architectures, Tiwari et al. pointed out instructions requiring memory accesses are power-expensive [Tiwa96b]. An effective way to reduce memory accesses is to better utilize registers. This idea was then confirmed by a manual register allocation scheme and for this particular case study, as much as 33% energy reduction was achieved. Commonly viewed as a milestone work in the research area, this publication presented additionally other perspectives regarding software optimization for low power, e.g. energy cost driven code generation and instruction reordering.

Lee et al. studied closely energy consumption for a Fujitsu embedded DSP core, and found that as oppose to large general-purpose processors, smaller and specialized microprocessors like DSPs are more subject to circuit state changes [Lee97]. This suggested that a greater energy savings are likely to obtain through instruction scheduling. Characteristics of the DSP (dual-memory access, packing two instructions into one) were also analyzed and utilized in

the optimization. Besides, as the multiplier was the major source of energy consumption, a technique to swap operands to reduce switching activities in this unit was proposed.

In tradition, instruction scheduling targets mainly issues regarding program execution time, and recent researches show another focus of interest, namely scheduling instructions under the guideline of power consumption. [STD94] presented a cold scheduling method to minimize transitions in the control path. [Lee00] targeted the very long instruction word (VLIW) architecture, and proposed the so-called “horizontal” and “vertical” scheduling to allow optimization of microinstructions within and even across instruction scopes.

The work of Petrov et al. was motivated by the observation that different register indices introduce varied numbers of transitions on the instruction bus and the register file address decoder. A register name adjustment method [PeOr03] was presented to traverse the space of equivalent register allocation solutions and choose from them the one with the minimal transitions. There are also research efforts that resort to instruction encoding techniques to reduce switching activities on the instruction bus. [STD94] compared the gray code with the binary code. [PeOr04] introduced a code transformation scheme that takes into account both the switching activities at an individual bus line and coupling activities at its neighboring lines, which has a particular significance in power dissipation under nano-meter technologies.

3.5 Conclusion

The ultra-large integration scale, escalating operating frequencies, and popularity of portable devices make it necessary to take energy/power consumption, in addition to the classical two parameters of area and performance, into full consideration during system design. In this chapter, we firstly present and model the power dissipation sources in ICs. As to the research area on power-aware design and testing, three kinds of metrics are usually concerned: energy consumption, average power consumption and peak power consumption. Next, we briefly review the techniques for power estimation and outlined the emerging topic related with our work, namely power estimation at the instruction level. Afterwards, a survey of techniques regarding low-power design is provided and we then emphasize the domain of “low power testing”, where methods, such as X-fillings, compaction and reordering that sparkle our work are introduced in detail.

Chapter 4 Software-based Self-test under Memory, Time and Power Constraints

4.1 Overview of the Proposed Approach

Just as presented in Chapter 2, our work utilizes processor instructions as a means to transport tests, which aim at detection of stuck-at faults, to the core; hence it belongs to the group of the software-based self-test schemes for microprocessors. However, we identify in this work that memory, test time and energy/power consumption are critical issues when the SBST method is applied to embedded systems. Thus, we proposed an approach for test program generation which optimizes these three factors at the same time while preserving the initial fault coverage. In short, we summarize the primary goals in this work as follows:

1) High fault coverage;
2) Automatic test program generation;
3) Short test application time;
4) Small memory requirement;
5) Low energy and power consumption

Although ATPG fails to reach sufficient fault coverage for the entire processor without any DFT mechanisms, it is capable of generating deterministic tests with high fault coverage for combinational modules in the processor. On the other side, deterministic test algorithms are also quite effective for particular components. For example, the register-files can be effectively tested with deterministic algorithms proposed initial for memory tests. Thus, to achieve high fault coverage, we can base on these deterministic module-level tests and map them into corresponding processor instructions.

The next question for us is whether it is possible to automate the “translation” process such that minimal human intervention is required. In order to answer the questions, we need to analyze the typical instructions used to realize the module-level and represent this common part with use of “templates”, which are similar as that introduced in 2.3.3. The final test program is then generated step-wise on template instantiation.

An embedded system is usually with a very stringent memory budget. Therefore, if we anticipate utilizing the generated programs not only for manufacturing test but also for periodic in-field test, the code size must be optimized to meet the small memory requirement. The possible solution to this problem can be to reduce the number of module-level test patterns based on compaction techniques, which brings us benefits concerning shortened test time and optimized energy consumption.

As mentioned previously, structural test causes several times more nodes to switch; peak power consumption as well as average power consumption should be dealt with special care during test. As to the SBST scheme, it is functional in nature; therefore, we can believe that the system specification of peak power consumption will not be violated. Nevertheless, delivering structural module-level test patterns, SBST routines consume still much more power than other applications with normal workload, and as a result, it is necessary to optimize test programs so as to avoid possible consequences like reduced system performance or overheating etc. due to high average power consumption. So, we set here low power consumption as one of our goals. The available program optimization techniques for low power give us a good starting point; however they are not completely suitable for the SBST programs, for they are supposed to ensure consistent program semantics throughout optimization whereas our interests are centralized on test. New perspectives to guarantee test effectiveness should be introduced into the conventional software optimization for low power. When the embedded system under test is with limited energy resources, energy consumption of test programs need be minimized. We can achieve this goal through reduction of test time.

Figure 4-1 illustrates the proposed scheme which basically consists of two phases. In the first phase, we target the first two goals, namely automatic generation of effective test programs. Instead of synthesizing test programs for the processor as a whole, we decompose the design and generate tests on a module basis, where deterministic algorithms and/or patterns can be

applied to ensure high module-level structural fault coverage. The processor-level tests, namely test programs, are then generated with instantiation of templates, where module-level patterns are mapped into instruction sequences.

In the second phase, we perform a variety of optimizations aiming at memory, test time and power consumption. For example compaction can be used to limit the number of applied patterns, which is beneficial to the purposes of lower memory and time usage. In this case, new programs have to be generated to incorporate the compacted test set. Software power consumption can be minimized through instruction rescheduling for an instance. Our later sections will present each of these two phases in more details.

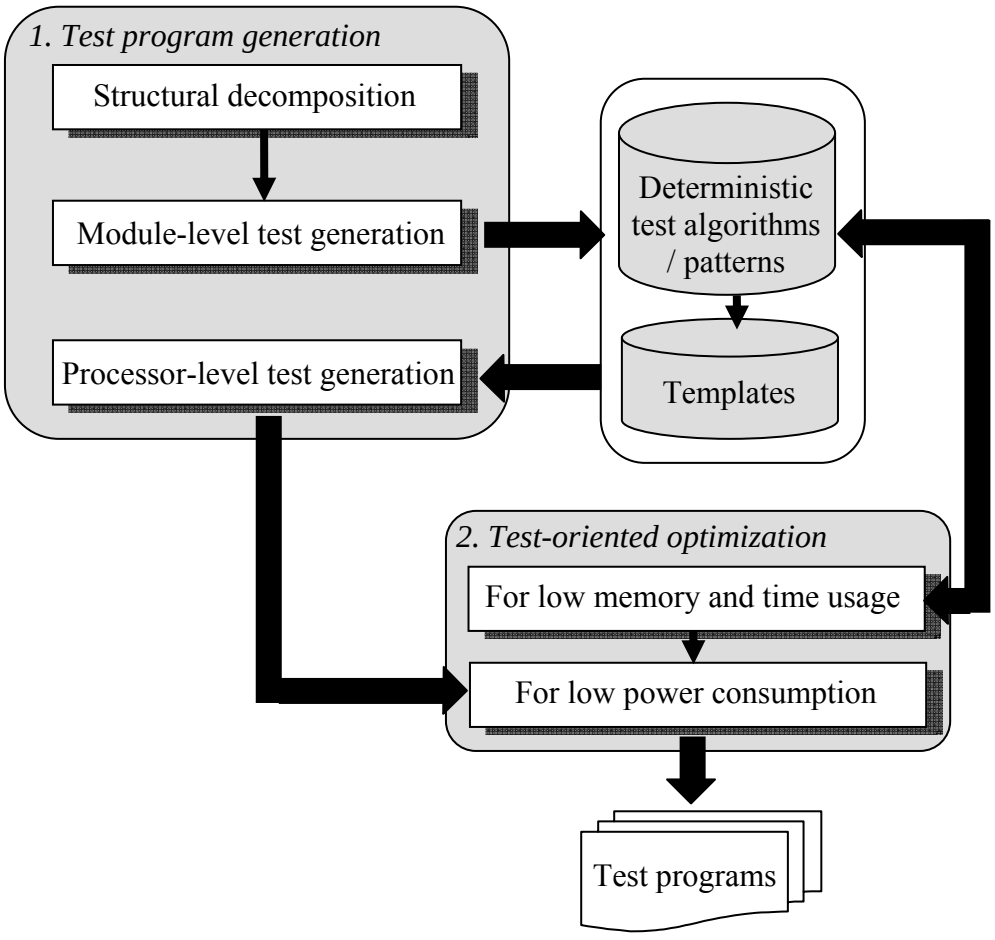


Figure 4-1: Overview of the proposed SBST scheme

4.2 Test Program Generation

4.2.1 Divide-and-conquer Test Strategy

Processors are so deep sequential circuits that even the most advanced automatic test generation tools fail to manage the complexity [GPZ04]. However, instead of aiming directly at the entire processor during program generation, we can decompose the processor into modules and then realize effective module-level tests with instructions. The benefits of such divide-and-conquer test strategy are multifold. First, deterministic test algorithms are thus applicable to particular modules, which in turn ensure high module-level fault coverage. Second, the instruction set architecture (ISA) describes the links between the modules and their related instructions. In other words, it is comparably easy to identify the instructions necessary to realize the test for the module under consideration. For example, to test the arithmetic and logic unit (ALU) we can use the computational instructions like addition ("ADD"), subtraction ("SUB") and so on, while testing the multiplier is necessary to use multiplication and/or division accordingly. Third, during application of a test program synthesized for a particular module, it is possible to detect as well faults in components that are primarily not targeted. Therefore, it is very important to decide reasonable test priorities among the modules, as it is not only an issue regarding fault coverage but also related with the efforts involved in program generation and test sizes.

The overall test generation strategy is depicted in Figure 4-2, which contains three phases:
1) Processor decomposition and test prioritization;
2) Module-level test generation;
3) Processor-level test generation.

In the first phase, we divide the processor into components and under consideration of the ISA evaluate their testability in terms of instruction accessibility, area and fault contribution. In the end, test priorities are associated with the modules. A high test priority indicates that the module is easy to be tested via instructions and/or with large area (fault) contributions. Then, we continue with module-level test generation, starting from the modules with high test priorities. Here, the conventional structural test techniques can be utilized. For an instance, ATPG is applicable to the modules that are combinational, or deterministic algorithms are adopted for register file tests. Afterwards, for each target module, we construct a

corresponding template which encapsulates instructions essential to test delivery. As an alternative, we can also construct a template that combines tests for several modules. The test programs are eventually generated via step-wise template instantiation and evaluated in terms of fault coverage. The generation flow repeats until either the desired fault coverage level is reached, or only the modules exists that are hard-to-test with instructions. Usually these modules are deeply sequential and sufficient fault coverage in them may usually require specific DFT mechanisms.

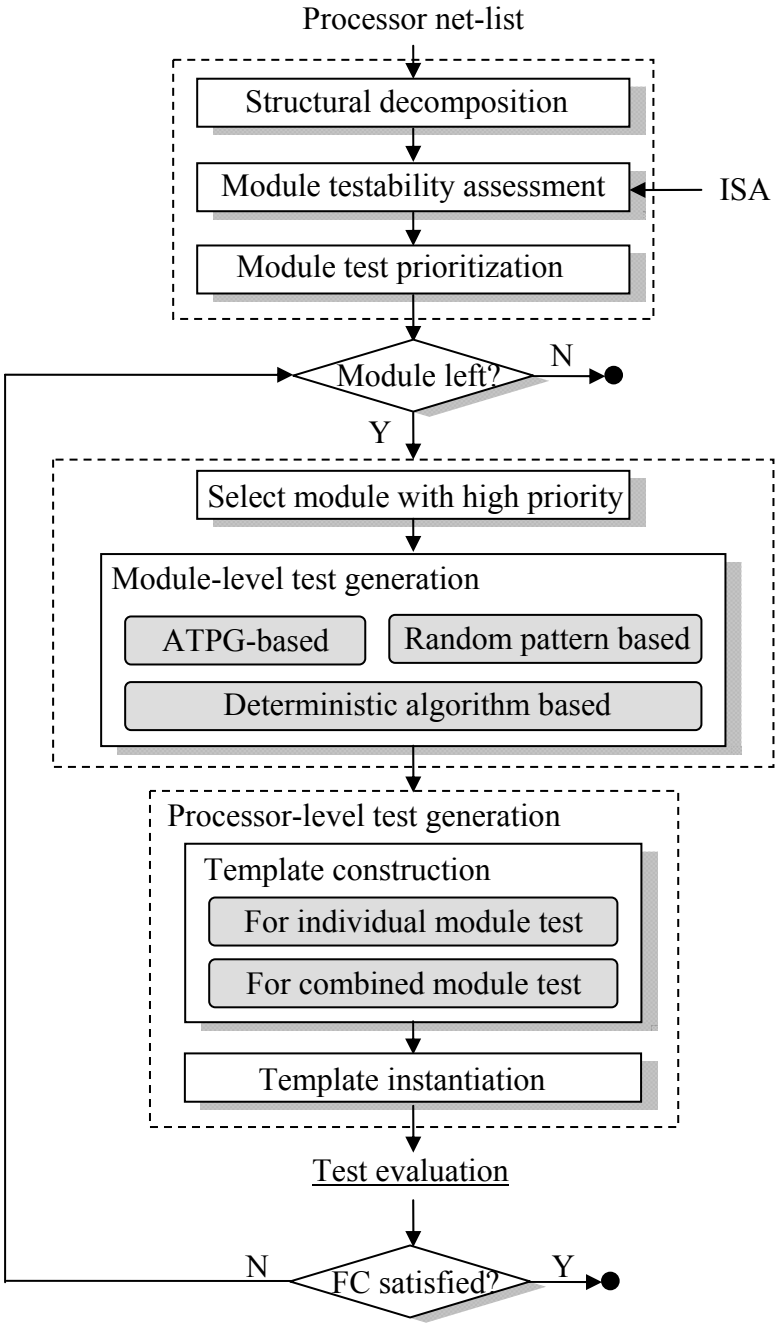


Figure 4-2: Divide-and-conquer test program generation

4.2.2 Processor Decomposition and Test Prioritization

With knowledge of the processor architecture, e.g. the hardware description at the register transfer level (RTL), we can decompose the processor into several functional blocks, like the ALU, the register file, the multiplier, the program counter, the pipeline, the controller and so on. These blocks exhibit different testability as well as influences on overall area and faults. For instances, register files usually take up a large portion of total hardware area in a processor, and at the same time, there are quite a lot of instructions that are able to apply values to or read values from this unit. Therefore, targeting such blocks early in program generation promisingly leads us to a high fault coverage level with relatively low efforts.

To prioritize the order for the modules in test generation, we mainly judge the following properties of the components:

- Instruction testability

By instruction testability, we refer to the degree of easiness to apply particular values to inputs of a component and observe the corresponding respond, either directly from the data and/or address bus or from the general-purpose registers, using assembly instructions. It is a critical facet in the software-based scheme that reflects whether a component is easily instruction testable or not.

In general, functional units like the ALU or a multiplier are easily instruction testable, since there are a variety of instructions able to activate the components with specific operands and correspondingly, each of these instructions can be the candidate to deliver the test for the module. Taking the ALU as an example, the instruction

Add Rd, Rs, Rt

can apply certain values stored in “Rs” and “Rt” to the inputs of the ALU, and upon the operation of “Add”, the corresponding response will be available in “Rd”. In other words, the ALU is controllable and observable with use of the “Add” instruction.

Modules like instruction decoder, the pipelines, etc are usually with low instruction testability. As to the decoder, its inputs are related with the OPCODE of the instruction under consideration, which means by properly selecting the instruction used, we can apply a specific value at the decoder. However, observation of the controlling signals generated after decoding

is not that straightforward, as none of existing instructions provide direct access to those signals. Similarly, modules like pipelines which are invisible to the programmer at all are not controllable and observable with use of instructions.

- Area and fault contribution

It is effective to target firstly the components with the largest contributions to the total processor area and faults during test program generation. Normally, modules like the general-purpose registers and other functional units in the data-path are the major fault and area contributors. Taking the Leon processor [Leon] from Gaisler Research for an instance, up to 72.17% of the processor area (also the same for the faults) excluding the cache memory is occupied by its register file with the window size of 2. This portion is increasing if a larger register window size is configured. Meanwhile, the integer unit takes up around 19.38% area for the same processor implementation and will be less impacted by various configurations. Needless to say, detection of faults in these components will have a significant influence on test effectiveness.

In light of the above criteria, we then prioritize component-level tests. As to a standard processor, we tackle test generation for instruction-visible, functional modules like the ALU, the register file, the program counter, etc., and leave out other components such as the control unit that constitute deeply sequential logic since sufficient structural fault coverage for them requires special design for testability means, e.g. [HeWu94].

4.2.3 Module-level Test Generation

High fault coverage at the module-level can be achieved with utilization of structural test patterns or deterministic algorithms. As to the functional components of the ALU or the shifter, it is suitable to test them with ATPG patterns, since commercial tools offer effective test sets for the combinational logic that can be directly parsed into corresponding instructions.

Suppose an ALU with the structure shown in Figure 4-3. At each time, one of the 8 operations is chosen through the 3-bit control lines (OP) and performed on the two 32-bit operands (A and B). Besides the computational result (S), a status signal (Z) which is set ‘1’ to indicate that the input data at A is zero.

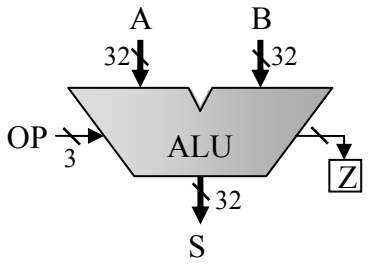


Figure 4-3: An example: a 32-bit ALU

Giving the net-list to a commercial ATPG tool, we get a test set: each pattern is 67-bit wide and conforms to the format that the first 64 bits correspond to the operands at A and B, and the last 3 bits specify the ALU operation. That is to say, if we activate the specified operation with the desired operands, the exact test pattern will be applied to the ALU. Then, the remaining problem is to observe the test response, which is distributed in the computational output S and the status output Z. Unlike the values in S that are directly accessible in the register file, the result signal after test application at Z can be indirectly observed with use of the conditional branch instruction "JZ".

Figure 4-4 explains the detailed steps how to come up with the code that is essential to deliver the desired pattern in MIPS-like assembly instructions.

Step 1: according to the format, annotate the test pattern to identify the necessary operands and the operation.
In this case, application of the exact test pattern requires the data of 0x712e0a5b and 0xbb2c0211 be assigned at the ALU inputs and at the same time the operation of "addition" be utilized.

Step 2: under consideration of the ISA, determine the candidate instructions that correspond to the required operation.
It is quite common that more than one instruction can trigger the required operation in the target module. For example, as to MIPS, {ADDI, ADDIU, ADD, ADDU} can activate the operation of addition in the ALU. At this point, each of them is capable of ALU testing. However, if we consider further their syntactic usages, some of them may not be appropriate. According to the MIPS syntax, ADDI and ADDIU should have a 16-bit immediate as one of the inputs, whereas our two operands have to be 32-bit. Thus, only ADD or ADDU is adequate in this context. As both of them are equivalent from the viewpoint of test, the

decision on which instruction to be used finally can be based on other criteria, such as the metric of power consumption. That is, if ADDU is more power efficient as ADD, then use ADD to realize the ALU test.

Step 3: identify the essential code for pattern application

Based on the previous account, we use two consecutive load instructions “li” (load 32-bit immediate) to prepare the operands ready in the registers “$r1” and “$r2”. Then use the power-efficient operation “ADDU” instead of “ADD” to apply the exact test pattern. In the end, the computational output is available in “$r3”.

Step 4: observe the test response

To observe the test response regarding the computational result, we can propagate the value stored in the register cell “$r3” to the data bus via the store instruction “sw”.

Note that the status signal Z is not related with the operation “ADDU” performed by the ALU. Instead, it is related with the value at the specific input A, in this case “$r1”. Thus, although we cannot capture Z as soon as test application (ADDU $r3, $r1, $r2) finishes, the use of the conditional jump instruction (“jz $r1”) will cause the same signal to appear at Z again. With creation of different memory images (either 16-bit 0’s or 1’s) via “ADDIU”, we can later evaluate the test response at Z in such an indirect way.

From the above description, we split the code used for implementation of the ALU test basically into the following sections:

1) loading the operands;
2) test application;
3) test response observation

This structure is quite common in delivery of all ALU test patterns, and thus can be standardized with introducing parametric fields into the instructions. All these parameters, or settable fields, are dependent on test patterns to be delivered. This is just the idea of “templates”, and we will cover this aspect in the later section.

An ALU test pattern:

01110001001011100000101001011011 10111011001011000000001000010001 001

Step 1: Annotation according to the pattern format

01110001001011100000101001011011 10111011001011000000001000010001 001

Operand A: (0x712e0a5b) *Operand B*: (0xbb2c0211) *Operator:* +

Step 2: Mapping the operator into assembly instruction(s) according to the instruction specification

{*ADD*, *ADDU*, ADDI, ADDIU}

Step 3: Essential code for pattern application

```
li      $r1, 0x712e0a5b
li      $r2, 0xbb2c0211
addu    $r3, $r1, $r2
```

Loading section

Application section

Step 4: Code for test observation

```
        sw      $r3, offset(base)
        addiu   $r4, $r0, $r0
        jz      $r1, status
        addiu   $r4, $r0, 0xffff
status:
        sw      $r4, offset'(base)
```

Computational result

Status result

Figure 4-4: An example: ALU test based on ATPG patterns

Pattern to test the module of the program counter (PC) are viewed at the instruction level as program addresses, and can be applied with use of "jump" instructions. Figure 4-5 shows a piece of code to test the PC with the user-specified pattern. Just as the ALU, structurally the test code consists of three parts, for loading, application of the desired pattern and observation of the response respectively. Note that 1) the directive ".org" is used to allocate a code segment to a specified memory location; 2) the instruction "la" is to load the address for the specified label. As to this example, 0xf0f0 is basically a test pattern for the PC which is firstly loaded to "$r1" and then applied to the program counter by the "jr" instruction. There are some critical issues to be considered during generation of "valid" yet "effective" test patterns for the PC:

- Using the program addressable memory without causing any conflict in code segments.
- Ranging over the usable memory in a balanced manner in order to achieve a satisfactory fault coverage level, i.e. both the upper and lower memory locations need to be equally addressed.

