NAIST-IS-DD0261204

Doctoral Dissertation

Instruction-Based Self-Testing of Performance Oriented Faults in Modern Processors

Virendra Singh

July 25, 2005

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Virendra Singh

Thesis committee: Professor Hideo Fujiwara, (Supervisor)
Professor Minoru Ito, (Member)
Professor Kenichi Matsumoto, (Member)

Dedicated to my
grandmother (Late) Smt. Padma Devi

"karmany evadhikaras te
ma phalesu kadachana
ma karma-phala-hetur bhur
ma te sango 'stv akarmani"

(From the Geeta – in Sanskrit)

English translation

On action alone be thy interest,
Never on its fruits.
Let not the fruits of action be thy motive,
Nor be thy attachment to inaction.

Instruction-Based Self-Testing of Performance Oriented Faults in Modern Processors[*]

Virendra Singh

Abstract

Aggressive microprocessor design methodologies using gigahertz clock and very deep sub-micron technology are necessitating the use of at-speed testing of small and distributed timing defects caused by process variations during manufacturing. A primary objective of such tests is to test for the faults that can lead to performance degradation, such as delay faults. However, at-speed testing using external tester is not an economically viable scheme and hardware BIST leads to unacceptable performance loss and area overhead. A new paradigm, instruction-based self-testing, can alleviate these problems as it uses processor instructions to deliver the test patterns and collect the test responses. Also, it has the ability to link to low level fault models and it is well-suited methodology for testing processors, embedded processor cores, IP cores, and SoCs. Moreover, the same test can also be used for online periodic testing of processors to improve the reliability.

This thesis proposes an instruction-based self-testing methodology for delay fault testing of modern processors in a chronological way by first dealing with non-pipelined processors, and then pipelined processors and superscalar processor architectures.

In order to test a non-pipelined processor a graph theoretic model, called instruction execution graph based on the register transfer level description of the processor is developed. This model, in conjunction with the structural and

[*] Doctoral Dissertation, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0261204, July 25, 2005.

functional information, is used to identify and classify all paths into functionally testable and untestable paths and to generate tests and test instruction sequences that can be applied in functional mode of operation of the processor. The completeness of the test method is guaranteed by extracting constraints for the paths that are testable.

The approach proposed above is expanded to include pipelined architectures. A new graph model, called pipeline instruction execution graph, is defined that captures the effect of executing multiple instructions concurrently. This graph model is then used to generate tests and test sequences for normal as well as forwarding paths between different stages of a pipelined processor.

Finally, the thesis explores path delay testing of superscalar architectures, one of the most complex architectures of the modern processors. Such architectures use out of order execution technique to enhance the throughput, which poses serious challenges to instruction-based testing. This thesis identifies test related issues. It proposes a superscalar processor model, called superscalar instruction execution graph, and provides a method of generating test programs that can force scheduler to execute the instructions in the desired order to test the processor.

The effectiveness of all the above stated approaches has been demonstrated through experimental results for some representative processors.

Keywords:

microprocessor testing, performance oriented faults, at-speed test, functional mode testing, delay fault testing, self-test

List of Publications

Journal Papers

1. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, “Delay Fault Testing of Processor Cores in Functional Mode,” *IEICE Trans. on Information and Systems*, Vol. E-88D, No. 3, pp. 610–618, Mar. 2005.

2. Michiko Inoue, Kazuko Kambe, Virendra Singh, and Hideo Fujiwara, ”Software-Based Self-Test of Processors for Stuck-at Faults and Path Delay Faults,” *Trans. of IEICE (DI)*, Vol. J88-D-I, No.3, pp. 1003–1011, Jun. 2005. (In Japanese)

International Conferences (Reviewed)

1. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, ”Software-Based Delay Fault Testing of Processor Cores,” *Proceedings of the IEEE 12th Asian Test Symposium*, pp. 68–71, Nov. 2003.

2. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, “Instruction-Based Delay Fault Testing of Processor Cores,” *Proceedings of the 17th International Conference on VLSI Design*, pp. 933–938, Jan. 2004.

3. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, ”Instruction-Based Delay Fault Self-Testing of Pipelined Processor Cores,” *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 5686–5689, May. 2005.

4. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, “Program-Based Testing of Superscalar Microprocessors ,” *Proceedings of the IEEE 14th North Atlantic Test Workshop*, pp. 79–86,May 2005.

5. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, ”Testing Superscalar Processors in Functional Mode,” *Proceedings of the 15th International Conference on Field Programmable Logic and Applications*, pp. 747–748, Aug. 2005.

Other Publications

1. Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara, “Program-Based Delay Fault Self-Testing of Processor Cores,” *Technical Report of IEICE (DC2003-37)*, Vol. 103, No. 476, pp. 19–24, Nov. 2003.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *".... Productivity increases as quality improves ... [but] inspection to improve quality is too late, ineffective, costly ... note that there are exceptions, circumstances in which mistakes and duds are inevitable but intolerable. An example is, I believe, manufacture of complicated integrated circuits. Separation from good ones from bad ones is the only way out ... It is important to carry out inspection at the right point for minimum total cost."*
>
> — W. Edward Deming, in the book, *Out of the crisis* [37]

The reliability is no longer a concern limited to the military, aerospace, and banking organizations where failure consequences may have catastrophic effect. Reliability and testing techniques have become of increasing interest to all other applications such as computers, telecommunications, consumer products, and automobiles, as today's systems are built around very high-speed processors to meet the consumer's demand of high performance and rich functionality. The expectation of zero failure can only be met if all manufacturing defects are eliminated. A key requirement for obtaining reliable electronic systems is the ability to determine that systems are error-free.

The complexity of VLSI technology has reached the point where we are trying to put billions transistors on a single chip, and implementing these with GHz clock

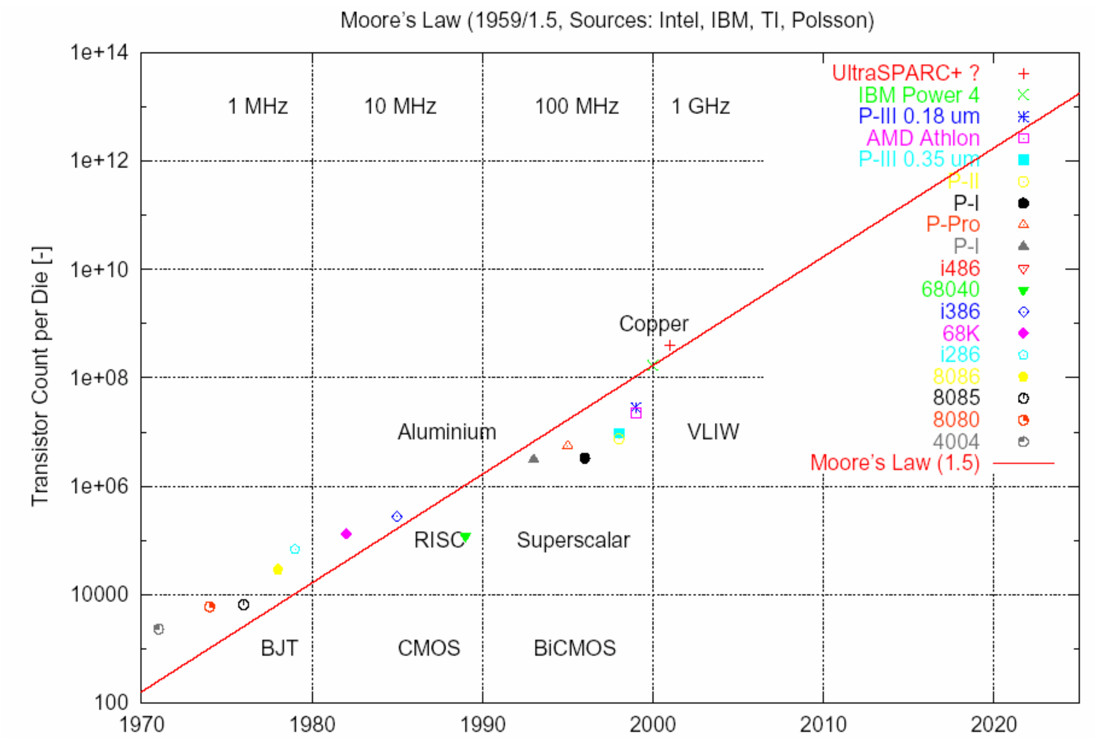


Figure 1.1. Complexity growth

frequency. Transistor feature sizes on a VLSI chip reduce roughly by 10.5% per year, resulting in a transistor density increase of roughly 22.1% every year. An almost equal amount of increase is provided by wafer and chip size increases and circuit design and process annotations. This is evident from Figure 1.1, which shows nearly 44% increase in the transistors on microprocessor chips every year.

According to the prediction of the 2003 International Technology Roadmap for Semiconductors (ITRS) [41], by year 2018 the half pitch of Dynamic Random Access memories (DRAMs), microprocessors and Application Specific ICs (ASIC) will drop to 18 nm and the microprocessor physical gate length will drop to 7nm. The implementation of correctly operating electronic circuits in such a small geometry - usually referred as Very Deep Submicron (VDSM) manufacturing technologies - was believed, just a few years ago, to be extremely difficult, if at all possible, due to hurdles imposed by the fundamental laws of the microelectronics physics when circuit elements are manufactured in such distances separating them. Despite the skeptical views, VDSM technologies are successfully used today to produce high performance circuits and continue providing evidence that Moore's law is still valid.

1.1. Motivation and What to Test for

As the chip density grows beyond millions of gates, VLSI circuit testing becomes a formidable task. A number of companies estimate that about 7% to 10% of the total cost is spent in single device testing [40]. This figure can rise to as high as 20% to 30% if the cost of the in-circuit testing and board-level testing is added. However, the most important cost can be the loss in time-to-market due-to hard to detect faults. Recently, studies show that a six months delay in time-to-market can reduce the profit by 34%. Thus, testing can pose serious problems in VLSI design.

To make testing more manageable, it is thus important to characterize the defects in the circuits, logical or electrical value on the nodes connecting various components of the circuit, that is, to represent failure modes by a logical value. This amounts to representing physical defects by model on logical level. If this mapping is one-to-one, there will be a large number of models to represent. As a model, the fault does not have to be an exact representation of the defects, but rather, be useful in detecting the defects. For example, the most common fault model assumes single stuck-at (SSA) lines even though it is very clear that this model does not accurately represent all actual physical failures. However, despite its popularity, the stuck-at fault is no longer sufficient for present circuits and technologies. Some of the stringent faults that cannot be covered by conventional stuck-type fault models are dielectric, conductor, and metallization failures [42]. These faults manifest themselves as performance oriented faults and remain challenging faults to test.

Aggressive timing requirement of high performance processors have introduced the need to test smaller timing defects and distributed faults caused by process variations. Delay fault testing determines the operational correctness of the circuits at its specified speed. Even if its steady behaviour is correct, it may not be reached in the allotted time. Delay fault testing exposes such circuit malfunctions.

For the proper operation of a circuit, the propagation delay of each sensitizable path in the circuit should be less than a specified limit (usually the clock interval). As the clock interval is gradually decreasing, the need and significance of delay fault testing is gradually increasing. Like all other faults, delay faults are also originated from manufacturing defects. Now, for the sake of the propagation delay in the circuit, it is not feasible to test every manufacturing defects and some sort of modeling is required. Several fault models of delay fault are proposed in the literature such as transition delay, path delay, and segment delay fault models. Some models use lumped delay assumption where as others use distributed delay assumption. Path delay fault model is more close to the reality but the number of paths grows exponentially as circuit size grows.

In some way, microprocessors test is a reflection or superset of the today's industrial test issues. Testing microprocessors presents a real challenge [35], for the following reasons.

- They have diverse and complex architectures
- They run at very high speed
- They are highly integrated
- They have complex memory and data architectures
- They use most advance processes
- They are high volume and cost sensitive

Testing of defects in high-speed circuits such as microprocessors requires high-speed testers. At-speed testing of processors using external tester is not an economically viable scheme. Moreover, the inherent inaccuracy of the ATE also leads to yield loss. Traditional hardware BIST moves the testing task from external tester to the internal hardware but this often needs design changes that can stretch the time to market. In addition, such methods lead to unacceptable performance loss and area overhead, and can also result into burn out of the chip due to excessive power consumption during the test. A new paradigm, instruction-based self-testing (also known as software-based self-testing) can alleviate the problems of both external tester and structural BIST. It links instruction-level test with the low level fault model. In order to apply test in functional mode, instruction-based self-testing uses processor instructions to deliver the test patterns and collect the test responses. Thus, being inherently non-intrusive, it does not require area and performance overheads and it is well suited test methodology for the testing of microprocessors and processor cores embedded deep inside a System-on-a-Chip (SoC). Furthermore, the test programs developed for this method can also be used for online periodic testing to improve the processor reliability. This thesis focuses on the delay fault testing of processors using instruction-based self-testing methodology. It proposes testing methodologies for modern processors in their chronological order.

1.2. Thesis Organization

The thesis is organized as follows. Chapter 2 is devoted to the taxonomy of microprocessor testing. It describes two methods of microprocessor testing, namely, functional testing and structural testing and their pros and cons. It further describes various fault models to test a microprocessor such as stuck-at fault model and, delay fault model. It also presents further classification of path delay faults, as the path delay fault model is the target model addressed in this thesis.

Chapter 3 is devoted to instruction-based self-testing technique. At first it presents the concept of instruction-based self-testing and the method of application of the test vectors in the functional mode of operation. Thereafter, a survey of previous work in the domain of instruction-based self-test targeting stuck-at faults and delay faults is presented.

Chapter 4 is devoted to the testing methodology of non-pipelined processors. At first, it describes non-pipelined processor architecture. Thereafter, it presents models for the datapath (called instruction execution graph) and the controller (state transition diagram). Based on these models, architectural constraint extraction and path classification methods are presented. The method to generate the test sequences to apply test vectors in functional mode are discussed near the end of this chapter. At last, in order to demonstrate the effectiveness of the method, experimental results for two representative processors are presented.

Chapter 5 is devoted to the testing methodology of pipelined processors. It starts with the architectural description of a pipelined processor. After architectural description, it presents a graph theoretic model of a pipelined processor (called pipeline instruction execution graph) which captures the pipelined behaviour of the processor in addition to the information about the flow of data and control. Constraint extraction, path classification and instruction sequence generation procedures are discussed. Finally, it presents the experimental results for two representative pipelined processors to demonstrate the effectiveness.

Chapter 6 is devoted to the testing methodology of superscalar processors. After a brief introduction of superscalar architecture, it identifies test issues related to this architecture. A graph model is presented to model dynamic pipeline behaviour of superscalar architectures. A method of generating test programs that can force scheduler to execute the instructions in the desired order is described. As in other chapters, in order to demonstrate the effectiveness of the methodology, experimental results for a representative superscalar processor are also presented.

Finally, this thesis concludes with Chapter 7 where the main accomplishments of the work are outlined and the future directions are identified.

1.3. Contributions of This Thesis

This thesis makes three basic contributions to the problem of testing high performance microprocessors.

First, this thesis proposes a graph theoretic model, called instruction execution graph, for a non-pipelined processor which is based on the ground breaking work by Thatte and Abraham [1]. Subsequently, this graph model is extended to capture the pipeline processor behaviour (pipelined instruction execution graph) and superscalar processor behaviour (superscalar instruction execution graph). Modeling of these behaviours is the first work in the direction of modeling static and dynamic pipeline behaviour of processors to facilitate testing.

Second, this thesis proposes a unified approach for testing of the forwarding paths and normal paths in a pipelined processor.To achieve this the graph model developed for various processor architectures are used. The thesis also presents a hierarchical test generation methodology, which classifies paths at RT level and generates test at gate level. Methods for instruction sequence generation so that the test generated to test different parts of a microprocessor can be applied in the functional mode of operation are also developed .

Third, test issues related to superscalar processor testing using instruction-based self-test methodology are identified and these are demonstrated, through examples, to be substantially different from the conventional test generation issues. The thesis developes various methods of generating test programs that can force a ou-of-order scheduler of a modern superscalar processor to execute the instructions in the desired order to make sure that the tests can be applied correctly.

Chapter 2

Taxonomy of Microprocessor Testing

2.1. Introduction

The role microprocessors play in the present society cannot be overemphasized. Since the introduction of Intel's 4004, microprocessors have proliferated in size, performance, and complexity. Today's microprocessors consist of hundreds of millions of transistors operating at extraordinarily high speeds. Test and verification of these high-performance devices continuously challenges engineers in every processor design cycle. This chapter presents the methods to test a microprocessor and fault models which are being used in microprocessors testing.

2.2. Functional Test

The main objective of functional testing is to validate the correct operation of a system with respect to its functional specifications. A functional model reflects the functional specifications of the system which is most of the time independent of its implementation. Therefore, functional test derived from a functional model can be used not only to check whether physical faults are present in the manufactured system, but also as design verification tests for checking that implementation is free of design errors.

This can be approached in two different ways. One approach assumes functional fault model and tries to generate tests that detect the faults defined by these models. In contrast to this, the other approach is not concerned with the possible types of faulty behaviour and tries to generate tests based only on the specified fault free behaviour. Between these two there is a third approach that defines an explicit fault model which assumes that almost any fault can occur.

Functional tests detecting almost any fault are said to be exhaustive, as they must completely exercise the fault free behaviour. Because of the length of the resulting tests, exhaustive testing can be applied in practice only to small circuits.

In general, functional testing methods are tightly coupled with a specific functional modeling technique. Thus the applicability of a functional testing method is limited to systems described via a particular modeling technique. Because there exists many widely different functional modeling techniques, it is unlikely that a generally applicable functional testing method can be developed. Moreover, deriving the functional model used in test generation is often manual, time consuming and error prone process. Another major problem is lack of means of evaluating the effectiveness of test sequences at the functional level.

A functional test performs the following tasks:

- Check the existence and responsiveness of all sub-systems.
- Check system specifications. For example, amount of memory available in a system.
- Check critical functions of a system. This part can be more or less elaborate depending on the environment in which the test is applied. For example, the functional test at manufacturing may execute many of the tests that have been used in simulation-based verification.

The functional test is usually executed by the software of the system. Software test coverage criteria, such as *statement coverage*, *branch coverage*, and *path coverage*, which are not good indicators of the hardware fault detection capability.

The function of a system usually consists of an operation perform on some data. A system test is then a collection of operation-data pairs. Some heuristics used to generate tests are:

Instruction-set fault model: The instruction decoder is assumed to malfunction, causing wrong instruction to execute. The data is selected so as to indicate error in result. These tests are found to thoroughly cover some portion of hardware. However, good coverage is not guaranteed for the complete system.

All instructions with random data: All instructions are exercised with randomly generated data. These tests can quickly cover some faults, but for other faults that have low probabilities of detection the coverage remains low. For adequate coverage, random test tends to be too long to be practical.

These procedures are similar to those often used to generate design verification tests, which otherwise tend to be very lengthy. A good functional test should be as short as possible and yet comprehensive enough to cover all failures that are ”likely to occur”.

The test is built through the design and development phases and its optimization continues through manufacturing, field trials, and system use. As actual faults are found and repaired, new test sequences are added. Also, some existing tests are not found to be useful in detecting actual faults are dropped.

In summary, advantages of functional testing are:

- Low-level details are not needed
- Functional verification pattern can be used
- Less test development cost

And, disadvantages of functional testing are:

- No relation with the structural fault
- Low defect coverage
- Long test sequences
- Lack of means of evaluation

2.3. Structural Tests

The drawback of functional testing is that it is not directly connected to actual structural testability of the processor, which is related to the physical defects. Structural test uses specific fault model to test the structure for errors. EDA tools can be used for automatic generation of test patterns (ATPG tools). Structural test generation can be performed only if a gate level model of the processor is available; and high fault coverage can be achieved for the target structural fault model with a small test set or a small test program that is executed in short time. Typically used fault models are stuck-at fault model, delay fault model, bridging fault model, and cross talk fault model. Some of the fault models are discussed in the next section.

Advantages of structural testing are:

- EDA tools can be used
- High fault coverage
- Small test sequence
- Fast test program

And, disadvantages of structural testing are:

- Needs gate level model of processor
- Higher test development cost
- Most of the time, it needs circuit modification

A structural fault can be tested in the following ways:

1. External tester based testing
2. Hardware-based self-testing
3. Instruction-based self-testing

2.3.1 External Tester Based Testing

It is a conventional method for the testing of high volume electronic chips after manufacturing. The test patterns previously stored in the tester memory are applied to an IC under test and operate it under this test input. The response of each of the applied test patterns is captured by the tester, and stored back in the tester memory. Finally, the stored response is compared with the known, correct response. Subsequently, the next test pattern is applied to the IC, and process is repeated until all patterns of the test set stored in the tester memory have been applied to the chip.

Figure 2.1 presents the idea of external tester based testing. The operating speed of today's chips (f_{chip}) is usually more than the operating speed of the tester (f_{tester}). The relation between these two frequencies is a critical factor that determines both the quality of the testing process using external tester and also test application time and subsequently test cost.

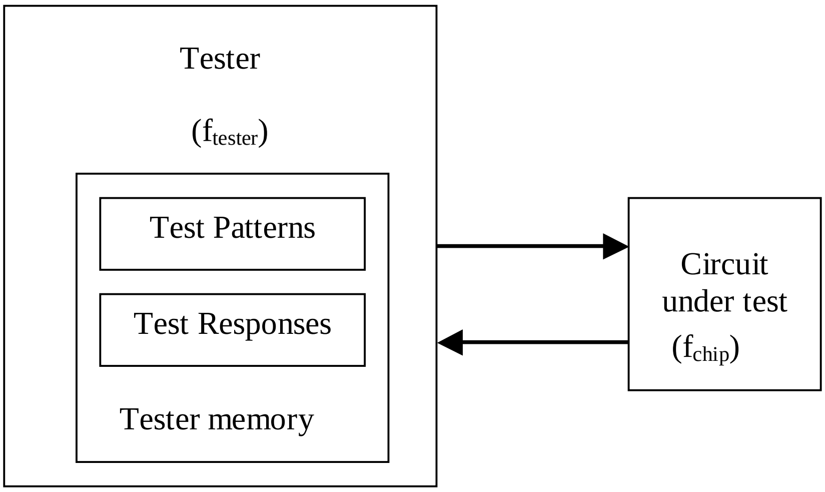


Figure 2.1. External tester based testing

The essence of the relation between the tester and the chip frequencies is that if we want to execute a high quality test to a chip and detect all (or most) physical failure mechanisms of modern manufacturing technologies, we must use a tester with a frequency f_{tester} which is close to or equal to the actual chip frequency

f_{chip}. This means, for a high frequency chip, a very expensive, high frequency tester must be used and this fact will increase the over all test and development cost of the chip. According to [6], if the current testing techniques are to be continued, the test equipment cost can rise towards $20 million.

Another cost-related consideration for external testing is the size of the tester's physical memory where test patterns and test responses are stored. In case, memory size is not large enough, multiple loading of memory is needed to apply entire test vector set, which in turn leads to extension of test application time. Extension of test time directly implies higher test cost.

When a complex chip design has to be tested by an external tester with test pattern generated by ATPG tools and possibly applied in scan-based fashion, following are the limitations of the external tester based testing:

1. *At speed testing:* Continuously widening gap between the tester operating frequency and the IC operating frequency does not allow the detection of a large percentage of physical failures in CMOS technology that manifest themselves as delay faults instead of logical faults. A very large set of physical mechanisms that lead to circuits not operating in the target speeds can only be detected if the chip is tested in the actual frequency in which it is expected to operate which is called *at-speed* testing. Even for the best available tester at any point of time, there will always be a faster IC in which performance related circuit malfunctions will remain untestable because it is testing next generation ICs which is often true for modern processors. Therefore, the fundamental target of manufacturing testing – detection of as many as possible physical defects that may lead the IC to malfunction – cannot be met under these conditions.

2. *Yield loss due to tester inaccuracy:* Testers are external devices that perform measurements on manufactured chips, and thus they suffer from severe measurement inaccuracy problems which for high speed designs of these days lead to serious production yield loss. A significant set of correctly operating chips are characterized as faulty and rejected just due to tester's inaccuracies in the performance measurement. According to [6], due to inherent inaccuracy of testers, at-speed testing of high speed processors may result in an unacceptably high yield loss of 48% by 2012.

3. *Yield loss due to over-testing:* Over testing is another source of yield loss. Scan-based testing may put the circuit in non-operation, and it can detect a fault that never appears in the functional mode. In many cases, the circuit is tested for potential faults that, even if they exist, they will never affect the normal circuit operation. The rejection of chips that have non-functional faults (like non-functionally sensitizable path delay faults) leads to further yield loss in addition to yield loss due to tester inaccuracy.

2.3.2 Hardware-Based Self-Testing

Traditional hardware-based self-testing or built-in self-testing (BIST) techniques have been proposed since several decodes ago [35] to resolve the bottlenecks that external tester based testing can not, by moving testing task from external tester to the internal hardware. It uses embedded hardware test generators and test response analyzers to generate and apply test pattern on chip at the speed of circuit, thereby eliminating the need for an external tester. The only necessary external action is the initiation of the self-testing execution.

Self-testing is defined as the ability of an electronic circuit to test itself, i.e. to excite potential fault sites and propagate their effect to observable locations outside the chip (Figure 2.2). The tasks of the test pattern generation and test response collection are both performed by internal circuit resources, and the fault inside them must also be detected.

The advantages of self-testing strategies compared to external tester based testing are:

- Expensive external tester is not needed
- Self-testing mechanisms have a much better access to internal resources than the external tester based mechanism
- It can be applied at the operating frequency. Hence at-speed testing becomes possible, which leads to high quality test.
- It can be applied at the operating frequency. Hence at-speed testing becomes possible, which leads to high quality test

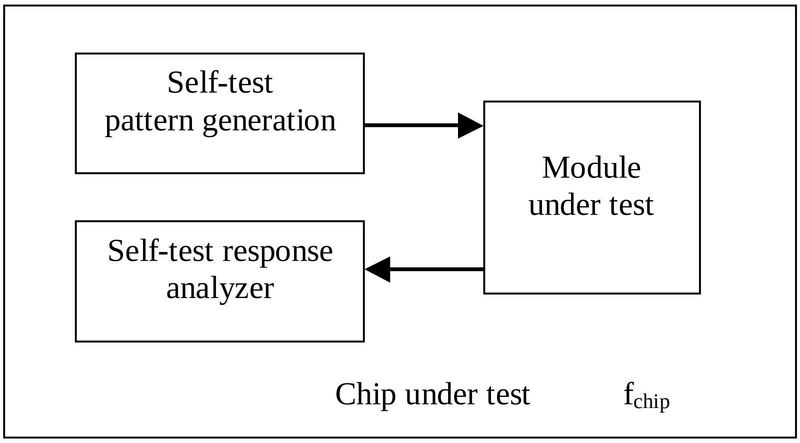


Figure 2.2. Built-in self-testing

- No yield loss due to tester inaccuracy

Memory BIST has been commonly used for testing embedded memory components, as it performs well due to deterministic nature of memory tests facilitated by regular structure of memory components. Logic BIST, however, faces many challenges because it relies on the generation and application of pseudo-random test patterns [6].