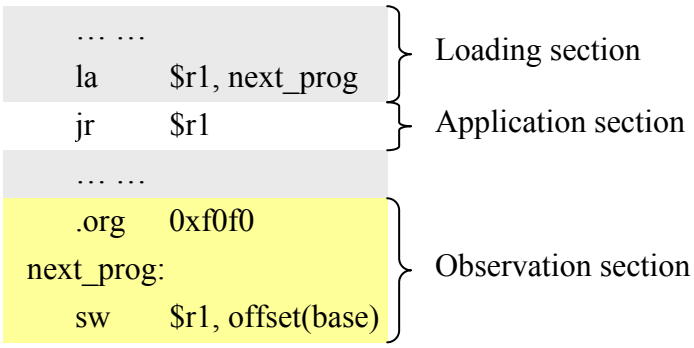


Figure 4-5: An example: PC test with a user-specified pattern

Deterministic test algorithms, for instance the modified algorithmic test sequence (MATS), exhibit their effectiveness in testing of stuck-at faults of memories. However, due to the structural regularity and similarity of the register file to the memory, they are also able to achieve high fault coverage for the register file. Thus, testing of this component can be based on deterministic algorithms. Figure 4-6 illustrates the essential steps of the MATS test and implementation of this algorithm in assembly code corresponds to the test program for the register file.

1)	Set all the registers to 0;
2)	R_0 = <01…01>;
3)	Execute $R_i = R_i + R_0$, where i is from 1 to (n − 1);
4)	Execute $R_i = R_i + R_i$, where i is from (n − 1) to 0.

Figure 4-6: MATS for the register file test

To implement concretely the above algorithm, specific properties of the target processor have to be considered. Taking MIPS for an example, as $r0 is hardwired to zero, the traverse through all register cells is only possible from $r1 to $r31. Additionally, some registers are usually used for special purposes, like $r31 for storage of the return address ($ra). Thus, to ensure a normal program execution, important register statuses should be banked up before

test starts. The exemplary test code with flat register accesses for MIPS-similar processors is given in Figure 4-7.

```
… …                                  // bank-up code for special registers: $ra ($r31), $sp ($r29), …
// step 1: initialize $r1 - $r31 to zero
add    $r1,    $r0,       $r0
… …
add    $r31,   $r0,       $r0
// step 2: ascending access
sw     $r1,    0($r0)              // save the initial status of $r1 at the address 0x0
li     $r1,    0x555555555         // load $r1 with the value 0x55555555
sw     $r2,    4($r0)              // save the initial status of $r2 at the address 0x4
add    $r2,    $r2,       $r1      // load $r2 with the value 0x55555555
… …
sw     $r31,   120($0)             // save the initial status of $r31 at the address 0x78
add    $r31,   $r31,      $r1      // load $r31 with the value 0x55555555
// step 3: descending access
sw     $r31,   120($r0)            // save the status of $r31 at the address 0x78
add    $r31,   $r31,      $r31     // load $r31 with the value 0xaaaaaaaa
sw     $r31,   120($r0)            // save the status of $r31 at the address 0x78
… …
sw     $r1,    0($r0)              // save the status of $r1 at the address 0x0
add    $r1,    $r1,       $r1      // load $r1 with the value 0xaaaaaaaa
sw     $r1,    0($r0)              // save the status of $r1 at the address 0x0
```

Figure 4-7: Exemplary implementation code: the MATS(++) for register files

4.2.4 Processor-Level Test Generation

In short, the primary task within this phase is to automate the process for test program generation of the target components. An interesting work in this direction was presented in [Chen03], which relies on the so-called templates to solve the automation problem. By definition, a template is a piece of code to represent a common structure for a module-level test and some instruction fields of it are left as settable on purpose. The final test programs are

then generated with template instantiation. In order to ensure the quality of templates, their work requires a large amount of efforts in generation of random values for the settable fields and template evaluation. In contrast, we assign values in a deterministic way, i.e. either dependent on test patterns or in line with the criteria to improve test quality, so that the space exploration efforts can be avoided.

It is possible to construct an individual template for each of target modules. Based on the specific code shown for the ALU and the program counter previously, we can extract in accordance the common structures in Figure 4-8 used to test them respectively. The settable fields are marked in brackets, and their values are determined in the following ways:

- Related with patterns to be delivered, e.g. [A], [B] and [OP] for the ALU and [val_pc] for the PC: parsing directly from the test patterns;
- Related with register usage, e.g. [f, x - y]: to achieve high coverage fault of the register file during the ALU and PC test, we should allocate available registers in a uniform way. To achieve this, we associate a counter with each register, and increment it once this particular register is allocated. Then, our goal is to keep all the counters growing an equivalent pace during test generation.
- Related with program labels and memory references, e.g. [status], [mem_loc] and [lab_pc]: the syntax requires the labels unique within the program scope. Moreover, if test responses should always be available in the memory, unique addresses have to be assigned.

```
    li      $r[x], [A]
    li      $r[y], [B]
    [op]    $r[z], $r[x], $r[y]
    sw      $r[z], ([mem_loc])
    addiu   $r[f], 0
    jz      $r[x], [status]
    addiu   $r[f], 0xffff
[status]:
    sw      $r[f], ([mem_loc'])
```

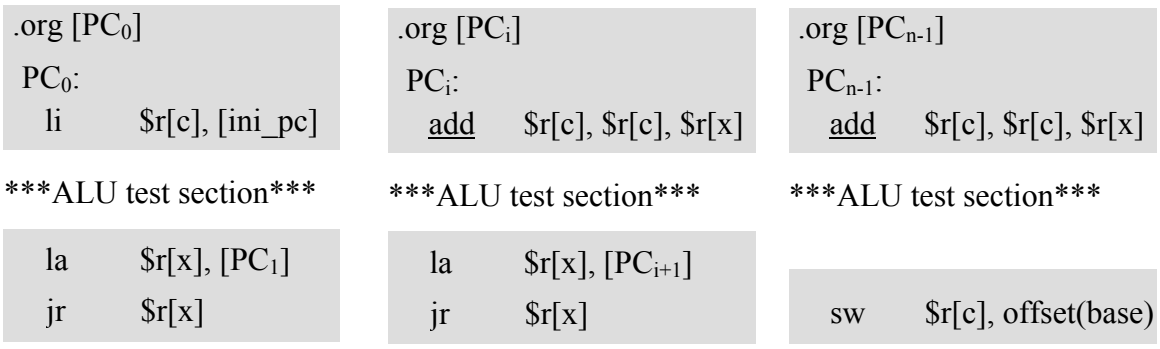
a) ALU template

```
    la      $r[x], [lab_pc]
    jr      $r[x]
    .org    [val_pc]
[lab_pc]:
    sw      $r[x], ([mem_loc])
```

b) PC template

Figure 4-8: Templates for individual module test

As an alternative, it is also possible to combine tests for several modules into a single template. For example, we can integrate the shown ALU and PC tests together. In order to reduce memory accesses, the PC responses are accumulated [BaPa99], that is, instead of storing directly the responses, they are accumulated with an operation, say addition, so that we only save once in the end the summed value. As a result, the chained test mechanism causes slight variances for the templates whose instances are at the beginning, in the middle and at the end of the test program. Accordingly, we call them the head, the intermediate and the tail template respectively, and illustrate their structures in Figure 4-9. As to the head template, an initial value (ini_pc) is firstly loaded into a register (r[c]) especially for the PC test. Then an ALU pattern is applied using the code conforming to the structure in Figure 4-9 a). The last two instructions apply the next pattern to the PC. An intermediate template bears the similar structure as that of the head, only replacing the first instruction with the one, in our case "add" underlined, for the accumulation purpose. Eventually, the accumulated response is store in the instance of a tail template.



a) the head template b) the intermediate template c) the tail template

Figure 4-9: Templates for integrated tests

With availability of the constructed templates, we can automate the process of test program generation. Specifically speaking, for a target module or a group of modules, we select correspondingly the suitable template, and determine each of its parametric fields under considerations of test patterns to be delivered or other criteria related with test quality, e.g. equal register allocation.

4.3 Test-oriented Optimization for Low Memory, Time and Power

In short, our basic interest is to ensure that no penalty of obtained fault coverage will occur during each step towards our optimization objectives, namely low memory usage, short test application and low power consumption. Although there are some existing literatures addressing software optimization for the mentioned goals, they are not applicable to the software-based self-test context, since test quality may possibly suffer. For this reason, we propose special optimization means that are suitable for test programs and integrate them into the program generation flow.

4.3.1 Optimization for Low Memory and Time

Test program optimization for low memory has a significant meaning especially when the software-based self-test scheme is implemented for embedded cores that run with restricted memory requirements. Meanwhile, it is also important to reduce test application time due to the following reasons. First, as to in-field testing, test programs are equipped in the target embedded system and compete with other routines for relatively limited resources like computation power. To avoid impacts on the normal workload, it is necessary to optimize test application time. Moreover, reduction of test time has another meaning in energy consumption which is critical for the systems operating autonomously under stringent energy budgets.

As mentioned in 3.4.3.2, compaction is a standard and effective means to reduce the sizes of test sets, which can benefit our goals as well. Our work explores the aspect that ATPG brings redundancy in test vectors, and eliminates accordingly the redundancy under consideration of the power metric, i.e. bit correlations among consecutive vectors. The heuristic proposed here belongs to the category of static compaction, namely the post-processing of the test set generated by ATPG.

We introduce the notations and symbols used in the proposed compaction algorithm as follows:

- TS: the original ATPG test set;
- TS': the compacted test set
- FS: the fault universe under consideration
- FS_{obj}: faults detected by "obj", which can be a test set (e.g. FS_{ts}) or a test pattern (FS_p).

- Hamming distance between two l-bit test vectors: $\text{Hamming}(p_s, p_t) = \sum_{i=0}^{l-1} (p_s^i \oplus p_t^i)$

The compaction heuristic is presented in Figure 4-10 and its computational complexity is O(n). With use of the compaction technique, we can achieve test data reduction and potential energy and power savings without loss of module-level fault coverage.

INPUTS:
- TS
- FS

OUTPUT:
- TS'

BEGIN

Initialization: $TS' = \phi,\ FS_{TS'} = \phi,\ p_{last} = p_{cur} = null$

Fault simulation: $\forall p \in TS,$ identify FS_p

While $FS_{TS'} \subset FS$ do
- Create the set $TMP = \{p \mid (p \in TS) \wedge (\mid FS_p \mid = \max)\}$
- If $(\mid TMP \mid = 1)$ then $p_{cur} = p, where\ p \in TMP$
- Elseif $(\mid TMP \mid > 1)$ then
  $p_{cur} = p,\ where\ ((p \in TMP)\ \wedge\ \min(\text{Hamming}(p_{last}, p)))$
- Update:
  - $TS' = TS' + \{p\}$
  - $FS_{TS'} = FS_{TS'} + \{f \mid (f \in FS_p) \wedge (f \notin FS_{TS'})\}$
  - $TS = TS - \{p\}$
  - $\forall p \in TS,\ FS_p = FS_p - \{f \mid (f \in FS_p) \wedge (f \in FS_{TS'})\}$
  - $p_{last} = p_{cur}$

END

Figure 4-10: Power-aware test pattern compaction

4.3.2 Optimization for Low Power

We take into account the researches on software optimization for low power, and contribute in the aspect that the new metric of test quality is introduced during the optimization process. In contrast to the conventional goal for software optimization focusing on program semantics, our attention is to ensure the same test quality. The following sub sections present our work in details.

4.3.2.1 Instruction Level Power Model

During optimization, it is necessary for us to compute the power consumption. For such purpose, in principle we have two options: simulation-integrated or model-based. The former uses a gate-level power analysis tool for controlling program optimization and is accordingly computationally expensive. As an alternative, through simulation of a set of systematically constructed routines, the second method builds a reference model for a target instruction set architecture in advance. Once created, the model is reusable to evaluate power consumption of a variety of programs with the same ISA. However, the accuracy of the built model influences to a large extent the real power savings.

Our work adopts the second method to build beforehand the reference model at the instruction level that specially describes power consumption due to switch of instruction contexts. The basic idea of our method is to concentrate firstly on a few of modules that change their circuit states for an instruction pair, and use Hamming Distances (HD) as a measurement characterizing their transitions. Then the relationship between these module-level transitions and average power consumption at the processor level is represented with statistical analysis based on a variety of samples from gate-level simulation. Thus, we get a much better estimation of dynamic power dissipation due to inter-instruction effects than by merely considering transitions in registers at the program model level [Lee00].

The switching activities of the instruction register (IR), the register files (RF) and the ALU are easily predictable. For example, transitions T_{IR} at the output of IR are related to instruction coding, and those for the RF, T_{RF}, and the ALU, T_{ALU}, depends on runtime data. For an n-bit instruction word, T_{IR} is:

$$T_{IR}(i,j) = HD(C_i, C_j) = \sum_{m=0}^{n-1} (C_i^m \oplus C_j^m) \qquad \text{Equation 4.1}$$

where C_i and C_j are binary code for instruction i and j respectively.

For a RF with two outputs, Q0 and Q1, let $Q0_i$ and $Q1_i$ be the values of output registers of instruction i, $Q0_{i\rightarrow j}$ and $Q1_{i\rightarrow j}$ be the temporary values due to (i, j). Then we model T_{RF} as:

$$T_{RF}(i,j) = HD(Q0_i, Q0_{i\to j}) + HD(Q0_{i\to j}, Q0_j) + HD(Q1_i, Q1_{i\to j}) + HD(Q1_{i\to j}, Q1_j)$$ Equation 4.2

Transitions of the ALU depend not only on the inputs (A, B, OP) but also the function it operates. For this reason, we consider transitions both at the inputs and the output (Q):

$$T_{ALU}(i,j) = HD(A_i, A_j) + HD(B_i, B_j) + HD(OP_i, OP_j) + HD(Q_i, Q_j)$$ Equation 4.3

Next, to find relations between average power consumption A(i, j) due to inter instruction effects for a given instruction pair (i, j) and the above depicted factors, we apply *regression analysis*, which is a standard statistical method to investigate relationships between *dependent* and *independent variables* [SeLe03]. In our case, the dependent variable is A(i, j), and the independent variables are T_{IR}, T_{RF}, and T_{ALU}, that is:

$$A(i,j) = f(T_{IR}, T_{RF}, T_{ALU}) + \varepsilon$$ Equation 4.4

ε represents approximation discrepancy. If a linear relationship is assumed, we further specify the above formula as:

$$A(i,j) = \beta_0 + \beta_1 \cdot T_{IR}(i,j) + \beta_2 \cdot T_{RF}(i,j) + \beta_3 \cdot T_{ALU}(i,j) + \varepsilon$$ Equation 4.5

β_0, …, β_3 are coefficients to be determined by the process of multiple regression analysis based on samples. A sample is an observation of the dependent and the independent variables. Equation 4.1 to 4.3 facilitate calculation for the values of independent variables. According to [Tiwa96b][Schu02], we explore Equation 4.6 to obtain values for the dependent variable, where A_{i+j} stands for the measured average power consumption of the loop (i, j), while A_{Base_i} and A_{Base_j} are the basic average power when instruction i or j executes stand-alone:

$$A(i,j) = A_{i+j} - \frac{1}{2}(A_{Base_i} + A_{Base_j})$$ Equation 4.6

Finally, we follow the steps below to concretely set up the reference power model:

- Simulate programs consisting loops of individual instruction to generate basic power costs A_{Base_i};
- Simulate programs consisting loops of instruction pairs to get values for power costs A_{i+j};
- Create samples with Equation 4.1, 4.2, 4.3 and 4.6;
- Perform multiple regression analysis to determine coefficients β_0, …, β_3.

With the transitions captured during gate level simulation, the model offers closer estimation than the bit-transition at the program level. A comprehensive statistical analysis for validating the model and parameters was done in [Schu02].

4.3.2.2 Optimization through Instruction Scheduling

Many publications already show that power consumption due to circuit state overhead can be minimized through instruction scheduling. However, when we consider the test context, these methods may not be appropriate. The code below exhibits the difference:

```
… …
1)    li     $r1,    512
2)    li     $r2,    65535
3)    add    $r3,    $r2,    $r1
4)    sw     $r3,    (result_0)
5)    li     $r3,    324
6)    li     $r4,    790
7)    sub    $r5,    $r3,    $r4
8)    sw     $r5,    (result_1)
… …
```

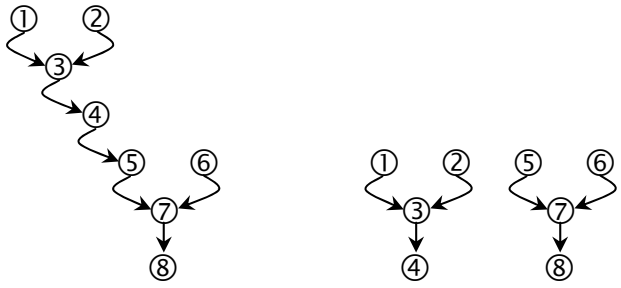
The dependency graph (DG) of the shown code in Figure 4-11 a) displays the correlations imposed by program semantics which are supposed to be maintained for the conventional optimization based on instruction scheduling. Since most of the order is fixed, there are only few degrees of freedom for reordering, for example exchanging instruction 1 and 2, and putting 6 somewhere before 7.

However, viewing the same code as part of the ALU test, we divide instructions into units, where a pattern is loaded, applied and relevant responses are stored. Such a unit, called a test

behavior, forms an atomic test for the ALU module. As to the above code, two test behaviors are extracted and shown in Figure 4-11 b). We notice here that fewer restrictions exist so that it is possible to reorder instructions:

– Inside the test behavior: e.g. changing the orders for 1 and 2 as to the first unit, or 5 and 6 for the second one;

– At the test behavior level: that is to say, to switch the order of test behaviors. For instance, from the test standpoint, it makes no difference if the set of instructions 1 to 4 are executed before or after the set of instruction 5 to 8. However, the freedom to change the orders among test behaviors may lead to minimization of transitions. Note that to avoid corruption of the test patterns or results, we should ensure that reordering does not take place across the borders of test behaviors.



a) DG b) Test behaviors

Figure 4-11: Program-oriented vs. test-oriented instruction scheduling

The test-oriented instruction scheduling algorithm is presented in Figure 4-12. As inputs, the original test program, the template used to construct the program and the reference model to predict power consumption for a given instruction pair should be available. The algorithm starts with extraction of test behaviors, basically a set of rules for instruction scheduling, in line with the template structure under consideration. Then we repeat the following steps until all the instructions get scheduled:

INPUTS:
- Instructions of a test program to be optimized (IS_{ori});
- Applied template structure;
- Reference power model (Power);

OUTPUT:
- Rescheduled instructions (IS_{opt})

BEGIN

Extract constraints (C);
While $IS_{ori} \neq \varnothing$
- get the last scheduled instruction j:
  $j = getLastElement(IS_{opt})$;
- determine the set of instructions ready to be scheduled:
  $readySet = getListOfInstructionsToBeScheduled(IS_{ori}, C, j)$;
- for each instruction in *readySet*, calculate the cost of power consumption with respect to j:
  $\forall i \in readySet,\ \text{cost}_i = Power(i, j)$;
- schedule the instruction with the minimal cost:
  $IS_{opt} = IS_{opt} + \{i\}$, where cost_i is minimal;
  $IS_{ori} = IS_{ori} - \{i\}$;
  update C

End while

END

Figure 4-12: Test-oriented instruction scheduling algorithm

1) Extract constraints (C): given the structure of the template, we split the original code into test behaviors (TBs). Thus, for a program applying the number of n test patterns to the ALU, then C = $\{TB_i \mid 1 \leq i \leq n\}$. Next, inside a test behavior, we consider further constraints among the instructions. Suppose instruction i should execute after instruction j, to represent this relationship, we use the clause (i: j).

   Therefore as to the code in Figure 4-11 b), namely IS_{ori} = {1, 2, 3, 4, 5, 6, 7, 8}, the corresponding constraint C is finally as follows:

   C = { TB1, TB2}

   $TB_1 = \{(1:null),(2:null),(3:1 \wedge 2),(4:3)\}$

   $TB_2 = \{(5:null),(6:null),(7:5 \wedge 6),(8:7)\}$

2) *getLastElement*: returns the instruction that is scheduled and inserted in IS_{opt} for the last time. When IS_{opt} is empty, a default instruction with binary code of all 0's is given.

3) *getListOfInstructionsToBeScheduled*: for the last scheduled instruction j,
   - if $(j \in TB_x) \wedge (TB_x \subseteq C)$, then
     readySet = {i | $(i \in TB_x) \wedge$ (i is unscheduled while the instructions appear on the right clause of i are scheduled)}
   - if $\forall TB_x$, where $TB_x \subseteq C$, $j \notin TB_x$, then
     readySet = {i | i is an instruction of any TB which is still in C and i satisfies: 1) i is unscheduled and 2) all instructions appear on its right clause are scheduled}

   According to the above rules, as an example, suppose the last scheduled instruction is ∅, then readySet = {1, 2, 5, 6}. However, if j is 5, which means that the last scheduled instruction is part of TB_2, then we can only select instructions from TB_2 for scheduling in order to avoid potential test corruption. Thus in this case, readySet = {6}.

4) *Power(i, j)*: computes the costs for each instruction i in readySet with respect to the last scheduled instruction j.

5) *Schedule the instruction with minimal cost and do necessary updates.*
   Scheduling is performed by moving the target instruction i from IS_{ori} to IS_{opt}. Afterwards, it is necessary to update the constraints defined in C. Do the update as follows:

   - Change: status (i) ← "scheduled"
   - For TB, which contains i, namely: $i \in TB$:
     If all of its instructions are "scheduled", then C = C – {TB}

4.3.2.3 Optimization through Don't-Care Bit Specification

Many publications use don't-care bits to reduce transitions in test patterns [SOT00]. A similar idea is explored in our work based on the existence of unused instruction bit fields. Figure 4-13 exemplifies a specification for an Add operation in SPARC v8 [SPARCv8].