1. For random-pattern-resistant circuits, the fault coverage achieved by pseudo-random testing may be low.

2. The insertion of the BIST circuitry used for generating and applying pseudo-random patterns may result in significant area overhead and performance penalty

3. The application of pseudo-random patterns often results in excessive power consumption in the BIST mode which may some times leads to burn out of the chip.

4. The application of random test patterns may drive the circuit under test into nonfunctional mode in which free flow of test data can be impeded by problems such as bus contentions.

Fault coverage could be improved by techniques such as deterministic BIST [38] or weighted random patterns [39]. In scan-based BIST, the test overhead may also be reduced by techniques such as test scheduling, reducing input activities, or filtering non-detecting vectors. BIST readiness may be achieved by design changes.

While logic BIST may perform well on industrial application specific integrated circuits (ASICs), its feasibility on microprocessors is yet to be investigated. First design changes needed for making a microprocessor BIST ready may come with unacceptable cost, such as manual effort and significant performance degradation. In addition, microprocessors are especially random-pattern resistant circuits. Due to, timing critical nature of microprocessors, test points insertion can not be acceptable as a solution to this problem, as they could introduce performance degradation in critical paths. Deterministic BIST, on the other hand, may lead to unacceptable area overhead, as the size of the on-chip hardware for encoding deterministic test patterns depends on the testability of the circuit. For at-speed testing complex issues related to timing, like multiple clock domain and clock skew, must be resolved.

Further, this method may be unacceptable for testing an optimized processor core embedded deep inside a System-on-a-Chip (SoC) due to its poor and limited accessibility.

2.3.3 Instruction-Based Self-Testing

A new paradigm, Instruction-Based Self-Testing could alleviate the problems of both external tester and structural BIST. It links instruction-level test with the low level fault model. In order to apply test in functional mode, it uses processor instructions to deliver the test patterns and collection of test responses. Therefore, it does not require to modify the circuit and is capable to apply test patterns at operating frequency.

Self-test routines are stored in instruction memory and data they need for execution are stored in data memory. Both transfers (instructions and data) are performed using external test equipment which can be as simple as a personal computer and as high as a high-end tester. Tests are applied to components of the processor (or processor core) during the execution of the self-test programs and test responses are stored back in the data memory.

Instruction-based self-test (functional self-test) methodology uses processor instructions and its functionality in order to test the processor core. Therefore, it has the following advantages over structural BIST:

- Non-intrusive approach
- Low cost
- No area overhead
- No performance penalty
- At-speed testing is possible
- No excessive power consumption during test
- Flexible and programmable

Thus, being inherently non-intrusive, it does not require area and performance overheads and it is well suited test methodology for the testing of processor cores embedded deep inside an SoC. Furthermore, the test programs developed for this method can also be used for online periodic testing to improve the processor reliability. This approach is discussed in detail in next chapter.

2.4. Fault Models

The consideration of possible faults in a digital circuit is undertaken in order to establish a minimum set of test vectors, which collectively will test that faults are or are not present. If none of the predefined faults are detected then circuit is considered to be fault free. There are several fault models presented in the literature to model various defects. This section presents two widely used fault models; namely stuck-at fault model and delay fault model, which deal with logical and timing defects respectively.

2.4.1 Stuck-at Fault Model

A stuck-at fault is assumed to affect only the interconnection between the gates. Each connecting line can have two types of faults: stuck-at-0 *(s-a-0)* and stuck-at-1 *(s-a-1)*. Thus, a line with *s-a-0* fault will always have a logic state 0 irrespective of the correct logic output of the gate driving it.

In general, several stuck-at faults can be simultaneously present in the circuit. A circuit with n lines can have 3^n-1 possible stuck line combinations, because each line can be in one of three states: *s-a-0, s-a-1*, or fault-free. All combinations except one having all lines in fault-free states are counted as faults. Clearly, even a moderate value of n will give an enormously large number of multiple stuck-at faults. It is common practice, therefore, to model only single stuck-at faults. An n-line circuit can have at most $2n$ single stuck-at faults. This number is further reduced by fault collapsing technique.

2.4.2 Delay Fault Models

These faults cause the combinational delay of the circuit to exceed the clock period. The manufacturing defects that lead to the timing faults are modeled by several fault models such as transition delay, gate delay, line delay, path delay and segment delay fault models based on the lumped delay or distributed delay assumptions. We discuss each model briefly.

Transition delay fault model: This model is based on the assumption that the delay fault affects only one gate in the circuit. Each gate may have two faults: slow to rise or slow to fall, i.e. the fault results in an increase in the propagation delay of the gate. Under the transition fault model, this increase in delay is large enough to prevent the transition from reaching any primary output within specified time. However, this assumption is not realistic and a disadvantage of this fault model. The advantages of this model are: i) the number of faults in the circuit is linear in terms of the number of gates, ii) the stuck-at fault test generation and fault simulation tools can be easily modified to handle transition faults.

Gate delay fault model: This fault model is based on the assumption that the delay fault is lumped at one gate in the circuit. Unlike the transition fault model, gate delay fault model does not assume that the fault affects the circuit performance independent of the propagation path through the fault site. It is assumed that only long paths through the circuit might cause performance degradation. The advantages of gate delay fault model are similar to those transition delay fault model.

Line delay fault model: Line delay fault model is a variation of gate delay fault model. This model tests rising or falling delay faults on a given line in the circuit. The fault is propagated through the longest sensitizable path passing through the line. The number of faults equals twice the number of lines in the circuit because each line is tested for rising and falling delay faults. Sensitizing the longest path through the target line allows the detection of the smallest delay fault on the target line.

Path delay fault model: The delay defect in the circuit is assumed to cause the cumulative delay of a combinational path, i.e. distributed delay in the circuit is assumed. The combinational path begins at a primary input or a clocked flip-flop, contains a connected chain of gates, and ends at a primary output or a clocked flip-flop. A path is considered faulty if its propagation delay exceeds a specified duration. The specified time duration can be the duration of the clock period (or phase), or the vector period. The propagation delay is the time that a signal event (transition) takes to traverse the path. Both switching delay of device and transport delays of interconnects on the path contribute to propagation delay. It is more realistic model, however, the major limitation of this model is the number of faults in the circuit can be very large - increases exponentially with circuit size.

Segment delay fault model: A segment delay fault model is somewhere in between path delay fault model and transition delay fault model. A segment is a partial path and this model assumes the delay fault of segment is large enough to cause a delay fault on all paths that include the segment. The number of segments for some given length is much lower than the number of paths in a circuit. This helps to reduce the number of faults while considering some sort of distributed delay, which is more realistic. The length of a segment can be decided based on available statistics about the manufacturing defects.

Classification of Path Delay Faults

Cheng and Chen [31], [27] classifies path delay faults as robust, non-robust, functional sensitizable, and functional unsensitizable, based on the sensitization criteria. The robust, non-robust, and functional sensitizable faults can affect circuit performance; hence circuit must be tested for these faults. On the other hand, functional unsensitizable fault can never independently affect the circuit performance and need not to be tested. Before explaining each of these fault types, we discuss some terminology used. An input to a gate is said to have a controlling value if it determines the output of the gate regardless the value of the other inputs of the gate, and its complement is said to be non-controlling value. For example, logic value 0 is the controlling value for the AND/NAND gate, and logic value 1 is non-controlling value for these gates. An input to a gate is called on-input with respect to a path P if it is on P. An input to a gate is called off-input with respect to a path P if gate is on P and the input is not an on-input.

Robust Testable Path Delay Faults: A path is called robust testable if there exist at least one vector pair (V_1, V_2) such that:

- It launches the desired logic transition at the beginning of P, and
- At each gate along P, if the on-input transitions to a controlling value, the off-inputs of the gate remain stable at non-controlling value or if the on-input transitions to a non-controlling value, the off-inputs of the gate either remain stable at non-controlling values or transitions to a non-controlling value.

The robust testability condition ensures that if the path is faulty then it is guaranteed to fail independent of the delays in the other paths of the circuit. Therefore, for testing purposes, robust tests are the highest quality tests and should be applied when they exist.

Non-Robust Testable Path Delay Faults: A robust untestable path is non-robust testable if there exists at least one vector pair (V_1, V_2) such that:

- It launches the desired logic transition at the primary input of the target path P, and
- All side inputs of the target path settle to non-controlling value under vector V_2.

A non-robust test to detect a delay fault along a path may be invalidated by the presence of delay faults along one or more side paths.

Functional Sensitizable Path Delay Faults: A non-robust untestable path P is functional Sensitizable if there exists at least one vector pair (V_1, V_2) such that,

- It launches the desired logic transition at the primary input of the target path P, and
- At least at one gate along P, if the on-input transition to a controlling value, the off-input(s) of the gate also transitions to a controlling value and the rest of the gates satisfy either robust or non-robust testability condition.

Functional Sensitization criterion requires the presence of multiple faults in the circuit in order to detect the target fault.

Functional Unsensitizable Path Delay Faults: Functional unsensitizable path delay faults are defined as the faults for which all possible vector pairs at least at one gate if the on-input transitions to a controlling value, the off-inputs of the gate remain stable at controlling values or if the on-input transitions to a non-controlling value, the off-inputs of the gate either remain stable at controlling values or transitions to a controlling value. Functional unsensitizable paths can never affect the circuit performance and need not be tested.

Relations between robust testable, non-robust testable, functional Sensitizable faults have been presented in Figure 2.3.

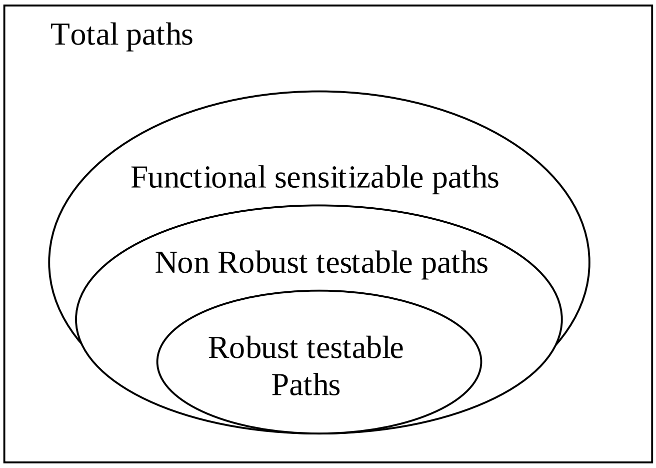


Figure 2.3. Testability classification of path delay faults

2.5. Conclusion

This chapter presented two methods for microprocessor testing, namely, functional testing, and structural testing. It is pointed out that functional testing needs long test sequence, and lacks in fault coverage evaluation method, although it can use design verification sequences. Hence, structural testing is a practical approach to test a processor. Various structural test application methods are presented. It is shown that instruction-based self-testing is the best suitable methodology for processor testing as it has the capability to apply test in functional mode of operation. Details of delay fault models have been presented as this thesis targets the timing defects in the processors which can lead to performance degradation of the processors.

Chapter 3

Instruction-Based Self-Testing

3.1. Introduction

As discussed in chapter 2, instruction-based self-testing (IBST) methodology addresses the problems faced by external tester based testing and built-in self-testing for testing high performance processors or processor cores embedded inside an SoC. This chapter describes instruction-based self-testing approach in detail and the previous work done in the domain of instruction-based self-testing for the testing of processors.

3.2. IBST Methodology

IBST links functional testing with gate level fault model. The concept of the IBST is illustrated in the Figure 3.1. It uses on chip resources and processor instructions to deliver the test patterns and collection of test responses. Self-test routines are stored in instruction memory and data they need for execution are stored in data memory. Both transfers (instructions and data) are performed using external test equipment which can be as simple as a personal computer and as high as a high-end tester. Tests are applied to components of the processor (or processor core) during the execution of the self-test programs and test responses are stored back in the data memory.

At first, the self-testing code is downloaded to the processor instruction memory of the processor via external tester which has access to the internal system bus. Alternatively, the self-test code may be "built-in" in the sense that it is permanently stored in the chip in a ROM or flash memory which is shown in the Figure 3.2. In this case, there is no need for downloading process and self-test code can be used many times for periodic/on-line testing of the processors in the field.

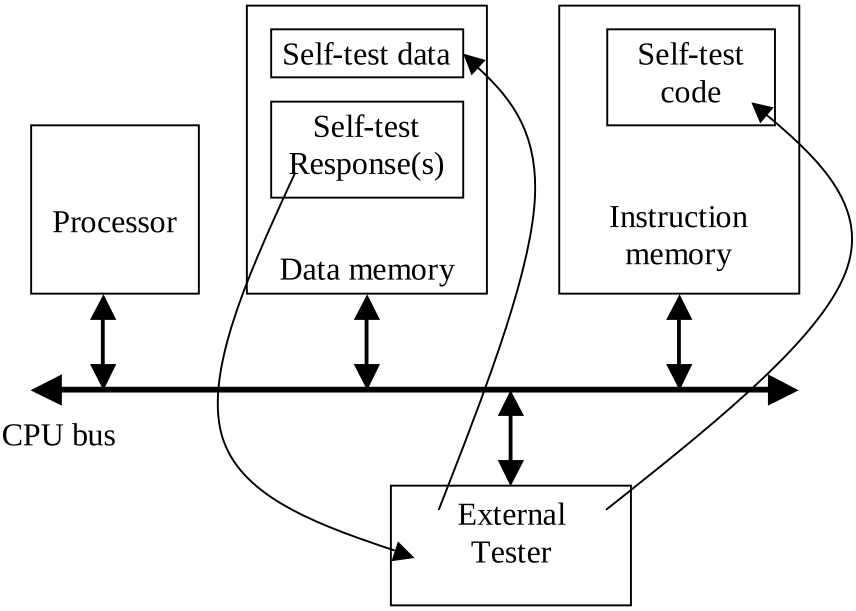


Figure 3.1. IBST with external tester

The test data is downloaded to the data memory of the processor via the same external tester. Self-test data may consist, among others, of: i) parameters, variables and constants of the self-test code, ii) test patterns that will be explicitly applied to internal processor modules or the paths between the registers, and iii) the expected fault-free test responses to be compared with the actual test responses. Downloading of self-test data does not exist if on-line testing is applied and the self-test program is permanently stored in the chip.

Once self-test code and data are transferred to processor memory, the control is transferred to self-test program which starts execution of self-test. Test patterns are applied to internal processor component via processor instructions to

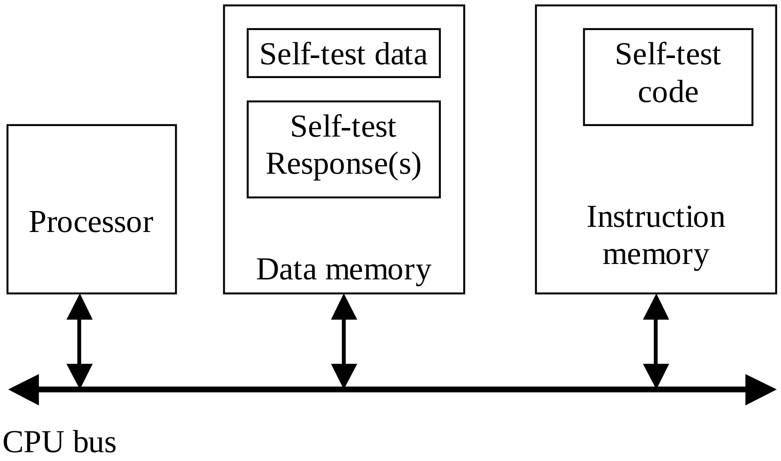


Figure 3.2. IBST without external tester

detect their faults. Test responses of the applied test instructions are collected in registers and/or data memory locations. Responses may be collected in the unrolled manner in the memory or may be compacted using known test response compaction algorithm. In the former case, more data memory is required and test application time may be longer, but, on the other hand, aliasing may be avoided. In later case, data memory requirements are smaller because only a few self-test signatures are collected, but aliasing problem may appear due to compaction.

After self-test code completes execution, the test responses previously collected in data memory, either as individual responses for test pattern or as compacted signatures are transferred to the external tester for evaluation.

In case of periodic, on-line testing there is no need to transfer self-test code, data and responses to and from external tester. Self-test code, data and expected response(s), are stored in the chip. Execution of self-test programs leads to a pass/fail indication which can be used subsequently for further actions on the system (repair, re-configuration, re-computation etc.).

Application of test patterns to processor via processor instructions consists of the following three steps, which are shown in Figure 3.3.

Test preparation: Test patterns are placed in locations (usually registers but also in memory locations) from which they can be easily applied to a processor component (the component under test or the path under test). This step may require more than one processor instructions.

Test application and response collection: Test patterns are applied to the processor's component (path) under test and component's (path's) response(s) are collected in locations (usually registers but may also be memory locations). This step usually takes one instruction.

Response extraction: Responses collected internally are exported towards data memory (if not already in the data memory by the test application instruction). This step may also require the execution of more than one instruction.

For example, a fault inside a shifter can be tested by the following instruction sequence:

I_1:	LOAD R1, mem[1]	– transfers test vector to register R1
		– from memory location mem[1]
I_2:	LOAD R2, mem[2]	– transfers test vector to register R2
		– from memory location mem[2]
I_3:	SLL R3, R1, R2	– applies test vector and store result in register R3
I_4:	STORE R3, mem[3]	– transfers result from register R3
		– to memory location mem[3]

The test patterns must be loaded at mem[1] and mem[2] memory locations. It transfers the response to mem[3] location, which can be compared with the correct response.

Due to inherent non-intrusive approach, IBST has following advantages:

- *No area overhead:* This approach uses only processor resources (functional units, processor buses, registers etc.) for test application and response collection. Hence, it doesn't lead to area overhead.

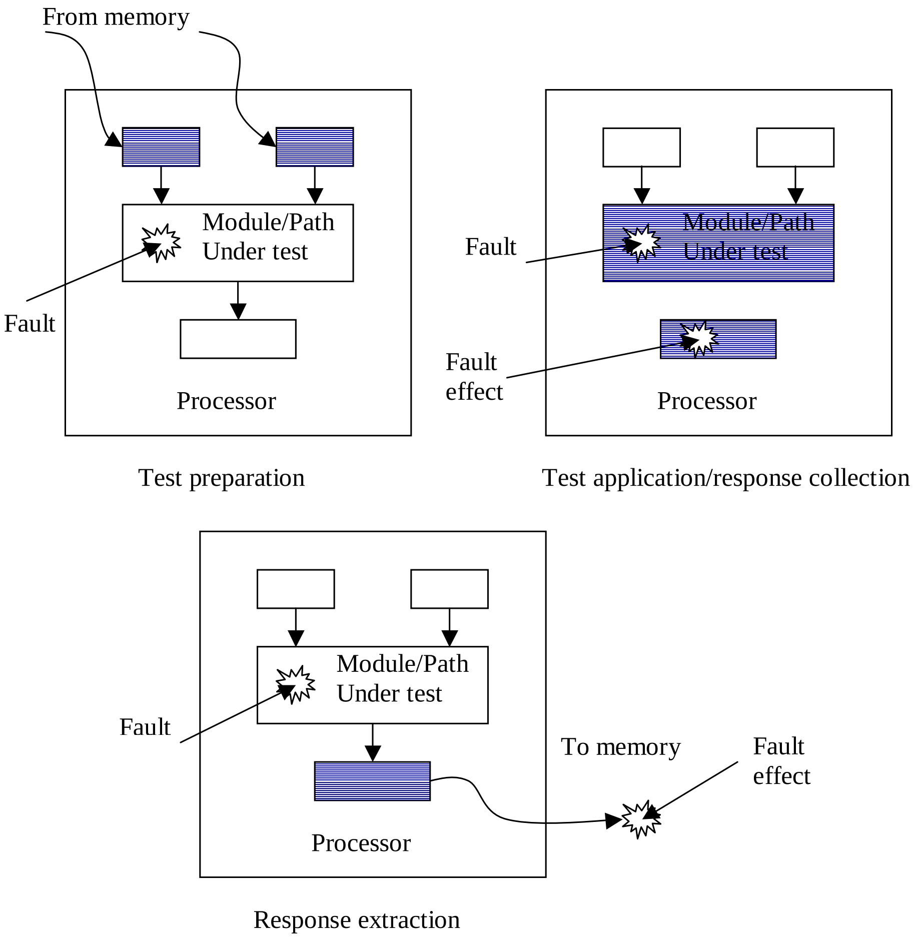


Figure 3.3. Application of instruction-based self-test

- *No performance penalty:* This approach does not modify the circuit under test; hence, it does not lead to performance penalty.
- *No excessive power dissipation:* This approach can be applied in functional mode of operation, instead of ortho-normal test mode. Hence, it cannot dissipate power greater than the rated power.
- *At-speed test:* This approach always applies test vectors at-speed as it uses functional mode of operation. Hence, it can be easily used for the testing of timing faults.

Due to the above stated advantages of IBST, it is a suitable testing methodology for processor testing. Next section describes the reported work in the domain of instruction-based self-testing of microprocessors.

3.3. Previous work

Many approaches have been reported in the literature to test a processor. This section provides a survey of IBST approaches proposed for the testing of microprocessors. We will first, enlist the approaches for non-pipelined architecture, followed by the approaches for pipelined architecture, targeting both stuck-at fault, and path delay faults. To the best of our knowledge no approach has been reported for the superscalar architecture. We believe this thesis proposes first approach to test a superscalar architecture using instruction-based testing.

For non-pipelined architecture:

In a landmark paper in the early 80's Thatte and Abraham [1] proposed a graph theoretic model for testing a microprocessor, and based on this and functional fault model, they developed test procedures to test a microprocessor. Saluja et al. [2] used timing and control information along with processor model [1] to reduce the test complexity. Brahme and Abraham [3] proposed an improved functional model to further reduce the test size. All these approaches use functional fault model but, little if any, fault grading was done on structural model in these approaches.

A number of instruction-based self-test approaches [4] – [14], targeting stuck-at faults for non-pipelined processors, have also been proposed. Shen and Abraham [4] proposed an approach based on instruction randomization. This approach generates a sequence of instructions that enumerates all combinations of operations and systematically selected operands. Batcher and Papachristou [5] also proposed an instruction randomization approach which combines the execution of microprocessor instructions with a small amount of on-chip hardware for randomization of instructions. Both these approaches [4],[5] give low fault coverage due to high level of abstraction and they generate large code sequence resulting in large test application time. Chen and Dey [6] used the concept of self-test signature in which they generate structural tests in the form of self-test signatures for functional modules by taking constraints into consideration. These self-test signatures are expanded into test sets using software LFSR during self-test and applied using a test application program. Responses are collected and compared with a response signature stored in memory. Due to pseudorandom nature of this methodology self-test code size and test application time are large. Moreover, efficiency of pseudorandom software based methodology depends on internal architecture and bit width. Paschalis et al. [7] use self-test routines for functional modules based on deterministic test sets for testing datapath of a processor. Similarly, Krantis et al. [8],[9] proposed a methodology based on instruction set architecture and RT level description while using deterministic test sets to test every functional component of the processor for all the operations performed by that component. Deterministic nature of these [7] – [9] approaches lead to reduced test code size but these methods find difficult to achieve high fault coverage for complex architectures. Kambe et al. [10] proposed a template based approach, that generates templates and uses fault simulation. It needs long sequence for hard to detect faults. The above stated approaches do not explicitly consider the controller.

An instruction-based self-test approach targeting delay faults was proposed by Lai et al. An instruction-based self-test approach targeting delay faults was proposed by Lai et al. [11] – [13]. This approach, first classifies a path to be functionally testable or untestable. The authors argue that delay defects on the functionally untestable paths will not cause any chip failure. They also suggest that gross delay defects should be tested by transition fault testing. In their method, datapath and controller are considered separately. Path classification is performed by extracting a set of constraints for the datapath logic and the controller. In constraint extraction procedure for datapath, all instruction pairs are enumerated and for each instruction pair all possible vector pairs that can be applied to the datapath are derived symbolically. These symbolic vector pairs represent the constraints for datapath testing. This requires a substantial effort to analyze all the instructions and all possible pairs of instructions even though it is not necessary to analyze all the pairs as shown in chapter 4 of this thesis. For controller, constraints in terms of legitimate bit patterns in registers and correlation between control signals and transition in registers are extracted. A procedure given in [11] is used to classify paths in controller which uses multiple time frames. This procedure uses sequential path classification methodology i.e., in order to classify a path it propagates the transition forward till PO and backward till PI in multiple time frames under the constraints. This is needed because it is not extracting the constraints provided by the state transitions. After classification of paths, constrained ATPG is used to generate the test patterns for testable paths. Lai and Cheng [14] proposed an approach for delay fault testing of a System-on-a-Chip using its own embedded processor instructions, and also proposed a methodology to include new test instructions for testability enhancement and test program size reduction.

For pipelined architecture:

A methodology for pipelined processor based on the test templates was presented by Chen et al. [19], which uses statistical regression for function mapping, but it leads to long program size for complex architectures due to its statistical nature. Kranitis et al. [20] also proposed a methodology for pipelined processors based on RTL description and instruction set architecture using deterministic tests for functional blocks, whereas Paschalis et al. [21] proposed an online periodic test methodology for pipeline processors. But these approaches [20] and [21] target only functional blocks; hence they are unable to achieve high fault coverage for complex architectures. Although [19] – [21] considered pipelined processor, the pipelined behaviour was not considered explicitly in these, as they are largely focused on the functional blocks. Also none of the above stated approach target controller explicitly.

To the best of our knowledge, no approach has been proposed in literature for testing of pipelined processors targeting delay faults so far. We believe this thesis proposes the first work towards modeling of pipeline behavior for testing of a microprocessor and a delay fault testing methodology of pipelined processors in functional mode.

3.4. Conclusion

This chapter described a new paradigm, instruction-based self-testing methodology, which is being proposed recently in order to overcome the limitations of external tester based and built-in self-test based testing of processors. It has been shown that this is the most suitable approach for the testing of processors and processor cores embedded deep inside an SoC. It also presented the approaches proposed in the literature for the testing of processors using instruction-based self-testing.

Chapter 4

Non-pipelined Processor Testing

4.1. Introduction

Today's SOCs are built around high performance processors to meet increasing consumers demand of rich functionality and performance with short turn around time. Design reuse is being regarded as the only way that allows designers to keep pace with the technological developments. It reduces the time to market and design effort significantly. However, it introduces test difficulties. Chapter 2 highlighted the need of at-speed delay fault testing of such a high performance processors or processor cores to make sure the performance of these high-speed processors. At-speed testing problem can be addressed by instruction-based self-testing, as described in chapter 3. An ad-hoc instruction-based self-testing approach targeting delay faults is proposed by Lai et al [11]. This approach extracts constraints by exhaustively searching instructions and instruction pairs. Moreover, results for the controller are not reported for this approach

This chapter proposes [16] – [18], an efficient and systematic methodology of delay fault self-testing of processors or processor cores using their instruction set. The proposed approach uses a graph theoretic model (represented as Instruction Execution Graph) of the datapath and a finite state machine model of the controller for the elimination of functionally untestable paths at the early stage without looking into the circuit details, and extraction of constraints for the paths that can potentially be tested. Parwan and DLX processors are used to demonstrate the effectiveness of our approach.