Assembly Language Syntax			
add	reg_{rs1},	reg_or_imm,	reg_{rd}

Format 1:

10	rd	000000	rs1	i=0	unused(zero)	rs2
31 29		24	18	13	12 4	0

Format 2:

10	rd	000000	rs1	i=1	imm
31 29		24	18	13	12 0

Figure 4-13: An example of unused instruction bits: Add in SPARC v8

According to its format specification, Add computes "r[rs1] + r[rs2]" if the 13th instruction bit is 0, otherwise an immediate is used whose value is carried in the 12 least significant bits, and "r[rd]" holds the result. In contrast to Format 2 where each instruction bit has a particular meaning, there are 8 bits in Format 1 starting from Bit 5 to Bit 12 defined as "unused" and by default assigned zeros. However, in some cases it is more power efficient, if other values are assigned to those bits, and this is the primary motivation of our second optimization. On the other side, as all of those unused bits do not carry any test-related information, the target program will not suffer from a fault coverage loss.

So, given the ISA, we are able to identify don't-cares (DCs) of each instruction under consideration. Then for an instruction (i, j), where i is fully specified, we annotate DCs in a way that switching activities due to (i, j) are minimized. The reference model built previously is used once again during DC specification. The detailed algorithm is illustrated in Figure 4-14.

INPUT:
- Instructions of a test program to be optimized (IS_{ori});
- Reference power model (Power);
- Instruction set specification (ISS)

OUTPUT:
- Optimized instructions (IS_{opt})

BEGIN

While $IS_{ori} \neq \varnothing$
- get the present instruction to be specified and the last optimized instruction:

  $i = getFirstElement(IS_{ori})$

  $j = getLastElement(IS_{opt})$;
- identify don't-care bits in i with reference to the ISS:

  $DCs = ISS(i)$;
- specify DCs to minimize power consumption due to (i, j)
- update the sets of un-optimized and optimized code:

  $IS_{opt} = IS_{opt} + \{i\}$;

  $IS_{ori} = IS_{ori} - \{i\}$

End while

END

Figure 4-14: Optimization algorithm based on unused bit specification

4.4 Conclusion

The software-based self-test approaches are attractive for programmable cores due to advantages of minimal area overhead and the ability of at-speed testing. The major problems of them centralize the engineering efforts spent in developing test programs with high structural fault coverage. Moreover, in order to be adequate for embedded systems, multiple facets of generated test programs such as memory usage, execution time, energy and power consumption, should be optimized. Our work firstly addresses the automation aspect by presenting a method to synthesize programs from module-level structural tests. Then to meet the aforementioned constraints of embedded applications, we propose optimization algorithms which can either be integrated into or performed as a step after the program generation process. Memory usage, execution time and energy consumption can be reduced with our compaction method, while power dissipation due to inter instruction effects is minimized through instruction scheduling and unused bit specification.

The key contribution of our work lies in that for the first time, we combine the two directions of the structural SBST and software optimization, and propose a novel method for test program synthesis which tackles fault coverage, memory requirements, test lengths and power consumption at the same time.

Chapter 5 Software-based Self-test for Application Specific Instruction Set Processors

5.1 Application Specific Instruction Set Processors

Multimedia applications such as video processing often spend plenty of computational time executing a small fraction of program code. As a result, it is quite significant to speed up the execution of these "hot-spots" so as to achieve a better system performance. In addition to the performance improvement, the demand for ultra low power consumption in embedded systems exists as another strong motivation that makes application-specific design optimizations a necessity. Conventionally, either general-purpose processors or dedicated hardware, i.e. application specific integrated circuits (ASICs), can be used to solve the domain-relevant problems, each of which has particular properties regarding efficiency, flexibility and cost.

In comparison with the aforementioned two solutions, the recently emerging design of application specific instruction set processors (ASIPs) offers a better tradeoff between performance and flexibility. Much freedom is left to the designer to customize the processor architecture in a way that the design goals, e.g. high performance or low power dissipation, are fulfilled. Meanwhile, in contrast to ASICs, programmability of ASIPs ensures that the cores are much more flexible. The main applications of ASIPs are found so far in micro-controllers or System-on-a chip designs for multimedia processing.

Basically, designers can tailor an ASIP to a dedicated domain through two means, namely configuration and extension. Configuration is usually the process to:

1) Determine whether to present particular blocks in the target system through block inclusion or exclusion. The common example for the optional modules can be floating point units, coprocessors, etc.
2) Specify appropriate attributes for predefined modules, such as the width of system buses, the cache architecture, the width of multipliers and so on.

Another design freedom is ensured with the system extensibility. That is, it is possible for an ASIP designer to extend with ease the basic core functionalities through implementation of user-defined logic. All of the customization decisions are made on the basis of analysis of the target application. Figure 5-1 illustrates a typical design flow of an ASIP.

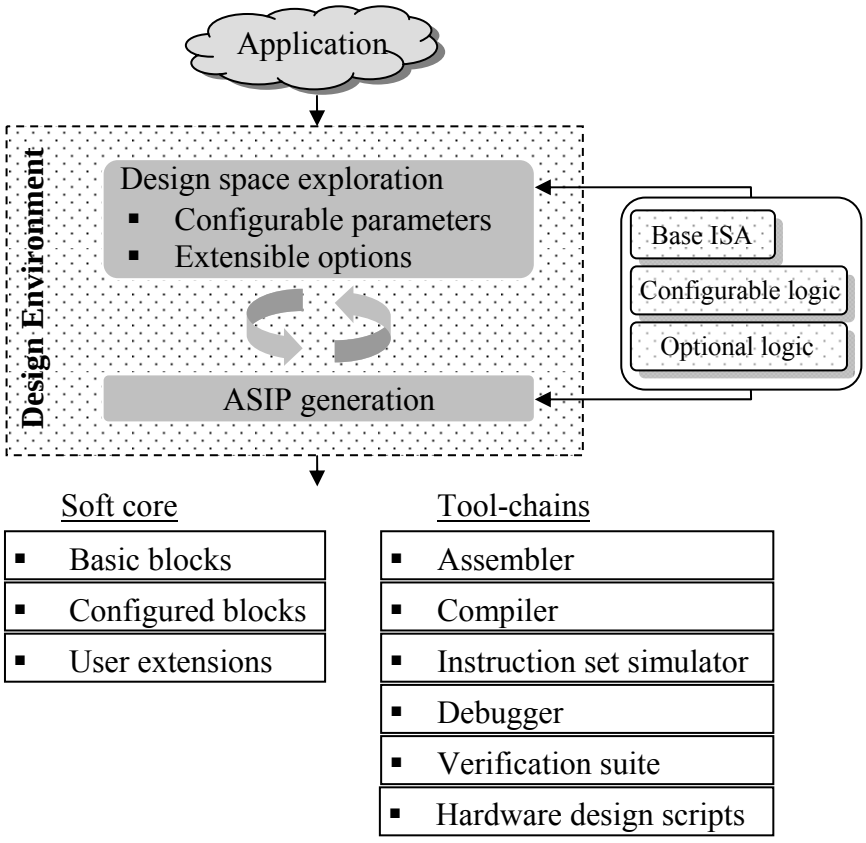


Figure 5-1: A typical design flow of an ASIP

ASIPs are commonly equipped with two resources: 1) an abundant logic library and 2) a design environment. The logic library, which includes a core with the base ISA, configurable blocks and optional modules, provides the starting point as well as implementation options during design exploration. On the other side, the design environment targets mainly the automation issue during design exploration and ASIP generation with reference to the existing library. As Figure 5-1 shows, the target application, available possibly in the form of a set of executable programs, is analyzed under the design framework. Then, the application-specific

characteristics are profiled and translated into design choices regarding the preferred configurable parameters, inclusion/exclusion of logic blocks, and the necessary extensions. All of these decisions are later used in the automatic ASIP generation, which outputs the final design as a soft core together with a tool-chain for software development and logic synthesis.

The major research emphasis for ASIPs till now concerns the aspect of automation for design exploration and core generation so as to meet the critical requirement of short time-to-market. Unfortunately, there are comparably few efforts spent on ASIP testing and in fact, this new design paradigm of configurable and extensible cores challenges the conventional test in the following ways. First, the design flexibility possibly results in a variety of ASIP cores that conform to a similar architecture but differ only in some minor blocks. Due to the similarity in structure of ASIP cores, it is preferred to reuse tests as many as possible rather than starting from scratch every time when test is considered. Second, as time-to-market and the price are crucial factors, test automation and cost reduction is inevitably necessary to ensure delivery of high-qualified ASIPs within a short time span and acceptable development budgets. The software-based self-test methodology is an effective scheme for programmable core testing with attractive features of reusability and low cost. Thus, it is natural for us to combine these two directions and address the following issues for ASIP testing.

- Test generation for ASIPs. As to the conventional SBST, the test generation for a processor core with a fixed structure mainly concentrates on effectively implementation of programs based on structural test patterns at the module-level. However, in account of timing issues for ASIP development, we consider here the feasibility to start test generation right after design configuration is done. That means, additionally, we have to tackle with issues regarding testing of blocks whose target structures are partially known, e.g. configurable modules, or only known in form of “extended” instructions. To make test generation a flexible and configurable process, the key point here is to parameterize properly the module-level tests and integrate them in a hierarchical way. Unlike the focus on functional verification using code coverage or statement coverage as metrics in [Rimo06], our attention is to generate software programs for structural faults.

- Test improvement through instruction extension. Lai firstly implemented an instruction-level DFT scheme that adds extra logic in the form of new instructions to enhance quality of self-test programs [LaCh01]. The process presented in the paper to incorporate the test-

related new instructions is manual-intensive and requires expertise knowledge on the target processor architecture. In fact, the idea itself can be viewed as a specific application of ASIP in the test domain. That is, new instructions are introduced to improve test quality contrarily to the primary metrics on performance and power consumption. Our point here is to propose a general framework based on ASIPs and instruction-level DFT which identifies the hard-to-test modules and adds extra instructions to improve the testability.

The following sections of this chapter will elaborate out work in the above-mentioned directions respectively.

5.2 Test Generation for ASIPs

We can interpret the task considered as follows: with availability of the target ASIP configuration, generate correspondingly the self-test programs that can effectively detect structural faults. At first, we classified the ASIP blocks into the following groups:

1) Base-core components that support a basic instruction set. Their existences are very important that ease largely the efforts in design exploration for different solutions and tradeoffs between hardware and software modules;

2) Optional blocks that can be excluded from or included into a particular processor instance;

3) Configurable blocks that can be adapted to the target application through specification of their implementation parameters, for instances, the width of system buses, the number of available general-purpose registers;

4) Optional/configurable blocks which are hybrid of the above two classes. An example can be an optional multiplier with a user-specifiable word width;

5) User-defined logic. A straightforward example can be a logic used to speed up execution of performance-critical code. As to programmers, such logic usually exists in the form of (an) extra instruction(s).

Thus, for a typical ASIP, it contains the blocks of group 1 and 3 (with fully specified design parameters), whereas the presence of modules in group 2, 4, and 5 is optional. Accordingly, from the standpoint of ASIP testing, we need to consider basically three test cases for:

1) Base core, whose structure is known beforehand and presents the same for all core variants;
2) Configurable blocks
3) Optional or user defined blocks.

The existing SBST can be applied to test the base core. The major difficulties come from the other two classes, which vary the structures of processor instances dependent on customization. Below, we exemplify two customization processes, one for a configurable module and the other for an optional logic, and analyze respectively their influences in processor structures.

Example 1: Customization through configuring the number of general-purpose registers

Assume an ASIP with a configurable register file, whose size range from 2 to 32. To meet the requirements of the target application domain, this number is configured to 16 which means that it is valid for a programmer to use R0 ~ R15 in his routine for data proceeding. References to other registers (R17 ~ R31) in assembly programs may not invoke compilation errors, but will eventually activate the unit for exception handling.

In this case, the customization finally results in a processor, which has:

i. The logic to control the register usages, which generates appropriate control signals for the register file if the reference is valid, otherwise trigger the exception handling unit;
ii. The register file with the specified size, in this case 16, in the data-path;

The customization is transparent to an assembly programmer.

Accordingly, testing of this customization is split into two contents: testing of the configured register file in the data-path and testing of the relevant control logic. The method introduced in Chapter 4 can be used to test the register file. Moreover, if the test can be properly parameterized in accordance with the configurable structure of the register file, then we can automatically generate the test program based on the customization specification. On the

other side, as to testing of the control logic, particularly the logic in exception handling unit, the method proposed in [Hatz05] can be used, which develops test code on purpose to activate the unit and then read back the responses from it.

Example 2: Customization through inclusion of an optional unit (a multiplier)

Inclusion of the optional multiplier means to add it into the data-path, and at the same time, modify the control logic so that corresponding signals to control its operations are generated. Unlike the first example, the customization of this kind is not transparent to the programmer, as multiplication instructions are now provided and can be used to trigger functionalities of the multiplier. This means, the instruction set is extended and the compiler is accordingly modified to be aware of the extensions.

In general, functional extensions, either through inclusion of existing optional units or implementation of specific user-defined logic, require modifications of both hardware and software that address:

i. Changes in the control logic;
ii. Changes in the data-path to incorporate the new computational block;
iii. Changes of the instruction set and the compiler.

So, we also split the tests for a "new" functional block into two aspects: detection of faults in the data-path can be mainly based on deterministic test patterns with utilization of its relevant instructions, e.g. multiplication or division as for the multiplier, while to detect faults in the control logic, we use the code to create purposely control and data hazards [BhWa01].

The overall test generation flow is outlined in Figure 5-2. After the design phase, we start generation of test programs based on the information regarding processor specifications and the present customizations. As to the modules that constitute the base core, conventional SBST approaches are applicable. Then, algorithms are parameterized such that test programs for configurable modules can be automatically generated in relation to the current settings. For designs with optional blocks or user-defined logic, we adopt a verification method to test the related hardware in control logic, and deliver deterministic test patterns with use of the relevant instructions to detect faults in the data-path.

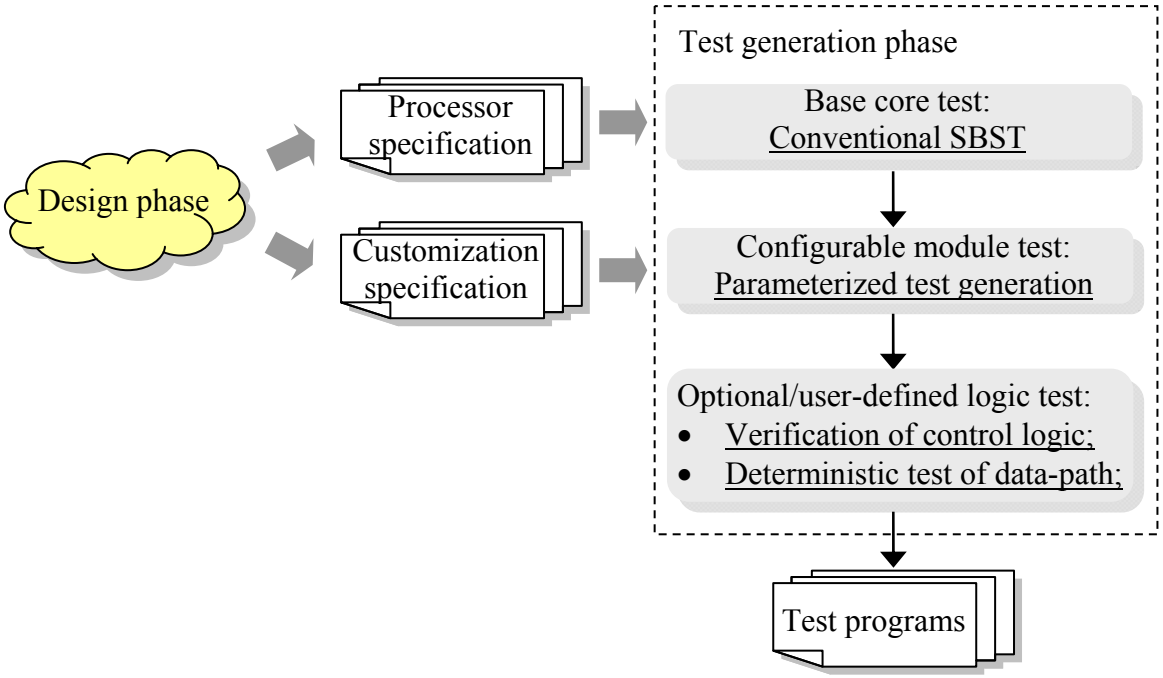


Figure 5-2: Test generation flow for ASIPs

5.2.1 Base Core Test

For the control logic of the base core, any customization directly translates into modification or extension of this unit. However, as to base-core components in the data-path, they have fixed structures and are not susceptible to design changes related with customization. This implies that we can reuse deterministic test patterns as well as relevant test programs to test the basic components for various processor instances. At this point, the conventional SBST approaches, e.g. our method presented in Chapter 4, are adequate.

5.2.2 Parameterized Test for Configurable Units

Parameterization is one of the key techniques to enable configurability. With specification of appropriate values of relevant parameters, the designer can easily configure and generate the logic to meet their needs. The common configurable variables of ASIPs can be, for example, the size of instruction and data caches, the number of registers, and as mentioned in the previous chapter, deterministic algorithms such as MATS test usually achieve a high structural fault coverage level for them. The basic idea to solve the problem of automatic test generation for configurable units is to make the generation process itself configurable such that tests can be tailored in accordance with the current design settings. In essence, MATS

tests specify the ways to access (read and write) each cell with complementary values. Implementation of the abstract scheme to a particular register file requires the information regarding 1) the target structure (the number of registers available, flat or windowed registers, etc.) and 2) instructions available for register operations.

Let us take one of MATS tests as example. The MATS+ scheme in [Nair79] is presented as follows: $\{\updownarrow(w0);\uparrow(r0,w1);\downarrow(r1,w0)\}$. The operation applied to the cell is given by "w" (write) and "r" (read), followed by the desired value (0/1). The manner to traverse all the cells is described by notations "↕", "↑", "↓":

↕: the order is irrelevant;

↑: ascending order;

↓: descending order

Or, the algorithm can be further generalized as: $\{\updownarrow(wa);\uparrow(ra,w\bar{a});\downarrow(r\bar{a},wa)\}$, where a is the background data and $\bar{a}$ is its inverted value.

General-purpose registers are often organized either flatly or grouped as windows. As to the former, all the registers are always "visible" to a program, and the latter only allows usage of registers in the current window aside from the global registers. Figure 5-3 exemplifies the pseudo code used to concretize the MATS+ algorithm for the flat. For simplicity, we assume here that the target ASIP has a 32-bit MIPS-similar instruction set and there are no registers reserved for special purposes. The parameters of "n_rf" and "n_rw" indicate the customization decision regarding the size of the register file during the ASIP design phase. In addition, to allow a flexible implementation of the MATS+ algorithm, three more variables are introduced:

ini_val: the user-specifiable value for the background data, that is, the value for "a" in the above algorithm;

inv_inst: the corresponding instruction used to invert "ini_val". For example, if ini_val is 0x55555555 for a 32-bit processor, then inv_inst can be one of the instructions {add, not}

base: the base address used for response storage

In Figure 5-3, the code in bold generates the assembly instructions of the final test program. The pseudo instruction "li" is used to load a 32-bit immediate to a register, which basically consists of a instruction sequence of "lui" and "ori" to load the upper 16-bits and the lower

16-bits of the immediate respectively. With “li” or “inv_inst”, the background data or its inverted value is “written” to the register cell, whereas the storage instruction “sw” is utilized to retrieve and propagate the responses.

```
Autogen_MATS+_flat (int n_rf, int ini_val, String inv_inst, int base)
int offset = 0;
BEGIN
        //initialize all registers
        for (int i = 0; i < n_rf; i++)
                write (“li $r” + i + “,” + ini_val);
        //ascending access
        for (int i = 0; i < n_rf; i++)
        {
                write (“sw $r” + i + “,” + offset + “(base)”);
                if (inv_inst is an op with two operands )
                        write (inv_inst + “ $r” + i + “, $r” + i);
                else
                        write (inv_inst + “ $r” + i + “, $r” + i + “, $r” + i);
                offset += 4;
        }
        //descending access
        for (int i = n_rf – 1; i  > 0; i– –)
        {
                write (“sw $r” + i + “,” + offset + “(base)”);
                write (“li $r” + i + “, ini_val”);
                offset –= 4;
        }
END
```

Figure 5-3: Configurable MATS+ test program generation for flat register files

As to test generation for flat register files shown on the upper part in Figure 5-3, suppose after the ASIP design phase, the size of register files is configured as 16 and there are no register cells reserved for special purposes. We further specify the rest parameters as: ini_val = 0x55555555, inv_inst = add, base = 0x70000000. Then finally we can get the assembly program implemented as below:

```
li      $r0,    0x55555555
… …
li      $r15,   0x55555555
sw      $r0,    0 (base)
```

```
add   $r0,   $r0,        $r0
… …
sw    $r15,  120 (base)
add   $r15,  $r15,       $r15
sw    $15,   120 (base)
li    $r15,  0x55555555
… …
sw    $r0,   0 (base)
li    $r0,   0x55555555
```

Register windows are another form of register file organizations with the intention to improve system performance on procedure calls. Commonly, there are 8 global registers and all the rest registers are organized into windows. Each window containing 24 registers is logically split into three groups, the "in" and "out" registers for transferring parameters across procedures, and the "local" registers for data processing inside of a routine. Moreover, windows are overlapped such that the "out" registers of a caller procedure become the "in" registers of a callee, and a special bit field in the processor status register (PSR), known as the current window point (CWP), is used to determine the available window. Some processor architectures, for example SPARC, leave the number of register windows as configurable ranging from 2 to 32 and thus eventually varying the number of total registers from 40 to 520.

Suppose a register window in Figure 5-4, where the global registers are from r0 to r7. Additionally, the register indexes grow as we move inside a register window from "in", "local" to "out", or from lower windows to the uppers. Thus, as to Window 1, its "in" section contains r8 ~ r15, "local" registers are r16 ~ r23 and "out" are with registers r24 ~ r31. The entire register window is circular, that is to say, the "out" section of Window (N – 1) is the "in" section of Window 0. Two assembly instructions are available to move from one window to its neighbor:

- *save*: increments the CWP, i.e. the upper window becomes active. Due to the circular structure, execution of "save" within Window (N – 1) will cause CWP to point to Window 0;
- *restore*: decrements the CWP, i.e the lower window becomes active, and execution of "restore" within Window 0 causes the switch to Window (N – 1).