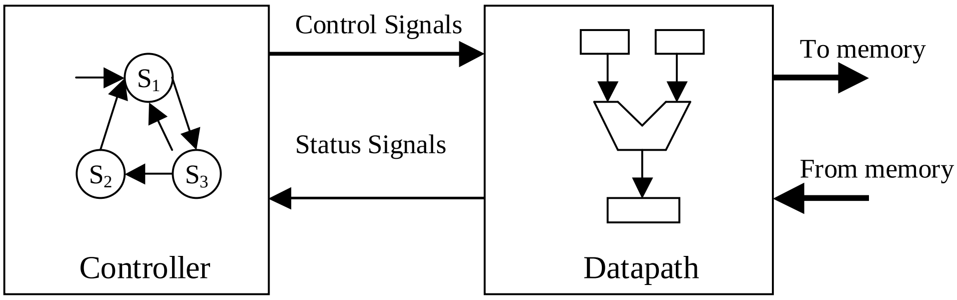


Figure 4.1. Block diagram of a non-pipelined processor

This chapter first describes non-pipelined processors architecture and overview of the approach. Thereafter it presents graph model and the testing methodologies for the datapath and the controller. Following test generation procedure, an instruction sequence generation procedure is presented. Finally experimental results are discussed.

4.2. Non-pipelined Processor Architecture

A microprocessor is a computer's central processing unit (CPU) implemented on a chip. The processor has two parts: a control part and a data part. The control part says what to do, and the data part does it. The control part decodes instructions and guides the processor through its internal states. The data part (or execution unit) contains the registers, arithmetic and logic unit, shifter, and other pieces that directly store or manipulate data. The control part directs operations in the execution unit. It consists of clock-phase generator, bus controller, and processor controller. The processor controller consists of an instruction decoder, and a finite-state machine to generate the control signal for every cycle.

Figure 4.1 shows a block diagram of the implementation of a processor.

Datapath

Datapath executes instructions using available architectural registers and functional units. Figure 4.2 shows a general architecture of the datapath of a non-pipelined processor.

In general, the execution of an instruction can be divided into three main phases as shown in Figure 4.3. The phases are *instruction_fetch*, decode_opfetch, and execute_opwrite. In the *instruction_fetch* phase, an instruction is retrieved from the memory and stored in the instruction register. The sequence of actions required to carry out this process can be grouped into three major steps.

1. Transfer the contents of the program counter to the memory address and increment the program counter. The program counter now contains the address of the next instruction to be fetched.

2. Transfer the content of the memory location specified by the memory location specified by the memory address register to the memory data register.

3. Transfer the contents of the memory data register to the instruction register.

In the *decode_opfetch phase*, the instruction in the instruction register is decoded, and if the instruction needs an operand, it is fetched and placed into the desired location.

The last phase, *execute_opwrite*, performs the desired operation and then stores the result in the specified location. Sometimes, no further action is required after the decode_opfetch phase. In these cases, the execute_opwrite phase is simply ignored.

The three phases described must be processed in the sequence.

Controller

The major parameters of interest to design a controller are:

- Speed

- Complexity

- Flexibility

The design of the control unit must provide for the fastest execution of instructions possible. Instruction execution speed obviously depends on the datapaths

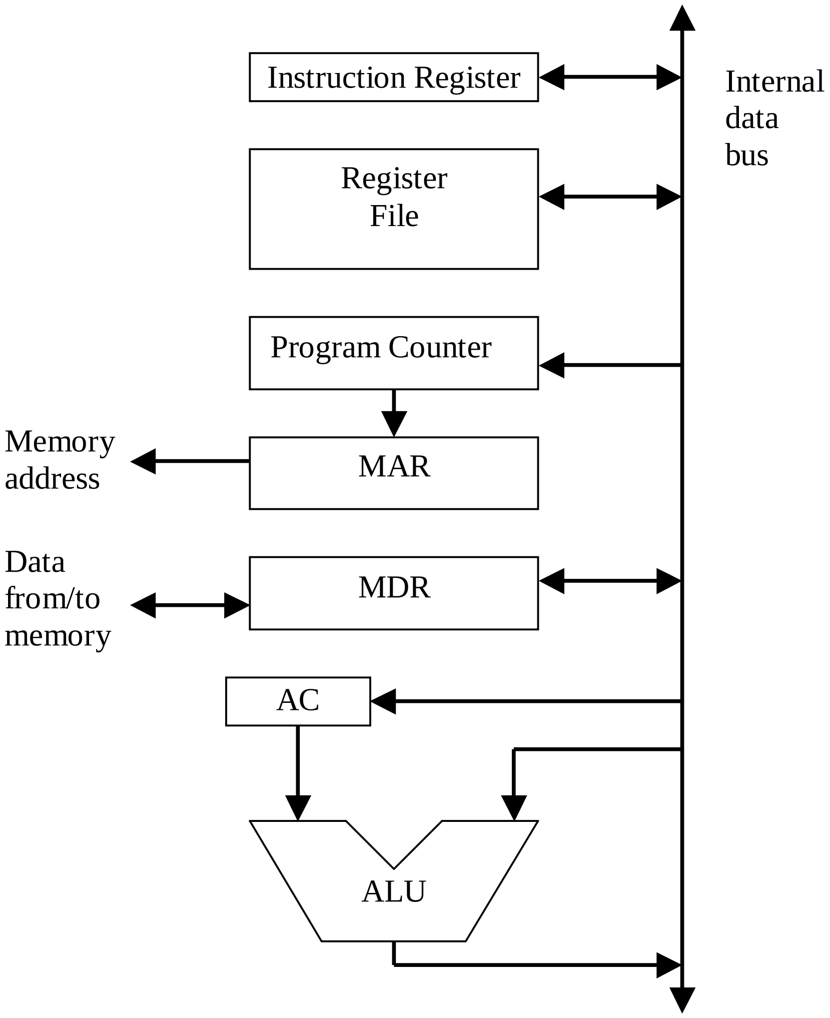


Figure 4.2. Datapath of a non-pipelined processor

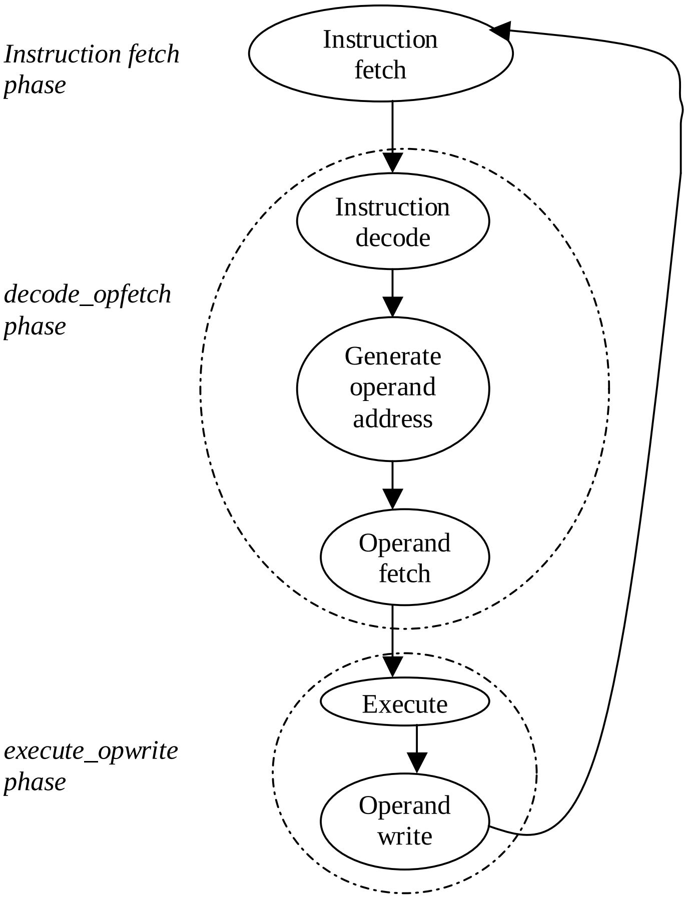


Figure 4.3. Phases of an instruction execution process

resources (i.e., number of buses etc.) available in the structure; the complexity of instruction itself (number of memory addresses, number of addressing modes, etc.); and the speed of hardware components in the circuit. The control unit should minimize the instruction cycle time (i.e., the time to fetch and execute an instruction) for each instruction.

The complexity of the control unit is predominantly a function of instruction size, although factor such as ALU complexity, the register complexity, and the processor bus structure influence it. Early design used complex instruction set architecture (complex instruction set computers – CISC) which supports complex addressing modes in order to ease out the programmers task. It was observed that on an average 20% to 30% of the instructions in an instruction set are not commonly used by application programmers. These observations led to the development of the reduced instruction set computer (RISC). RISC architectures use large number of instruction with fewer addressing mode, hence, reduce the controller complexity. Today, nearly all the processors are using RISC architecture.
Two popular implementation techniques of the control unit are:

- Hardwired control
- Micro-programmed control

Note that each instruction cycle corresponds to a sequence of micro-operations (or register transfer operations) brought about by the control unit. These sequences are produced by a set of gates and flip-flops in a hardwired control unit, or from the programs located in the micro-programmed ROM in a micro-programmed control unit. The micro-programmed control unit offers flexibility in terms of tailoring the instruction set for a particular application. The hardware implementations, on the other hand, offer higher speed. Almost all hardware control units are implemented as synchronous units whose operation is control by a single clock signal.

Generally, modern RISC processors use hardwired control and these are implemented using finite state machine based controller. In this work, we consider a finite state machine based controller. The controller, as shown in Figure 4.4, consists of an instruction register (IR) to hold the fetched instruction until it completes, a status register (SR) to hold status signals received from the datapath, and a present state register (PSR) to hold the current state of the sate machine which is used by next state logic to update the state and output logic to generate control signals in each state.

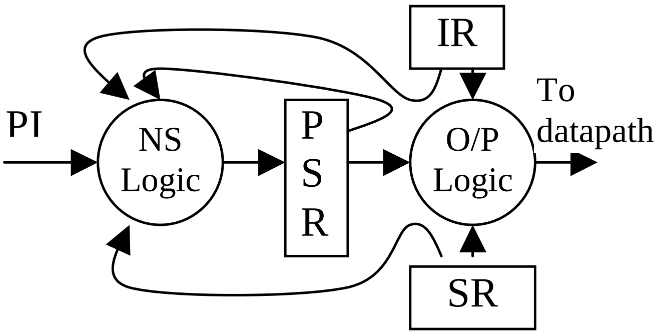


Figure 4.4. Structural organization of a controller

In order to execute an instruction, the controller should generate control signals to carry out the operations shown in Figure 4.3 sequentially. The state diagram can be used to implement a hardwired circuit as a Mealy type or as a Moore type finite state machine.

Parwan Processor

We will use Parwan processor [28] as a running example in this chapter to explain many of the concepts. Parwan processor is an accumulator based 8–bit processor with a 12–bit address bus. It has 17 instructions, listed in Table 4.1, and it supports both direct and indirect addressing modes. The structural organization of the processor is shown in Figure 4.5 and the state diagram of the controller in the Parwan processor is shown in Figure 4.6.

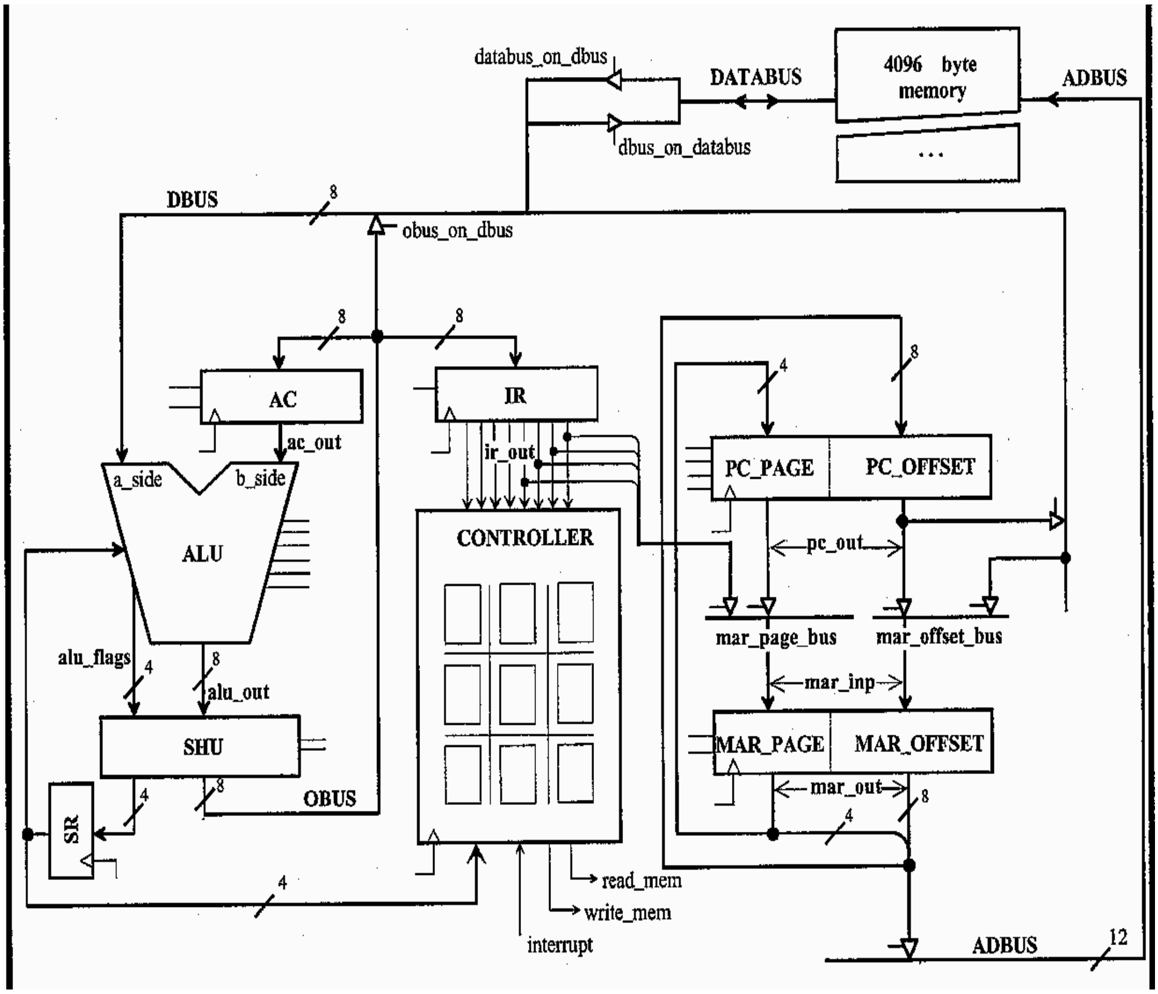


Figure 4.5. Structural organization of Parwan processor

Table 4.1. Instruction set of Parwan processor

1. LDA	5. JMP	9. BRA_C	13. CLA	17. ASR
2. AND	6. STA	10. BRA_Z	14. CMA	
3. ADD	7. JSR	11. BRA_N	15. CMC	
4. SUB	8. BRA_V	12. NOP	16. ASL	

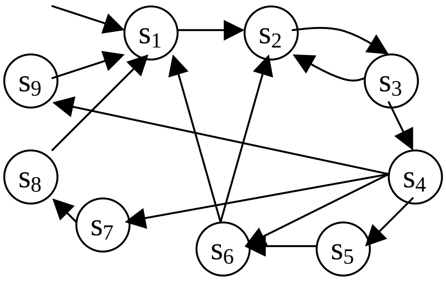


Figure 4.6. State diagram of Parwan processor

4.3. Overview of the Proposed Work

Our methodology considers datapath and controller separately as both of these have different design characteristics. The activities in the datapath are controlled by the controller. The controller is also constrained by state transitions and signals from the datapath. Hence, only a subset of structurally applicable test vectors may be applied in the functional mode. Path delay fault model is used.

We model datapath by a new graph theoretic model called Instruction Execution Graph (IE-graph) that can be constructed from the instruction set architecture and RT level description of the processor. In our formulation of the test problem IE-Graph is used to classify all paths as *functionally untestable* (FUT) paths or *potentially functionally testable* (PFT) paths, and to extract the constraints imposed on the datapath for PFT paths. First, constraints on the control signals that can be applied on the paths between a pair of registers in consecutive cycles are extracted. Next, constraints on justifiable data inputs (registers) are extracted. Following these, a combinational constrained Automatic Test Pattern

Generator (ATPG) is used to generate test vectors under the extracted constraints. Thus, in this approach only those vectors are generated that can be applied functionally. Further, the search space is significantly small as only those states are used during test generation which can cause data transfer to take place on a path between a pair of registers.

For testing the controller, the constraints are extracted in the form of state transitions from its RT level description. These constraints also include the values of status signals in the status register and instruction code in the instruction register of the processor. After extracting the constraints, paths are classified as FUT paths or PFT paths. Combinational constrained ATPG is used to generate the test vectors for the paths classified as PFT paths. As the vectors must be generated with constraints on the states and inputs to the controller (contents of the instruction register and status register), the number of time frames that are required for sequential test generation are eliminated. In the final phase test instructions are generated using the knowledge of the control signals and contents of the instruction register. Justification and observation instruction sequence generation processes are based on heuristics which minimize the number of instructions and/or the test application time. In order to test the processor core, the test program (a sequence of generated instructions) is loaded into the memory. The processor fetches the instructions from the memory and after execution, the results are transferred to the memory.

Throughout this chapter the following concepts and notation will be used.

Definition 1 A *path* [31] is defined as an ordered set of gates $\{g_0, g_1, ..., g_n\}$, where g_0 is a primary input or output from a FF, and gn is a primary output or input to a FF. Output of a gate g_i is an input to gate g_{i+1} (0<i<n-1). □

Definition 2 A path is (enhanced-scan or standard-scan) *structurally testable* [11] if there exists a structural test for the path, which can be applied through the (enhanced or standard) full-scan chain. □

Definition 3 A path is *functionally testable* [11] if there exists a functional test for that path, otherwise the path is functionally untestable. □

A functional two-pattern test does not exist to test a path implies that there does not exist an instruction or an instruction sequence to apply the required test

in functional mode of operation. Clearly, functionally untestable paths are never activated in normal (functional) operational mode and we need not target these paths in our approach. We use the following notation to represent signal values.
c : represents a bit which has the same value as in the previous timeframe.
x: represents a bit that can be assigned either a logic 0 or a logic 1 value at will.
d: represents a bit which is not cared by state transition. It is the same as x, except that legitimate bit pattern in the register has to be justified.
R: represents rising transition.
F: represents falling transition.

Definition 4 A constraint *P* is represented by a vector pair and each element of *P* can be 0, 1, x, c, or d. □

Definition 5 A constraint P is said to *cover* a constraint Q if P = Q or Q can be obtained from P by assigning 0 and 1 values to x's in P. □

4.4. Testing Methodology

4.4.1 Datapath Testing

In this section we deal with only those paths that are relevant to data transfer between registers in the datapath. The paths which include the logic in the controller are considered in the next section, even if they start from and end at some registers in the datapath.

Datapath is modeled by an IE-Graph. This is based on the concept of S-Graph proposed in [1]. However, unlike S-Graph, the IE-Graph contains information about data transfer activities associated with an instruction as well as the state during which a given action takes place. IE-Graph is constructed from the instruction set architecture and register transfer level description and includes architecture registers of the datapath.

Nodes of the IE-graph are:

1. registers,

2. two special nodes IN and OUT, which model external world such as memory and I/O devices,

3. part of registers which can be independently readable and writable, and
4. equivalent registers (set of registers which behave in the same way with instruction set, as defined by [3], such as registers in a register file).

A directed edge between two nodes is drawn iff there exists at least one instruction which is responsible for transfering data (with or without manipulation) over the paths between two nodes (registers). Each edge is marked with a set of [state, instruction(s)] pairs, which are responsible for the data transfer between the pair of nodes.

An IE-graph of Parwan processor is shown in Figure 4.7.

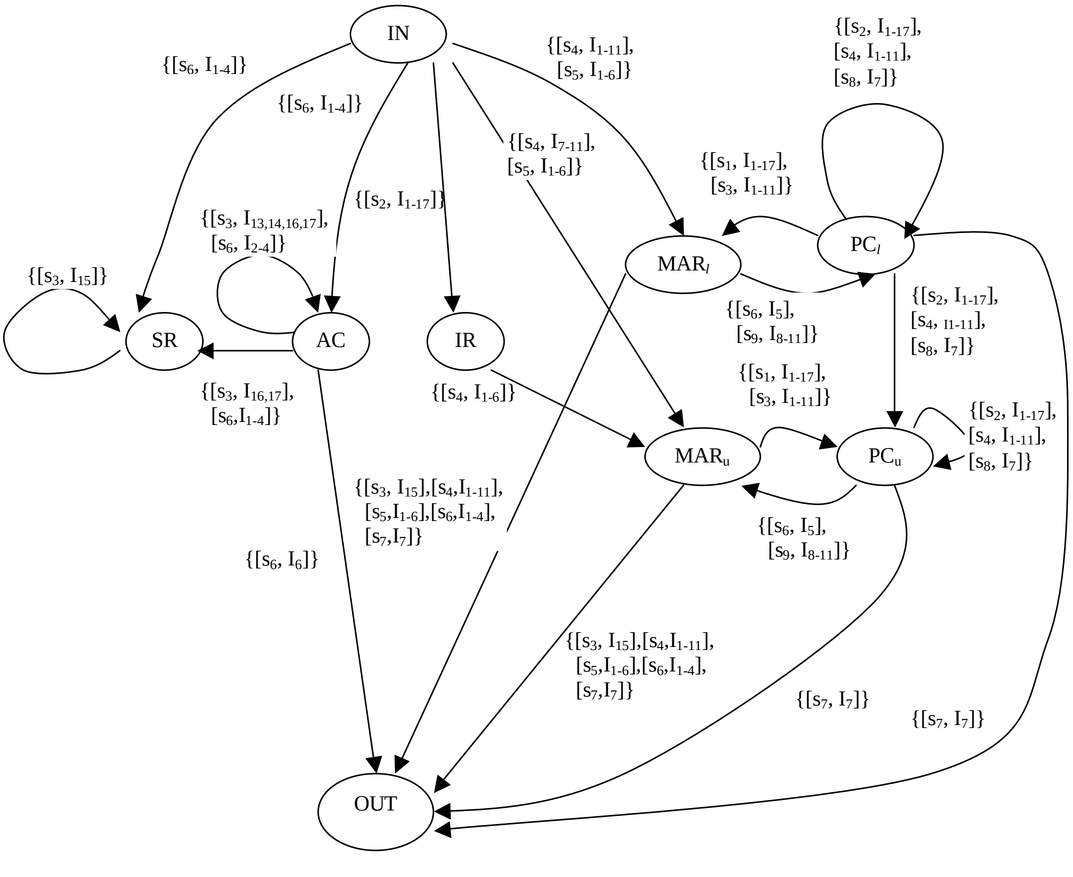


Figure 4.7. IE-graph of Parwan processor

Test vector generation process uses instruction set architecture, RT level description, and gate level netlist. It is a two-step process. The first step is constraint extraction process and the second step is test vector generation process.

Constraint Extraction and Path Classification

There are two types of constraints imposed on the datapath by the controller: i) control constraints, and ii) data constraints. Control constraints are imposed on control signals, which are responsible for transferring data between two registers. These constraints are obtained from IE-graph and RT level description. Data constraints, on the other hand, are the constraints on the justifiable data in the registers under control constraints, which are obtained from RT level description.

Definition 6 Let there be an edge from node R_i to R_o, marked with $[s_l,\ I_p]$. The marked state s_l is defined as a *latching state* for the paths represented by that edge. □

Data transfer activity from register R_i to R_o takes place in state sl during the execution of instruction I_p and register R_o will be latched. Hence, state s_l is defined as a latching state.

Lemma 1 Let $<V_1,\ V_2>$ be a test vector pair for a path from register R_i to a register R_o, where test vector V_1 is followed by V_2 and the edge between these registers is marked with a set of state-instruction pairs $\{[s_l,\ I_p]\}$. This vector pair can be a test vector pair in functional mode only if there exists at least one state-instruction pair $[s_l,\ I_p] \in \{[s_l,\ I_p]\}$, such that

1. vector V_2 can be applied in the latching state s_l of the instruction I_p, and
2. vector V_1 can be applied in the state just before the latching state s_l of the instruction I_p.

□

Note that every instruction is a sequence of state transitions and the latching state(s) in this sequence for a register pair is well defined. However, if the latching state happens to be the very first state of an instruction then the last state of every instruction needs to be considered as the state that immediately precedes the latching state.

During the latching state s_l, data transfer (with or without manipulation) from register R_i to R_o takes place and the result is latched in register R_o. Therefore, we can apply the second vector only in the latching state (say s_l) and the first vector must be applied in a state just before the latching state (say s_j). Two consecutive states s_j and s_l provide the control constraints, and control signals in these states during the execution of instruction(s) marked with the latching state are obtained from RT level description. Constraints on the states during which we can apply the test vectors $<V_1$, $V_2>$ take care of justification of the control signals in the functional mode of testing. Data constraints in the form of justifiable data in the input register of the register pair and other registers required for the execution of marked instruction are obtained from RT level description.

Lemma 2 Paths from register R_i to R_o are *functionally untestable* if the following conditions exist,

1. R_i is not an IN node, and
2. R_i has no incoming edge marked with the state just before the latching state (s_l) of the instruction Ip for any $[s_l, I_p]$ marked on the edge (R_i, R_o).

□

If conditions stated in Lemma 2 exist then transition cannot be launched from register R_i. Hence, the paths between a register pair R_i and R_o are FUT paths. Otherwise, these paths are classified as PFT paths and we need to extract the data constraints for the these paths. Covering relation, defined in section 4.3, helps reduce the number of constraints.

Following examples should help clear the above concepts.