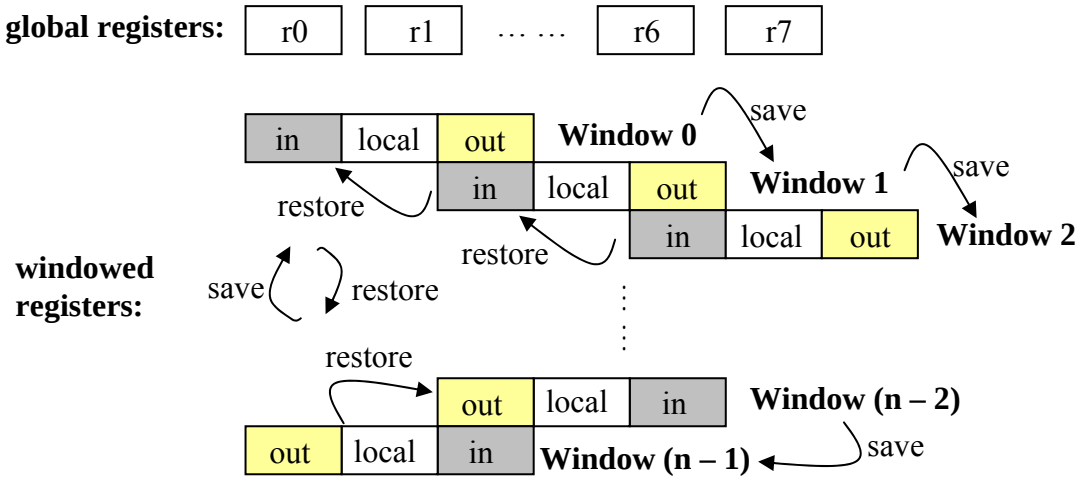


Figure 5-4: The assumed register window

Under the assumption of the mentioned register window, we develop the code shown in Figure 5-5 to generate test programs based on the MATS+ algorithm. Compared with the code for the flat register files, the major difference lies in the way to traverse the register cells. To access in an ascending order, we have to obey the following rules:

1) Start with the global registers and then the windowed registers;
2) Inside a window, start with the "in" registers and then the "locals";
3) Use "save" to increment the CWP

Correspondingly, the descending access requires to:

1) Process the windowed registers and the global ones;
2) Inside a window, start with the "outs" and then the "locals";
3) Use "restore" to decrement the CWP

```
Autogen_MATS+_windowed (int n_rw, int ini_val, int base)
int offset = 0;
BEGIN
        // initialize all registers: first global then windowed registers
        for (int i = 0; i < 8; i++)
                write ("li $g" + i + "," + ini_val);
        for (int i = 0; i < n_rw; i++) {
                for (int j = 0; j < 8; j++) //access input registers
                        write ("li $i" + j + "," + ini_val);
                for (int j = 0; j < 8; j++) //access local registers
                        write ("li $l" + j + "," + ini_val);
                //advance the register window
                write ("save");
        }
```

```
        //ascending access: first global then windowed registers
        for (int i = 0; i < 8; i++) {
                write ("sw $g" + i + "," + offset + "(base)");
                write ("not $g" + i + ", $g" + i);
                offset += 4;
        }
        for (int i = 0; i < n_rw; i++) {
                for (int j = 0; j < 8; j++){ //access input registers
                        write ("sw $i" + j + "," + offset + "(base)");
                        write ("not $i + j + ", $i" + j);
                        offset += 4;
                }
                for (int j = 0; j < 8; j++){ //access local registers
                        write ("sw $l" + j + "," + offset + "(base)");
                        write ("not $l" + j + ", $l" + j);
                        offset += 4;
                }
                // advance the window when the most upper is not reached
                if (i < n_rw)
                        write ("save");
        }

        //descending access: first windowed then global registers
        for (int i = n_rw – 1; i > 0; i– –){
                for (int j = 7; j > 0; j– –){ //access output registers
                        write ("sw $o" + j + "," + offset + "(base)");
                        write ("li $o" + j + "," + ini_val);
                        offset –= 4;
                }
                for (int j = 7; j > 0; j– –){ //access local registers
                        write ("sw $l" + j + "," + offset + "(base)");
                        write ("li $l" + j + "," + ini_val);
                        offset –= 4;
                }
                //switch the window when the lowest is not reached
                if ( i > 0)
                        write ("restore");
        }
        for (int i = 7; i > 0; i– –)
        {
                write ("sw $g" + i + "," + offset + "(base)");
                write ("li $g" + i + "," + ini_val);
                offset –= 4;
        }
END
```

Figure 5-5: Configurable MATS+ test program generation for windowed register files

5.2.3 Test for Optional and User-defined Units

An ASIP core is usually extended with optional or user-defined units to achieve optimized performance for its application. Common examples of optional modules can be multipliers, floating-point units or co-processors. Inclusion of these blocks into the final processor instance directly correlates with structural modifications, and meanwhile from the program level, the basic instruction set is also extended with new instructions to activate these supplementary functions. Instead of utilization of existing functionalities supported by the optional blocks to meet application-specific demands, the user can extend the system capability through implementation of own dedicated logic and it is unavoidable to alter the processor hardware and the instruction architecture.

In all, to incorporate an extra functional unit (either existing or user-defined logic), both the control logic and the data-path of the base core need to be modified. It is apparent that the instruction decoding unit should be extended to process the new instructions, and control signals for the functional block itself as well as any relevant components during operations, e.g. the enabling signal to activate the data-forwarding unit for the read-after-write data dependency, are properly generated. As to test for this additional hardware in the control logic, we can base the work on verification approaches [PEK00][BhWa01][Rimo06], which systematically create the code to reveal system behaviours during normal instruction execution and also during the occurrences of control or data hazards. As to test of the functional block in the data-path, we can use 1) the corresponding instructions to activate this particular unit and 2) deterministic test patterns as operands that target directly at structural faults. Putting the above ideas together, we draw up the steps to generate tests for optional or user-defined logic in Figure 5-6. The key point to enable automatic test generation is to construct in advance a variety of templates shown as Template 1 to 3, each of which contains a set of instructions for a certain test purpose. We will detail their structures later on.

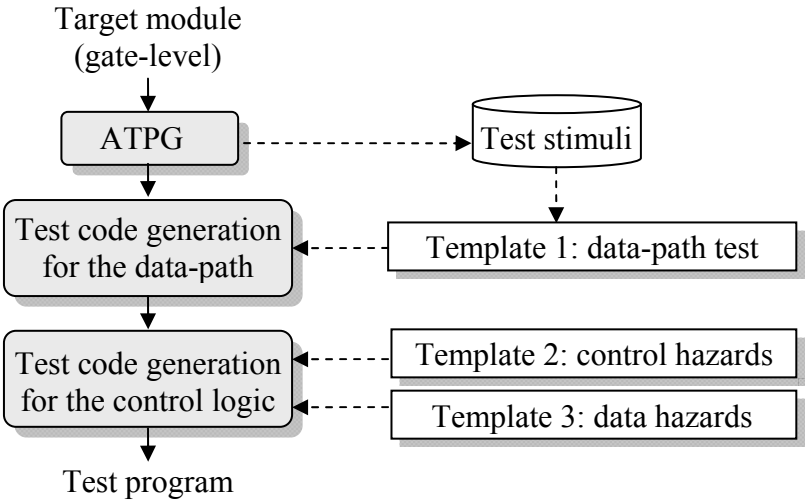


Figure 5-6: Test generation for optional or user-defined logic

For the given gate-level description of the target module, an ATPG process is applied to generate the deterministic test set, which are used as operands to test the data-path. At this point, it differentiates itself from standard verification approaches that operate with random values. Each instance of Template 1 maps a test pattern into its corresponding instructions. Next, we deal with faults in the control logic. For fault sensitization we construct the code in the way to trigger a variety of hazards in the control logic and the data-path. A branch-taken instruction introduces control hazards that cause pipeline stalls and termination of its successor instructions. Therefore, as proposed in [BhWa01], we can generate instruction sequences (Template 2) that consist of our target instructions preceded by branch-taken instructions to study the control hazards. Meanwhile, the code with the true data dependency (read-after-write) causes hazards in the data-path, and activates functions of the bypass logic. Accordingly, Template 3 is constructed for this purpose.

For convenience to explain the above mentioned template structures, we exemplify with an optional multiplier as the target module. The corresponding template implementations (in MIPS assembly) are illustrated in Figure 5-7 respectively.

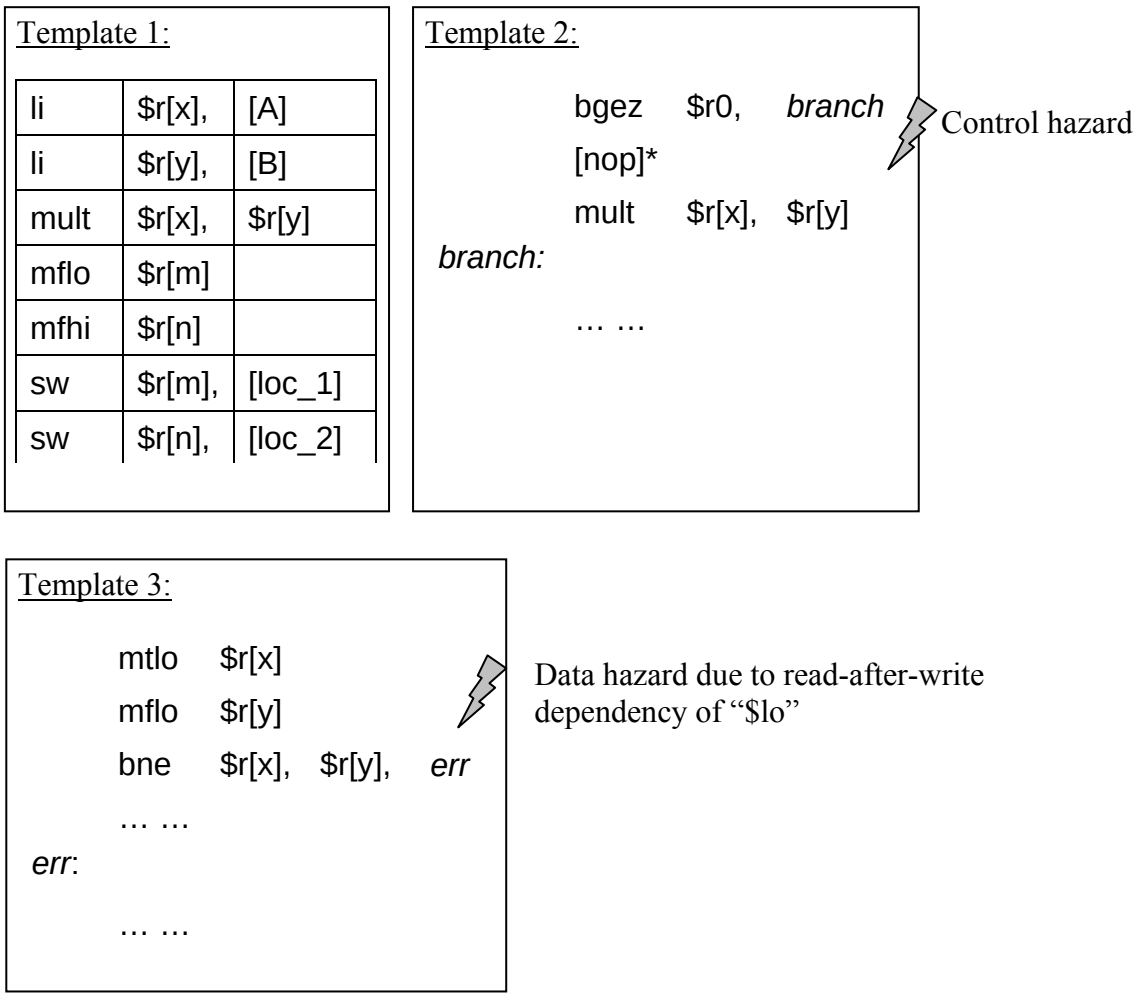


Figure 5-7: An example: templates for an optional multiplier

The multiplier has two special registers (%low and %high) to hold the lower and upper part of computational results. The instructions ("mflo", "mfhi", "mtlo" and "mthi") shown in the above exemplary templates are used to read out data from or move data to these registers. As to template 1, it basically follows a similar structure presented in Chapter 4 to test the multiplier with deterministic test pattern. The code can logically be split into sections for loading the pre-determined operands ("A" and "B") to certain registers, invoking the required operation ("mult") and propagating the result stored in %low and %high to the data bus. Template 2 uses the branch instruction "bgez" (branch on greater than or equal to zero), whose condition is always true due to $r0 that is hardwired to logic '0'. As a consequence, this branch-taken instruction introduces the control hazard that terminates the processing of instruction "mult". Through the number of "nop" instructions inserted in the program, we can control the pipeline stage in which "mult" is cancelled. In template 3, a true data dependency (read after write) exists between the first two instructions due to the common reference to the

register %low. As for a system without the bypass mechanism, the second instruction “mflo” that reads the value in %low is stalled in the pipeline until the write instruction “mtlo” eventually completes. However, pipeline stalls of this kind can be avoided with hazard detection and data forwarding logic. Any faulty execution of these two instructions will lead to unequal values in registers $r[x] and $r[y], and be implied with the branch instruction “bne” (branch on not equal).

5.3 Test Improvement through Instruction Extension

Design-for-test (DFT) is a term for design techniques that insert additional hardware to improve testability and reduce test costs of complex microelectronic products. The well known DFT techniques include test point insertions, scan-based designs, built-in-self-test and so on. In addition to these hardware-centralized techniques, Lai proposed a novel DFT method [LaCh01] that is adequate to software-based self-test. The basic goals for such DFT scheme are: 1) to improve the fault coverage and 2) to reduce the size and runtime of the synthesized test programs. Test logic is added in the form of the instructions so that common problems for standard DFT techniques such as complex timing issues during design modification and high power consumption during test application can be alleviated.

At its core, the idea of the instruction-level DFT coincides with ASIPs. In general, as to ASIPs, the primary instruction set is extended to meet the specific design objectives, which commonly can be low power consumption, or high system performance through acceleration of “hot-spot” code execution. However, the instruction-level DFT scheme tackles particular test issues, or more precisely speaking, quality of the SBST for microprocessor testing. For this reason, on the one side, we can regard this DFT scheme as a special case study, which exhibits application of ASIPs to the test domain; on the other side, to facilitate this special focus on test quality improvement, extensions to the traditional ASIP design flow become necessity.

Thus, we combine for the first time these two research directions and focus on the aspect to ease the instruction-level DFT approach by Lai on the basis of the existing ASIP framework and its necessary extensions. The overall scheme is illustrate in Figure 5-8, which highlights the new features to the conventional ASIPs.

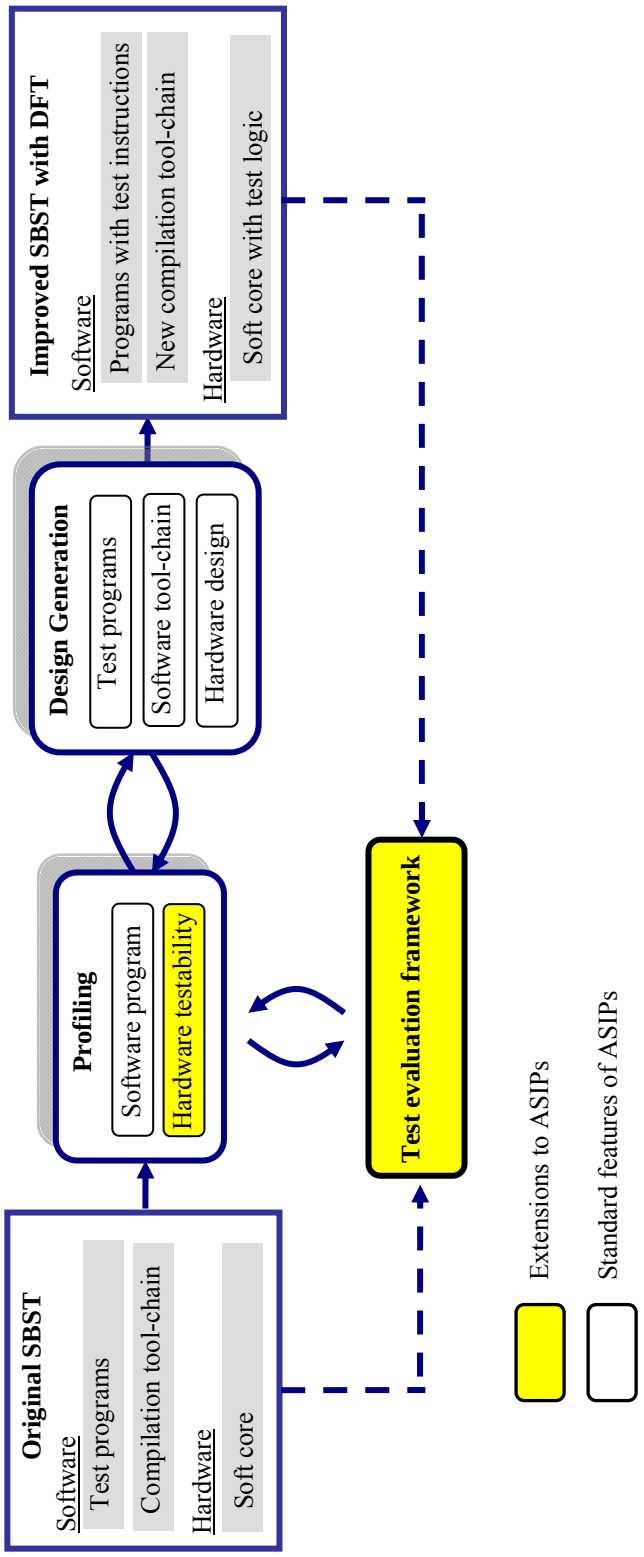


Figure 5 – 8: Extended ASIP framework for instruction-level DFT

Taking the information of the original SBST as the input, the instruction-level DFT scheme starts with profiling the relevant "test qualities" measured in terms of 1) the program features regarding execution time and the code size, and 2) structural fault coverage for the entire processor as well as modules. The ability of software profiling provided by the standard ASIP design flow is sufficient and reusable here for analysis of the first quality measurement. As to the second characterization, we need to add a test evaluation framework to the scheme, which analyzes hardware testability through simulation of the given self-test programs. Based on the profiled information, the hardware design is modified to incorporate user-defined test logic which can be activated through new instructions. Besides, other modifications are also necessary to make the new instructions recognizable to the software tool-chain and usable in the SBST programs. To obtain the optimal result, usually there are several iterations between the steps of profiling and design modification. In the end, the SBST with the improved structure is generated.

As mentioned already, the issues regarding instruction extension for small code sizes and short execution time are intensively covered by existing ASIP studies. So here we concentrate on the other aspect, namely implementation of new instructions for fault coverage improvement.

In general, a module is defined as "controllable" in SBST, if there exists an instruction sequence to move desired data from the memory to this unit; and for "observable" modules, their outputs can be transferred to the memory with use of instructions. According to the definitions, the general-purpose register file of a standard microprocessor is fully testable, as there are several ways to apply particular patterns to the registers, e.g. using "load-from-data-memory" or "load-from-immediate" instructions, and at the same time read out the corresponding outputs with "store" instructions. Outlined in Figure 5-9, the basic idea is to create appropriate paths between the hard-to-test modules and the register file so that the modules become controllable and observable via instructions as well.

Figure 5-9 shows two kinds of hard-to-test modules. For the module on the left, since there are paths from the module output either to the memory or to the register file (RF), observation of this module is not a problem. However, as its input is not connected to any controllable sites, e.g. the RF, it is hard to apply a desired value to the input ports and as result faults in it

are difficult to be detected. To improve its testability, a new instruction is introduced to move data from the RF to the MUT. This means, at the hardware level, we create an additional path to connect the RF and the MUT. As to the module on the right, it is hard to observe as no paths exist between it and any of observable sites (RF). So, in this case, test instructions are implemented to transfer the output of the MUT directly to the RF.

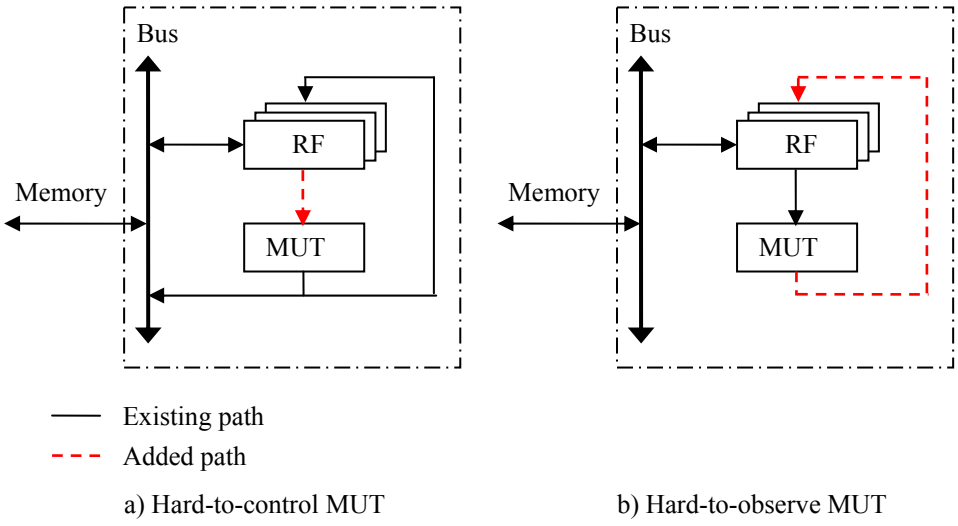


Figure 5-9: Schemes for hard-to-test Modules

The overall steps for test instruction implementation are elaborated in Figure 5-10, which concern the phases of profiling and design generation in **Error! Reference source not found.** Taking the netlist of the target processor as the input, we simulate the original SBST test program and identify modules with insufficient fault coverage, each of which can be the candidate to improve testability through the instruction level DFT. A further investigation is then carried out to find out the structural reasons for the low fault coverage, i.e. whether it is due to low controllability and/or low observability. The object with low coverage and high fault contribution becomes our target for test improvement through instruction extension. On the other hand, the original instruction set architecture is analyzed to identify available coding schemes for the new test instructions. Information from these two aspects is given to the implementation phase, which modifies the control logic and the data-path in line with the idea presented in Figure 5-9. As suggested in [LaCh01], existing hardware should be reused as much as possible to reduce area overhead due to the design changes. Meanwhile, the instruction set is extended with the coding scheme for the new test instruction.

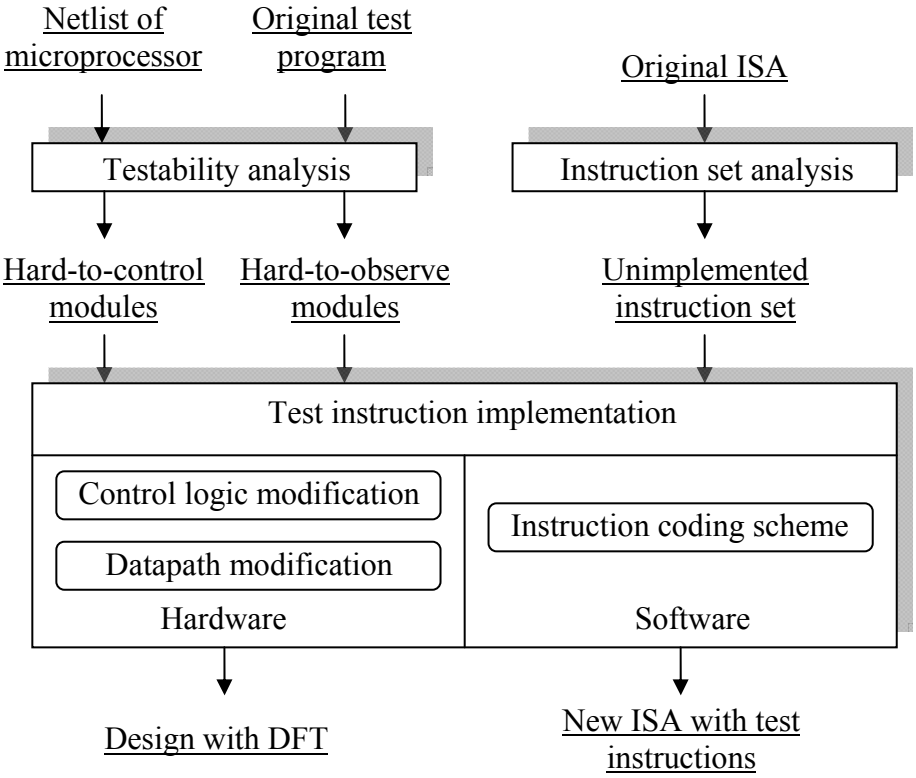


Figure 5-10: Steps in test instruction implementation

5.4 Conclusions

The new design era of ASIPs enable customization of a processor core to meet specific domain requirements. Despite the current emphases in ASIPs on automatic design exploration and generation, the test aspect is somewhat overlooked. As a result, we discuss here in details the interesting topics that are worthy of research efforts. Firstly, to ensure short time-to-market, test of ASIPs should be automated. In this chapter, we demonstrate application of software-based self-test strategies to ASIPs and propose a scheme to address the problem of automatic test program generation. Meanwhile, we combine the two research directions, namely the instruction-level DFT method proposed by Lai and the ASIPs, and present a framework that extends the conventional ASIP design flow to support the DFT implementation.

Chapter 6 Experimental Results

In this chapter, we discuss application of the software-based self-test methods to a few of microprocessors. As there are no benchmark processors widely acknowledged in this research area, we select some microprocessors from a large variety of public-accessible cores, each of which reveals particular issues during implementation of self-test schemes. The first two case studies on general-purpose processors with and without pipelines are intended to exhibit optimization facets regarding test quality and power consumption during test program generation, while the last on the configurable core exhibits the aspects regarding test program generation for on-chip user-defined logic and utilization of the processor to deliver the patterns for the target modules. A short conclusion will be given at the end of this chapter.

6.1 Hapra: 32-bit RISC Processor Core for Academics

In this section, we present the practical facets regarding implementation of the power-, memory-optimized self-test scheme to a 32-bit processor core that conforms to the reduced instruction set computing (RISC) load-store architecture and is developed mainly for the educational purpose [Hapra].

6.1.1 About the Processor Core

The block diagram of the processor design is shown in Figure 6-1. The data bus and the address bus are both 32-bit wide. As we see from this figure, the core is mainly made up of the following functional components: the ALU, the register file (RF) with 32 general-purpose registers, the program counter (PC), the instruction register (IR), the controller, and a few of

multipliers. Receiving two data inputs from the register file, the ALU can perform one of 8 operations selected by the controlling lines at a time. The PC holds the memory address where the current instruction is stored and in order to calculate the address for the next instruction, the PC can either add the value by a constant step size of 1, or receive the address from the register file upon a jump instruction. The opcode of the instruction to be executed is passed on from the IR to the controller, which then generates corresponding signals to control operations of all related components.

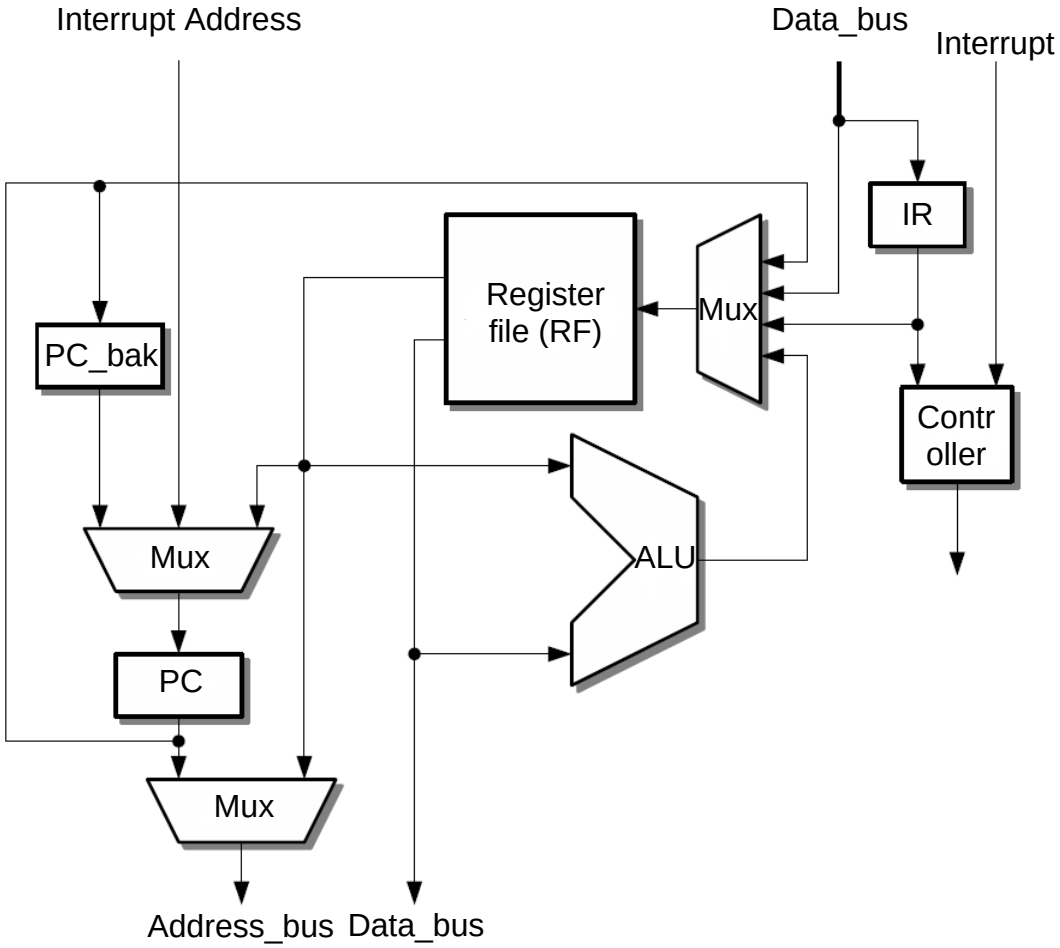


Figure 6-1: Block diagram of the Hapra processor

The processor has a concise instruction set (in total 17 instructions) supporting memory and register operations, arithmetic/logic/shift operations, unconditional and conditional jumps and procedure calls. The assembly program development for the Hapra processor is simplified with the integrated interface called HASE (Hapra Assembler Software Environment), with which programs can be edited, compiled, debugged and simulated. More relevant to our specific needs, the environment offers the functionality to dump the memory image of a given assembly program which is necessary for circuit simulation with commercial CAD tools.

6.1.2 Test Program Synthesis

Following the criteria presented in Chapter 4 to select appropriate target components for test program generation, we synthesize at first the core and analyse area contribution of each component. The pie diagram in Figure 6-2 shows the result in percentage for the major components. As we can see from the diagram, the RF takes up the largest portion, around 87.41%, of the total processor area, and then the ALU is another major contributor with about 4.51%. It is obvious that generating test programs for these two components is more effective in reaching a high fault coverage level for the entire processor than concentrating on other modules with minor area as well as fault influences. Besides, they are easily accessed with instructions. Despite its small area contribution, we select nevertheless the PC as our target due to two reasons. Firstly, this component is with high instruction-accessibility, and secondly, the use of jump or branch instructions to deliver PC test patterns can possibly increase fault coverage in the control logic, as they belong to another instruction category that is different from those used in the ALU and RF tests.

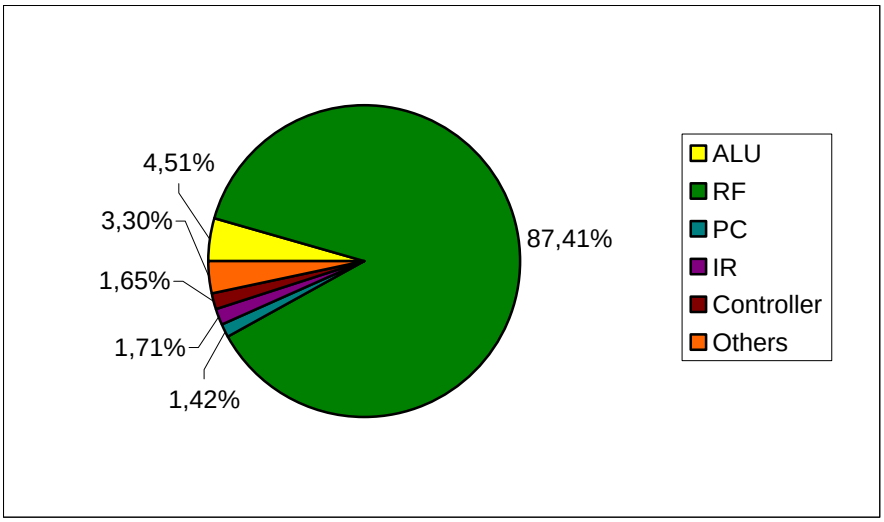


Figure 6-2: Analysis of area contributions of Hapra components

During synthesis of the self-test programs, we implement specific strategies for the above three modules. As to the ALU, we intend to apply deterministic test patterns generated by ATPG. For the PC, its tests are integrated into the ALU routine, and in this way, we can detect faults in multiple modules within a single test program. To match the sizes of test sets for these two modules, we generate patterns for the PC both deterministically and randomly.

With the knowledge of the template structure used for the combined test, we firstly generate a few of program addresses (patterns for the PC), which 1) will not cause any damages for program execution due to e.g. overlapped instruction segments and 2) trigger transitions in the PC as many as possible when applied. The random patterns are only necessary when the number of the deterministic PC patterns is still less than that of the ALU. For the RF test, we implement the MATS algorithm.

6.1.3 Experiments

6.1.3.1 Framework for Test and Power Evaluation

The evaluation framework is an extension of [ChDe00] to support measurements of gate-level fault coverage and energy consumption in terms of switching activities. The average power of a test program is then calculated with division of the energy consumption by its execution cycles.

As outlined in Figure 6-3, the RTL processor core is synthesized with a logic synthesis tool into the gate-level netlist. During the gate-level simulation, the test bench integrates the processor implementation and the test program binary produced through cross-compilation. Two kinds of information are recorded at this step for evaluations of test coverage and energy consumption, as highlighted in the figure with light and dark grey boxes. Primary inputs of the processor are captured and translated into patterns which are later fault simulated to evaluate the structural fault coverage. Switching activities throughout the whole simulation process are tracked in a file called Value Change Dump (VCD). Based on such file, we can use commercial tools, e.g. Primepower [Prim], for energy estimation, if layout information of the technology is available. As an alternative, it is straightforward to sum up recorded transitions and treat the final number as energy consumption for the simulated code directly. In our experiment, we adopt the second way for energy measurement.

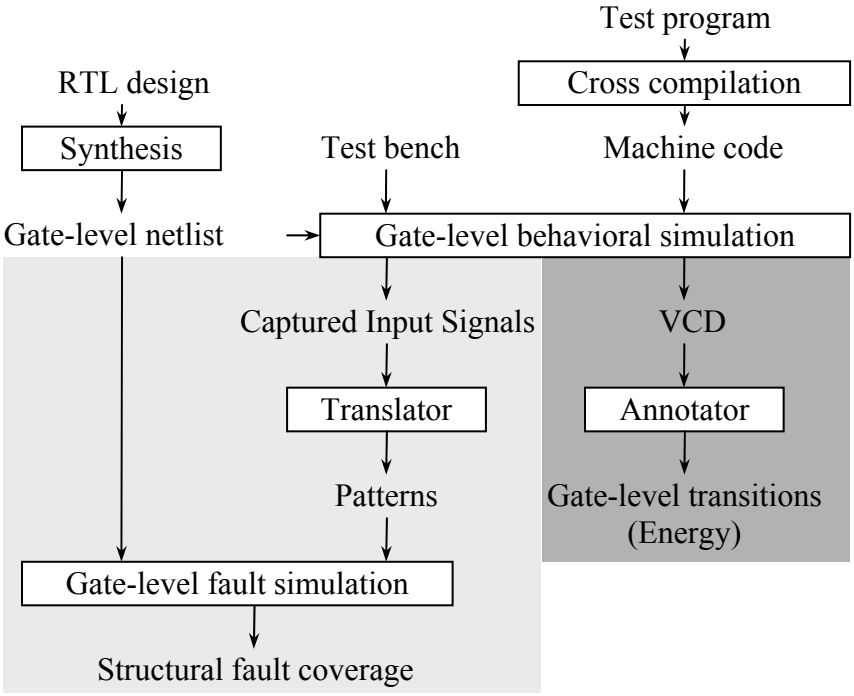


Figure 6-3: Evaluation Framework

6.1.3.2 Experiment Setting-ups

To evaluate our proposed method in Chapter 4, we mainly concern the issues regarding:

- Structural fault coverage both for the target modules and the entire processor;
- Memory requirements, power consumption and test cycles before and after optimization;
- Impacts of all the optimization means on the fault coverage level.

In line with this, we set up the experiments as follows:

1) Perform ATPG for the ALU alone to provide the deterministic patterns. The test set achieves 99.90% fault coverage of this module;
2) Implement a MATS test for the RF, which is used together with programs generated in step 3 to 5;
3) Generate the PC test set with the size equal to that of the ALU in 1), and synthesize the ALU and PC test sets into a test program (*Ori.TP*), which reflects the rudimentary qualities of our algorithm;
4) Compact ALU patterns in step 1, update PC patterns accordingly, and then re-synthesize the new sets into another program (Compacted). At this step, we basically target at reduction of memory requirement and test time;
5) Further optimize power consumption through the means of

5.1) Reordering the code generated in step 4 under test constraints. The new code is referred as Reordered;

5.2) Specifying don't-care bits of the code in step 5.1), and the new code is termed as *X-filling*;

6) In the absence of program synthesis algorithms e.g. [Kran03a], we conduct a sequential ATPG for comparison. Since the fault coverage is too low to accept, a program with randomized instructions is added. The number of test cycles for them together is equal to that of *Ori.TP* generated in step 3.

6.1.3.3 Results and Analysis

In regard to the time spent on the aforementioned experimental setups, steps for ATPG, program synthesis and power optimization actually last only a few seconds, while compaction accounts for the major part of the time required.

The statistics on fault coverage, test cycles and memory requirements are reported in Table 6-1. The column of *Ori.TP* indicates that the test program achieves high fault coverage for the targeted modules as well as for the entire processor. The column labelled as "Optimized" represents the data for the cases of *Compacted*, *Reordered* and *X-filling*. An identical value for the fault coverage is observed for them, which implies that we successfully retain the fault coverage during the two-stage optimization for low power. From the viewpoint of structural fault detection, our method, no matter with or without further optimization for test cycles, memories and power consumption, can generate effective test programs that obviously outperform the use of sequential ATPG together with randomized instructions, as the latter only reaches 19.64% fault coverage for the entire processor. In fact, sequential ATPG alone already leads to 18.14% processor fault coverage with 1,457 test cycles, while the improvement by the randomized program, which contributes to a much larger portion of cycles, is not remarkable. This observation is just in line with the well-known conclusion that processor cores are random-pattern resistant.

As to the aspect regarding optimization for short test cycles as well as low memory requirement, Row 7 and Row 9 indicate that with pattern compaction, we can reduce around 30% test cycles and 25% memories without distinct impact on the overall fault coverage. We

omit calculation of the memory requirement for the combined test of sequential ATPG and the random program due to its mixed form.

Table 6-1: Comparison in fault coverage, test cycles and memory requirements

		Ori.TP	Optimized	ATPG + randomized
Fault coverage	ALU	99.90%	99.90%	2.87%
	PC	92.38%	92.38%	66.40%
	RF	98.46%	98.03%	17.96%
	Proc.	96.70%	96.33%	19.64%
Test cycles	No.	13,886	9,628	13,886
	Ratio	1.0	0.6935	1.0
Memories	Bytes	61,550	46,050	--
	Ratio	1.0	0.7482	--

Table 6-2 details the outcome regarding energy and average power consumption. The last four rows clearly show a persistent reduction of these two parameters through every optimization step. In particular, row three indicates that compaction not only results in shortening test length, which means energy reduction as well, but also in minimizing average power dissipation by concerning switching activities of registers throughout the process. Moreover, as shown in the first row, our methodology also outperforms the combination of the sequential ATPG and the randomized program in this aspect. According to the experiment, the sequential ATPG patterns causes 1,215,280 transitions in total, which consequently bring the average power consumption (transitions per test cycle) to 834.10. Meanwhile, the randomized program consumes average power of 525.40, which is higher than any of our test programs but much lower than those structural patterns. The reason for this observation is, the structural test activates signal transitions as many as possible, while high correlation among instructions ensures much lower power consumption even for the case where random instructions with random data are used. Hence, we come to the conclusion that the SBST is low power in nature.

Table 6-2: Comparison in energy and average power consumption

	Energy		Power	
	# of transitions	Ratio	# of transitions per cycle	Ratio
ATPG + randomized	7,745,492	1.0	557,79	1.0
Ori.TP	6,205,081	0.8011	446.86	0.8011
Compacted	4,081,133	0.5269	423.88	0.7599
Reordered	3,904,869	0.5041	405.57	0.7271
X-filling	3,710,389	0.4790	385.37	0.6909

6.2 Plasma: MIPS I-Compatible RISC Processor Core

This section details the application of the SBST scheme to the pipelined processor core, called Plasma [Plasma]. The primary experiment shows that the scheme proposed in Chapter 4 is quite effective in detecting data-path faults, whereas the coverage for the control logic is beyond satisfaction. For this reason, it is of significance to improve test quality of the SBST by explicitly taking into account the fault detection in the control logic during test program synthesis.

6.2.1 About the Processor Core

Created and released by Opencores.Org, the Plasma CPU is a synthesizable 32-bit RISC microprocessor and executes all MIPS I [1] user mode instructions except unaligned load and store operations. As illustrated in Figure 6-4, the functional blocks in the data-path of the Plasma core include: the ALU, the multiplier (Mult), the shifter, the register file (Reg_bank) and the multiplexer (Bus_mux), while the other modules such as the memory access controller (Mem_ctrl), the instruction decode logic (Control) and the program counter (PC_next) form the control logic. The number of the pipeline stages of the CPU can be two or three, depending on whether an additional stage for memory access is needed.

[1] MIPS I is a registered trademark of MIPS Technologies, Inc.

Regarding the register usage, there are 32 32-bit general-purpose registers and the register R0 is hard-wired to logic '0'. For multiplication and division operations, two special registers LO and HI are used to hold the computational results. Besides the release of the CPU as a soft code in VHDL, OpenCores.Org provides an assembly program (opcodes.asm) which enumerates all the implemented MIPS I for verification and regression test.

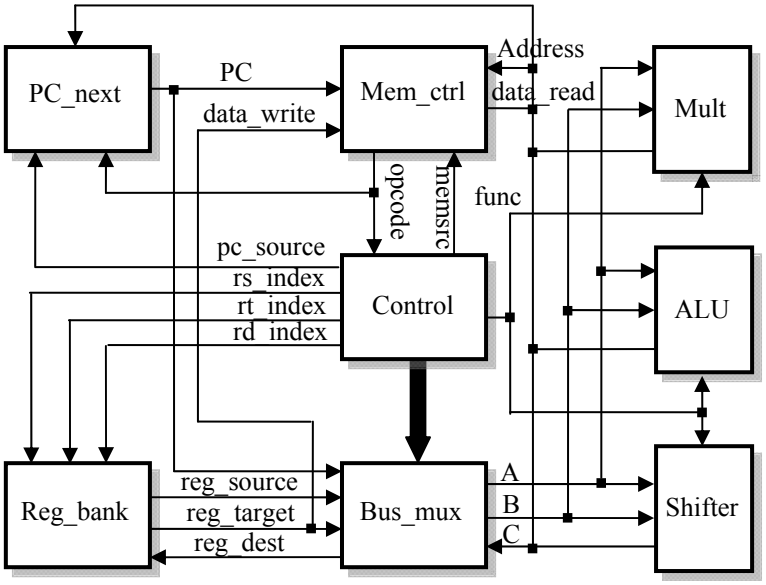


Figure 6-4: Block diagram of the Plasma CPU

As reported by Opencores.Org, this core already sees some practical applications, such as a web server running on a Plasma CPU on a Xilinx FPGA (Field Programmable Gate Array), and the use of Plasma RISC CPU to control four communication robots using Xilinx Virtex FPGAs.

6.2.2 Test Program Synthesis for Control Logic

Firstly, to evaluate the effectiveness of the method proposed in Chapter 4 for the pipelined processor, we apply it to synthesize test programs that are mainly for the data-path components. The target modules are: the ALU, the register file, the shifter, the multiplier. In addition, we also generate a test routine for the PC with use of random program addresses. Table 6-3 outlines the obtained result.