Example 1 Constraints on paths between AC and AC:
The edge between nodes AC and AC is marked with $\{[s_3, (I_{13-14}, I_{16-17})], [s_6, I_{2-4}]\}$, as shown in Figure 2. The previous states of s_3 and s_6 are s_2 and s_4(or s_5), respectively (as shown in Fig. 1). AC is neither an IN node nor it has any incoming

edge which is marked with just previous state of its latching state s_3 or s_6. Therefore, using Lemma 2 we can conclude that paths from AC to AC are functionally untestable. □

Example 2 Paths from IN to AC:
The edge between nodes IN and AC is marked with $[s_6, I_{1-4}]$, as shown in Figure 1. These paths are PFT paths in accordance with Lemma 2, as input node is an IN node, and the latching state for these paths is s_6. Therefore, control constraints are the control signals generated in state s_4 or s_5 followed by s_6 for the instructions I_1, I_2, I_3 or I_4. This is obtained from IE-graph and RT level description. The states s_4 and s_5 are the previous state of the latching state s_6 (as shown in Fig. 4.6). Data constraints can be obtained in the state s_4 followed by state s_6 or state s_5 followed by s_6, for the instructions I_1, I_2, I_3 and I_4. Data constraints for the instruction I_3 are shown in Table 4.2. This table shows the constraints for ALU control signals (ALU ctrl) and SHU control signals (SHU ctrl) as control constraints and constraints for IN and AC as data constraints in consecutive two time frames corresponding to s_4 and s_6 states of the instruction I_3. We consider these constraints as ALU and SHU lie in the paths and AC is the other input needed by the instruction I_3. The extracted data constrained for IN and AC shows that any value can be justified in IN, where as AC must have the same value across two time frames. Here we assume that when input to a combinational logic is in high impedance state then it can hold the logic value that is applied before the high impedance state. Parwan processor uses tristate buses which are responsible for the constraints on IN node. □

Table 4.2. Data constraints for the paths between IN and AC

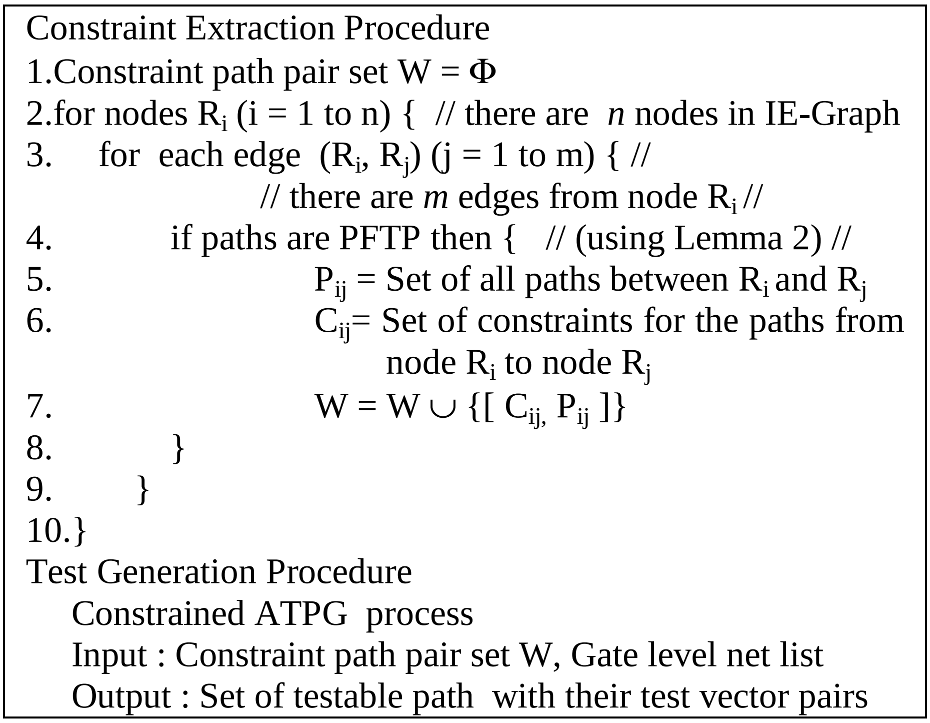
State	I_3(ADD)			
	ALU ctrl	SHU ctrl	IN	AC (other i/p)
s_4	000	00	xxxx_xxxx	xxxx_xxxx
s_6	101	00	xxxx_xxxx	cccc_cccc

For instruction I_3, both control constraints, s_4 followed by s_6 and s_5 followed by s_6, are identical. Hence, using the covering relation one of these two constraints can be eliminated. All other constraints are extracted similarly.

Test Vector Generation Procedure

Constrained ATPG is used to generate the test vectors for the PFT paths under the extracted constraints. Path lists between a register pair and their corresponding constraints are provided as inputs to an ATPG along with gate level netlist and it returns the test vectors for the testable path.

A procedure to extract the constraints and test generation is given in Figure 4.8. This procedure systematically extracts the constraints using IE-Graph and uses constrained ATPG to generate the test vectors.



```
Constraint Extraction Procedure
Constraint path pair set W = Φ
for nodes $R_i$ (i = 1 to n) {  // there are n nodes in IE-Graph
    for each edge ($R_i$, $R_j$) (j = 1 to m) { //
                // there are m edges from node $R_i$ //
        if paths are PFTP then {   // (using Lemma 2) //
            $P_{ij}$ = Set of all paths between $R_i$ and $R_j$
            $C_{ij}$= Set of constraints for the paths from
                node $R_i$ to node $R_j$
            W = W ∪ {[ $C_{ij,}$ $P_{ij}$ ]}
        }
    }
}
Test Generation Procedure
    Constrained ATPG process
    Input : Constraint path pair set W, Gate level net list
    Output : Set of testable path with their test vector pairs
```

Figure 4.8. Constraintextraction and test generation procedures

4.4.2 Controller Testing

As described in section 4.2, controller is a sequential circuit that is normally implemented as a Mealy type or a Moore type finite state machine. Structural organization of the controller is shown in Fig 4.4. In this section, we treat all paths which include logic elements in the controller. Test vectors applicable in functional mode of operation to the controller are restricted by the state transitions. If for a path there exists no sequence of valid state transitions which can launch a transition and propagate it along the path then that path is a functionally untestable path, even though that may be structurally testable. Therefore, we extract constraints on state transitions prior to test generation.

Constraint Extraction

Change of state of controller is determined by the contents of registers (IR and SR), inputs (PI), and the present state. Input from registers IR and SR (i.e., registers other than the present state register (PSR)) are treated as Constrained Primary Input (CPI). Therefore, we need to extract two types of constraints: i) constraints on state transition, and ii) constraint on legitimate values in IR and SR registers, because these are treated as constrained primary input.

1. Constraints on state transitions:
   Constraints on state transitions can be extracted by extracting possible valid state transition under legitimate values in IR, SR and input, by using instruction set architecture and RT level description. We demonstrate this using Parwan processor as an example. Table 4.3 shows a part of the state transition table of Parwan processor.

   This table shows that when present state is s_1 then next state will be either s_1 or s_2 depending on the value of input, and independent of values in IR and SR registers. During these state transitions (s_1 to s_1 or s_2) register IR and SR can have any legitimate value in the present state (s_1) and must have the same values in next state (s_1 or s_2). Hence, we cannot launch transition from IR and SR during these state transitions. When present state is s_2 then next state is always s_3. IR can have any legitimate value in

Table 4.3. State transition table of Parwan processor (partial)

PS	NS	IR (PS)	IR (NS)	SR (PS)	SR (NS)	In (PS)
s_1	s_1	dddd_dddd	cccc_cccc	dddd	cccc	1
	s_2	dddd_dddd	cccc_cccc	dddd	cccc	0
s_2	s_3	dddd_dddd	0xxx_xxxx	dddd	cccc	d
			100x_xxxx	dddd	cccc	d
			101x_xxxx	dddd	cccc	d
			110x_xxxx	dddd	cccc	d
			1110_0000	dddd	cccc	d
			1110_0001	dddd	cccc	d
			1110_0010	dddd	cccc	d
			1110_0100	dddd	cccc	d
			1110_1000	dddd	cccc	d
			1110_1001	dddd	cccc	d

present state (s_2) as well as in next state (s_3). Therefore, transition can be launched from IR during the state transition s_2 to s_3.

2. Constraints on legitimate values in IR and SR registers (registers other than the present state register):
A set of legitimate values in the registers other than the present state register can be obtained from its instruction set architecture and RT level description. For example, the legitimate bit patterns that the register IR of the Parwan processor can have are specified as {IR, < 0xxx_xxxx, 10xx_xxxx, 110x_xxxx, 1111_0100, 1111_0010, 1111_0001, 1110_0000, 1110_0001, 1110_0010, 1110_0100, 1110_1000, 1110_1001>}.

Path Classification

After extraction of constraints, each path is classified as PFT path or FUT path. This process uses state transition diagram and gate level implementation. There are three types of paths in a controller

1. PSR to PSR
2. PI or CPI (registers IR and SR) to PSR
3. PI, CPI, or PSR to a register in datapath.

Paths from PSR to PSR are only responsible for sequential behavior of the controller circuit. For path classification, we construct a table that shows transition on bits in PSR and other registers with state transitions. Table 4.4 shows transition on bits in PSR with state transitions for Parwan processor when states are binary encoded.

Table 4.4. Transition on bits in PSR with state transition (Parwan processor)

bit		s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
b_3	R				s_9					
	F									s_1
b_2	R				s_5,s_6,s_7					
	F						s_1,s_2		s_1	
b_1	R		s_3							
	F				s_5,s_6,s_9					
b_0	R	s_2		s_4		s_6		s_8		
	F		s_3		s_5,s_7,s_9		s_1		s_1	

This table shows that there can be rising transition on bit b_3 only when there is a state transition from state s_4 to s_9. Similarly, there can be falling transition on b_3 only when there is a state transition from state s_9 to s_1.

Lemma 3 Paths between bit i in register R_1 and bit j in register R_2 (registers R_1 and R_2 need not be different) in controller circuit are functionally untestable paths for a transition (rising or falling) if

1. there does not exist a valid state transition s_m to s_n to launch the transition at bit i, or

2. there does not exist a state transition s_p to s_q which can receive the launched transition (launched during the state transition s_m to s_n) or its inverse (receive f alling transition when rising is launched) at bit j, such that $s_n = s_p$.

□

A path is functionally testable if we can create a transition and propagate its effect along the path. If the conditions stated in Lemma 3 exist then we either cannot launch a transition or cannot propagate the created transition. Hence, the paths between bit i in register R_1 and bit j in register R_2 are FUT paths. Otherwise, these paths are classified as PFT paths because transitions can be created and may be propagated if values in other registers are justifiable. We also get precise constraints under which these paths can be tested using state transition table.

1. Paths from PSR to PSR (Paths from controller to controller):
   A path between bit i and bit j in PSR can be classified as follows for rising transition, using Lemma 3. We consider 3 consecutive time frames as shown in the following table. Activities at bit i and j in PSR and required state transitions are listed in Table 4.5. These paths are PFT paths iff either s_m to s_n to s_r, or s_m to s_p to s_q state transition sequence exists.

Table 4.5. Activities at bit i and j in PSR

Time frame	k	$k+1$	$k+2$
bit i	0	1	x
bit j	x	0 (1)	1 (0)
state	s_m	$s_n(s_p)$	$s_r(s_q)$

Example 3 PSR to PSR paths classification in Parwan processor when states are binary encoded. Table 4.4 shows transition on bits in PSR with state transitions. Paths from b_2 to b_1 (for rising transition) are classified as FUT paths because no one state transition sequence exists to test these

paths, where as paths from b_2 to b_1 (for falling transition) are classified as PFT paths because a state transition sequence s_6 to s_2 to s_3 exists. State transition sequence s_6 to s_2 to s_3 is an exact constraint for these paths (b_2 to b_1, falling transition) under which these can be tested if other values are justifiable. Similarly, we can find out all PFT paths, which are the potential candidates for the next phase. □

2. Paths from PI or CPI to bit i in PSR (Paths from input or datapath to controller):

   (a) Paths from PI to bit i in PSR are classified as PFT paths, iff there exists a state sequence (s_m to s_n), which can receive a transition at bit i.

   (b) Paths from CPI to bit i in PSR are classified as PFT paths, iff there exist a valid state transition (s_m to s_n) to create a transition at CPI (register IR or SR) and there exists a valid state transition (s_n to s_p) at bit i of PSR (according to Lemma 2).

3. Paths from CPI or PSR to a register in datapath (Paths from controller to datapath):

   (a) there exists a state sequence (s_m to s_n) which can launch a transition at bit i in PSR or CPI, and

   (b) the register in the datapath, where these paths terminate, has an incoming edge, marked with state sn in IE-Graph and state s_m and s_n are two consecutive states of the marked instruction I_p.

Test Vector Generation

A constrained combinational ATPG is used to generate the test vectors for the paths, which are, classified as PFT paths under the extracted constraints. ATPG is given with a set of PFT paths along with their respective constraints. ATPG will return the test vectors if a path is testable under constraints.

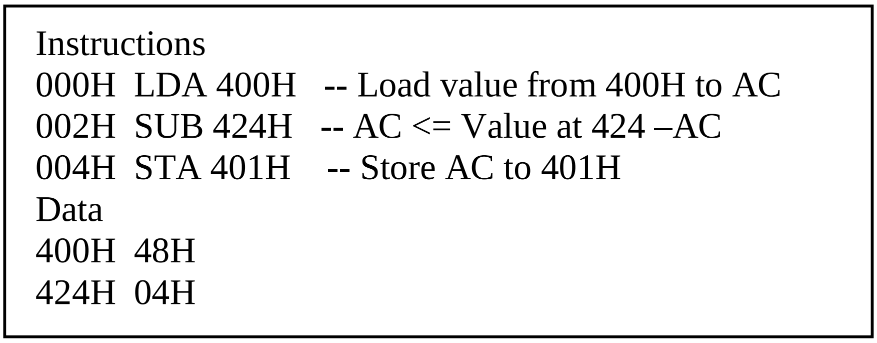
This approach extracts the constraints in the form of state transitions and classifies the paths as functionally untestable or potentially functionally testable.

Functionally untestable paths are removed from the path list. It uses combinational constraint ATPG to generate test vectors. Therefore, we need not consider multiple time frames for all the paths like a sequential ATPG, as sequential behavior is taken care of by the state transition in our approach. This also reduces the complexity of test generation. Note that the vectors generated by us are valid instructions and/or data.

4.4.3 Test Instruction Sequence Generation

The generated test vector pairs as explained in the preceding section are assigned to control signals and registers. Control signals and value(s) in IR in two consecutive time frames give the test instruction(s). Data in registers and in memory, which will be used by the test instruction, must be justified, using justification instruction. The result from the output register must be transferred to memory using observation instructions.

For example, consider a test vector pair (V_1, V_2), where
$V_1 = \{$ALU ctrl=000, SHU ctrl=00, AC=48H, IN=24H$\}$,
$V_2 = \{$ALU ctrl=111, SHU ctrl=00, AC=48H, IN=04H$\}$.



```
Instructions
000H  LDA 400H   -- Load value from 400H to AC
002H  SUB 424H   -- AC <= Value at 424 –AC
004H  STA 401H    -- Store AC to 401H
Data
400H  48H
424H  04H
```

Figure 4.9. Instruction sequence

Figure 4.9 shows the generated test instructions for this example. In this figure, each line shows the memory location followed by an instruction mnemonic (with comments) or a content of the location. The generated test implies that a

SUB instruction (which provides ALU ctrl = 000, SHU ctrl = 00 at s_4, and ALU ctrl = 111, SHU ctrl = 00 at s_6) is applied as a test instruction.

The lower order 8 bit of the memory address used by the SUB instruction (i.e., 424H) must be 24H (value of IN in V_1) and the value stored at this location must be 04H (value of IN in V_2). The value of AC is 48H in V_1. This implies that we should justify the value at AC prior to SUB instruction. This is achieved by LDA instruction that loads 48H to AC. The test instruction SUB stores the result in AC. We can observe the result by STA instruction (store AC) that transfers the result to memory.

We can generate a test instruction sequence for every test vector pair in the above stated manner.

4.5. Experimental Results

A constraint extraction procedure for the datapath and the controller has been implemented in C language. Constrained ATPG for delay fault testing has also been implemented in C language, as commercially available ATPGs are not capable of handling our constraints.

We have applied our methodology to Parwan processor [28] and DLX processor [29]. The synthesized version of the Parwan processor contains 888 gates and 53 flip-flops and DLX processor contains 16152 gates and 1446 flip-flops. IE-graphs for these processors are constructed and functionally untestable paths are identified. For example, the paths from AC to AC, AC to SR, AC to OUT, SR to SR, SR to OUT and PC to PC in the datapath of the Parwan processor are found to be untestable. We extract constraints for rest of the paths using IE-Graph and RT level description. Similarly, state transition tables for both of the processors are constructed which show the constraints on the controllers.

We generated test patterns for Robust (Rob), Non Robust (NR) [31] and Functional Sensitizable (FS) [31] paths under the extracted constraints using our constrained ATPG. The test vectors are generated in the following order: Rob, NR, and FS. For each path, first it generates a test vector pair for the robust test, if exist under the constraints; otherwise go for NR test followed by FS test. Here we consider, the paths which are starting from some register in datapath (e.g., IR or SR) going through the controller and terminating at some register in datapath, as a part of the controller. These paths are large in number (about 98-99% of the total paths in the controller). Results are shown in the tables 4.6 and 4.7.

The results show that 37% of paths in Parwan datapath, 46% paths in Parwan controller, and 17% paths in DLX datapath are identified as functionally untestable paths and these are eliminated during the first phase without using circuit details. However, all the paths in the controller of the DLX processor are determined to be potentially functionally testable paths because of the completely specified state space in the state encoding (64 states are encoded in 6 bits). We have extracted the constraints for these paths efficiently and achieved 100% fault efficiency. The paths shown as functionally untestable in 9^{th} row of the table 4.6 and 4.7 include the paths which are declared functionally untestable in first phase. NR testable paths include the robust testable paths, and FS testable paths include Robust and NR testable paths. The CPU time shown in table 4.6 and 4.7 is the total time taken by ATPG to generate Rob, NR and FS test vectors.

Table 4.8 shows that our methodology achieves higher fault coverage for the datapath of Parwan processor as compared to [12] because we are considering at microinstruction level during the extraction of constraints for the potentially testable paths. Note that [12] uses a different synthesized version of Parwan processor with 168 sequential elements in order to separate out controller and datapath to make it better testable and reduction of number of paths, where as we are using original Parwan processor. Similarly, their DLX processor is also differently synthesized. The results for the controller are not shown in [12]. Our approach can eliminate substantial number of paths without looking into the circuit details, where as [12] uses the circuit details for path classification. Moreover, [12] has not shown the results other the results for NR test for the datapath of Parwan and DLX processors.

Table 4.6. Results for Parwan processor

	Datapath			Controller		
	Rob	NR	FS	Rob	NR	FS
Total Path	5,217	5,217	5,217	174,362	174,362	174,362
No. of faults	10,434	10,434	10,434	348,724	348,724	348,724
Faults declared untestable in first phase	3,902	3,902	3,902	162,812	162,812	162,812
Percentage of eliminated faults	37.4	37.4	37.4	46.6	46.6	46.6
No. of functionally testable faults	156	1,653	2,618	407	2,417	3,520
Fault coverage (%)	1.5	15.8	25.0	0.1	0.7	1.0
No. of unctionally untestable faults	10,278	8,781	7,816	348,3173	346,307	345,204
Fault efficiency (%)	100	100	100	100	100	100
CPU time (ATPG)	3 minutes 41 sec			3 hours 27 minutes		

Table 4.7. Results for DLX processor

	Datapath			Controller		
	Rob	NR	FS	Rob	NR	FS
Total Path	264,906	264,906	264,906	743,411	743,411	743,411
No. of faults	529,812	529,812	529,812	1468,822	1468,822	1468,822
Faults declared untestable in first phase	34,924	34,924	34,924	0	0	0
Percentage of eliminated faults	17.0	17.0	17.0	0	0	0
No. of functionally testable faults	19,354	34,924	44,324	15,146	42,295	112,735
Fault coverage (%)	3.6	6.5	8.3	1.0	2.8	7.6
No. of unctionally untestable faults	510,458	494,888	485,488	1453,676	1426,527	1356,087
Fault efficiency (%)	100	100	100	100	100	100
CPU time (ATPG)	1 hour 32 minutes			15 hours 18 minutes		

Table 4.8. Comparison with earleir work (Fault Coverage)

		Parwan		DLX	
		Lai [10] work	Our work	Lai [10] work	Our work
Datapath	Rob	–	1.5	–	3.6
	NR	3.7	15.8	7.2	6.5
	FS	–	25.0	–	8.3
Controller	Rob	–	0.1	–	1.0
	NR	–	0.7	–	2.8
	FS	–	1.0	–	7.6

4.6. Conclusion

A systematic approach for the delay fault testing of processor core using its instruction set has been presented in this chapter. A graph theoretic model for data path has been developed. This model is used with the RT level description to eliminate the functionally untestable paths at the early stage and it is also used for extraction of constraints. Controller is modeled as a finite state machine and constraints on state transitions are extracted. This eliminates the need for multiple time frame consideration for test generation, and hence reduces the test generation complexity. Our experimental results show that our test generation process can efficiently generate test vectors for functionally testable paths which can be applied by test instructions.

Chapter 5

Pipelined Processor Testing

5.1. Introduction

Although nearly all modern processors use pipelined architecture, yet no method has been proposed in literature to model these for the purpose of test generation. This chapter proposes a graph theoretic model of pipelined processors and develops a systematic approach to path delay fault testing of such processors using the processor instruction set. To the best of our knowledge no approach has been proposed for delay fault testing of pipelined processors in functional mode.

This chapter describes the pipelined processor architecture first and then overview of the approach. It describes a graph theoretic model of a pipelined processor in section 5.4. Section 5.5 describes the testing methodology for datapath and controller. Experimental results are presented in section 5.6 to demonstrate the effectiveness of the proposed approach.

5.2. Pipelined Processor Architecture

Pipelining [29] is a powerful implementation technique for enhancing system throughput without requiring massive replication of hardware. This architectural approach allows the simultaneous execution of several instructions. Pipelining is transparent to the programmer; it exploits parallelism at the instruction level by overlapping the execution process of instructions. It is analogous to an assembly line where workers perform a specific task and pass the partially completed product to the next worker. Pipelining is a technique that is now widely employed in the design of instruction set of processors. This chapter focuses on the scalar pipelined processors. The current trend is towards very deep pipelines. Pipeline depth has increased from less than 10 to more than 20. Deep pipelines are necessary to achieve very high clock frequencies. This has been very effective in gaining greater processor performance. There are some indications that this trend will continue.

The primary motivation for pipelining is to increase the throughput of a system with little increase in hardware. Pipelining involves partitioning of the system into multiple stages with added buffering between the stages. These stages and the inter-stage buffers constitute the *pipeline*. The computation carried out by the original system is decomposed into k sub-computations, carried out in k stages of the pipeline. A new task in pipeline can start into the pipeline as soon as the previous task has traversed the first stage. The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of assembly line tries to balance the time for each step in the process. Given that the total number of tasks to be processed is very large, the throughput of the pipelined system can potentially approach k times that of a non-pipelined processor in case of balanced pipeline. This potential increase in the performance by a factor of k by simply adding new buffers in a k–stage pipeline is the primary attraction of the pipelined design.

A pipeline stage performs a particular function and produces an intermediate result. It consists of an input latch, also called a register or buffer, followed by a processing circuit (A processing circuit can be combinational or sequential circuit). The processing circuit of a given stage is connected to the input latch of the next stage as shown in Figure 5.1. A clock pulse, every stage transfers its intermediate result to the input latch of the next stage. In this way, the final result is produced after the input data have passes through the entire pipeline, completing one stage per clock pulse. The period of the clock pulse should be large enough to provide sufficient for a signal to traverse through the slowest stage, which is called the bottleneck (i.e., the stage needing the longest amount of the time to complete).

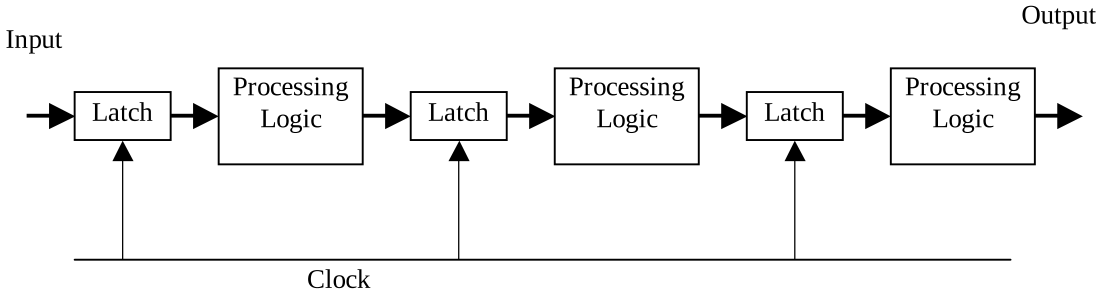


Figure 5.1. Basic structure of a pipeline

As described in chapter 4, the process of executing an instruction involve several steps. First, the control unit of a processor fetches the instruction from the cache (or from memory). The control unit decodes the instruction to determine the type of operation to be performed. When the operation requires operands, the control unit also determines the address of each operand and fetches them from cache (or memory). Next, the operation is performed on the operands and, finally, the result is stored in the specified location. A five-stage instruction pipeline is shown in Fig. 5.2, which consists of following stages:

1. Instruction Fetch (IF): Retrieval of instructions from cache (or main memory).

2. Instruction Decoding (ID): Identification of operations to be performed, and operand read.

3. Execution (EX): Perform operations on the operands.

4. Memory (MEM): Read from or write to memory.

5. Write-back (WB): Updating the destination operands.

An instruction pipeline overlaps the process of the preceding stages for different instructions to achieve a much lower total completion time, on average, for a series of instructions. As an example, consider Table 5.1, which shows the execution of four instructions in an instruction pipeline. During the first cycle,

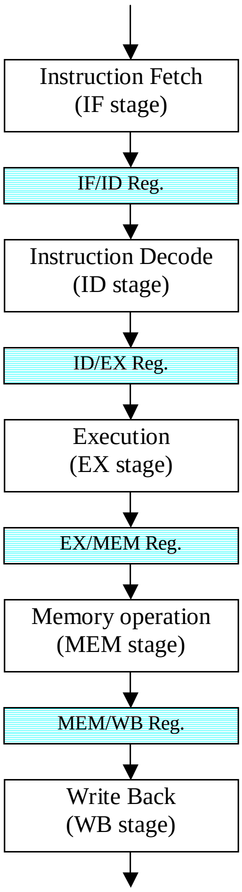


Figure 5.2. Five-stage instruction pipeline

or clock pause, instruction i_1 is fetched from memory. Within the second cycle, instruction i_1 is decoded while instruction i_2 is fetched. This process continues until all the instructions are executed. The last instruction finishes the write back stage after 8 clock cycles.

The major hurdle of Pipelining – Pipeline hazards

There are situations, called hazards that prevent the next instruction in the instruction stream from executing during its designated clock period. Hazards reduce the performance from the ideal speed up gained by pipelining. There are three classes of the hazards:

Table 5.1. Execution of instruction in a five stage pipeline

Instruction	1	2	3	4	5	6	7	8
i_1	IF	ID	EX	MEM	WB			
i_2		IF	ID	EX	MEM	WB		
i_3			IF	ID	EX	MEM	WB	
i_4				IF	ID	EX	MEM	WB

1. Structural hazards arise from source conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

2. Data hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

3. Control hazards arise from the overlapping of branches and other instructions that change the program flow.

Hazards in pipelines can make it necessary to stall the pipeline, which in turn degrade the performance. Eliminating a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed. When an instruction is stalled, all instructions issued later than the stalled instruction – and hence not as far along the pipeline – are also stalled. Instructions issued earlier than the stalled instruction – and hence farther along in the pipeline – must continue, since otherwise the hazard will never clear. As a result no new instructions are fetched during the stall.