Table 6-3: Fault coverage of the test programs for the data-path modules

	# of faults	Fault coverage	# of cycles	Program size (KB)
u1_pc_next	2118	81.26%	475	1.95
u2_mem_ctrl	3925	61.63%	--	--
u3_control	1579	69.22%	--	--
u4_reg_bank	37528	95.20%	308	1.66
u5_bus_mux	2344	62.54%	--	--
u6_alu	2632	99.58%	612	4.44
u7_shifter	3331	100%	361	2.33
u8_mult	11094	87.57%	3740	8.29
u9_pipeline	4232	72.54%	--	--
Interconnects	1664	95.19%	--	--
Processor	70447	89.07%	5496	18.67

We can see from the table that the proposed method can effectively detect faults in the data-path modules. Additionally, although not explicitly targeted, interconnects can also be well tested with the synthesized programs. However, the coverage for the components that are part of the control logic is not satisfying. This gives us a good reason to think about synthesis of test programs that can appropriately detect faults in the control logic so as to further improve test quality.

Here are two aspects regarding the SBST scheme that have to be considered. First, not all module-level structural test patterns can be realized with instructions. A straightforward example can be the patterns whose applications require the use of “invalid” opcodes. Second, when realizing these deliverable patterns in instructions, we can only detect a subset of faults that are along the instruction-sensitized path. Therefore, we target in the work the *functionally testable faults*, that is, the faults correlating with instruction-realizable patterns and along the sensitized path during system-level pattern application.

As a part of the control logic, the instruction decoder of the Plasma core is implemented as a combinational logic which receives opcodes from the memory access module and generates corresponding signals to control the behaviors of concerned units. An ATPG is hence an ideal solution to provide us an effective test set, where desired opcodes are specified. Based on the

ATPG patterns, we can avoid the inefficient way, as the traditional approach does, to generate a test program for the control logic with enumeration of all instructions and all addressing modes. Figure 6.5 sketches the major steps in this regard. The gate-level netlist of the combinational decoder logic is given to an ATPG process. Then with reference to the instruction set architecture, we classify the patterns into two sets: the functional set consists of patterns that can be translated into instructions and the un-functional set with patterns corresponding to invalid opcodes. Our program generation will only consider the functional set.

According to the MIPS assembly format, instructions are divided into three groups: R-type contains instructions that operate on register operands, such as arithmetic and logic with all operands in registers; I-type includes instructions with an immediate operand; and J-type consists of jump instructions, which require a 26-bit field to specify the target address of the jump. However, in view of the functionalities of instructions, we can categorize them into classes: 1) arithmetic, logic, shift and move operations; 2) branch and jump operations; and 3) load and store instructions. During template construction, we take the latter classification into account and for each instruction class, we devise accordingly a template which forms a unit of basic instructions necessary to sensitize the functionally testable faults and propagate them to an observable site (either to a general-purpose register cell which can later be easily accessed, or directly to the processor's outputs of the address or data buses). In the end, the test program will be synthesized based on the functional set and the templates. A better result can be achieved with combination of the test programs for the data-path modules, as we mentioned previously.

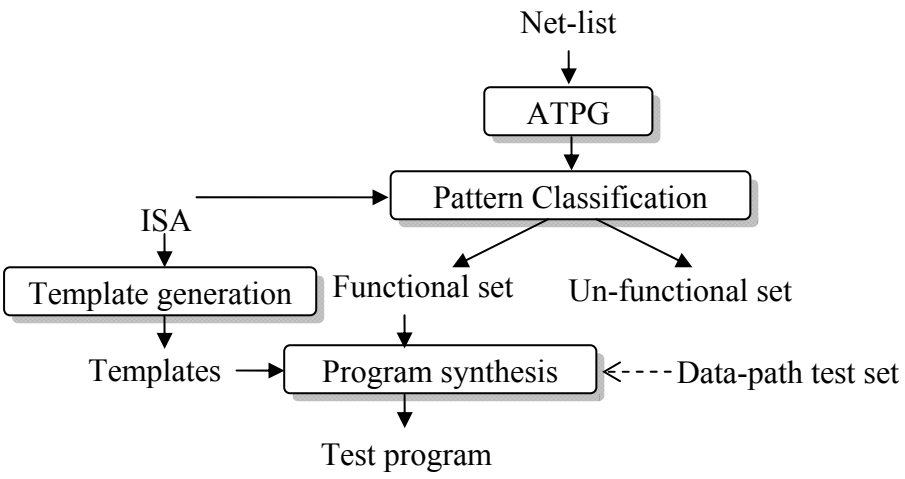


Figure 6-5: The flow of test program synthesis for the control logic

6.2.3 Results and Analysis

We have applied the overall methodology to the Plasma core. Giving the net-list of the instruction decoder to Flextest, we get 80 ATPG test patterns, 41 of which are functional. Table 6-4 provides the results regarding fault coverage of these test patterns. The total 80 test patterns can achieve 97.71% fault coverage of the decoder logic, and however when only 41 functional patterns are applied, the coverage can only be 87.26%, which is also the upper bound of coverage that a synthesized test program for this unit can obtain. Further analyzing these 41 functional patterns, we identify that 26 are arithmetic, logic, shift or move operations, 9 address branch/jump instructions and the rest 6 are loads or stores.

Table 6-4: Test patterns vs. Fault coverage

		# test patterns	FC
Pattern type	Functional	41	87.26%
	Un-functional	39	--
	Total	80	97.71%

We then synthesize these 41 functional test patterns into a test program (TP) by instantiating templates. For comparison, we adopt the program (opcode) released with the Plasma core. Provided for regression test of each variation of the processor, this program exercises all the supported MIPS instructions. The relevant statistics are listed in Table 6-5, and as to fault coverage, we consider the complete fault set. We compare the results from three aspects: the memory requirement, the number of test cycles and the achieved fault coverage. As shown in the second row, the proposed method requires much less memory for storing test data. Moreover, the third row indicates the number of test cycles is dramatically reduced by our method. From the standpoint of fault coverage, a notably better result is achieved by our algorithm than the instruction-exhausted method.

Table 6-5: Comparison: the proposed method vs. the instruction-exhausted method

			TP	opcode
Memory requirement (KByte)			2.48	9.24
# of Test cycles			531	2055
Fault coverage	Control logic	u1_pc_next	74.79%	55.43%
		u2_mem_ctrl	79.59%	70.96%
		u3_control	82.20%	83.09%
		u5_bus_mux	72.65%	65.27%
		u9_pipeline	81.19%	78.14%
	Data-path	u6_alu	84.23%	65.73%
		u7_shifter	45.15%	24.38%
		u8_mult	61.54%	65.34%
		u4_reg_bank	56.65%	29.33%
	Interconnects		90.75%	87.26%
	Processor		63.11%	45.93%

Finally we combine all the tests, namely both for the data-path components and the control logic, to see what the fault coverage level is. Table 6-6 presents the corresponding results. As indicated in the figure, a notable improvement of the processor fault coverage as well as module coverage is observed with around 9.66% increase of test cycles and 13.28% increase of the program size.

Table 6-6: The results for the data-path and control logic test programs together

	Fault coverage	# of test cycles	Program size (KB)
u1_pc_next	87.63%	475	1.95
u2_mem_ctrl	80.28%	--	--
u3_control	82.84%	531	2.48
u4_reg_bank	95.20%	308	1.66
u5_bus_mux	77.26%	--	--
u6_alu	99.58%	612	4.44
u7_shifter	100%	361	2.33
u8_mult	87.58%	3740	8.29
u9_pipeline	81.59%	--	--
Interconnects	96.39%	--	--
Processor	91.29%	6027	21.15

6.3 Leon: Configurable SPARC v8-Compatible Processor Core

In this section, we exhibit application of the software-based self-test scheme to the Leon processor, which is user-configurable and extensible. The major focus here is to utilize the processor to test its extended logic.

6.3.1 About the Processor Core

Designed for embedded applications, the 32-bit Leon2 [Leon] processor conforms to the SPARC Vv8 architecture [SPARCv8]. The design is released as a soft core implemented in VHDL, and fully synthesizable with most synthesis tools. The block diagram in Figure 6-6 is excerpted from the processor manual and illustrates the major functional components: the integer unit realizing the SPARC V8 instruction set, the floating-point unit (FPU), the co-processor (CP), the cache sub-system, the memory management unit (MMU), the debug support unit (DSU), the memory interface, the timers, the on-chip watchdog, two 8-bit Universal Asynchronous Receiver Transmitters (UARTs), interrupt controller, a 32-bit parallel I/O port, AMBA on-chip buses, a PCI interface, an Ethernet MAC, and on-chip RAM.

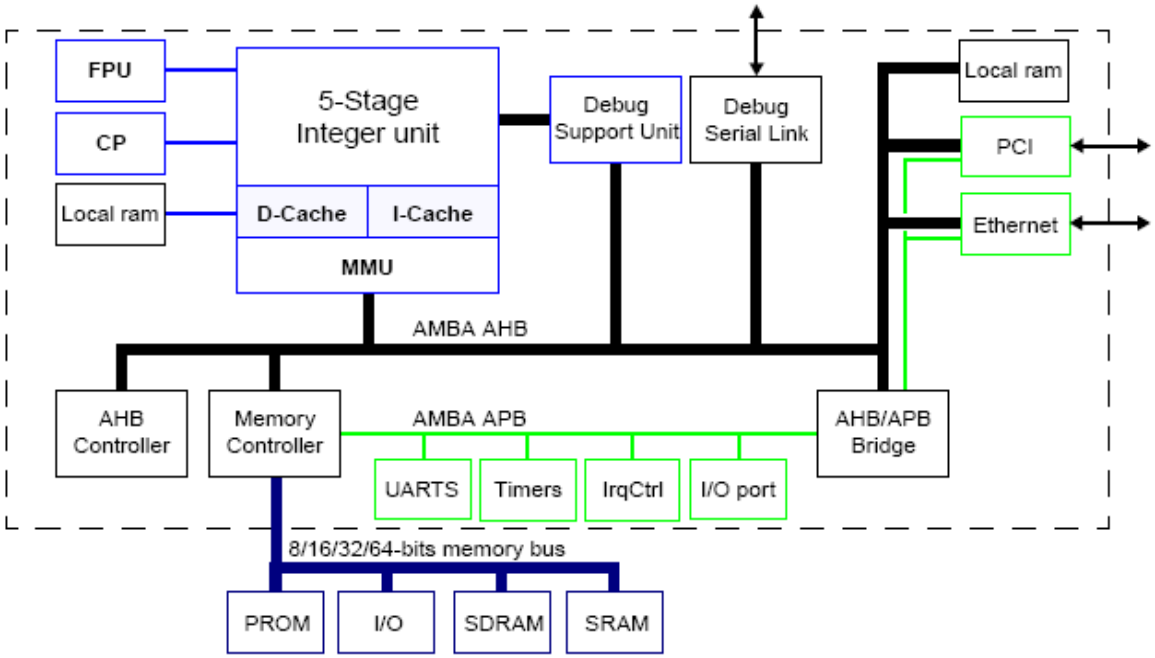


Figure 6-6: Leon2 block diagram excerpted from [Leon2]

The Leon processor features itself in the following attributes:

- ✓ Configurability

  The core has a few optional components that the user can decide whether to present in the final design instance or not, for example, the FPU and the CP. Besides, some module parameters can also be specified by the user. For an instance, the number of the register window of the integer unit is configurable, whose value ranges within the limit of the SPARC standard, namely from 2 to 32, and has a default setting of 8. As to the configuration process, users can invoke the graphical interface to configure the core and by the end of the process, the corresponding hardware descriptions are automatically generated. Or as an alternative solution, configuration can be done by directly modification of the relevant VHDL files.

- ✓ Extensibility

  With the flexible implementation of AMBA AHB and APB on-chip buses, the functionality of the processor can be easily extended with inclusion of user-defined modules to the system. However, addition of new IP cores requires design changes that must be done fully by hand.

- ✓ Scripts and tool-chains to assist the design flow

  The Leon environment provides comprehensive scripts to support hardware design at various abstraction levels with the mainstreaming commercial tools.

6.3.2 Test Program Synthesis for Extended Logic in Leon

As presented already, with the flexible bus architecture, the Leon environment allows the user to add extra logic with ease to the core. Accordingly, here within this experiment, we implement the test scheme for the on-chip user-defined logic based on the processor. The idea is sketched in Figure 6-7. A test program for the extended IP is generated and stored in the memory. During program execution, the microprocessor fetches instructions from the memory and delivers the desired test patterns to the target IP. Test responses are finally transferred back to the processor and saved to the memory for later analysis.

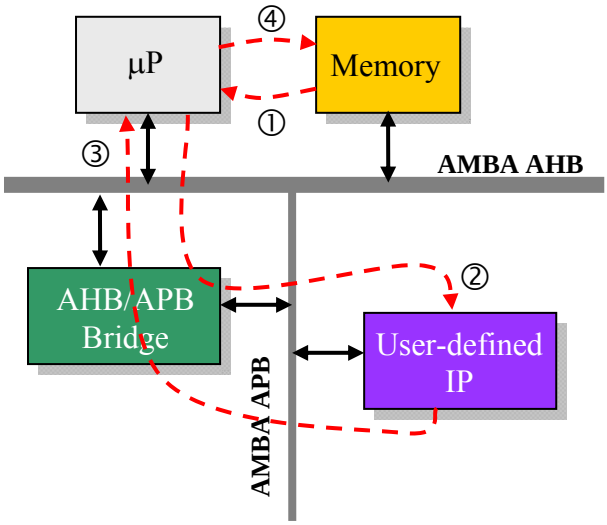


Figure 6-7: Processor-based test scheme for the on-chip user-defined IPs

In line with the AHB/APB standard, high-speed units such as the processor core, the memory controller and the AHB/APB bridge are connected to the AHB bus, where by default the processor plays the role as the bus master and the other two as slaves. All the peripheral IPs are linked to the APB bus and allocated with local indexes. In order to let the IPs be addressable from the processor point of view, their corresponding indexes are then mapped into AHB addresses (ranging from 0x80000000 to 0x8FFFFFFF) by the APB bus master, namely the AHB/APB bridge. Our scheme relies on these IP-specific AHB addresses to communicate with the target modules for the test purpose.

Suppose that we have a unit with the address “0x800003CC”, to access this unit, we can implement the code as shown in Figure 6-8 either in C or in SPARC assembler. From the top to the bottom, the highlighted code segments fulfill the tasks to 1) point the address of the target IP; 2) send a test pattern to the IP via writing to the address; and 3) acquire the response via reading from the address. In this way, given a set of test patterns for an IP, we can delivery them only with the processor instructions, thus avoiding utilization of dedicated test hardware as logic BIST approaches do.

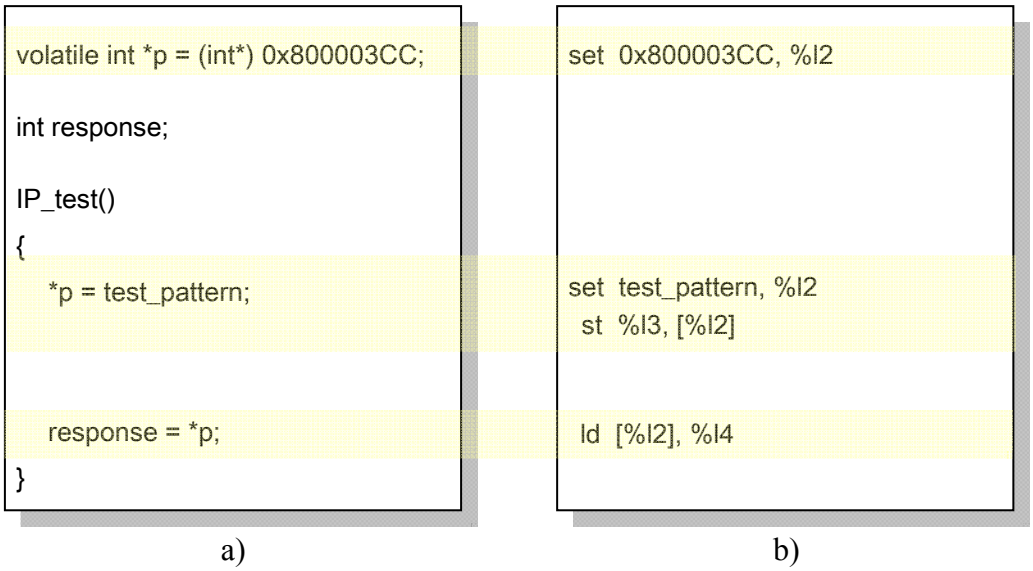


Figure 6-8: Testing the IP a) in C or b) SPARC assembler

6.3.3 Results and Analysis

To evaluate the above scheme, we firstly extend the Leon with addition of an extra peripheral, which includes two registers with the interfaces conforming to the AMBA specification. In order to make them both accessible to the processor, we allocate them each with an APB address.

As the next step, we generate a test program that basically delivers 4 32-bit data (0x0, 0x55555555, 0xAAAAAAAA, 0xFFFFFFFF) to the two registers. After cross-compilation, we obtain the executable of the routine which is loaded by the testbench into the simulation environment. All the signals at the logic's primary inputs are listed down, out of which the test patterns are extracted. We finally perform fault simulation with Flextest.

Our results show that with 1903 test cycles, we can reach the fault coverage of this on-chip unit as 93.55%. The major advantages of this scheme are its minimal hardware overhead in realizing the IP test, and the at-speed ability.

6.4 Test Improvement through Instruction Extension

This section discusses in details our case study on test instructions with use of a variant of the HAPRA microprocessor, which has the identical architecture but a narrower width of address

and data buses. Our main focus to introduce new instructions is to improve fault coverage for the hard-to-test components of the system.

6.4.1 Test Analysis

We first need to analyze test quality of the primary SBST scheme so that the modules with low testability can be identified, and then test instructions are added to increase fault coverage for them. Application of the self-test programs to the target microprocessor can achieve up to 95.72% stuck-at fault coverage at the processor level, and the coverage for the major data-path components such as the ALU and the register file reaches 99.81% and 99.14% respectively. In general, data-path modules are easy to be accessed with instructions, thus with high instruction-testability; and at the same time, they often contribute to a large portion of the total processor area. Therefore, it is appropriate to synthesize firstly test programs for data-path components from the standpoints of test efficiency and engineering efforts. However, since modules of the control logic are with low controllability and/or observability, faults in them cannot be sufficiently detected using the primary SBST scheme. Taking the instruction register (IR) as an example, the fault coverage is only 73.85%. So, one of the possible ways to upgrade the coverage is to define a new instruction which improves the testability of this unit.

The input to the IR is the binary code of the instruction fetched from the memory. This means that a desired value can be applied to this unit if we properly select the instruction to be used. In other words, we can say the IR is instruction-controllable. The major difficulty for test arises from observation of its output. Structurally, the IR output flows both into the control logic and the data-path. That is, the signals concerning the op-code are connected to the decoder. As they are further embedded in the control logic, these signals are hard to be observed. On the other hand, the IR outputs regarding immediates are forwarded to the data-path, and it is possible to observe them using suitable instructions. For example, as to our target microprocessor, “ldhi” (load the 8 most-significant bits of an immediate) will extract the lowest 8 bits from its binary code and copy this value to the corresponding most-significant bits of the specified register. Nevertheless, in all the instruction register is with relative low observability.

6.4.2 Test Instruction Implementation

As mentioned in Chapter 5, implementation of test instructions addresses two aspects: 1) at the software side, identification of the available instructions, determination of the coding schemes, and modification of the compiler to recognize the new instruction; and 2) at the hardware side, modification of the design, which usually requires changes of the control logic and the data-path structure. According to the test analysis in the previous section, the IR is with low observability. To improve its testability in this regard, our primary plan is to latch every value that passes through the IR and defined a new "test instruction", whose execution will move the latched value to a general-purpose register.

Given the instruction set architecture, we can select one from the unimplemented instructions and re-define it as our test instruction. Its coding scheme is illustrated in Figure 6-9. With execution of the "mvir" instruction, the latched IR value will be forwarded to the specified destination register, "rd", which is one of register cells in the general-purpose register file. Accordingly, to encode the instruction, we assign the highest 5 bits, "11101" in the figure, as the opcode for the instruction. For the target microprocessor, there are 8 general-purpose registers, which means we need 3 bits to specify our destination register "rd". All the rest bits are unused, and by default assigned all with 0's.

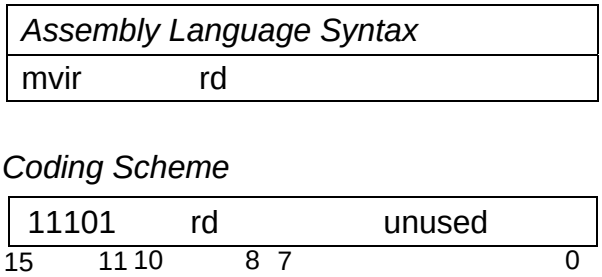


Figure 6-9: Coding scheme for the new test instruction

Figure 6-10 presents the hardware implementation for the new test instruction. The IR output is transported to a latch, which is controlled under the signal "T_EN" generated from the control logic. When the instruction from the original instruction set of the microprocessor is executed, the controller sets T_EN to '0', which causes the corresponding binary code to be latched in the Latch unit and the data from the normal path to be passed on to the register file (RF). As the other case, when the test instruction is applied, the controller sets accordingly

T_EN to ‘1’ to propagate the already-latched value to the specified “rd” register in the register file.

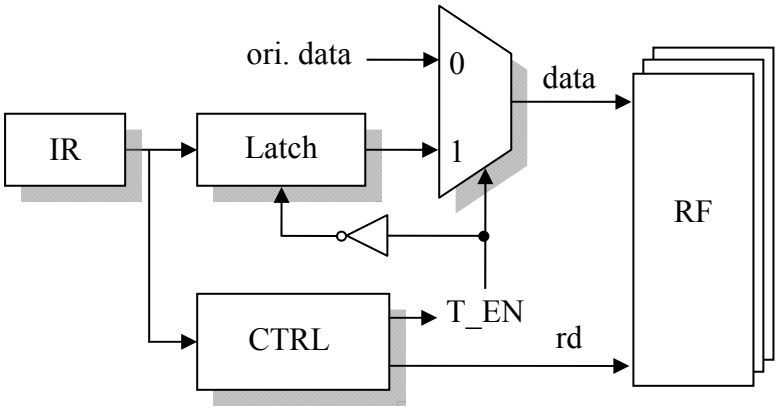


Figure 6-10: Hardware implementation for the new test instruction

6.4.3 Results and Analysis

To realize the aforementioned scheme for the test instruction of the IR, we firstly modify the compiler and relevant tools, e.g. disassembler for dumping a compiled code into the memory file. Then we alter the VHDL files of the microprocessor to incorporate the test logic, and synthesize the design with the DFT into the gate-level netlist. Table 6-7 presents the comparison in terms of hardware overhead. We label the result for the original design as “Ori.”, while that for the design with the test instruction is named as “IL-DFT”. Compared with the original design containing 2,532 equivalent logic AND gates, our DFT implementation requires only 5.3% increase of the gate count.