Structural Hazards: When a machine is pipelined, the overlapped executions requires pipelining of functional units and duplications of resources to allow all possible combinations in the instruction in the pipeline. If some combinations of the instructions cannot be accommodated because of resource conflict, the machine is said to have a structural hazard. The most common instances of structural hazards arise when functional unit is not fully pipelined or some resource has not been duplicated enough. The designer allows structural hazards in order to reduce the cost of the system and latency of the unit. Pipelining all functional units, or duplicating them, may be too costly.

Data Hazards: A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write access to operands so that the order differs from the order seen by the sequential executing instruction on a non-pipelined machine.

This problem can be solved by a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*), where result of the preceding instruction is forwarded to later dependent instruction without going through write back. It makes controller quite complex.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i occurring before j. The possible data hazards are:

- RAW (Read after write) – j tries to read a source before i write it, so j incorrectly gets the old value. This is most common type hazard and the kind that we use forwarding to overcome.

- WAW (Write after write) – j tries to write an operand before it is written by i. The writes end up being performed in wrong order, leaving the value written by i rather than the value written by j. This hazard is present only in pipelines that write in more than one pipeline stage (or allow an instruction to precede even when a previous instruction is stalled).

- WAR (Write after read) – j tries to write a destination before it is read by i, so i incorrectly gets the new value. These are rare hazards.

Control Hazards: Control hazards occur when an instruction such as branch instruction, causes a change in the program flow.

A general organization of 5 stage pipeline processor is shown in Figure 5.3.

We will use a 16–bit, 5–stage pipelined processor VPRO design which has most common 24 instructions to demonstrate the concept. This processor uses load/store RISC architecture. It has register type, immediate type, and jump type instruction formats. This represents the features of most of the pipelined RISC processors. Its instruction set architecture and structural organization is given in the appendix.

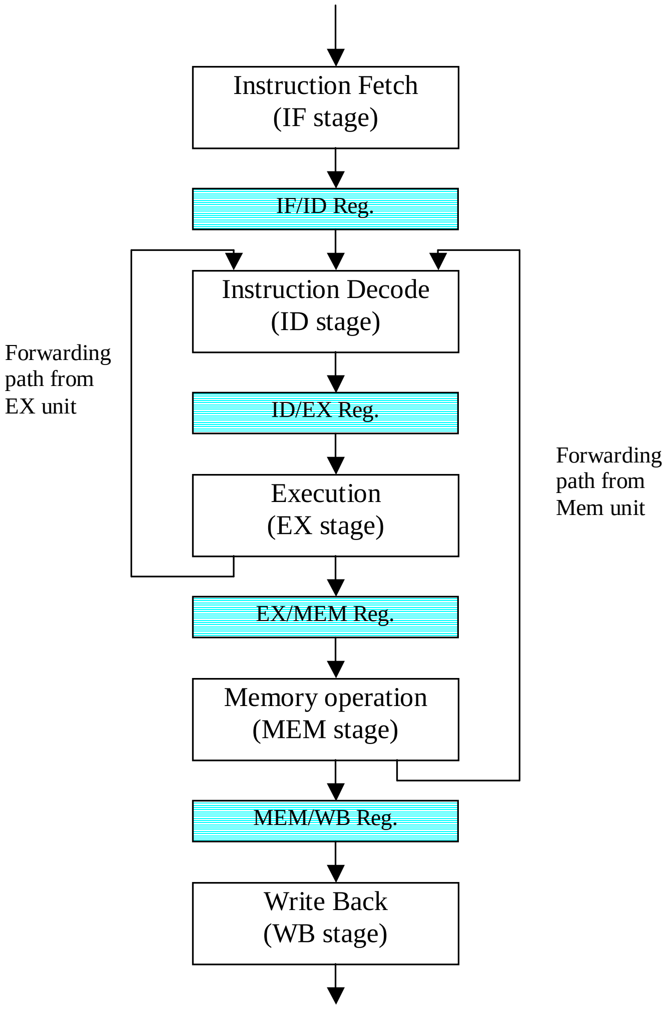


Figure 5.3. Five-stage instruction pipeline with forwarding

5.3. Overview of the Proposed Approach

The objective of this work [22] , [23] is to develop a procedure for delay fault testing of a pipeline processor core that can be used to generate tests for the functional mode of operation of a processor using its instruction set. The main objectives of this work are:

- Develop a graph theoretic model for pipeline behaviour using the RT level description of the processor
- Provide a systematic approach to test the processor based on the developed model
- Evaluate the method using experimental studies

This chapter presents a unified approach to test all normal and bypassing/ forwarding paths in the datapath and all paths in the controller by using a graph model of the behaviour of the processor. A hierarchical approach is presented for the test generation which classifies paths at RT level and extracts the constraints for potentially functionally testable paths to generate test vectors at gate level using constrained ATPG. Path delay [31] fault model is used in this work.

As discussed in the last section, unlike a non-pipelined processor which completes execution of one instruction before the execution of the next instruction, in a pipelined processor multiple instructions can be in various stages of execution. These stages can be viewed as independent hardware units and all the stages execute instructions concurrently. In order to support concurrent execution of instructions, necessary data and control signals are carried along as an instruction progresses in the pipeline stages. Simultaneous execution of multiple instructions can lead to data, control and structural hazards. Data bypassing is a commonly used technique to resolve data hazards; stalling is used for the unresolved hazards. Data flows from the first pipeline stage to the last pipeline stage during the normal execution (without any hazard). The simultaneous execution of multiple instructions in various stages and the use of data forwarding/bypassing mechanism make the behaviour of pipelined processor complex.

It is very difficult to separate datapath and controller parts clearly in a pipelined processor as every pipeline stage carries all the data and control signals required by the pipeline stages ahead of it. Nonetheless, our model defines them clearly and considers the paths in the datapath part and the control part separately. The data transfer activities between the architectural registers and data and address (memory address and register address) part of the pipeline registers is assumed to be in the datapath. The paths, which go through the control logic, are considered in the control part. The activities in datapath are controlled by the control signals which are carried forward with the data; thus the function of datapath is constrained by the controller. Hence, only a subset of structurally applicable test vectors may be applied in the functional mode of operation due to the presence of constraints.

A graph theoretic model called pipeline instruction execution graph (PIE-graph), has been developed that is constructed by using the instruction set architecture and RT level description. It is based on instruction execution graph (IE-graph), described in chapter 4, for non-pipelined processors. This graph models the complex pipeline behaviour. Our present model classifies paths as functionally testable (FT), functionally untestable (FUT), potentially functionally testable (PFT), and parity check functionally untestable (PCFUT) paths. After the classification, it extracts constraints for the PFT and PCFUT paths. First, constraints on the control signals in one or more relevant pipeline stages are extracted and then the constraints on justifiable data in the data registers or pipeline registers under the control constraints are extracted. PCFUT paths are further classified as FUT paths or PFT paths. A combinational constrained ATPG is used for the test vector generation for the PFT paths. We can get test sequences without using ATPG for FT paths, and no test sequence is needed for FUT paths. For testing the controller, the constraints on the legitimate values for a group of control signals are extracted by using the RT level description. PIE-graph is used along with these constraints for further extraction of control and data constraints for target control paths, and their classification. Constrained ATPG is used to generate the test vectors. Finally, instruction sequences to apply the generated test vectors, are generated by using the knowledge of the control signals of various pipeline stages and the PIE-graph. Test program generation flow is shown in Figure 5.4.

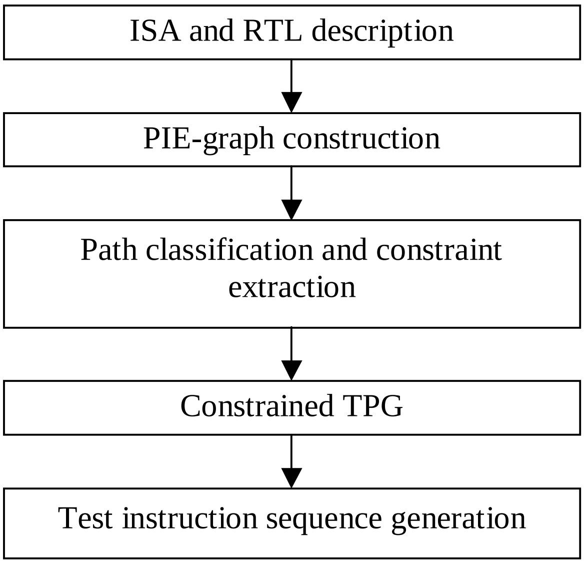

ISA and RTL description
PIE-graph construction
Path classification and constraint extraction
Constrained TPG
Test instruction sequence generation

Figure 5.4. Test program flow

5.4. Pipeline Instruction Execution Graph

Data transfer activities between the data registers of a pipelined processor can be modeled by PIE-graph, which as stated earlier, is based on IE-graph, described in chapter 4, for non-pipelined processors. IE-graph models the behaviour of a simple non-pipelined, FSM based processor. PIE-graph can be constructed from the instruction set architecture, and RTL description of a processor. It captures the pipeline behaviour. This includes architecture registers and data and address part of the pipeline registers. Note that this does not include the control part of the pipeline registers.

Definition 7 The number of pipeline stages bypassed by a path is defined as *distance* associated with the path. □

We noticed that many paths directly transfer data to the next stage using simple interconnects or through a set of multiplexers. Keeping this in mind we classified logic into three types: i) interconnect (I), ii) multiplexers (M), and iii) processing logic (L). This classification simplifies the test generation process.

This information can be obtained from RTL description even when full structural description is not available.

Nodes of the PIE-graph are:

1. architectural registers,
2. part of architectural registers which can be independently readable and writeable,
3. equivalent registers (set of registers which behave identically as a group with instruction set, such as register file),
4. two special nodes, IN and OUT, which model the external world such as memory and IO devices, and
5. data and address (memory address and register address) part of pipeline registers.

A directed edge between two nodes is drawn iff there exists at least one instruction responsible to transfer data (with or without manipulation) over the edge (paths) between the two registers corresponding to nodes. Each edge is marked with a 4-tuple [⟨*instruction set*⟩, ⟨*stage from, stage to*⟩, ⟨*distance*⟩, ⟨*logic type*⟩]. This 4-tuple signifies that a set of instructions ⟨*instruction set*⟩ is responsible for the data transfer from ⟨*stage from*⟩ stage to ⟨*stage to*⟩ stage through the logic type ⟨*logic type*⟩, and the pair of instructions for delay testing must be separated by the cycles specified by the ⟨*distance*⟩.

If a pipeline register is a source node then the pipeline stage succeeding it will be ⟨*stage from*⟩ stage; otherwise, the stage controlling the data transfer activities in the register will be in ⟨*stage from*⟩. If a pipeline register is a destination node, then the stage just before it (whose data it is latching) will be ⟨*stage to*⟩ stage, otherwise the stage that controls the data write activity in the register is ⟨*stage to*⟩. Data transfer activities inside a pipeline stage are modeled by keeping the same ⟨*stage from*⟩ and ⟨*stage to*⟩ and zero distance, and the data transfer activities across the pipeline stages (mainly bypassing paths) are modeled by using ⟨*stage*

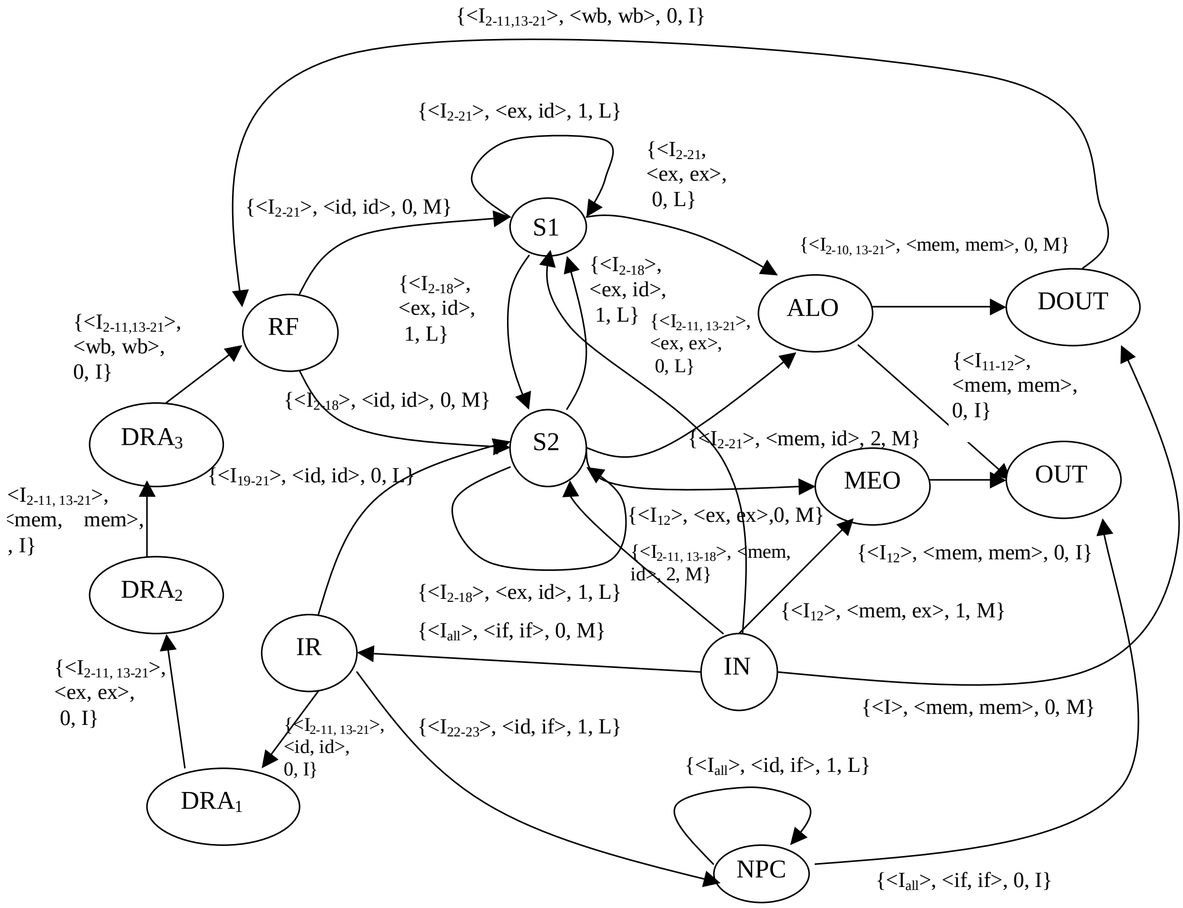

{<$I_{2-11,13-21}$>, <wb, wb>, 0, I}
{<I_{2-21}>, <ex, id>, 1, L}
{<I_{2-21}, <ex, ex>, 0, L}
{<I_{2-21}>, <id, id>, 0, M}
S1
{<$I_{2-10, 13-21}$>, <mem, mem>, 0, M}
{<I_{2-18}>, <ex, id>, 1, L}
{<I_{2-18}>, <ex, id>, 1, L}
ALO
DOUT
RF
{<$I_{2-11,13-21}$>, <wb, wb>, 0, I}
{<$I_{2-11, 13-21}$>, <ex, ex>, 0, L}
{<I_{11-12}>, <mem, mem>, 0, I}
{<I_{2-18}>, <id, id>, 0, M}
DRA$_3$
S2
{<I_{2-21}>, <mem, id>, 2, M}
{<I_{19-21}>, <id, id>, 0, L}
MEO
OUT
<$I_{2-11, 13-21}$>, <mem, mem>, , I}
{<I_{12}>, <ex, ex>,0, M}
{<$I_{2-11, 13-18}$>, <mem, id>, 2, M}
{<I_{12}>, <mem, mem>, 0, I}
DRA$_2$
{<I_{2-18}>, <ex, id>, 1, L}
{<I_{12}>, <mem, ex>, 1, M}
{<I_{all}>, <if, if>, 0, M}
IR
IN
{<$I_{2-11, 13-21}$>, <ex, ex>, 0, I}
{<I>, <mem, mem>, 0, M}
{<$I_{2-11, 13-21}$>, <id, id>, 0, I}
{<I_{22-23}>, <id, if>, 1, L}
{<I_{all}>, <id, if>, 1, L}
DRA$_1$
NPC
{<I_{all}>, <if, if>, 0, I}

Figure 5.5. PIE-graph of VPRO processor

from⟩, ⟨*stage to*⟩ and appropriate distance to create hazard in order to transfer data over that.

A complete PIE-graph of 5 stage pipelined VPRO processor is shown in Fig. 5.5.

5.5. Testing Methodology

5.5.1 Datapath Testing

This section deals with the paths that transfer data between architectural registers or data and address part of the pipeline registers, which are significant in number. Other paths will be considered in the control part. Datapath of a pipeline processor is modeled by PIE-graph, and is used for the constraint extraction, path classification, and instruction sequence generation.

We assume that any instruction can be followed by any other instruction in a pipeline stage except those instructions, which always need stall after the execution such as unconditional jumps. In order to test a path from register R_i to register R_o, we must create a transition at R_i and capture the transferred data at R_o. Although there may be paths with *distance* > 0 (bypass paths) to R_i, there is guaranteed to be a path with zero distance (normal path) which brings the same values as the bypass paths, and hence we only need to consider the normal path ($d = 0$) for data transfer to R_i. We also allow the propagation of data to R_o through normal paths except from R_i. This observation prunes the search space substantially.

Definition 8 Instructions, which behave identically within a pipeline stage, are defined as equivalent instructions, for that stage. □

For example, ADD and INC behave identically in EX stage of VPRO processor, hence, these are the equivalent instructions in EX stage. Similarly, instructions (ADD, ADDU, ADDI, ADDUI, LW, LH, LB, SW, SH, SB) are the equivalent instructions for EX stage of the pipelined DLX processor. We can use these equivalent instructions to reduce the marked instructions which in turn reduce the constraint extraction and test generation effort. We make a table of the equivalent instructions for every stage, which is also used during instruction sequence generation.

Example 4 The bypass paths from memory to register S1 of VPRO processor (represented by an edge between IN and S1) can be tested by the following instruction sequence:

I_1: LOAD R1, R5 – [R1] $\Leftarrow$ Mem[R5]

I_2: LOAD R2, R6 – [R2] $\Leftarrow$ Mem[R6]

I_3: ADD R3, R1, R0 – [R3] $\Leftarrow$ [R1] + [R0]

I_4: ADD R4, R2, R0 – [R4] $\Leftarrow$ [R2] + [R0]

The edge between IN and S1 is marked with distance 2 means these paths bypass two pipeline stages. We need four instructions to test these paths. The instructions I_1 and I_2 launch a transition from memory and propagate the launched transition in *mem* stage, and the instructions I_3 and I_4 propagate the transition in the *decode* stage. Finally, the instruction I_4 transfers the result to register R4. During first cycle, the instructions I_1 and I_3 execute concurrently in *mem* stage and *decode* stage respectively and during second cycle, the instructions I_2 and I_4 execute concurrently in *mem* stage and *decode* stage respectively. Therefore, this sequence can test the paths from IN to S1.

□

A path from register R_i to register R_o, marked with $[\langle\ I_{set}\ \rangle$, $\langle S_j$, $S_i\rangle$, d, $LT]$, where $LT \in \{I, M, L\}$, can be tested by a test instruction sequence (IP_1, IP_2, ID_1, ... ,ID_{d-2}, IS_1, IS_2), where ID_1, ID_2, ... , ID_{d-2} are the $(d-2)$ filler instructions. To test a path, we need constraints for both stages S_j and S_i in two consecutive cycles (as shown in Fig. 5.6). Instruction pair (IP_1, IP_2) is responsible to create a transition at register R_i and allows it to propagate in S_j stage. Instruction pair (IS_1, IS_2) is responsible to propagate the created transition in S_i stage and finally instruction IS_2 latches the result in register R_o. The instructions IP_1 and IS_1 must be executed concurrently in the stages S_j and S_i respectively. Similarly, the instructions IP_2 and IS_2 must be executed concurrently in the stage S_j and S_i. Other instructions (ID_1, ... ,ID_{d-2}) are used to provide the proper distance between the instructions IP_2 and IS_1, so that a transition along the path can be excited, propagated and the result will be latched. We assume that data from another stages (ex. data from S' in Fig. 5.6) come through MUX and such data do not affect the data transfer along the target path. Therefore, ATPG does not care these values and we do not need to extract their constraints. A sequence of $d+2$ instructions is needed to test these paths. Note that $IP_2 = IS_1$ if $d = 1$, and $IP_1 = IS_1$ and $IP_2 = IS_2$ if $d = 0$.

Instruction pair (IP_1, IP_2) must be marked on any zero distance (with $d = 0$) in-edge of R_i and instruction pair (IS_1, IS_2) must be marked on the target path (edge between R_i and R_o). If source node is IN node then any load instruction

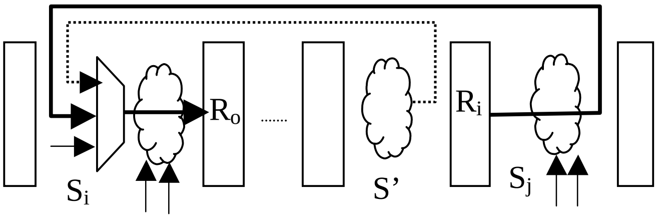


Figure 5.6. Target path and pipeline stages

can be used as IP_1 or IP_2. If target path is inside a pipeline stage ($d = 0$) then we don't need dummy instructions, and are left with an instruction pair (IS_1, IS_2). The instructions IS_1 and IS_2 must be marked on any zero distance in-edge of R_i and at target edge respectively.

Path Classification and Constraint Extraction

Our approach classifies all paths into four categories, which are: (i) functionally testable (FT) paths, ii) functionally untestable (FUT) paths, iii) potentially functionally testable (PFT) paths, and iv) parity check functionally untestable (PCFUT) paths. We can get a test sequence without using ATPG for FT paths, whereas we don't need to generate test for FUT paths. For the rest of the categories we need to extract architectural constraints.

There are two types of constraints: i) control constraints, and ii) data constraints. Control constraints are the constraints on control signals, which are responsible to transfer data between two nodes. These are obtained from PIE-graph. Data constraints are the constraints on justifiable data under the control constraints. Control constraints are extracted as instruction pairs (IP_1, IP_2) and (IS_1, IS_2). Note that Non-Robust test [31] does not take care of first vector for off inputs. Therefore, we need to extract a set of instructions for IP_2 and IS_2 instead of instruction pairs (IP_1, IP_2) and (IS_1, IS_2). We can easily get the set of instructions for IP_2 and IS_2 from the instructions marked on the input edge to R_i and on target edge respectively.

Let there be an edge between nodes R_i and R_o, marked with [$\langle I_{set}\rangle$, $\langle S_j$, $S_i\rangle$, d, LT]. Constraints for paths of various types of logic are extracted as follows:

1. when logic type is interconnect 'I':
   These paths are generally used to carry forward data to the next stage and are always with $d = 0$. R_o has only one in-edge, which is from R_i. An instruction sequence (IS_1, IS_2) is needed to test. Any instruction marked on the zero distance ($d = 0$) in-edge of register R_i, can be used as IS_1, and any instruction marked in the target edge can be used as IS_2. These two instructions give the constraint on the control signals in S_i stage. R_o has no other in-edge; hence, it will not observe any data constraint. These paths can be tested as interconnects test. Therefore, these paths are classified as FT paths.

2. when logic type is multiplexer 'M':
   These paths pass through a set of MUXs and behave as interconnects if control signals are properly assigned. Therefore, under the control constraints (proper assignment of MUX select signals), data constraints are not applicable, as other paths to R_o will automatically be deselected with the proper assignment of MUXs control signals. These paths can be tested as interconnect test.
   We consider two different distance cases separately:

   (a) when $d = 0$ (Normal flow inside a pipeline stage):
   These paths transfer the data inside the same stage. Therefore an instruction pair (IS_1, IS_2) is needed to test these paths. Any instruction marked on the zero distance ($d = 0$) in-edge of R_i can be used as IS_1, and any instruction marked on the target edge can be used as IS_2. Instruction pair (IS_1, IS_2) gives the constraints on the control signals in S_i stage. These paths always find a sequence of instructions without any data constraints, hence classified as FT paths.

   (b) when $d > 0$ (data flow across the pipeline stages, i.e., forwarding path):
   These paths are responsible for the data transfer across the pipeline stages. These paths are classified as FUT paths if these are marked with $d = 1$ and have a self-loop because a transition cannot be launched. Other paths can be tested by an instruction sequence (IP_1, IP_2, ID_1, ... , ID_{d-2}, IS_1, IS_2). Instructions IP_1 and IP_2 must be marked on

any of zero distance in-edge of R_i, and instructions IS_1 and IS_2 must be marked on the target edge. Therefore, the instruction pair (IP_1, IP_2) gives the control constraints on the control signals of stage S_j, and the instruction pair IS_1, IS_2 gives the constraints on the control signals in stage S_i. These paths are classified as FT paths.

3. when logic type is processing logic 'L':
This includes the paths which pass through the combinational logic. Let an edge between two registers R_i and R_o be marked with $[\langle I_{set1}\rangle$, $\langle S_j$, $S_i\rangle$, d, $LT]$. Following edges and nodes must be considered: i) all the in-edges to R_o with distance d and logic type 'L' (having some instructions common with I_{set1}), ii) all the in-edges to R_o with zero distance, logic type 'L', and have some instruction common with I_{set1}, and iii) all zero distance ($d = 0$) in-edges to R_i.
All those registers which have out-edge to R_o (with distance d – same as distance of target path, logic type 'L', and some instruction common with the target edge) provide the data constraints for the propagation of created transition in S_j stage. All those registers which have out-edge to R_o (with zero distance, logic type 'L', and have some instructions common with the target path) provide data constraints for the propagation of the created transition in S_i stage. Fig. 5.7 shows the edges and nodes which are needed to be considered. Note that $I_{set1} \cap I_{set2} \neq \phi$, and $I_{set1} \cap I_{set3} \neq \phi$.

We consider two different distance cases separately:

(a) when $d = 0$ (Normal flow inside a pipeline stage):
This includes the paths in a single stage. We need an instruction pair (IS_1, IS_2) for testing. IS_1 can be any instruction among the instructions marked on the in-edge of R_i, and IS_2 can be any instruction among the instructions marked on the target edge. An instruction pair (IS_1, IS_2) gives the constraint on control signal in S_i stage. We have to find out the data constraints on all those registers which have zero distance in-edge to R_o with logic type 'L' using PIE-graph and RTL description. Let R_o has an in-edge from register R_p which is marked

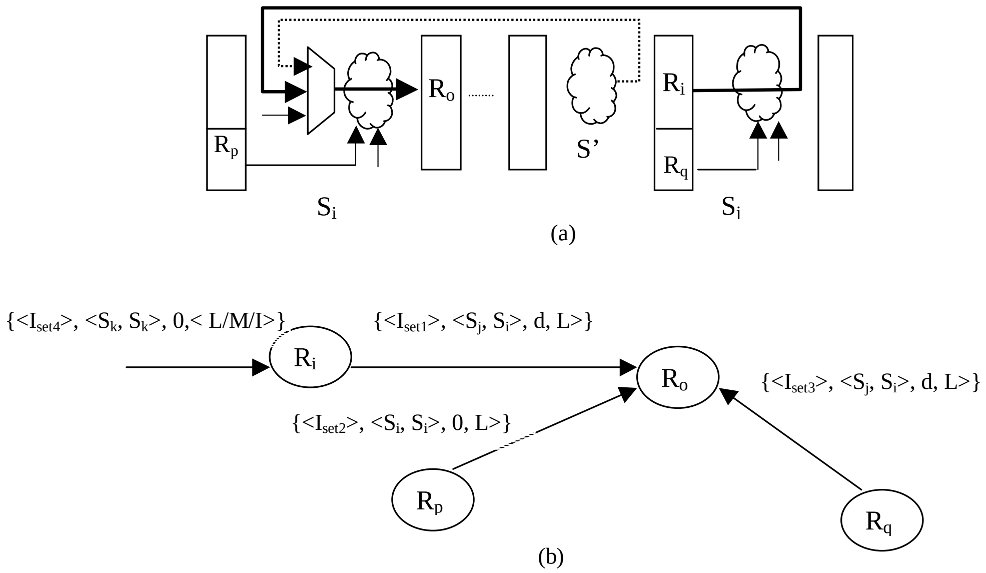

Rp
Ro
Ri
Rq
S'
Si
Si
(a)
{<Iset4>, <Sk, Sk>, 0,< L/M/I>}
{<Iset1>, <Sj, Si>, d, L>}
Ri
Ro
{<Iset3>, <Sj, Si>, d, L>}
{<Iset2>, <Si, Si>, 0, L>}
Rp
Rq
(b)

Figure 5.7. Edge consideration for constraint extraction (a) structural organization (b) edges in PIE-graph

with common instructions with the target edge. Data constraints for these registers can be obtained as follows.

If the selected instruction IS_2 is not marked on any of the in-edge of R_p, then the register R_p must have constant value across two time frames (under IS_1 and IS_2).

(b) when $d > 0$ (data flow across pipeline stages, i.e., forwarding path): These paths are responsible to transfer data across the pipeline stages. These paths need $(d+2)$ consecutive instructions $(IP_1$, IP_2, ID_1, . . . , ID_{d-2}, IS_1, $IS_2)$ to test, which consists of $(d-2)$ filler instructions to excite these paths. IP_1 and IP_2 can be any instruction marked at any zero distance in-edge of R_i, and IS_1 and IS_2 can be any instruction marked at target edge. Instruction pair $(IP_1$, $IP_2)$ gives the control constraints on the control signals of S_j stage, and instruction pair $(IS_1$, $IS_2)$ gives the constraint on the control signals of S_i stage. Note that $IP_1 = IS_1$ for $d = 1$, and we need an instruction sequence $(IP_1$, IS_1, $IS_2)$ to test a path.

If an edge between register R_i to R_i is marked with logic type 'L' and $d = 1$, then the paths from bit i to bit i of register R_i represents a self-loop. The paths between bit i to bit i of the register R_i can be functionally testable only when there is an odd inversion parity exists in the path, i.e, when odd number of gates which can invert the logic (e.g., NOT, NOR etc.) exists, otherwise, these paths are functionally untestable. These paths are classified as PCFUT. Many paths of such kind exist in the circuit, such as paths in the pass logic of ALU, paths in shifter, paths in logic operation block of ALU etc. Rest of the paths are classified as PFT paths.

Instruction pair (IP_1, IP_2) imposes constraints on those registers which have out-edge to R_o with distance d, logic type 'L' and some instructions common with the target edge. Instruction pair (IS_1, IS_2) imposes data constraints to those registers which have zero distance out-edge of logic type 'L' to register R_o with some instruction common with the target edge. Let register R_q has out-edge to R_o which is marked with distance d, logic type 'L', and has some common instruction with target edge, and a register R_p has zero distance out-edge to R_o which is marked with logic type 'L', and has some common instructions with target edge. Register R_q must have constant value across two time frames (under IP_1 and IP_2) if selected instruction IP_2 is not marked on any in-edge of R_q. Register R_p must have constant value across two time frames (under IS_1 and IS_2) if selected IS_2 is not marked on any the in-edge of R_p.

Example 5 Paths from the node S1 to ALO in VPRO processor:
Paths from S1 to ALO are marked with $[\langle I_{2-21}\rangle, \langle ex, ex\rangle, 0, L]$ as shown in Fig. 5.8. This implies that any instruction from I_2 to I_{21} can transfer data on the target path. We consider only normal data flow for the paths other than the target paths. Register S1 has only one zero distance edge marked with the tuple $[\langle I_{2-21}\rangle, \langle ex, ex\rangle, 0, L]$. Therefore, IS_1 can be any instruction from I_2 to I_{21}, and IS_2 can also be any instruction from I_2 to I_{21}. We can extract constraints in terms of instruction pairs and convert them into pairs of control signals (*alu control* and *comparator control*). Register S2 also has an out-edge with ALO

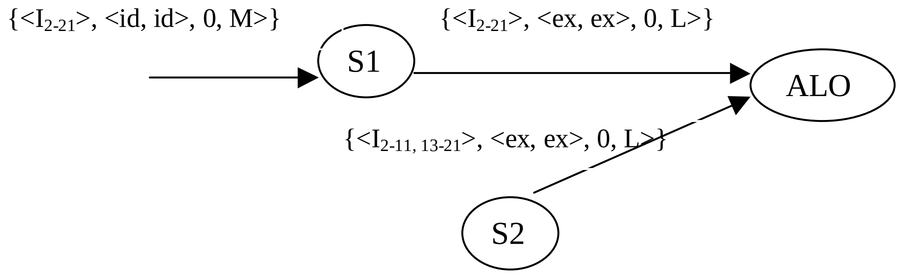

{<I2-21>, <id, id>, 0, M>}
S1
{<I2-21>, <ex, ex>, 0, L>}
ALO
{<I2-11, 13-21>, <ex, ex>, 0, L>}
S2

Figure 5.8. Edge consideration for the paths between S1 and ALO

marked with $[\langle I_{2-11,13-21}\rangle,\ \langle ex, ex\rangle,\ 0,\ L]$, which provides the data constraints. Since register S2 does not have any instruction marked at the in-edge which is not marked at the zero distance in-edge of S1, these paths do not observe data constraints. Since there are no constraints on data registers, we can find the values for control signals (ALU control signals and comparator signals) for the set of valid instructions for IS_2 and IS_1, i.e set of control signals for instructions I_1 to I_{21}. All the possible combination of control signals under IS_1 and IS_2 are the control constraints. Eight sets of control signals have been extracted under IS_1 and the same eight sets for IS_2. □

Test vector generation

Inversion parity test program, which checks the parity of the path, is used to further classify PCFUT paths into FUT paths or PFT paths. The above stated procedure can be use to simplify the circuit for ATPG. Constrained ATPG is used to generate test vectors for all the PFT paths by using extracted constraints. ATPG is given with PFT paths and their respective extracted constraints, and it returns the test vectors if paths are functionally testable or identifies these to be untestable.

5.5.2 Controller Testing

This section deals with the paths that contribute to control signals. Any path, which goes through control logic, is dealt in this section.In order to execute an instruction, the instruction is decoded by the decode unit (in decode stage). The decode unit dispatches control signals along with the required data to the pipeline stages ahead. Therefore, each pipeline stage has control signals that are not structured in nature but most of the time these can be grouped together in a small group.

Path Classification and Constraint Extraction

In our approach, small grouping of control signals is used to find constraints. We need to extract two types of constraints: i) constraints on the legitimate value of the group of control signals, and ii) constraints on inter group signals in a pipeline stage.

1. *Constraints on the legitimacy of signals:* Control signals generally form a group of small number of signals, where every possible value is not valid. Therefore we need to extract all the legitimate values. Test patterns must be generated under the legitimate values. For example, comparator control (*comp_ctrl*) signals in VPRO are grouped in a group of 3 bits, and legitimate values are ⟨0XX, 10X, and 110⟩.

2. *Constraints on inter group signals:* It is not sufficient to consider only the legitimate values for a group of signals but we need also to consider the legitimacy of the inter group signals in a pipeline stage, as all the possible combinations are not valid. We extract these in terms of instructions, i.e., map the control signals to the instruction which can generate the particular combination and all possible combinations are extracted. For example, in VPRO when ALU ctrl (*alu_ctrl*) signal is 0000 the comparator control (*comp_ctrl*) signal must be 000. Here onwards we will discuss how we can use these constraints for the test generation.

The part of a pipeline register, which carries the control signals is called control register. There may be paths between control register (CR) to control register, control register to data register (DR), data register (such as IR) to control register, or data register to data register through control logic. Paths between CR to CR are used to carry the control signals for the pipeline stages ahead. These paths usually connected directly and these can always be tested as interconnect test. Hence, these paths are classified as FT paths. Test vectors are generated under above stated constraints. Paths from data register to control register usually present in decode stage. These paths are classified as PFT paths, and test vectors for these paths can be generated under above stated constraints. Paths from CR to DR are the paths which pass through the combinational logic and these are significant in number. We construct a table which shows the transition on some bit in CR with instructions after exclusion of equivalent instructions.

Let there be a path between a bit i of control register C_k, and data register R_o. Constraints can be extracted in the following manner:

1. when control register C_k and data register are in the same stage:
   It needs an instruction sequence of two instructions (IS_1, IS_2). All those instruction pairs that can produce a transition at bit i and those are also marked on the in-edge of the register R_o can be the test instructions (IS_1, IS_2). All the data registers that have zero distance out-edge to R_o (have some common instruction with the selected potential instruction pairs) are needed to check for data constraints. Data constraints can be obtained in the same way as we obtain for datapath. These paths are classified as PFT paths.

2. when control register C_k and data register R_o are in different stages:
   Register R_o must have an edge from a register R_i that lies in the same stage in which C_k lies. This edge gives us the distance (say d), and we need a $(d + 2)$ instruction sequence (IP_1, IP_2, ID_1, . . . ,ID_{d-2}, IS_1, IS_2) to apply a test. All those instructions which can produce a transition at bit i of C_k and marked on any of the in–edge of those registers which have out-edge to R_o with same distance d, can act as IP_1, and IP_2. Constraints on those registers which have out-edge to R_o (with distance d) must be considered under IP_1, and IP_2. All those instructions which are marked on the in-edge of R_o (with distance $= d$) can act as IS_1 and IS_2, and data

constraints on those registers which have zero distance out-edge to R_o must be considered under the control constraints of IS_1, IS_2 instructions. These data constraints can be obtained in the same way as in datapath. These paths are classified as PFT paths.

Paths between DR to DR through control logic, usually carry control signals to multiplexers in the forwarding paths. Let there be a path between registers R_i and R_o.

1. when both registers R_i and R_o are in the same stage:
   The instructions marked on zero distance in-edges of regsiter R_o provide the control constraints (for IS_1 and IS_2). These paths do not observe data constraints and these are classified as PFT paths.
   Depending on the multiplexer control signals, an in-edge to R_o is selected which forwards data during this test. Let that edge be marked with distance d. Hence test can be applied by an instruction sequence (IP_1, IP_2, ID_1, . . . ,ID_{d-2}, IS_1, IS_2).

2. when both registers R_i and R_o are in different stages:
   Register R_o must have an edge from a register R_j that lies in the same stage in which R_i lies. The edge between R_j to R_o gives us the distance (say d). The test vectors can be applied through an instruction sequence (IP_1, IP_2, ID_1, . . . ,ID_{d-2}, IS_1, IS_2). All the instruction marked on the edge between R_j to R_o can act as IS_1 and IS_2. All the instructions marked on the in-edges of R_j can act as IP_1 and IP_2. These paths do not observe data constraints and these are classified as PFT paths.

Test Vector Generation

Constrained ATPG is used to generate test vectors for all the PFT paths under the extracted constraints. ATPG is given with PFT paths and their respective extracted constraints, and it returns the test vectors if paths are functionally testable or identifies these to be untestable.

5.5.3 Test Instruction Sequence Generation

The generated test vector pairs as explained above are assigned to control signals and registers. A sequence of instructions is needed to apply these test vectors. This process needs following three steps:

1. Test instruction sequence generation
2. Justification instruction sequence generation
3. Observation instruction sequence generation

Test instruction sequence generation step, generates a sequence of instructions which is responsible to launch the transition, propagate the launched transition, and latch the result provided that desired data are available in the appropriate registers. These data are made available by the justification instruction sequence. Finally, the result must be transferred to memory by a sequence of instructions, called observation sequence.

1. *Test instruction sequence generation:*
   We have generated test vectors for the paths between a register pair, under extracted constraints, and the information regarding the registers is available to us. We can use this information and PIE-graph to generate test instructions, which make this process simpler. It is clear from the earlier discussion that if an edge between registers R_i and R_o is marked $[\langle I_{set} \rangle$, $\langle S_j,\ S_i \rangle$, d, $LT]$, then we need an instruction sequence (IP_1, IP_2, ID_1, . . . , ID_{d-2}, IS_1, IS_2) to apply the test vectors provided that test vectors are available in desired registers. Instructions IP_1 (when $d > 0$) and IP_2 (when $d > 1$) are decided by the control signals of the stage S_j, and instructions IS_1 and IS_2 are decided by the control signals of S_i stage. If there are more than one potential candidates for these instructions then we must select easy to observe instruction (such as STORE) for IS_2, and easy to justify instruction for the rest. Once IP_1, IP_2, IS_1, and IS_2 instructions are decided, we fill the rest of the instructions by NOP instructions which can be later on replaced by the justification instructions for IS_1 and IS_2 that can reduce the number of instructions.

Table 5.2. Register mapping - VPRO processor

Register	I_1	I_2	I_3	I_4	I_{12}
S1	–	RF/RF	RF/RF	RF/RF	RF/mem
S2	–	RF/RF	RF/RF	RF/RF	RF/mem
ALO	–	RF/RF	RF/RF	RF/RF	RF/mem
MEO	–	–	–	–	RF/mem

2. *Justification instructions:*

   During the test instruction sequence generation, we assumed that desired data are available in appropriate registers. Now, we need to generate a sequence of instructions to justify the data in the registers.

   We cannot directly justify data in the pipeline registers. Therefore, we map back the data to either memory or architecture registers where we can justify easily. In order to do this, we construct a table that maps every pipeline data/address register to justifiable register or memory and to output register or memory when we execute some particular instruction. Such a table for VPRO is shown in part which contain some of the mapped input/output in the table 5.2.

   We use this table to find the register/memory where we need to justify data. Simple justification instructions are used. A special routine developed specifically for justification of a value in a special register, is used to justify the value in the special register.

3. *Observation instructions:*

   Result from the register R_o must be transferred to memory. The instruction IS_2 transfers data to some output register or memory, and that information we can get from the table stated above. If it is transferred to memory then we don't need any observation instruction, otherwise we need to transfer data from register file to memory using STORE instruction. Special routine is used for the data transfer from the special registers and the control registers

Example 6 Consider a test vector pair for a path between registers S1 and ALO of VPRO processor
V_1 = {S1 = 0024H, S2 = 0428H, alu_ctrl = 0100, comp_ctrl = 000}, and
V_2 = {S1 = 0004H, S2 = 0224H, alu_ctrl = 0101, comp_ctrl = 000}
The test instruction sequence will be

```
ADD  R5,  R1,  R2

ADD  R6,  R3,  R4
```

where [R1] = 0024H, [R2] = 0004H, [R3] = 0428H, and [R4] = 0224H

This edge is marked with distance = 0. Therefore, we need two test instructions IS_1 and IS_2 which can be obtained from the control signals of the execution stage. Control signals imply that the instruction IS_1 must be ADD (provide ALU ctrl = 0100, CMP ctrl = 000) and IS_2 must be SUB (provide ALU ctrl = 0101, CMP ctrl = 000). We map pipeline register S1 and S2 to memory or architectural registers using mapping table. Both these registers are mapped to register file. We must choose four different registers from register file to justify the value. Let R1 and R2 are chosen for S1, and R3 and R4 are chosen for S2. The content of R1, R2, R3, and R4 must be 0024H, 0004H, 0428H, and 0224H respectively. We must also map the output pipeline register to memory or architectural register using mapping table. ALO is mapped to register file. Therefore, we must once again choose some register from the register file, and let it be R5 (it can also be R1 or any other register). Test instructions will be ADD R5, R1, R3, and SUB R5, R2, R4. In order to observe the result we have to transfer data to memory. Observation instruction will be STORE instruction. In order to justify data in registers R1, R2, R3, and R4 we need LOAD instructions.

□

The above stated method can generate a test sequence to apply the generated test vectors under architectural constraints. However, in theory it is possible that a fault effect (error) that is captured in a register may be masked when the results are propagated for storage in the memory, or even the justification sequence may not be valid. But, we believe that the likelihood of this to happen is very little and in any case such fault masking can be identified by fault simulation. If fault is masked by the justification sequence then the fault effect from the justification sequence can be directly transferred to memory by an observation sequence without going through test instruction sequence, because fault is already excited by the justification instruction sequence. If fault is masked by the observation sequence then it can be eliminated by insertion of some dummy instruction(s) between test instruction sequence and the observation instruction sequence. Hence, fault masking can be eliminated.

5.6. Experimental Results

We have applied our methodology to two processors namely 16 bit 5 stage pipelined VPRO processor and 32 bit 5 stage pipelined DLX processor. VPRO processor has been synthesized using 2345 gates and 268 sequential elements, and pipelined DLX processor [29] is synthesized with 34,347 gates and 1898 sequential elements. Complete PIE- graphs for both of the processors are constructed by using instruction set architecture and RT level description. PIE- graph is used for the constraint extraction and the path classification. Our developed constrained ATPG for path delay faults is used, as commercially available ATPG are not capable of handling required constraints.

Results for VPRO and DLX processors for the Non Robust (NR) and Functional Sensitizable (FS) [31] tests are shown in the tables 5.3 and 5.4, respectively. The order of generation of test vectors is NR test followed by FS test. For each path, at first, ATPG generates a test vector pair for the NR test, if exists under the extracted architectural constraints. Otherwise, it generates test vector pair for the FS test if exists under architectural constraints. FS testable paths include the NR testable paths. Here we considered a path that goes through the control logic as a part of the controller. The results show that only a small fraction (about 24%) of paths are functionally testable. However, we achieve 100% fault efficiency in the test generation. These test vectors are generated under architectural constraints; hence these can be applied through instruction sequences. As pointed out in Section 5.5.3, these instruction sequences may lead to fault masking due to observation and justification sequences. Fault masking can be identified by fault simulation and eliminated as explained in Section 5.5.3.

Table 5.3. Results for VPRO processor

	Datapath		Controller	
	NR	FS	NR	FS
No. of paths	112,752	112,752	98,786	98,786
No. of faults	225,504	225,504	197,572	197,572
No. of functionally testable paths	32,134	52,092	27,512	42,282
No. of functionally untestable paths	193,370	173,412	170,060	155,290
Fault coverage (%)	14.2	23.1	13.9	21.4
Fault efficiency (%)	100	100	100	100

Table 5.4. Results for pipelined DLX processor

	Datapath		Controller	
	NR	FS	NR	FS
No. of paths	372,459	372,459	190,542	190,542
No. of faults	744,918	744,918	381,084	381,084
No. of functionally testable paths	148,718	185,247	57,502	89,974
No. of functionally untestable paths	596,200	559,671	323,582	2 91,110
Fault coverage (%)	19.9	24.8	15.0	23.6
Fault efficiency (%)	100	100	100	100

We now estimate the size of the test programs to test these processors. In order to apply a test vector pair, a sequence consisting of approximately 15 instructions is sufficient as follows. We need a sequence of about 8 instructions to load operands, about 4 test instructions - without filler instructions, one observation instruction, and about 2 instruction to load the memory locations. Therefore, a maximum of 2,858,130 instructions are needed to apply the generated 190,542 test vector pairs for DLX processor. Assuming each instruction to be 4 bytes, we need 10.9 MB storage space. Such a test program will take 37 milliseconds to run on a 100 MHz implementation of a DLX processor, assuming average 30% stalls during execution. Although these numbers can be reduced substantially by merging some vector pairs. Nonetheless, these figures (storage requirement and test time) show that this approach is suitable for self-testing of processors and it can also be applied for periodic on-line testing.

5.7. Conclusion

In this chapter we presented a systematic hierarchical approach for the delay fault testing of pipelined processor cores using their instruction set. To achieve this we developed a graph theoretical model using the RTL description of the processor to captures the complex pipeline behaviour. The graph model is used to extract architecture constraints for test generation. The extraction process can also identify some functionally untestable paths at this stage. The test generator uses the gate level description of the design and the extracted constraints to generate test vectors. In order to apply these generated test vectors in functional mode for at-speed test, a test instruction sequence generation procedure is developed. The graph model also assists the test instruction sequence generation process. Effectiveness of this approach is demonstrated through experimental results on two representative pipelined processors. The estimated test program size and test application time are suitable for on-line periodic testing. Hence, the proposed approach can also be used for on-line periodic testing which can further improve the reliability of the system in the field.

Chapter 6

Superscalar Processor Testing

6.1. Introduction

Instruction-based self-testing approach enables at-speed testing without any performance penalty. However, testing superscalar processors using this approach faces serious challenges, as these architectures discover the instruction-level parallelism on the fly, and use out-of-order execution, to achieve high throughput. This chapter identifies test challenges for the testing of superscalar architectures using instruction-based self-testing. A graph theoretic model is presented to model the superscalar behavior. Procedures for generating test programs which make sure that generated test vectors are applied in the correct order to test each testable path, are developed. Test results for a superscalar DLX (DLX-SV) processor are presented to demonstrate the effectiveness of the approach. To the best of our knowledge, this is the first work for superscalar processor testing using its instruction set.

This chapter first discusses superscalar architecture in Section 6.2, followed by test issues for this architecture in Section 6.3. Section 6.4 presents a graph model and testing methodology is discussed in section 6.5. Finally, it discusses the experimental results in Section 6.6.

6.2. Superscalar Architecture

Scalar pipelines are characterized by a single instruction pipeline of k stages. All instructions, regardless of type, traverse through the same set of pipeline stages. At the most one instruction can be resident in each pipeline stage at any one time and the instructions advance through lock step fashion. Fundamental limitations of scalar pipeline architecture are:

- The maximum throughput for a scalar architecture is bounded by one instruction per cycle
- The unification of all instruction types into one pipeline can yield an inefficient design
- The stalling of a lockstep or rigid scalar architecture induces unnecessary pipeline bubbles

Superscalar pipeline [33], [34] can be viewed as natural descendant of the scalar pipelines and involve extensions to alleviate the three limitations with scalar pipelines. Superscalar pipelines are parallel pipeline, instead of scalar pipelines, in that they are able to initiate the processing of multiple instructions in every machine cycle. In addition, superscalar pipelines are diversified pipelines in employing multiple and heterogeneous functional units in their execution stage(s). Finally, superscalar pipelines can be implemented as dynamic pipelines which discover instruction level parallelism, in order to achieve the best possible performance without requiring reordering of instructions by the compiler. A six stages superscalar pipeline is shown in the figure 6.1. These six stages are fetch, decode, dispatch, execute, complete and retire. The execute stage can include multiple (pipelined) functional units of different latencies. This necessitates the dispatch stage to distribute instruction of different types to their corresponding functional units. With out-of-order execution of instruction of instructions in the execute stage, the complete stage is needed to reorder the instructions and ensure the in-order updating machine state. Note also that there are multi-entry buffers separating these six stages. Complexity of buffers can vary depending on their functionality and location in the superscalar pipeline.

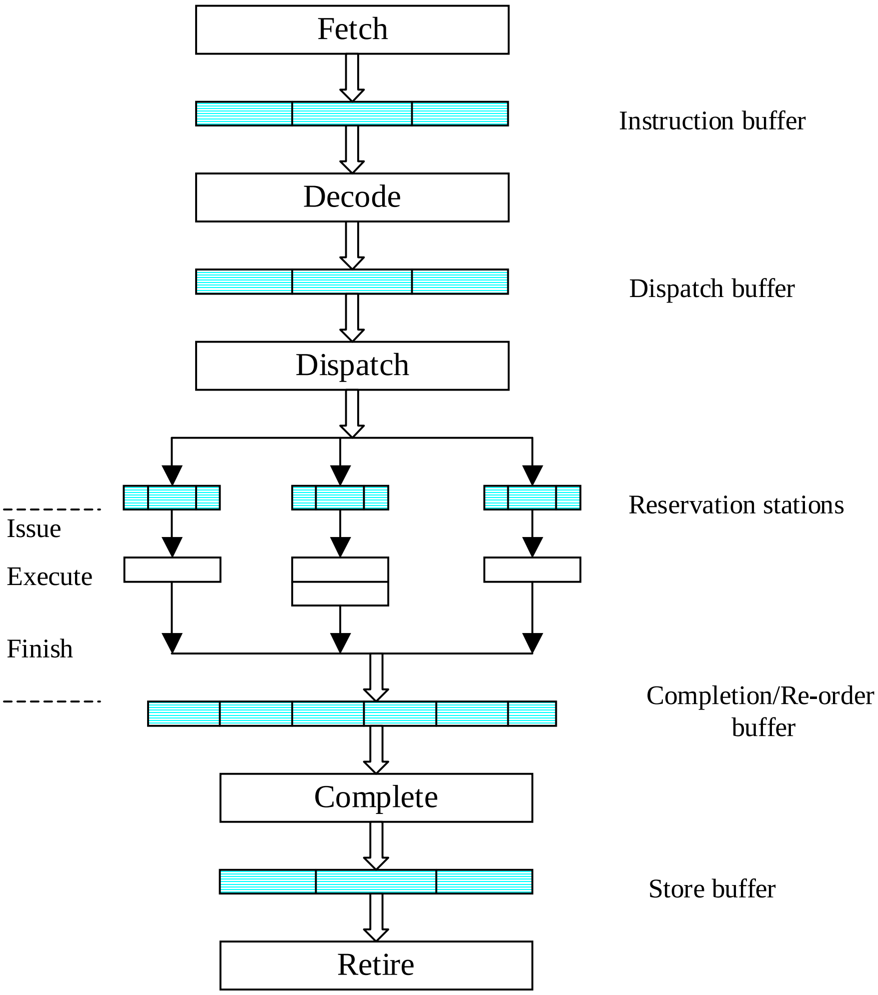


Figure 6.1. A 6-stage dynamic pipeline

Unlike a scalar pipeline, a superscalar pipeline, being a parallel pipeline, is capable to fetch multiple instructions from the I-cache in every machine cycle. Given a superscalar pipeline of width w, its fetch stage can fetch w instructions from I-cache, that means the physical organization of the I-cache must be wide enough that each row of the I-cache array can store w instructions and that an entire row can be accessed at a time. The primary objective of the fetch stage to maximize the instructions-fetching bandwidth. Instruction decoding involves the identification of the individual instructions, determination of instruction types, and detection inter-instruction dependences among the group of instructions that has been fetched but not dispatched yet. The complexity of the instruction-decoding task is strongly influenced by two factors, namely, the ISA, and width of the parallel pipeline.