Table 6-7: Comparison in Area Overhead

	Ori.	IL-DFT
# of gates	2,532	2,666
Ratio	1.0	1.053

Afterwards, to evaluate the effectiveness of the new instruction, we implemented a test program for the IR, and fault simulate this same code both on two designs. Note that the original microprocessor basically treats our new instruction as an unimplemented instruction (functionally equivalent to NOPs), whereas for the DFT design, this instruction will activate the path for the IR test. Figure 6-11 illustrates a notably higher fault coverage for the design

with the test instruction, which reaches 85.32%, while the primary coverage for the original is only 73.85%.

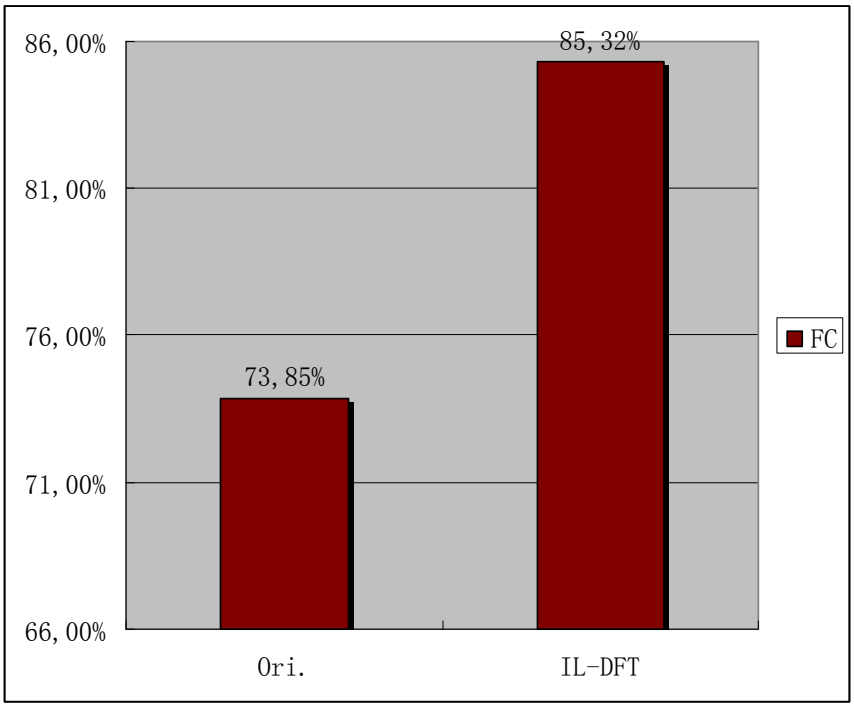


Figure 6-11: Comparison in Fault Coverage for the Instruction Register

This experiment clearly shows the effectiveness of the test instructions. Due to lack of appropriate tools, in our case study, we almost have to manually cope with modifications both at the software and hardware sides by ourselves. However, in ASIPs, an amount of efforts in implementation of the test instruction and relevant hardware can be saved, as ASIPs provide already powerful and automated functionalities in this regard.

6.5 Conclusion

In this chapter, we discuss practical aspects regarding application of the proposed software-based self-test scheme to some public processor cores. In our first case study, we synthesize test programs for an academic processor core, and optimize the routines such that memory footprints, test lengths, and power/energy consumptions are reduced with minimal, or if possible, no impact on fault coverage. As to the second case study, we concentrate on a pipelined core and exhibit the way to further improve test quality of our self-test approach by consideration of the test for the control logic. According to the experiments, our approach

results in a notably shorter test length, smaller memory requirements and better fault coverage than the usual method which generates a test program for the control logic by enumeration of all valid instructions together with addressing modes. In our experiment on the Leon microprocessor, we implement the scheme to test its on-chip user-defined peripherals. At last, we detail the experiment on instruction-level DFT, which defines a new instruction to increase testability of the target microprocessor. The result indicates that a notable increase of fault coverage can be achieved with a slight area overhead.

Chapter 7 Conclusion

In this chapter, we firstly summarize the work done in the scope of this thesis. Then, we point out the aspects worthy of extensive research efforts in the future so as to make the software-based scheme more adequate and effective for testing microprocessors during manufacture as well in-field operation.

7.1 Summary of Our Work

The rapid increase of complexity and performance of modern microprocessors causes the test economics a critical issue in production. The conventional practice of external testers during manufacturing, i.e. ATE-based test, is susceptible to the cost-and-quality influences. The external test of a microprocessor with millions of transistors and the gigahertz operating frequency usually demands extremely high expense for an ATE with vast memory volumes and fast speed, while the test quality is far from satisfaction due to the widened frequency gap between the tester and the testee.

Under this situation, self-test is proposed as an alternative to solve the facing problems. The major research efforts of this area were spent on the hardware aspect, which incorporated the circuit under test with extra dedicated hardware for the test purpose. Although such a method exhibits its effectiveness in testing random logic, it is not appropriate for microprocessors due to the feature of random-pattern resistance. Moreover, as the structures of microprocessors are well optimized to meet specific requirements, design modifications to incorporate additional logic are not desirable. Recently, the focuses in the self-test area are cast on the software side as well, which utilize the processor's instruction set as a means to deliver pre-designed test patterns. Due to its non-intrusive and functional nature, the software-based self-test scheme

can avoid design modification and reduce power consumption during test. In addition, the generated test programs can be reused in the whole system life cycle.

Our work also looks into the software-based self-test and proposes novel methods to address the pertinent issues on application of the scheme to modern microprocessor designs.

1) Automatic test program generation:

The primary objective is to synthesize test programs in a way that high test quality is obtained while minimal engineering efforts are involved. Accordingly, our method achieves high structural fault coverage with use of deterministic test pattern, and automation of the test program generation process is realized on the basis of templates, which map the low level patterns into corresponding instruction sequences for the test purpose. A wide variety of processor architectures are considered here, from the non-pipelined structure to the pipelined, and from the general-purpose processor to the application-specific instruction processor which allows the core user to configure the structure to meet specific requirements of the target application domain.

2) Test-oriented program optimization:

In view of the broad use of microprocessors in embedded systems where memory, execution time and power consumption are crucial factors, we propose a test-oriented optimization algorithm which reduces the memory, power consumption and test application time of the generated test programs without sacrifice of the structural fault coverage. In order to minimize the number of deterministic patterns to be applied, we firstly compact the test set with special care paid to ensure the same fault detection capability. The generated programs are then optimized through instruction re-scheduling and unused bit specification. Different from the existing program optimization techniques, our main objective is to retain the fault coverage rather than the program semantics. Although minimization of power consumption is here one of our targets, the proposed method is applicable to the opposite scenario, i.e. the stress testing with the intention to cause as many as possible circuit node switches.

3) Test quality improvement through instruction extension

We exhibit the way to improve test quality, namely the fault coverage for the hard-to-test components, through instruction extension. The idea itself conforms to the application-

specific instruction-set processor (ASIP), and therefore the process for introducing test instructions can be greatly eased under the ASIP framework.

In the experiments, first of all, we applied the power-, memory-optimized self-test scheme to a 32-bit RISC processor, called Hapra, which is developed mainly for the educational purpose. The results indicate that this methodology achieves high structural fault coverage by targeting at a subset of modules of the processor. The program optimization algorithm reduces the memory requirement, the test application time, and energy/power consumption without fault coverage penalty.

The second case study on a public-accessible processor core, namely Plasma, which implements the most of MIPS I instruction set and has a pipelined structure. The primary experiment confirms again the effectiveness of the proposed approach in detecting faults within the data-path modules, and however shows that coverage for the control logic needs to be enhanced. For this reason, we extend our initial method by targeting at control logic during test program synthesis. Compared with the common way for testing the control logic in the software-based schemes which are based on exhaustive use of instructions together with addressing modes, our approach achieves notably better results in terms of small memory, short test length and high fault coverage.

The third case study, we take into account the microprocessor called Leon, which is in line with the SPARC V8 architecture. A few of modules of the Leon processor are configurable such that the core user can specify the parameters to meet particular design requirements. In addition, with the flexible AMBA bus architecture, the functionality of the Leon can be extended through inclusion of user-defined IP blocks. In our experiment, we implement the scheme to test the on-chip user-defined peripherals and the results demonstrate the benefits of the proposed scheme in terms of low hardware overhead and at-speed test.

In addition, to show the strength of test improvement through instruction extension, we present our case study which introduces a new instruction to forward the output of the instruction register to the register file. A significant improvement of fault coverage is achieved with slight increase of hardware.

7.2 Future Work

Despite the positive attributes of the methods proposed here, there are still quite a few of challenging issues worthy of further research efforts so as to bring the software-based self-test scheme to a more realistic utilization.

✓ Scalability

Most of the present research in SBST is limited to comparably small-scaled microprocessors. It is still an open question how to manage the complexity of the SBST when scaling to processors with millions or even billions of transistors, and with sophisticated architectures, e.g. superscalar, and features like out-of-order execution. All of these properties are challenging both the test program synthesis and the program optimization.

✓ Automation

A certain degree of automation for test program generation is achieved with the use of templates. However, the template structures are strongly linked to specific modules under particular processor architectures. It is challenging yet very significant to generalize the template structures so that they are applicable to a variety of processors with minimal alterations. This aspect is even important for the ASIPs where configurations and user-defined extensions are commonplace.

✓ Characterization of test quality in the ASIP environment

With the prevailing focuses on performance and power consumption, the existing ASIP environments are usually equipped with the functionality to characterize features of the target applications in these two aspects. If test quality is aimed as well, it is necessary to extend the ASIP environment to characterize test-relevant properties.

✓ Test-oriented design space exploration in ASIP

With the above mentioned functionality of the extended ASIP environment, we can further strengthen its capability in design space exploration with the main task to propose promising instruction candidates for testability improvement of the target microprocessor.

Chapter 8 References

[ABF90] M. Abramovici, M. Breuer, and A. Friedman, “Digital Systems Testing and Testable Design”, New York: Computer Science Press (W. H. Freeman and Co.), 1990, ISBN 0-7167-8179-4

[AFK88] M. S. Abadir, J. Ferguson, and T. Kirkland, “Logic Design Verification via Test Generation”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 7, 1988, pp. 138 – 148

[Alid94] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, “Precomputation-based Sequential Logic Optimization for Low Power”, IEEE Transactions on VLSI Systems, Vol. 2, No. 4, Dec. 1994, pp. 426 – 436

[Bade06a] N. Badereddine, P. Girard, S. Pravossoudovitch, C. Landrault, A. Virazel, and H.-J. Wunderlich, Minimizing Peak Power Consumption during Scan Testing: Test Pattern Modification with X Filling Heuristics, Proceedings of the Conference on Design & Test of Integrated Systems in Nanoscale Technology (DTIS), Sep. 5 -7, 2006, pp. 359 - 364

[Bade06b] N. Badereddine, P. Girard, S. Pravossoudovitch, C. Landrault, A. Virazel, and H.-J. Wunderlich, "Structural-Based Power-Aware Assignment of Don't Cares for Peak Power Reduction during Scan Testing", Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-Soc), Oct. 16 - 18, 2006, pp. 403 - 408

[BaPa99] K. Batcher, and C. Papachristou, “Instruction Randomization Self Test for Processor Cores”, Proceedings of IEEE VLSI Test Symposium (VTS), 1999, pp. 24 – 40

[Bell82] C. Bellon, A. Liothin, S. Sadier, G. Saucier, R. Velayco, F. Grillot, and M. Issenman, "Automatic Generation of Microprocessor Test Programs", Proceedings of the 19th Design Automation Conference (DAC), 1982, pp. 566 – 573

[BeMi95a] L. Benini, and G. D. Micheli, "Transformation and Synthesis of FSMs for Low Power Gated Clock Implementation", Proceedings of the International Symposium on Low Power Design, April 1995, pp. 21 – 26

[BeMi95b] L. Benini, and G. D. Micheli, "State Assignment for Low Power Dissipation", IEEE Journal of Solid-State Circuits, Vol. 30 , No. 3, Mar. 1995, pp. 258 – 267

[Beni99] L. Benini, G. Paleologo, A. Bugliolo, and G. De. Micheli, "Policy Optimization for Dynamic Power Management", IEEE Transactions on Computer-Aided Design, Vol. 18, No. 6, June 1999, pp. 813 – 833

[Bern06] P. Bernardi, E. Sanchez, M. Schillaci, G. Squillero, and M. S. Reorda, "An Effective Technique for Minimizing the Cost of Processor Software-Based Diagnosis in SoCs", Proceedings of Design, Automation and Test in Europe (DATE), 2006, pp. 412 – 417

[Bern07] P. Bernardi, M. Grosso, E. Sanchez, and M. S. Reorda, "On the Automatic Generation of Test Programs for Path-Delay Faults in Microprocessor Cores", Proceedings of the 12th IEEE European Test Symposium (ETS), 2007, pp. 179 – 184

[BFR94] L. Benini, M. Favalli, and B. Ricco, "Analysis of Hazard Contribution to Power Dissipation in CMOS IC's", Proceedings of the 1994 International Workshop on Low Power Desing, April 1994, pp. 27 – 32

[BhWa01] N. Bhattacharyya, A. Wang, "Automatic Test Generation for Micro-Architectural Verification of Configurable Microprocessor Cores with User Extensions", Proceedings of the 6th IEEE International High-Level Design Validation and Test Workshop (HLDVT'01), 2001, pp. 14 – 15

[BGK89] F. Brglez, C. Gloster, and G. Kedem, "Hardware-Based Weighted Random Pattern Generation for Boundary Scan", IEEE International Test Conference (ITC), 1989, pp. 264 – 274

[BMS87] H. Bardell, W. H. McAneey, and J. Savir, "Built-In Test for VLSI, Wiley-Interscience", New York, 1987

[BrAb84] D. Brahme, and J. A. Abraham, "Functional Testing of Microprocessors", IEEE Transactions on Computers, Vol. C-33, 1984, pp. 475 – 485

[BuAg00] M. L. Bushnell, and V. D. Agrawal, “Essentials of Electronic Testing”, Kluwer Academic Publishers, 2000, ISBN: 978-0-7923-7991-1

[Burc93] R. Burch, F. Najm, P. Yang, and T. Trick, “A Monte Carlo Approach for Power Estimation”, IEEE Transactions on VLSI, Vol. 1, No. 1. Mar. 1993, pp. 63 – 71

[Butl04] K. M. Butler, J. Saxena, T. Fryars, and G. Hetherington, "Minimizing Power Consumption in Scan Testing: Pattern Generation and DFT Techniques", Proceedings of the International Test Conference (ITC'04), 2004, pp. 355 - 364

[CBD01] L. Chen, X. Bai, and S. Dey, “Testing for Interconnect Crosstalk Defects Using On-Chip Embedded Processor Cores”, Proceedings of Design Automation Conference (DAC), 2001, pp. 317 – 320

[ChBr95] A. P. Chandrakasan, and R. W. Brodersen, “Minimizing Power Consumption in Digital CMOS Circuits”, Proceedings of the IEEE, Vol. 83, No. 4, April 1995, pp. 498 – 523

[ChCh01] K.-W. Choi, and A. Chatterjee, “Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems”, Proceedings of the 14th International Symposium on System Synthesis (ISSS), 2001, pp. 147 – 152

[ChDa94] S. Chakravarty, and V. Dabholkar, “Minimizing Power Dissipation in Scan Circuits During Test Application”, Proceedings of IEEE International Workshop on Low Power Design, 1994, pp. 51 – 56

[ChDe00] L. Chen, and S. Dey, “DEFUSE: A Deterministic Functional Self-Test Methodology for Processors”, Proceedings of the 18th IEEE VLSI Test Symposium, April 2000, pp. 255 – 262

[ChDe01] L. Chen, and S. Dey, “Software-Based Self-Testing Methodology for Processor Cores”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 3, March 2001, pp. 369 – 380

[ChDe02] L. Chen, and S. Dey, “Software-Based Diagnosis for Processors”, Proceedings of Design Automation Conference (DAC), June 10 – 14, 2002, pp. 259 – 262

[Chen03] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, “A Scalable Software-Based Self-Test Methodology for Programmable Processors”, Proceedings of Design Automation Conference (DAC), 2003, pp. 548 – 553

[ChGa99] C. Chakrabarti, and D. Gaitonde, “Instruction Level Power Model of Microcontrollers”, Proceedings of the IEEE International Symposium on Circuits and Systems, 1999, pp. 76 – 79

[CIR87] J. L. Carter, V. S. Iyengar, and B. K. Rosen, “Efficient Test Coverage Determination for Delay Faults”, Proceedings of International Test Conference (ITC), Sept. 1987, pp. 418 – 427

[Ciri87] M. A. Cirit, “Estimating Dynamic Power Consumption of CMOS Circuits”, Proceedings of IEEE International Conference on Computer-Aided Design, Nov. 1987, pp. 534 – 537

[Corn97] F. Corno, P. Prinetto, M. Rebaudengo, and M. S. Reorda, “New Static Compaction Techniques of Test Sequences for Sequential Circuits”, the Proceedings of the 1997 European Conference on Design and Test, 1997, pp. 37 – 43

[Corn01] F. Corno, M. S. Reorda, G. Squillero, and M. Violante, “On the Test of Microprocessor IP Cores”, Proceedings of Design, Automation and Test in Europe (DATE), 2001, pp. 209 – 213

[Corn02] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, “Automatic Test Program Generation from RT-Level Microprocessor Description”, Proceedings of International Symposium on Quality Electronic Design, 2002, pp. 120 – 125

[Corn03] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, “Fully Automatic Test Program Generation for Microprocessor Cores”, Proceedings of Design, Automation and Test in Europe (DATE), 2003, pp. 11006 – 11011

[Corn04] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, “Automatic Test Program Generation: A Case Study”, IEEE Design & Test of Computers, Vol. 21, No. 2, 2004, pp. 102 – 109

[Cric79] G. Crichton, “Testing Microprocessors”, IEEE Journal of Solid-State Circuits, Vol. SC-14, No. 3, June 1979, pp. 609 – 613

[CRS03] F. Corno, M. S. Reorda, and G. Squillero, “Automatic Test Program Generation for Pipelined Processors”, Proceedings of the 18th Annual ACM Symposium on Applied Computing, 2003, pp. 736 – 740

[Cuvi99] M. Cuviello, S. Dey, X. Bai, and Y. Zhao, “Fault Modeling and Simulation for Crosstalk in System-on-chip Interconnects”, Proceedings of the International Conference on Computer-Aided Design (ICCAD), Nov. 1999, pp. 297 – 303

[Dahb98] V. Dabholkar, S. Chakravarty, I. Pomeranz, and S. M. Reddy, "Techniques for Reducing Power Dissipation During Test Application in Full Scan Circuits", IEEE Transactions on Computer-Aided Design, vol. 17, no. 12, 1998, pp. 1325 – 1333

[Ding98] C.-S. Ding. Q. Wu, C.-T. Hsiesh, and M. Pedram, "Stratified Random Sampling for Power Estimation", IEEE Transactions on VLSI, Vol. 17, No. 6, June 1998, pp. 465 – 471

[EiLi83] B. Eichelberger, and E. Lindbloom, "Random pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test", IBM Journal of Research and Development, Vol. 27, No. 3, May 1983, pp. 265 – 272

[Eldr59] R. D. Eldred, "Test Routines Based on Symbolic Logical Statements", in J. Assoc. Comput. Mach., Vol. 6, No. 1, 1959, pp. 33 – 36

[GBT03] S. Ghosh, S. Basu, and N. A. Touba, "Joint Minimization of Power and Area in Scan Testing by Scan Cell Reordering", Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03), 2003, pp. 246 – 249

[GeWu97] S. Gerstendoerfer and H.-J. Wunderlich, "Minimized Power Consumption for Scan-Based BIST", Proceedings of the International Test Conference (ITC), 1999, pp. 77 – 84

[Gira98] P. Girard, C. Landrault, S. Pravossoudovitch, and D. Severac, "Reducing Power Consumption during Test Application by Test Vector Ordering", Proceedings of International Symposium Circuits and Systems (ISCAS 98), Part II, IEEE CS Press, 1998, pp. 296 – 299

[Gira99] P. Girard, L. Guiller, C. Landrault, and S. Pravossoudovitch, "A Test Vector Ordering Technique for Switching Activity Reduction during Test Operation", Proceedings of 9th Great Lakes Symposium on VLSI (GLS-VLSI 99), 1999, pp. 24 – 27

[Gira01] P. Girard, L. Guiller, C. Landrault, S. Pravossoudovitch, and H.-J. Wunderlich, "A Modified Clock Scheme for a Low Power BIST Test Pattern Generator", the Proceedings of the 19th IEEE VLSI Test Symposium (VTS), 2001, pp. 306 – 311

[GoVe92] A. J. van de Goor, and Th. J. W. Verhallen, “Functional Testing of Current Microprocessors (applied to the Intel i860TM)”, Proceedings of International Test Conference (ITC), 1992, pp. 684 – 695

[GPZ04] D. Gizopoulos, A. Paschalis, and Y. Zorian, “Embedded Processor-Based Self-Test”, Kluwer Academic Publishers, ISBN 1-4020-2785-0

[Hapra] http://www.iti.uni-stuttgart.de/~imhofml/hapra08

[Hata97] K. Hatayama, K. Hikone, T. Miyazaki, and H. Yamada, “A Practical Approach to Instruction-Based Test Generation for Functional Modules of VLSI Processors”, Proceedings of the 15th IEEE VLSI Test Symposium (VTS), 1997, pp. 17 – 22

[Hatz05] M. Hatzimihail, M. Psarakis, G. Xenoulis, D. Gizopoulos, and A. M. Paschalis, “Software-Based Self-Test for Pipelined Processors: A Case Study”, Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT), 2005, pp. 535 – 543

[Henk03] J. Henkel, “Closing the SoC Design Gap”, Computer, Vol. 36, No. 9, Sep. 2003, pp. 119 – 121