Unlike, scalar pipelines, superscalar pipelines are diversified pipelines that employ multiple heterogeneous functional units in the execution unit. Different functional units can execute different types of instructions. Hence, decoded instructions must be routed to relevant functional units. This job is carried out by dispatch stage. Fetch and decode stages operate in centralized fashion, i.e., all instructions are managed by the same controller. Fetch unit fetches multiple instructions from same I-cache and deposited into the same buffer. In order to detect inter-instruction dependency, instructions fetched in a cycle must be decoded in centralized fashion. On the other hand, all the functional units in a diversified pipeline can operate independently in distributed fashion, once inter-instruction dependency is resolved. Consequently, going from decode stage to execution stage, there is a change from centralized processing of instructions to distributed processing of instructions. This change is carried out by, and is the reason for, the instruction dispatch stage in a superscalar pipeline. Another mechanism that is necessary between instruction decoding and instruction execution is temporary buffering of instructions. In order to execute an instruction, all the required operands must be ready. In a superscalar architecture it is possible that some of these operands are not yet ready because earlier instructions that update these operands have not finished their execution. A solution to this problem, without using stall, is to fetch those register operands which are ready and go ahead and advance these instructions into a separate buffer to wait those register operands that are not ready. When all register operands are ready, those instructions can then exit this buffer and be issued into the functional unit for execution. These instruction buffers are called *reservation stations*. The use of reservation station decouples instruction decode and instruction execution. Reservation stations can be implemented as *centralized reservation station* which is shared by all the functional units, or as *distributed reservation stations* for every functional units.

Instruction execution by heterogeneous functional units is a heart of superscalar architecture. These functional units perform more efficiently by specializing them for executing specific instruction types. In real superscalar processor architecture, the total number of functional units exceeds the actual width of the parallel pipeline. Because of specialization and heterogeneity of the functional units the total number of pipeline functional units must exceed the width of the superscalar pipeline to avoid having instruction execution portion become the bottleneck due to excessive structural dependences related to unavailability of certain functional unit types. Large number of functional units result into additional hardware complexity due to the need of forwarding results from the output of functional units to the input of reservation stations.

After being executed in out-of-order, an instruction must be written back in program order. This task is performed by completion buffer (or Re-Order Buffer) which is managed as a circular queue with instructions arranged according to program order. An instruction is considered *completed* when it finishes the execution and updates the machine states. An instruction finishes execution when it exits the functional unit and enters the completion buffer. Subsequently, it exits the completion buffer and becomes completed. In the complete stage, it writes back result to architectural register. With instruction that actually update memory locations, there can be a time period when they are architecturally completed and when the memory locations are updated. For example, store instruction can be architecturally completed when it exits the completion buffer and enters the store buffer to wait for the availability of the bus cycle in order to write to D-cache. This store instruction is considered retired when it exits the store buffer and update the D-cache.

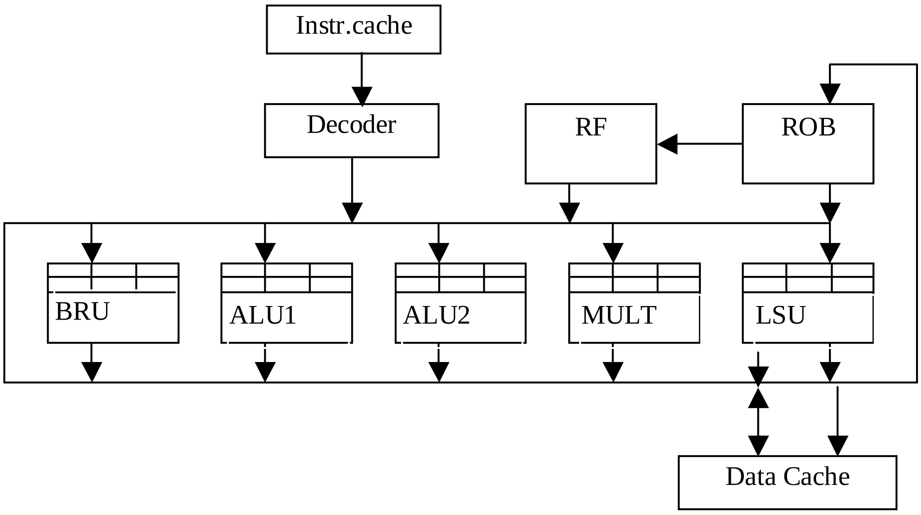


Figure 6.2. A Superscalar Organization of the DLX-SV

There are many possible superscalar organizations. Typically a superscalar organization consists of instruction fetch and branch prediction unit, decode and register renaming unit, instruction issue unit, execution unit, and commit unit. In this work we consider a most common organization of a superscalar processor which use distributed reservation station for each functional unit. The Re-order buffer (ROB) is used to commit the instructions. Figure 6.2 shows an organization of the DLX superscalar processor. For simplicity of presentation, we will use this particular organization to explain the various concepts. We believe, all the concepts can easily be generalized to other organizations.

Pipeline Vs Superscalar Processors

Scalar pipelines are characterized by a single instruction pipeline of k stages. All instructions, regardless of type, traverse through the same set of pipeline stages. At the most one instruction can be resident in each pipeline stage at any one time and the instructions advance through lock step fashion. Whereas, superscalar processors go beyond just a single-instruction pipeline by being able to simultaneously advance multiple instructions through the pipeline stages. They incorporate multiple functional units to achieve greater concurrency of processing multiple instructions for higher instruction execution throughput, often quantified as instructions per cycle (IPC). Another fundamental attribute of the superscalar processors is their ability to execute instructions in an order different from the order specified by the original program.

6.3. Test Issues and Overview of the Approach

This work is aimed at delay fault testing of superscalar processors. The objective is to generate tests and test sequences that can be applied in the functional mode of operation, using path delay fault model [31]. We believe that this [24] – [25] is the first work towards the modeling of the superscalar (dynamic pipeline) behaviour for the purposes of testing of a superscalar processor. This chapter describes some of the important issues that are pertinent to testing superscalar architectures.

We use an example superscalar DLX processor to demonstrate the concept. This processor uses a branch history table with 2 history bits to predict branch. It fetches four instructions and commits at most four instructions per cycle. Execution unit has 5 functional units (2 ALU, 1 Multiplier, 1 Branch unit, and 1 Load Store unit). Every unit has its own reservation station with 2 entries and ROB is implemented as a circular queue with 32 entries.

Superscalar Test Issues

Instruction based testing faces serious challenges due to the out of order execution with multiple functional units and in-order commit behavior, because it is the processor scheduler who decides the order of instruction execution, on the fly, and not the program that executes on the processor. This means that even if we have a test vector sequence generated under architectural constraints, when we apply such a sequence, there is no guarantee that the sequence will indeed be executed by the same functional unit for which it was meant to be. In fact, in a superscalar processor, the instructions in the sequence may be executed on a different functional unit and possibly in different order of instructions. Further, superscalar architecture uses buffers and queues, which makes it a challenging task to ensure that a given instruction resides at a given location in the buffer or queue with appropriate data at a given time. We explain this through the following example.

Example 7 Consider a 4 instruction wide fetch superscalar implemented with 2 ALU, 1 Multiplier, 1 Shifter, 1 Load, 1 Store and 1 Branch Unit, where every unit has individual reservation station with 2 entries, and ROB has 32 entries. Processor instructions are represented as (I Rd, Rs1, Rs2) where I specifies operation, Rd is the destination, and Rs1 and Rs2 are the two source operands. Let a path through ALU be tested by an instruction sequence ADD followed by SUB. This path is from the reservation station to the reorder buffer. Let the desired operands be placed in the registers R2 and R3 for the ADD instruction and in registers R6 and R7 for the SUB instruction. Conventionally, we apply the test vectors in the following sequence:

I_1: ADD R1, R2, R3 – processor schedules this instruction to ALU1

I_2: SUB R5, R6, R7 – processor schedules this instruction to ALU2

The processor may schedule instructions I1 and I2 to two different ALUs. Therefore, this sequence will not apply the desired test to any of the ALUs. We will get the correct result in spite of having a faulty path, because the fault is not excited. A possible partial solution to this problem is to concurrently test the two ALU's by the following program segment.

I_1: ADD R1, R2, R3 – processor schedules this instruction to ALU1

I_2: ADD R21, R2, R3 – processor schedules this instruction to ALU2

I_3: SUB R5, R6, R7 – processor schedules this instruction to ALU1

I_4: SUB R25, R6, R7 – processor schedules this instruction to ALU2

This can apply the test sequence to both the ALUs provided that these instructions are aligned, i.e., all these 4 instructions are fetched simultaneously. We can achieve this by having branch instruction preceding this set. Now, these instructions can be applied in our desired order. However, reservation station has two entries and first two instructions will be placed in the first entries of respective reservation stations and next two instructions will be placed in the second entries of the corresponding reservation stations. Therefore, the transition will not be launched and the path will remain untested. Again, a possible partial solution is to insert two instructions between I2 and I3 which are being scheduled to some other functional units. Therefore, the partial solution which can test the path from the first entry of reservation station to ROB is:

I_1:		J 2000H	
I_2:	2000H	ADD R1, R2, R3	– Processor schedules it for – ALU1 (stays at 1st position in RS)
I_3:		ADD R21, R2, R3	– Processor schedules it for – ALU2 (stays at 1st position in RS)
I_4:		MULT R10, R11, R12	– Processor schedules it – for Multiplier (Filler instr.)
I_5:		SW R1, 100 (R15)	– Processor schedules it for – Load store unit (Filler instruction)
I_6:		SUB R5, R6, R7	– Processor schedules it for – ALU1 (stays at 1st position in RS)
I_7:		SUB R25, R6, R7	– Processor schedules it for – ALU2 (stays at 1st position in RS)

□

This way, we can make sure that the desired transitions will be created and propagated. Still the consideration to make sure that a result will be transferred to some particular entry of ROB is not looked at in this example. This simple example demonstrates the need for carefully developing a test sequence. The situation becomes even more complex when we consider feedback paths (due to the presence of forwarding logic) in the out of order execution engine.

Overview of the Approach

In order to test the processor, we consider paths in datapath part and controller part separately. Clearly it is very difficult to separate out datapath and controller in superscalar processor as every stage carries data and control signals. We define data transfer activities between architectural registers, and data and address part of pipeline registers, buffers and queues, as a part of the datapath. All other paths are considered as a part of the controller.

A graph theoretic model called Superscalar Instruction Execution graph (SIE-graph) has been developed that is constructed by using RT level description and instruction set architecture. This graph model is an extension of our pipeline instruction execution graph [15] – [16]. This graph models the complex superscalar behaviour. The paths in datapath are classified as functionally testable, functionally untestable, and potentially functionally testable. The graph is used to extract the constraints. Combinational constrained ATPG is used to generate test vectors for potentially functionally testable paths. Vectors thus generated can be applied in functional mode using carefully crafted instruction sequences generated under architectural constraints. The test vectors so generated are mapped to control signals and registers. Processor instructions are used as vehicles to deliver test patterns and collect test responses. It was indicated earlier that a superscalar processor executes instructions out-of program order using multiple functional units and it is the processor scheduler that decides, on the fly, which instruction will be executed by which functional unit. Therefore, we need to carefully craft the test instruction sequence that can force scheduler to execute in our desired order as well as on a given functional unit. We have developed a methodology to generate an instruction sequence for every path based on the graph that forces scheduler to execute instructions in our desired order. We limited ourselves to Non Robust testing of the path delay faults.

6.4. Superscalar Instruction Execution Graph

The pipeline instruction execution graph (PIE-graph), described in chapter 5, is extended to capture the superscalar behaviour. SIE-graph is used for constraint extraction, path classification, and test instruction sequence generation.

SIE-graph can be constructed from RTL description and instruction set architecture. This includes the architectural registers, data and address part of the pipeline registers, buffers (Reservation Station), and queues (Re-Order Buffer).

Note that this does not include control part of the registers, buffers, and queues.

Nodes of SIE graph are:

1. Architectural Registers

2. Part of architectural registers if it is independent readable and writable

3. Equivalent registers (Set of registers that behave identically with the instruction set, such as register file, and stacks)

4. Two special nodes, IN and OUT, which models the external world such as memory or IO devices

5. Data and address part of the pipeline registers

6. Data and address part of buffers (like Reservation Station)

7. Data and address part of queues (like ROB

There are four types of nodes in SIE-graph, which are special type (IN and OUT), register type (R), buffer type (B), and queue type (Q). Every node is labeled with its type and its attribute. The number of entries in buffers or queues are the attributes. Every node, except special node, is labeled with its name, node type, and the attributes of the type if any. For example, a node representing ROB with 16 entries is labeled as (ROB, Q, 16).

A directed edge between two nodes is drawn iff there exists at least one instruction responsible to transfer data (with or without manipulation) between corresponding two registers. Each edge is marked with a $4-$ tuple [$\langle$ *instruction set*$\rangle$, $\langle$*stage from*, *stage to*$\rangle$, $\langle$*logic type*$\rangle$, $\langle$*cardinality*$\rangle$]. This $4-$ tuple signifies that instructions from the $\langle$ *instruction set*$\rangle$ are responsible for the transfer data from $\langle$*stage from*$\rangle$ to $\langle$*stage to*$\rangle$ through the logic specified by $\langle$*logic type*$\rangle$. Logic classification for logic type is based on our observation that many paths directly transfer data to the next stage using simple interconnects or through multiplexers. Keeping this in mind we classify logic in three types, namely interconnect (I), multiplexers (M), and processing logic (L). This classification simplifies the test generation process.

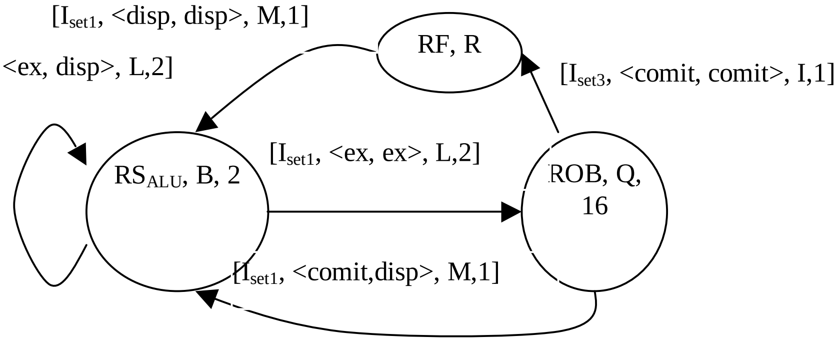

[Iset1, <disp, disp>, M,1]
[Iset1, <ex, disp>, L,2]
RF, R
[Iset3, <comit, comit>, I,1]
RSALU, B, 2
[Iset1, <ex, ex>, L,2]
ROB, Q, 16
[Iset1, <comit,disp>, M,1]

Figure 6.3. Part of SIE-graph of DLX-SV processor

Superscalar processors often use multiple identical functional units. Edges for these are merged and a cardinality of the edge is specified as ⟨*cardinality*⟩. SIE-graph for a part of the superscalar DLX processor is shown in the Fig. 6.3.

6.5. Testing Methodology

6.5.1 Datapath Testing

In this section we consider the paths that transfer the data between architectural registers, data and address part of the pipeline registers, buffers and queues. These paths are significant in number. Other paths are considered in the control section.

We assume that any instruction can follow any other instruction. Data forwarding takes place through the multiplexers. So, the data that can be received through forwarding path can also be received by the normal paths.

Path Classification and Constraint Extraction

A path can be, i) functionally testable (FT) path, ii) functionally untestable (FUT) path, or iii) potentially functionally testable (PFT) path. In order to test a path, we extract the architectural constraints by using SIE-graph. There are two types of constraints, i) control constraints and ii) data constraints. Control constraints are the constraints on the control signals, which are responsible to transfer data between two nodes. These are obtained from the instructions marked on the corresponding edge on SIE-graph. Data constraints are the constraints on the justifiable data under the extracted control constraints. Data constraints are not applicable to Non Robust (NR) testable paths.

1. *when logic type is interconnect 'I':*
   These paths always carry data for the preceeding stages. Therefore, these paths do not observe data constraints and can be tested as interconnects. These paths are classified as FT paths.

2. *when logic type is multiplexer 'M':*
   These paths pass through a set of multiplexers and behave as interconnects if control signals are properly assigned. Therefore these can also be tested as interconnects and classified as FT paths.

3. *when logic type is processing logic 'L':*
   These paths transfer data to destination node after manipulation.

   (a) Normal paths:
   These paths carry manipulated data inside the same pipeline stage. An instruction pair (I_{V1}, I_{V2}) is needed to test such a path. Any pair of instructions marked on the target path can be a test instruction, and can be used as I_{V1} and I_{V2}. The constraints on the control signals of the modules in the target path are extracted under this set of instructions. Non Robust test do not observe any data constraints. For Robust test, we must consider the data constraints on the nodes which have out–edge to the target destination node inside the same stage and have some common instruction with the target edge. These paths are classified as PFT paths.

   (b) Forwarding paths:
   These paths carry data to the other stages. Therefore, they need a sequence of three instructions (I_{V1}, I_{V2}, and I_{V3}) to test, where instruction I_{V1} and I_{V2} must be marked on the in–edge of the target source node, and I_{V3} must be marked on the target edge. Non robust

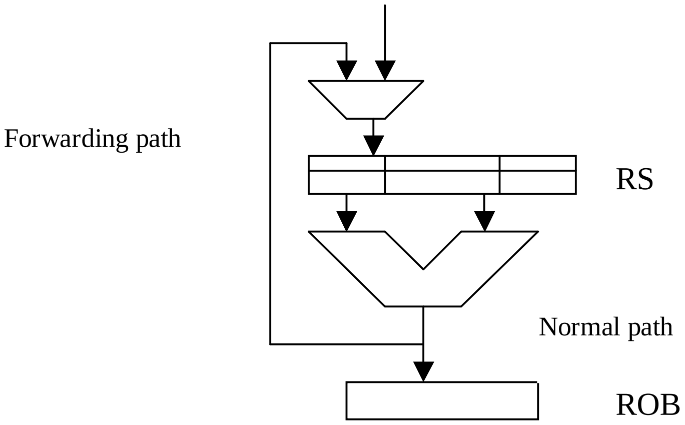


Figure 6.4. Forwarding and normal paths

test do not observe any data constraints. These are also classified as PFT paths.

The forwarding paths from the i^{th} entry of a buffer to the i^{th} entry of the same buffer are classified as FUT paths because a transition cannot be launched and propagated through this path.

As shown in the Fig. 6.4, the forwarding paths always go through MUX and the MUX can always be set to forward data. Therefore, the forwarding paths dominate the normal paths. Hence, it reduces the test generation effort.

Test Generation

After extraction of constraints, constrained ATPG is used to generate the tests for the potentially functionally testable paths. The ATPG returns the test vectors for the functionally testable paths.

Test Instruction Sequence Generation

The generated test vectors are mapped to the control signals and the registers. An instruction sequence is needed to apply the test vectors, justify the valus, and transfer the results to memory. We need to carefully craft an instruction sequence which can force scheduler to apply the test patterns in desired order. Test instruction generation procedures are explained through examples. In the examples we consider a processor that has 2 ALUs (1 ALU is considered for $p = 1$ case to demonstrate some concepts), with 2–entry reservation station and 32 entry ROB. The fetch width of the processor is 4 instructions. The other functional units in the processor are multiplier, load store unit, and branch unit. We assume that the ADD instruction followed by the SUB instruction is a test instruction pair.

Paths from node N_i to N_o (register type nodes): Fetch and the decode stages usually consist of these paths. These stages are in-order processing stages. Let us consider that a test instruction sequence (I_{V1}, I_{V2}) is needed to test a path. Let the superscalar width be w, and the cardinality of the edge be p. The instruction pair (I_{V1}, I_{V2}) can be applied by an instruction sequence [p number of I_1 instructions, $(w - p)$ other instructions except branching instructions, p number of I_2 instructions].

Paths from a buffer type node N_i to queue type node N_o: These paths originate from a reservation station and terminate at ROB. Let a reservation station has k entries and there be p number of identical functional units. The node representing the RS is labeled as (N_i, B, k). Let ROB be labeled with (N_o, Q, l). Derivation of test sequence for a path from i^{th} entry of RS to j^{th} entry of ROB is explained for two cases through examples.

1. when $p = 1$

Example 8 A path from 2^{nd} entry in RS to 6^{nd} entry in ROB can be tested by the following instruction sequence. We assume that the processor has one ALU.

I_1:		J 2000H	– Instruction for the alignment
I_2:	2000H	MULT R7, R8, R9	– Instr. for dependency

I_3:	AND R10, R7, R11	–Instr. to occupy 1st entry
I_4:	ADD R1, R2, R3	– Instruction I_{V1}
I_5:	SW R7, R13, R14	– Filler instruction
I_6:	SW R12, R15, R16	– Filler instruction
I_7:	SUB R4, R5, R6	– Instruction I_{V2}

The first jump instruction flushes the RS and the ROB (assuming this entry is seen first time). The next 4 entries will be fetched in next cycle. The AND instruction (I_3) will be placed at first entry of the RS and the ADD instruction (I_4) will be placed in the second entry of the RS. During second cycle instruction I_4 will be executed. Next four instructions will be fetched in the second cycle and instruction I_7 will be placed in the second entry of the RS. During the third cycle, instruction I_7 will be executed and will transfer the result to the 6th entry of the ROB. Hence, the path from the 2nd entry of RS to the 6th entry of ROB is tested.

□

Branch instruction, [(int(j/w)-1)*w] instructions at branch address which are not marked on the edge, instruction for dependency creation (should not be marked on the edge), $(i - 1)$ instructions marked from the instructions marked on the edge with dependency to the instructions which are not marked on the edge, I_{V1} instruction, $(w - i - 1)$ instructions which are not marked on the instruction, (rem (j-1/w)) instructions which are not marked the edge, I_{V2} instruction. In case of $1 < j < w$, $j = l + w$. We need a previously unseen branch instruction to align instructions and flush RS and ROB to make sure that desired data transfer takes place.

2. when $p > 1$ (Multiple identical functional units exist)
   In case of multiple identical units, our approach tests these units simultaneously. Here, we assume that $p \leq w$, i.e., the number of multiple units is less than or equal to the fetch width. Let a test sequence I_{V1}, I_{V2}) be required to test a path from the RS to the ROB.

Example 9 A path from 2nd entry in RS to 9nd and 10nd entries in ROB can be tested by the following instruction sequence.

I_1:		J 2000H	– Instruction for the alignment
I_2:	2000H	LW R8, 100(R10)	
I_3:		MULT R7, R8, R9	– For dependency creation
I_4:		AND R11,R7,R12	– Schedules to 1^{st} entry in RS of ALU1
I_5:		AND R13,R7,R14	– Schedules to 1^{st} entry in RS of ALU2
I_6:		ADD R1, R2, R3	– Instruction I_{V1}
			– Schedules to 2^{nd} entry in RS of ALU1
I_7:		ADD R21, R2, R3	–Instruction I_{V1}
			– Schedules to 2^{nd} entry in RS of ALU2
I_8:		SW R11, 100(R15)	– Filler instruction
I_9:		SW R13, 104(R14	– Filler instruction
I_{10}:		SUB R24, R5, R6	– Instruction I_{V2}
			– Schedules to 2^{nd} entry in RS of ALU1
			– Transfers the result to the 9^{th} entry of ROB
I_{11}:		SUB R4, R5, R6	– Instruction I_{V2}
			– Schedules to 2^{nd} entry in RS of ALU2
			– Transfers the result to the 10^{th} entry of ROB

The procedure to test a path from any entry in a buffer to any entry in queue can be generalized without much difficulty.

□

Paths from a buffer type node N_i to buffer type node N_o (Forwarding paths): These paths are responsible to forward data to the instructions residing in the RS without going through commit stage. These paths dominate the normal paths, i.e, a test for a forwarding path can also test the corresponding normal path. Hence, normal paths can be tested along with forwarding paths by using observation sequence for normal paths. An instruction sequence (I_{V1}, I_{V2}, I_{V3}) can test a path from RS (i^{th} entry) to the same RS (j^{th} entry) if it is applied in the following manner. This instruction sequence will test both normal paths and forwarding paths.

Example 10 A path from the 1^{st} entry in RS to the 2^{nd} entry in RS can be tested by the following instruction sequence.

I_1:		J 2000H	– Instruction for the alignment
I_2:	2000H	ADD R1, R2, R3	– Instruction I_{V1}
I_3:		ADD R21, R2, R3	
I_4:		SW R1, 100 (R9)	
I_5:		SW R21, 104 (R14)	– Filler instruction
I_6:		SUB R4, R5, R6	- Instruction I_{V2}
I_7:		SUB R24, R5, R6	- Instruction I_{V2}
I_8:		ORA R7, R4, R8	- Instruction I_{V3}
I_9:		ORA R7, R24, R8	- Instruction I_{V3}

□

Paths from a queue type node N_i to buffer type node N_o (From ROB to RS):

These are the paths which forward the data from the ROB to the RS. Following example explain the procedure to test a path from i^{th} entry in ROB to j^{th} entry in RS.

Example 11 A path from 3^{rd} entry of ROB to 2^{nd} entry in RS of multiplier unit can be tested by the following instruction sequence.

I_1:		J 2000H	– Instruction for the alignment
I_2:	2000H	LW R1, 100(R10)	
I_3:		AND R11,R7,R12	
I_4:		ADD R1, R2, R3	– Instruction I_{V1} schedule to
			– 3^{rd} entry in ROB
I_5:		J 2100H	
I_6:	2100H	LW R11, 100 (R15)	– Filler instruction
I_7:		AND R14, R12, R13	
I_8:		SUB R4, R5, R6	– Instruction I_{V2} schedule to
			– 3^{rd} entry in ROB
I_9:		SW R4, 104(R14)	– Filler instruction
I_{10}:		MULT R7, R8, R9	– Schedule to 1^{st} entry
			– in RS of multiplier
I_{11}		MULT R10, R4, R11	– Instruction I_{V3} schedule to
			– 2^{nd} entry in RS of Multiplier

The paths from ROB to register file can also be tested along with these paths by observing the result of I_{V2}.

□

6.5.2 Controller Testing

Instruction decoder dispatches the control signals with the data, which are used by the stages ahead. These control signals are often not structured. However they form a small group. We use this grouping to find the constraints. There are two types of constraints. i) intra group constraints, and ii) inter group constraints.

1. *Intra-group signal constraints:*
   Some combinations of value on a small group of signals are not valid combination. Therefore, we need to extract all the legitimate values. For example, test control (*test_ctrl*) signals in DLX-SV are grouped in a group of 3 bits, and legitimate values are ⟨0XX, 10X, and 110⟩.

2. *Inter group signal constraints:*
   We extract these constraints in terms of instructions, i.e., map to the instruction which can generate the particular combination and all possible combinations are extracted. For example, in DLX-SV when ALU ctrl (*alu_ctrl*) signal is 0000 the test control (*test_ctrl*) signal must be 000.

The part of a pipeline register, which carries the control signals is called control register. There are paths between control register (CR) to control register, control register to data register (DR), or data register (such as IR) to control register. The paths between CR-to-CR are used to carry the control signals for the pipeline stages ahead. These paths are connected directly and can be tested as interconnects. Paths from CR to DR are the paths which pass through the combinational logic such as paths from control register of RS to the ROB. Following example shows the procedure to test such paths.

Example 12 A path from 1^{st} entry of CR in RS to 6^{th} entry in ROB can be tested by the following instruction sequence.

I_1:		J 2000H	– Instruction for the alignment
I_2:	2000H	ADD R1, R2, R3	– Instruction I_{V1}
I_3:		MULT R10, R11, R12	
I_4:		SW R1, 100 (R15)	
I_5:		SW R10, 104 (R15)	– Filler instruction
I_6:		SUB R4, R5, R6	– Instruction I_{V2}
I_7:		SUB R4, R5, R6	– Instruction I_{V2}

□

Similarly, the test procedures for the paths from the control part of RS to the data part of RS (feed back paths) and other paths can also be developed as explained in the previous section.

6.6. Experimental Results

In order to demonstrate the effectiveness of our approach, we implemented a superscalar version of DLX processor (DLX-SV) with 5 functional units (2 ALU, 1 Multiplier, 1 Branch Unit, and 1 Load Store unit). Each functional unit has 2-entry reservation station. ROB is implemented as 16 entry circular queue. It can fetch two instructions and can commit at most 2 instructions.

Using RT-level description, SIE-graph is constructed. Based on the SIE-graph, paths are classified, and constraints for the potentially functionally testable paths have been extracted. Some of the paths are classified as functionally untestable at this stage. Test vectors are generated using constrained ATPG. Table 6.1 shows the results of the Non Robust tests [31] for the data path of DLX-SV processor. The results achieved so far indicate that high fault efficiency can be achieved to ensure the performance of processors. Although fault coverage is low, but that is normally true for path delay faults even if full scan techniques are employed. In order to apply these test patterns an instruction sequences can be generated using the procedures given in this chapter.

Table 6.1. Results for datapath of DLX-SV processor (NR test)

No. of paths [1]	2,559,284
No. of Faults	5,118,568
No. of functionally testable paths	1,576,122
No. of functionally untestable faults	3,542,446
Fault coverage (%)	30.7
Fault efficiency (%)	100

6.7. Conclusion

This chapter presented an at-speed testing methodology for testing of superscalar processors. It highlighted the test challenges pertinent to testing of superscalar architectures in the functional mode of operation. A graph theoretic model is developed to extract the constraints. We have developed a method of generating test programs that can force the processor scheduler to execute program in our desired order. Hence, these procedures can apply test vectors in the functional mode of operation. In order to show the effectiveness of the methodology, results for a superscalar DLX (DLX-SV) processor are presented.

[1] Except paths which go through multiplier unit

Chapter 7

Conclusion and Future Work

This chapter first summarizes the work completed in this thesis and then presents the future directions for this work. In particular two extensions of our existing work, namely application of these methods to design verification and study of simultaneously threaded processors, are perceived and described under the *future work* section.

7.1. Summary of the Thesis

In order to ensure the performance of a processor after manufacturing, it must be tested for the smaller timing defects and distributed faults caused by statistical process variations. This thesis presented an instruction-based self-testing methodology for modern processors in their chronological order of complexity. We believe that the instruction-based self-testing is one of the important and most suitable test methodology for testing of high performance processors as it applies test vectors through processor instructions in functional mode of operation, as illustrated in chapter 3. However, it needs an efficient method to extract the architectural constraints and test vector generation, so that generated tests can be coded (possibly automatically) in terms of valid processor instructions. This thesis presented processor models, which facilitate the test generation for processors under architectural constraints and the generated tests can always be coded into valid instructions.

In chapter 4, a systematic approach for the delay fault testing of non-pipelined processors has been presented. A graph theoretic model for data path has been developed. This graph model is constructed with the help of RT level description and instruction set architecture of the processor. This model, in conjunction with RT level description, is used to classify paths and eliminate the functionally untestable paths at the early stage of test generation process without looking into circuit details. Our method can efficiently extract the constraints for the remaining paths using graph model, as this model has information about the data transfer activities. A constrained ATPG is implemented to generate the test vectors under the extracted constraints so that these vectors can easily be applied by the instruction sequences. Controller is modeled as a finite state machine and constraints on state transitions are extracted. This eliminates the need for multiple time frame consideration for test generation, and hence reduces the test generation complexity. Experimental results show that our test generation process can efficiently generate test vectors for functionally testable paths which can be applied by test instructions. Results also show that a significant number of paths are identified as functionally untestable at higher level without using circuit details, which in turn reduces test generation effort significantly.

Chapter 5 presented a systematic approach for the delay fault testing of a pipelined processor using its instruction set. To the best of our knowledge, this is the first work that modeled the pipeline behaviour of a processor for the purpose of test generation. Once again, the pipeline behaviour is modeled by a graph theoretic model using RT level description and instruction set architecture. A hierarchical test generation procedure is presented. It classifies paths at higher level using graph model and efficiently extracts the constraints for potentially functionally testable paths and parity checkable functionally testable paths using RT level description and graph model. This model also assists the test instruction sequence generation process. Some paths can be declared as functionally untestable paths at the early stage. Effectiveness of this method is demonstrated through experiments on two representative pipelined processors.

Chapter 6 presented instruction-based delay fault self-testing methodology for superscalar processors. Testing superscalar processors using instruction-based self-testing approach faces serious challenges, as these architectures discover the instruction-level parallelism on the fly, and use out-of-order execution, to achieve high throughput. This chapter highlighted the test challenges pertinent to testing of superscalar architectures in the functional mode of operation. As for the pipelined processors, a graph theoretic model is developed to extract the constraints. We have developed a method of generating test sequences that can force the processor scheduler to execute program in our desired order because it is the processor scheduler which decides the order of execution on the fly. Hence, these procedures can apply test vectors in the functional mode of operation. In order to show the effectiveness of the methodology, results for a superscalar DLX (DLX-SV) processor are presented.

7.2. Future Work

This thesis presented systematic approaches for the testing of modern processors, which can easily be automated. However, the method of extraction of graph model of a processor from the RT level description and instruction set architecture is still manual, because of the different styles used in writing RT level descriptions. However, the complexity of graph is not high, therefore manual generation of graph model is manageable. But, in order to automate the entire process to develop a tool, the graph extraction process must also be automated.

We have developed algorithms to identify testable and untestable path delay faults in the datapath as well as in the control unit of the processors. Our algorithms use a constraint based test generator which guarantees to find all functionally testable paths and is highly efficient. However, we need to develop algorithms to automatically generate test sequences that can be used to apply the required vectors. Chapter 6 of this thesis identifies the complexity of this problem, through numerous examples, if the underlying processor is a superscalar processor. We believe that we have enumerated nearly all cases that can arise during testing of a superscalar processor and we can generate the test sequences for each case of interest but this process needs to be automated and efficient algorithms to achieve this are required. Methods to overlap sequences, to generate memory and performance efficient programs, without loss of coverage, still remain to be developed.

This thesis presented efficient instruction-based self-testing methodologies that can apply test vectors in functional modes through instructions. This work can be extended in two ways:

1. **For design verification:** Design verification is considered one of the serious bottlenecks for modern processors. There are two broad approaches to hardware design verification: Formal verification and simulation-based verification. Formal methods use mathematical proofs to verify the correctness. Simulation-based method tries to uncover design errors by detecting the behaviour of the faulty circuits when deterministic or pseudorandom test vectors are applied. The verification process for the modern complex designs is largely based on simulation methods using pseudorandom vectors, which take long time to uncover a design errors. Our method generates test vectors that can be applied in functional mode of operation by linking high level information with gate level information. Hence, it can also be extended for design verification by using error modeling techniques which can generate deterministic test vectors. Deterministic methodology gives better coverage. Some error modeling [43], [44] techniques have already been proposed in literature.

2. **For next generation processor architecture testing:** Simultaneous Multithreaded Processor (SMT) architecture is a natural descendant of superscalar architecture. It combines the hardware features of superscalar architecture with the multithreaded processors. From superscalar, it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware state of several programs (or threads). The result is a processor that can issue multiple instructions, from multiple threads each cycle, to achieve better performance for variety of workloads. Thus the SMT architecture by being inherently superscalar, poses all the challenges to instruction based testing that superscalar processor faces. Moreover, it is the processor fetch unit that decides, on the fly, which instructions from which thread are being fetched and issued in next cycle, hence it make it even more complex. A simple way to apply our approach for testing SMT processors is by operating it in single thread mode.

Once it starts to operate in single thread mode, it behaves like a superscalar processor. However, the coverage of data transfer paths regarding other threads in the fetch unit needs to be investigated. More over, an advantage of functional testing is that it can also be used for online testing of SMT processors, as they have capability to run multiple threads simultaneously. One thread can be dedicated to be a test thread to achieve online testing.

Introduction of one test thread that runs periodically can also help speed path fixing. When error is detected, the processor either increases its voltage or decreases its operating frequency, as most of the modern processors are able to run at multiple voltages and frequencies. The tests are run to see if it fixes the problems. This process can be repeated until the problem is fixed or some threshold is reached and system is shutdown. This technique could increase the reliability of the systems and also decrease the number of systems that would need repair.

Appendix

A. VPRO Processor

VPRO is a 16 bit, 5-stage pipelined RISC processor. It has 24 most common instructions. It uses load/store architecture. It consists of 8 general purpose 16 bit registers.

Instruction set

1. NOP	5. OR	9. SRL	13. SEQ	17. SGT	21. DEC
2. ADD	6. XOR	10. SRA	14. SNE	18. SGE	22. BEQ
3. SUB	7. MOV	11. LOAD	15. SLT	19. MVI	23. BNE
4. AND	8. SLL	12. STORE	16. SLE	20. INC	24. JUMP

Instruction Set Architecture

It consists of 3 type of instruction

1. Register- Register type instructions

OP	Rs1	Rs2	Rd	Fn2	Fn1
(2 – bit)	(3 – bit)	(3 – bit)	(3 – bit)	(3 – bit)	(2 – bit)

2. Immediate Instruction

OP	Rs1	Immediate	Unused	Fn1
(2 – bit)	(3 – bit)	(8 – bit)	(1 – bit)	(2 – bit)

3. Jump Instruction

OP	Immediate
(2 – bit)	(14 – bit)

Instruction Encoding

Instructions	Operation	Op	Fn1	Fn2	Type
NOP	No operation	00	00	000	R
ADD Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] + [Rs2]	00	00	001	R
SUB Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] - [Rs2]	00	00	011	R
AND Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] and [Rs2]	00	00	100	R
OR Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] or [Rs2]	00	00	101	R
XOR Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] xor [Rs2]	00	00	110	R
MOV Rd, Rs1	[Rd] ⇐ [Rs1]	00	00	111	R
SLL Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] sll [Rs2]	00	01	000	R
SRL Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] srl [Rs2]	00	01	001	R
SRA Rd, Rs1, Rs2	[Rd] ⇐ [Rs1] sra [Rs2]	00	01	010	R
LOAD Rd, Rs1	[Rd] ⇐ mem[Rs1]	00	10	000	R
STORE Rd, Rs1	Mem [Rs1] ⇐ [Rd]	00	10	001	R
SEQ Rd, Rs1, Rs2	[Rd]⇐ 1 if Rs1 = Rs2 else 0	00	11	001	R
SNE Rd, Rs1, Rs2	[Rd] ⇐ 1 if Rs1 ≠ Rs2 else 0	00	11	010	R
SLT Rd, Rs1, Rs2	[Rd]⇐ 1 if Rs1 < Rs2 else 0	00	11	011	R
SLE Rd, Rs1, Rs2	[Rd] ⇐ 1 if Rs1 ≤ Rs2 else 0	00	11	100	R
SGT Rd, Rs1, Rs2	[Rd] ⇐ 1 if Rs1 > Rs2 else 0	00	11	101	R
SGE Rd, Rs1, Rs2	[Rd] ⇐ 1 if Rs1 ≥ Rs2 else 1	00	11	110	R
MVI Rd, #imm8	[Rd] ⇐ [Imm8]	01	00	–	I
INC Rd, #imm8	[Rd] ⇐ [Rd] + [Imm8]	01	10	–	I
DEC Rd, #imm8	[Rd] ⇐ [Rd] - [Imm8]	01	11	–	I
BEQ Rs1, #imm8	PC ⇐ PC+ [Imm8] if Rs1 = 0	10	00	–	I
BNE Rs1, #imm8	PC ⇐ PC+ [Imm8] if Rs1 ≠ 0	10	00	–	I
JUMP #imm14	PC ⇐ PC+ [Imm14]	11	–	–	J

Structural Organization

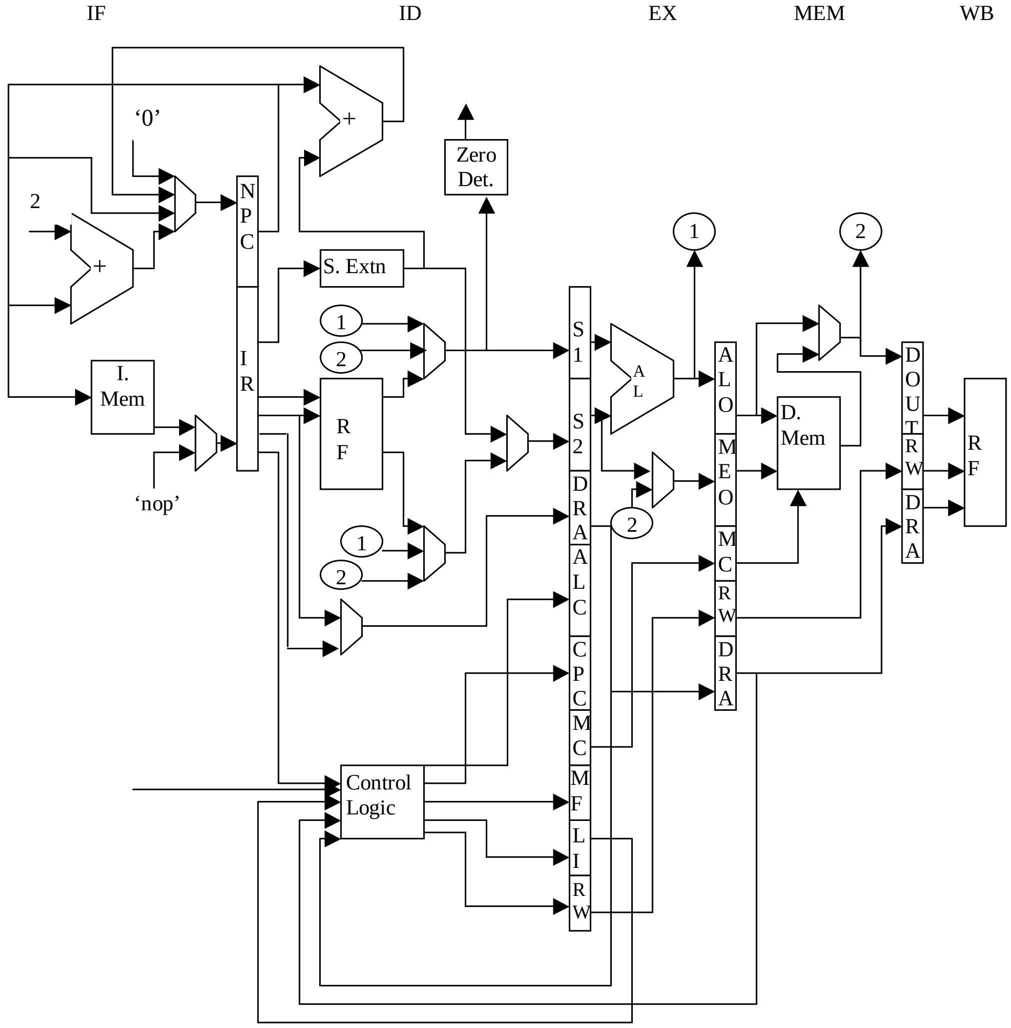


Figure A.1. Structural Organization of VPRO Processor

Acknowledgements

I would like to thank my supervisor, Professor Hideo Fujiwara, for his guidance and support during my graduate studies, and for the invaluable training I received as his student. His influence is sure to stay with me for long time and help me in my career in many ways. I feel honored as his student, and I cherished the freedom he offered me to explore and develop new ideas. This freedom kept my interest alive in my research throughout my course of study at Nara Institute of Science and Technology (NAIST).

I would also like to thank the members of the thesis committee, Professor Minoru Ito and Professor Kenichi Matsumoto for their valuable comments and important suggestions concerning this thesis.

I am highly thankful to Associate Professor Michiko Inoue, whose continuous guidance, valuable suggestions and encouragement, played an important role in the completion of this thesis. Despite her bad health towards the end of this thesis, her willingness to help always encouraged me.

I am also thankful to Assistant Professors Satoshi Ohtake and Tomokazu Yoneda for their friendly discussion and continuous cooperation.

I would like to express my heartiest gratitude to Professor Kewal K. Saluja, University of Wisconsin-Madison, USA, for his constant guidance and encouragement throughout the course of this research. He gave extremely insightful advice aimed at making me target higher and higher in doing research. He influenced almost every corner of my life. I feel honored and privileged to have worked with him for this thesis.

I would also like to extend my gratitude to Ms. Yoshiko Hirayama and Ms. Saeko Ono for helping me in different official correspondence.

I am grateful to Prof. Samiha Mourad, Santa Clara University, and Prof.

Jacob Savir, New Jersey Institute of Technology, for their friendly discussions.

I am indebted to Dr. Erik Larson, Dr. Md. Altaf-Ul-Amin, Dr. Shintaro Nagai, Dr. Tsuyoshi Iwagaki, Dr. Kazuko Kambe, Dr. Thomas Clouqueur,Mr. Yusuke saga, Mr. Zhiqiang You, Ms. Chia Yee Ooi, and Mr. Yoshiyuki Nakamura for their support and cooperation which made my stay very pleasant and comfortable. Thanks are also due to all other present and past members of the Fujiwara Laboratory for making my stay, within the confines of laboratory, pleasurable.

I am deeply indebted to the members of STARC (Semiconductor Technology Academic Research Centre), Mr. Tokonori Kozawa (STARC), Mr. Shunsuke Miyamoto (STARC), Dr. Kazumi Hatayama (Renesas), Mr. Mamoru Mukuno (Sanyo), Mr. Ryoji Sakurai (Sharp), and Mr. Yashuyuki Nozuyama (Toshiba) for their valuable comments on each and every part of this work.

This work was supported in part by the STARC, JSPS (Japan Society for Promotion of Science) under Grant-in-Aid for Scientific Research and Foundation of NAIST under the grant for activity of education and research. I express my gratitude to all these sponsoring organizations.

I am grateful to Ministry of Education, Culture, Sports, Science and Technology (MEXT), Government of Japan for supporting me with Japanese government scholarship through out my course of study. I am also grateful to Ministry of Human Resource Development, Government of India, for providing me a chance to pursue doctoral degree at NAIST, Japan.

My institute, Central Electronics Engineering Research Institute (CEERI), Pilani, India, also deserve thanks for providing me study leave to pursue higher study at NAIST, Japan. I am thankful to Dr. Shamim Ahmad, Dr. Chandrashekhar, Mr. Raj Singh, and Dr. R.S. Shekhawat, for their suggestions and cooperation.

I would like to express my gratitude to Prof. N.S. Verma for his continuous guidance during undergraduate study and motivating me to pursue doctoral course in Japan. I am also thankful to Prof. M.S. Gaur and Prof. Rakesh Bairathi for their encouragement.

My parents, Mr. Rameshwar Singh and Mrs. Phool Kaur, also deserve thanks not only for the love and support they have given me – over phone, through e-mail – for last three years, but also every thing they have done for me during last

thirty two years. I am also grateful to my sister Saroj and brother Rajesh for their cooperation and support I received through out my life. Thanks are also due to other members of my family.

I would also like to express my sincere gratitude to my friend Sushil Kabra for his help and support, I received as and when I needed. Last but not least, I would like to thank Mr. Prem Prakash Kothari, my school teacher, whose teachings always motivated me to take up challenging work.

References

[1] S. M. Thatte and J. A. Abraham, “Test generation for microprocessors,” *IEEE Trans. on Computers*, Vol. C–29, No.6, pp. 429–441, Jun. 1980.

[2] K. K. Saluja, L. Shen, and S. Y. H. Su, “A simplified algorithm for testing microprocessors,” *Proc. of the International Test Conference*, pp. 668–675, 1983.

[3] D. Brahme and J. A. Abraham, “Functional testing of microprocessors,” *IEEE Trans. on Computers*, vol. 33, No. 6, pp. 475–484, Jun. 1984.

[4] J. Shen and J. A. Abraham, “Native mode functional test generation for processors with applications to self test and design validation,” *Proc. of the International Test Conference*, pp. 990–999, 1998.

[5] K. Batcher and C. Papachristou, “Instruction randomization self test for processor cores,” *Proc. of the VLSI Test Symposium* , pp. 34–40, 1999.

[6] L. Chen, and S. Dey, “Software-based self-testing methodology for processor cores,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 20, No. 3, pp. 369–380, Mar. 2001.

[7] A. Paschalis, D. Gizopoulos, N. Krantis, M. Psarakis, and Y. Zorian, “Deterministic software-based self-testing of embedded processor cores,” *Proc. of the Design Automation and Test in Europe*, pp. 92–96, 2001.

[8] N. Krantis, D. Gizopoulos, A. Paschalis, and Y. Zorian, “Instruction-based self-testing of processor cores,” *Proc. of the VLSI Test Symposium* , pp. 223–228, 2002.

[9] N. Krantis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-based self-testing of processor cores," *Journal of Electronic Testing: Theory and Application (JETTA)*, Vol. 19, pp. 103–112, 2003.

[10] K. Kambe, M. Inoue, and H. Fujiwara, "Efficient template generation for instruction-based self-test of processor cores," *Proc. of the IEEE Asian Test Symposium*, pp. 152–157, 2004.

[11] W. -C. Lai, A. Krstic, and K. -T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," *Proc. of the VLSI Test Symposium* , pp. 15–20, 2000.

[12] W. -C. Lai, A. Krstic, and K. -T. Cheng, "Test program synthesis for path delay faults in microprocessor cores," *Proc. of the International Test Conference* , pp. 1080–1089, 2000.

[13] W. -C. Lai, A. Krstic, and K. -T. Cheng, "Functionally testable path delay faults on a microprocessor," *IEEE Design & Test of Computers*, Vol. 17, No. 4, pp. 6–14, Oct-Dec 2000.

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

[15] 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 & Test of Computers*, Vol. 19, No. 4, pp. 18–27, Jul-Aug 2002.

[16] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Software-based delay fault testing of processor cores," *Proc. of the IEEE Asian Test Symposium*, pp. 68–71, 2003.

[17] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based delay fault testing of processor cores," *Proc. of the International Conference on VLSI Design* , pp. 933–938, 2004.

[18] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Delay fault testing of processor cores in functional mode," *IEICE Trans. on Information & Systems*, Vol. E-88D, No. 3, pp. 610–618, Mar. 2005.

[19] L. Chen, S. Ravi, A. Raghunath, and S. Dey, "A scalable software-based self-test methodology for programmable processors," *Proc. of the Design Automation Conference* , pp. 548–553, 2003.

[20] N. Krantis, G. Xenoulis, A. Paschalis, D. Gizopolous, Y. Zorian, "Application and analysis of RT-level software-based self-testing for embedded processor cores," *Proc. of the International Test Conference*, pp. 431–440, 2003.

[21] A. Paschalis, D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *Proc. of the Design and Test in Europe*, pp. 578-583, 2004.

[22] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based delay fault self-testing of pipelined processor cores," *Proc. of the IEEE International Symposium on Circuits and Systems*, pp. 5686–5689, 2005.

[23] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Trans. on VLSI Systems*, Vol. , No. , pp. xx–xx, 20xx. (Submitted)

[24] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Program-based testing of superscalar microprocessors," *Proc. of the IEEE 14th North Atlantic Test Workshop*, pp. 79–86, 2005.

[25] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Testing superscalar processors in functional mode," *Proc. of the 15th International Conference on Field Programmable Logic and Applications*, 2005. (To appear)

[26] A. Krstic, S. T. Chakradhar, and K. -T. Cheng, "Testable path delay fault cover for sequential circuts," *Journal of Information Science and Engineering*, Vol. 16, pp. 673-686, 2000.

[27] K. -T. Cheng, and H. -C. Chen, "Classification and identification of non-robust untestable path delay faults," *IEEE Trans. on CAD of Integrated Circuits and Systems*,Vol. 15, No. 8, pp. 845-853, 1996.

[28] Z. Navabi, *VHDL: Analysis and modeling of digital systems*, McGraw-Hill, NY, 1997.

[29] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 1996.

[30] M. Gumm, "VLSI Design Course: VHDL-Modeling and Synthesis of DLXS RISC Processor," University of Stuttgart, Germany, Dec. 1995.

[31] A. Krstic and K. -T. Cheng, *Delay fault testing for VLSI circuits*, Kluwer Academic Publishers, 1998.

[32] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded processor-based self-test*, Kluwer Academic Publishers, 2004.

[33] J. E. Smith, and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. of the IEEE*, Vol. 83, No. 12, pp. 1609–1624, Dec 1995.

[34] J. P. Shen, and M. H. Lipasti, *Modern processor design*,Mc. Graw Hill Publication, 2004.

[35] W. Needham, "Microprocessor testing today," *IEEE Design & Test of Computers*, Vol. 15, No. 3, pp. 56–57, Jul 1998.

[36] R. Tupuri, A. Krishnamachari, and J. A. Abraham, "Test generation for gigahertz processors using automatic functional constraint extractor," *Proc. of the Design Automation Conference*, pp. 647–652, 1999.

[37] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory & mixed-signal VLSI circuits*, Kluwer Academic Publishers, 2000.

[38] S. Hellebrand, and H. -J. Wunderlich, "Mixed-mode BIST using embedded processors," *Proc. of the International Test Conference*, pp. 195–204, 1996.

[39] F. Brglez, C. Gloster, and G. Kedem, "Built-in self-test with weighted random-patterm hardware," *Proc. of the IEEE International Conference on Computer Design*, pp. 161-167, 1990.

[40] M. T. Lee, *High-level test synthesis of digital VLSI circuits*, Artech House Publisher, 1997.

[41] International Technology Roadmap for Semiconductors (ITRS). On-line reference. http://public.itrs.net/

[42] A. K. Sharma, *Semiconductor memories: Technology, Testing , and Reliability*, IEEE Press, 1997.

[43] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mugde, and R. B. Brown, "High-level design verification of microprocessors via error modeling," *ACM Trans. on Design Automation of Electronics Systems*, Vol. 3, No. 4, pp. 581–599, Oct 1998.

[44] M. N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs," *Proc. of the International Test Conference*, pp. 138–147, 2003.