[Hira03] T. Hiraide, K. O. Boateng, H. Konishi, K. Itaya, M. Emori, H. Yamanaka, and T. Mochiyama, "BIST-Aided Scan Test - A New Method for Test Cost Reduction", Proceedings of VLSI Test Symposium (VTS'03), 2003, pp. 359 - 364

[HeWu94] S. Hellebrand, H.-J. Wunderlich, “Synthesis of Self-Testable Controllers”, Proceedings of European Design Automation Conference (EDAC/ETC/EuroAsic), Mar. 1994, pp. 580 – 585

[HWH96] S. Hellebrand, H.-J. Wunderlich, and A. Hertwig, “Mixed-Mode BIST Using Embedded Processors”, Proceedings of the IEEE International Test Conference (ITC), 1996, pp. 195 – 204

[ITRS03] The International Technology Roadmap for Semiconductors (ITRS), 2003 Edition, http://www.itrs.net/Links/2003ITRS/Home2003.htm

[ITRS06] The International Technology Roadmap for Semiconductors (ITRS), 2006 Edition, http://www.itrs.net/links/2006Update/2006UpdateFinal.htm

[JaVe00] M. F. Jacome, and G. de Veciana, "Design Challenges for New Application-Specific Processors", IEEE Design & Test of Computers, Vol. 17, No. 2, Apr. 2000, pp. 40 – 50

[KaMe84] M. G. Karpovsky, and R. G. van Meter, "An Approach to the Testing of Microprocessors", Proceedings of the 21st Design Automation Conference, 1984, pp. 196 – 202

[KaRa00] R. Kannah, and C. P. Ravikumar, "Functional Testing of Microprocessors with Graded Fault Coverage", Proceedings of the 9th Asian Test Symposium, 2000, pp. 204 – 208

[Karp81] M. G. Karpovsky, "An Approach for Fault-Detection and Fault-Correction in Distributed Systems Computing Numerical Functions", IEEE Transactions on Computers, Vol. C-30, No. 12, Dec. 1981, pp. 947 – 954

[Kran02a] N. Kranitis, A. M. Paschalis, D. Gizopoulos, Y. Zorian, "Instruction-Based Self-Testing of Processor Cores", Proceedings of IEEE VLSI Test Symposium (VTS), 2002, pp. 223 – 228

[Kran02b] N. Kranitis, A. M. Paschalis, D. Gizopoulos, Y. Zorian, "Effective Software Self-Test Methodology for Processor Cores", Proceedings of Design, Automation and Test in Europe (DATE), 2002, pp. 592 – 597

[Kran03a] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Low-Cost Software-Based Self-Testing of RISC Processor Cores", Proceedings of Design, Automation and Test in Europe (DATE), 2003, pp. 10714 – 10719

[Kran03b] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores", Proceedings of the International Test Conference (ITC), 2003, pp. 431 – 440

[Kran05] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-Based Self-Testing of Embedded Processors", IEEE Transactions on Computers, Vol. 54, No. 4, April 2005, pp. 461 – 475

[Kran06] N. Kranitis, A. Merentitis, N. Laoutaris, G. Theodorou, A. M. Paschalis, D. Gizopoulos, and C. Halatsis, "Optimal Periodic Testing of Intermittent Faults in Embedded Pipelined Processor Applications", Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2006, pp. 65 – 70

[KrCh98] A. Krstic, and K.-T. Cheng, "Delay Fault Testing for VLSI Circuits", Boston: Kluwer Academic Publishers, 1998, ISBN: 978-0-7923-8295-9

[Krst02a] A. Krstic, L. Chen, W.-C. Lai, K.-T. Cheng, and S. Dey, "Embedded Software-Based Self-Test for Programmable Core-Based Designs", IEEE Design and Test of Computers, Vol. 19, 2002, pp. 18 – 27

[Krst02b] A. Krstic, W.-C. Lai, L. Chen, K.-T. Cheng, and S. Dey, "Embedded Software-Based Self-Testing for SoC Design", Proceedings of 39th Design Automation Conference (DAC), 2002, pp. 355 – 360

[LaCh01] W.-C. Lai, and K.-T. Cheng, "Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip", Proceedings of the 38th Conference on Design Automation (DAC), 2001, pp. 59 – 64

[LDM05] Z. Liu, K. Dickson, and J. V. McCanny, "Application-Specific Instruction Set Processor for SoC Implementation of Modern Signal Processing Algorithms", IEEE Transactions on Circuits and Systems, Vol. 52, No. 4, Apr. 2005, pp. 755 – 765

[Lee97] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software", IEEE Transactions on VLSI Systems, Vol. 5, No. 1, Mar. 1997, pp. 123 – 135

[Lee00] C. Lee, J. K. Lee, T. T. Hwang, and S.-C. Tsai, "Compiler Optimization on Instruction Scheduling for Low Power", Proceedings of the 13th International Symposium on System Synthesis, 2000, pp. 55. – 60

[Lee01] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, "An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors", Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN) workshop on Language, Compilers and Tool Support for Embedded Systems, 2001, pp. 1 – 10

[Lee03] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler Optimization on VLIW Instruction Scheduling for Low Power", Transactions on Design Automation of Electronic Systems, Vol. 8, No. 2, Apr. 2003, pp. 252 – 268

[Leon] http://www.gaisler.com/cms/

[Leon2] Leon2 User's Manual, http://www.gaisler.com/doc/leon2-1.0.30-xst.pdf

[LeSa00] S. Lee, and T. Sakurai, "Run-time Power Control Scheme Using Software Feedback Loop for Low-power Real-time Applications", Proceedings of Asia South-Pacific Design Automation Conference, Jan. 2000, pp. 381 – 386

[LHC01] W.-C Lai, J.-R. Huang, and K.-T. Cheng, "Embedded-Software-Based Approach to Testing Crosstalk-Induced Faults at On-Chip Buses", Proceedings of IEEE VLSI Test Symposium (VTS), 2001, pp. 204 – 209

[LiHo88] C.-S. Lin, and H.-F. Ho, "Automatic Functional Test Program Generation for Microprocessors", Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 605 – 608

[LKC00a] W.-C. Lai, A. Krstic, and K.-T. Cheng, "On Testing the Path Delay Faults of a Microprocessor Using Its Instruction Set", Proceedings of VLSI Test Symposium (VTS), 2000, pp. 15 – 22

[LKC00b] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test Program Synthesis for Path Delay Faults in Microprocessor Cores", Proceedings of the International Test Conference (ITC), 2000, pp. 1080 – 1089

[MAH96] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "A Unified Framework for Design Validation and Manufacturing Test", Proceedings of International Test Conference, 1996, pp. 875 – 884

[MaAg97] A. K. Majhi, and V. D. Agrawal, "Tutorial: Delay Fault Models and Coverage", Proceedings of 11th International Conference on VLSI Design: VLSI for Signal Processing, 1998, pp. 364 – 369

[Majh96] A. K. Majhi, J. Jacob, L. M. Patnaik, and V. D. Agrawal, "On Test Coverage of Path Delay Faults", Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication, 1996, pp. 418 – 421

[Mini] miniMIPS CPU Core, available at: http://www.opencores.org/projects/minimips

[MMP94] R. Marculescu, D. Marculescu, and M. Pedram, "Switching Activity Analysis Considering Spatiotemporal Correlations", Proceedings of the International Conference on Computer Aided Design (ICCAD), 1994, pp. 294 – 299

[MMP99] R. Marculescu, D. Marculescu, and M. Pedram, "Sequence Compaction for Power Estimation: Theory and Practice", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 7, Jul 1999, pp. 973 – 993

[MoDe96] J. Monteiro, and S. Devadas, "Techniques for Power Estimation and Optimization at the Logic Level: A Survey", Journal of VLSI Processing Systems, Vol. 13, Issue 2 – 3, Aug./Sept. 1996, pp. 259 – 276

[MoZo00] S. Mourad, and Y. Zorian, "Principles of Testing Electronic Systems", John Wiley & Sons, Inc., 2000, ISBN 0-471-31931-7

[Nair79] R. Nair, "An Optimal Algorithm for Testing Stuck-at Faults Random Access Memories", IEEE Transactions on Computers, vol. C-28, no. 3, 1979, pp. 258 – 261

[Najm91] F. N. Najm, "Transition Density, a Stochastic Measure of Activity in Digital Circuits", Proceedings of the 28th Conference on ACM/IEEE Design Automation, 1991, pp. 644 – 649

[Najm94] F. N. Najm, "A Survey of Power Estimation Techniques in VLSI Circuits", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No. 4, Dec. 1994, pp. 446 – 455

[NeMe97] W. Nebel, and J. Mermet, "Low Power Design in Deep Submicron Electronics", Kluwer Academic Publishers, 1997

[OlKa94] E. Olson, and S. M. Kang, "Low-Power State Assignment for Finite State Machines", Proceedings of International Workshop on Low Power Design, 1994, pp. 63 – 68

[OPEN] Plasma: http://www.opencores.org/projects.cgi/web/mips/overview

[Pari04] A. Parikh, S. Kim, M. Kandemir, M. Vijaykrishnan, and M.J. Irwin, "Instruction Scheduling for Low Power", Journal of VLSI Signal Processing, Vol. 37, No. 1, May 2004, pp. 129 – 149

[Pate98] J. H. Patel, "Stuck-at Fault: A Fault Model for the Next Millennium", Proceedings of International Test Conference (ITC), 1998, pp. 1166

[Pedr01] M. Pedram, "Power Optimization and Management in Embedded Systems", Proceedings of the Design Automation Conference, 2001, pp. 239 – 244

[Pedr96] M. Pedram, "Power Minimization in IC Design: Principles and Applications", ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 1, No. 1, 1996, pp. 3 – 56

[PEK00] M. Puig-Medina, G. Ezer, and P. Konas, “Verification of Configurable Processor Cores”, Proceedings of Design Automation Conference (DAC), 2000, pp. 426 – 431

[PeOr03] P. Petrov, and A. Orailoglu, “Compiler-Based Register Name Adjustment for Low-Power Embedded Processors”, Proceedings of the International Conference on Computer Aided Design (ICCAD), 2003, pp. 523 – 527

[PeOr04] P. Petrov, and A. Orailoglu, “Low-Power Instruction Bus Encoding for Embedded Processors”, IEEE Transactions on VLSI Systems, Vol. 12, No. 8, Aug. 2004, pp. 812 – 826

[PeRa02] M. Pedram, and J. M. Rabaey, “Power Aware Design Methodologies”, Kluwer Adademic Publishers, 2002, ISBN: 1-4020-7152-3

[PeVa97] M. Pedram, and H. Vaishnav, “Power Optimization in VLSI Layout: A Survey”, Journal of VLSI Signal Processing Systems, Vol. 5, No. 3, March 1997, pp. 221 – 232

[Plasma] Plasma – most MIPS I (TM) opcodes: Overview, http://www.opencores.org/projects.cgi/web/mips/overview

[Prim] “Data Sheet: PrimePower Full-Chip Dynamic Power Analysis for Multimillion-Gate Design”, Synopsys, Inc.

[Psar06] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, Proceedings of Design Automation Conference (DAC), 2006, pp. 393 – 398

[RaGu00] D. Ramanathan, and R. Gupta, “System Level Online Power Management Algorithms”, Design, Automation and Test in Europe, 2000, pp. 606 – 611

[RaPe96] J. Rabaey, and M. Pedram, “Low Power Design Methodologies”, Kluwer Academic Publishers, 1996

[Reme06] S. Remersaro, X. Lin, Z. Zhang, S. M. Reddy, I. Pomeranz, and J. Rajski, "Preferred Fill: A Scalable Method to Reduce Capture Power for Scan Based Designs", Proceedings of International Test Conference (ITC'06), 2006, pp. 1 - 10

[Rimo06] M. Rimon, Y. Lichtenstein, A. Adir, I. Jaeger, M. Vinov, S. Johnson, and D.Jani, “Addressing Test Generation Challenges for Configurable Processor Verification”, IEEE International High Level Design and Test Workshop, 2006, pp. 95 – 101

[Sami00] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "Instruction-level Power Estimation for Embedded VLIW Cores", Proceedings of the International Conference on Hardware Software Codesign, 2000, pp. 34 – 38

[Saxe03] J. Saxena, K. M. Butler, V. B. Jayaram, S. Kundu, N. V. Arvind, P. Sreeprakash, and M. Hachinger, "A Case Study of IR-Drop in Structured At-Speed Testing", Proceedings of International Test Conference (ITC), 2003, pp. 1098 – 1104

[SCB96] M. Srivastava, A. Chandrakasan, and R. Brodersen, "Predictive System Shut-Down and Other Architectural Technique for Energy Efficient Programmable Computation", IEEE Transactions on VLSI Systems, Vol. 4, No. 1, Mar. 1996, pp. 42 – 55

[Schu02] M. Schuller, "Study of the Switching Activity of RISC-Processors Exemplified by the Leon-Processor", Thesis No. 2042, 2002, Faculty of Computer Science, University of Stuttgart, Germany (in German)

[SeLe03] George A. F. Seber, and Alan J. Lee, "Linear Regression Analysis, 2nd Edition", Wiley & Sons, ISBN: 978-0471415404

[ShAb98] J. Shen, and J. A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation", Proceedings of International Test Conference (ITC), 1998, pp. 990 – 999

[Shiu01] W.-T. Shiue, "Retargetable Compilation for Low Power", Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES), 2001, pp. 254 – 259

[ShSu88] L. Shen, and S. Y. H. Su, "A Functional Testing Method for Microprocessors", IEEE Transactions on Computers, Vol. 37, No. 10, 1988, pp. 1288 – 1293

[SiCh02] A. Sinha, and A. Chandrakasan, "Software Energy Profiling", chapter 17 of the book "Power Aware Computing", Kluwer Academic-Plenum Publishers, ISBN 0-306-46786-0, 2002

[Simu01] T. Simunic, L. Benini, P. Glynn, and G. De Micheli, "Event-Driven Power Management", IEEE Transaction on CAD, July 2001, pp. 840 – 857

[Sing03] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Software-Based Delay Fault Testing of Processor Cores", Proceedings of the Asian Test Symposium (ATS), 2003, pp. 68 – 71

[Sing05a] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Delay Fault Testing of Processor Cores in Functional Mode", IEICE Transactions on Information and Systems, Vol. E88 – D, No. 3, Mar. 2005, pp. 610 – 618

[Sing05b] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-Based Delay Fault Self-Testing of Pipelined Processor Cores", IEEE International Symposium on Circuits and Systems (ISCAS), 2005, pp. 5686 – 5689

[Sing05c] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Testing Superscalar Processors in Functional Mode", Proceedings of the International Conference on Field Programmable Logic and Applications, 2005, pp. 747 – 748

[SKL01] D. Shin, J. Kim, and S. Lee, "Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis", Proceedings of 38th Design Automation Conference, Jun. 2001, pp. 438 – 443

[SLW95] W.-Z. Shen, J.-Y. Lin, and F.-W. Wang, "Transistor Reordering Rules for Power Reduction in CMOS Gates", Proceedings of the 1st Asia-Pacific design Automation Conference, Aug. 1995, pp. 1 – 5

[Smal94] C. Small, "Shrinking Devices Put the Squeeze on System Packaging", EDN, vol. 39, no. 4, Feb. 17, 1994, pp. 41 – 46

[SOT00] R. Sankaralingam, R. R. Oruganti, and N. A. Touba, "Static Compaction Techniques to Control Scan Vector Power Dissipation", Proceedings of the 18th IEEE VLSI Test Symposium, 2000, pp. 35 – 40

[SPARCv8] "The SPARC Architecture Manual, Version 8", SPARC International, Inc., available under the URL: http://www.sparc.com/standards/V8.pdf

[SPG02] D. Soudris, C. Piguet, and C. Goutis, "Designing CMOS Circuits for Low Power", Kluwer Academic Publishers, 2002

[STD94] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors", Proceedings of IEEE COMCON, 1994, pp. 489 – 498

[TAM95] V. Tiwari, P. Ashar, and S. Malik, "Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design", International Symposium on Low Power Design, April 1995, pp. 221 – 226

[TAS89] E.-S. A. Talkhan, A. M. H. Ahmed, and A. E. Salama, "Microprocessors Functional Testing Techniques", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 3, 1989, pp. 316 – 318

[TAS00] R. S. Tupuri, J. A. Abraham, and D. G. Saab, “Hierarchical Test Generation for Systems on a Chip”, Proceedings of the 13th International Conference on VLSI Design, 2000, pp. 198 – 203

[TBG05] M. Tuna, M. Benabdenbi, and A. Greiner, “STESI: A New Software-Based Strategy for Testing SoCs Containing Wrapped IP Cores”, Proceedings of the 15th International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES), 2005, pp. 459 – 464

[ThAb80] S. M. Thatte, and J. A. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, Vol. C-29, 1980, pp. 429 – 441

[Tiwa96a] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, “Instruction Level Power Analysis and Optimization of Software”, Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication, 1996, pp. 326 – 328

[Tiwa96b] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, “Instruction Level Power Analysis and Optimization of Software”, Journal of VLSI Signal Processing, Vol. 13, No. 2-3, Aug./Sept. 1996, pp. 223 – 238

[TMW94] V. Tiwari, S. Malik, and A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No. 4, Dec 1994, pp. 437 – 445

[Tomi98] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, “Instruction Scheduling for Power Reduction in Processor-Based System Design”, Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 1998, pp. 855 – 860

[TPD93] C. Tsui, M. Pedram, and A. Despain, “Efficient Estimation of Dynamic Power Dissipation under Real Delay Model”, Proceedings of the International Conference on Computer-Aided Design (ICCAD), 1993, pp. 224 – 228

[Tsui95] C.-Y. Tsui, J. Monteiro, M. Pedram, S. Devadas, and A. Despain, “Power Estimation for Sequential Logic Circuits”, Transactions of VLSI, Vol. 3, No. 3, Sept. 1995, pp. 404 – 416

[Tsui98] C. Tsui, M. Pedram, C. Chen, and A. Despain, “Low Power State Assignment Targeting Two- and Multilevel Logic Implementations”, IEEE transactions on CAD, Vol. 17, No. 12, Dec. 1998, pp. 1281 – 1291

[TuAb97] R. S. Tupuri, and J. A. Abraham, “A Novel Functional Test Generation Method for Processors using Commercial ATPG”, Proceedings of the International Test Conference (ITC), 1997, pp. 743 – 752

[VaPe93] H. Vaishnav, and M. Pedram, “PCUBE: A Performance Driven Placement Algorithm for Low Power Designs”, Proceedings of the European Design Automation Conference, Sept. 1993, pp. 72 – 77

[Waic89] J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, O. P. Forlenza, “A Method for Generating Weighted Random Test Patterns”, IBM Journal of Research and Development, vol. 33, No. 2, March 1989, pp. 149 – 161

[WaRo96] C. Y. Wang, and K. Roy, “Maximum Power Estimation for CMOS Circuits Using Deterministic and Statistical Approaches”, Proceedings of the 9th International Conference on VLSI Design: VLSI Mobile Communication, 1996, pp. 364

[Weis94] M. Weiser, B. Welch, A. Demers, and S. Shenker, “Scheduling for Reduced CPU Energy”, Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, Vol. 1, Article No. 2, 1994, pp. 13 – 23

[Wen05] C. H.-P. Wen, L.-C. Wang, K.-T. Cheng, W.-T. Liu, and C.-C. Chen, “Simulation-Based Target Test Generation Techniques for Improving the Robustness of a Software-Based-Self-Test Methodology”, Proceedings of the IEEE International Test Conference (ITC), 2005, pp. 936 – 945

[Wohl03] P. Wohl, J. A. Waicukauski, S. Patel, and M. B. Amin, "Efficient Compression and Application of Deterministic Patterns in a Logic BIST Architecture", Proceedings of the 40th Conference on Design Automation (DAC'03), 2003, pp. 566 - 569

[Wun90] H.-J. Wunderlich, “Multiple Distributions for Biased Random Test Patterns”, IEEE Transactions on Computer-Aided Design, Vol. 9, No. 6, June 1990, pp. 594 – 602

[Wun98] H.-J. Wunderlich, “BIST for Systems-on-a-Chip”, Integration - The VLSI Journal, Vol. 26, No. 1 – 2 (December 1998), ISSN: 0167 – 9260, pp. 55 – 78

[WWC06] C. H.-P. Wen, L.-C. Wang, and K.-T. Cheng, “Simulation-Based Functional Test Generation for Embedded Processors”, IEEE Transactions on Computers, Vol. 55, No. 11, 2006, pp. 1 – 9

[Zach03] S. T. Yachariah, Y.-S. Chang, S. Kundu, and C. Tirumurti, “On Modeling Cross-Talk Faults”, Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2003, pp. 10490 – 10495

[ZDR00] Y. Zorian, S. Dey, and M. J. Rodgers, “Test of Future System-on-Chips”, proceeding of the 2000 International Conference on Computer-Aided Design, Nov., 2000, pp. 392 – 398

[ZhPa98] W. Zhao, and C. Papachristou, “Testing DSP Cores Based on Self-Test Programs”, Proceedings of Design, Automation and Test in Europe (DATE), 1998, pp. 166 – 172

[ZhWu05] J. Zhou, and H.-J. Wunderlich, “Software-basierender Selbsttest von Prozessorkernen unter Verlustleistungsbeshraenkung”, GI Jahrestagung (1) 2005, pp. 441

[ZhWu06] J. Zhou, and H.-J. Wunderlich, “Software-Based Self-Test of Processors under Power Constraints”, Proceedings of Design, Automation and Test in Europe (DATE), 2006, pp. 430 – 435

[Zori93] Y. Zorian, “A Distributed BIST Control Scheme for Complex VLSI Devices”, Proceedings of IEEE VLSI Test Symposium (VTS), 1993, pp. 4 – 9

Appendix: Author's Biography

Jun Zhou received the bachelor degree in "Computer Science" and the master's degree in "Computer Software and Theory" at Beihang University (former: Beijing University of Aeronautics and Astronautics), P. R. China in July, 2000 and March, 2003 respectively.

She participated in the Jade-Bird Project sponsored by National 863 Program, P. R. China, in which she took part in development of a White-box testing tool for C/C++/Java and developed a tool for testing of component-based software systems.

From May 2003 to May 2008, she was working as teaching assistant and pursuing the doctor degree at Institut für Technische Informatik, Universität Stuttgart, Germany. She was involved in the project „Leistungs- und Energiebeschränkung im Selbsttest" from the program „Grundlagen und Verfahren verlustamer Informationsverarbeitung (VIVA)" sponsored by Deutsche Forschungsgemeinschaft (DFG) under the project number Wu 245 / 2-1.