

國立臺灣大學電機資訊學院電機工程學系



碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

應用於軟體自我測試之電路老化缺陷偵測

Software-Based Self-Test for Aging Defect Detection

林子翔

Tzu-Hsiang Lin

指導教授：黃俊郎 博士

Advisor: Jiun-Lang Huang, Ph.D.

中華民國 107 年 7 月

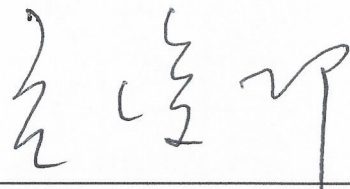
July, 2018

國立臺灣大學碩士學位論文
口試委員會審定書

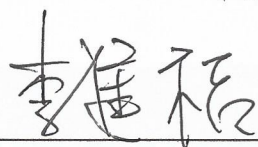
應用於軟體自我測試之電路老化缺陷偵測
Software-Based Self-Test for Aging Defect Detection

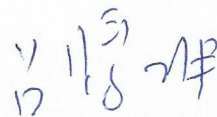
本論文係林子翔君 (R05943090) 在國立臺灣大學電子工程學
研究所完成之碩士學位論文，於民國 107 年 07 月 26 日承下列考
試委員審查通過及口試及格，特此證明

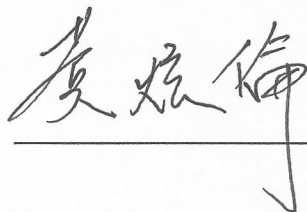
口試委員：



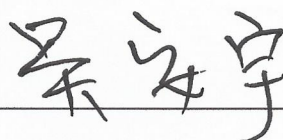
(指導教授)







系主任、所長





Thanks to my parents, friends, and those we helped me...

中文摘要



由於傳統結構性測試的不足，應用軟體自我測試(software-based self-test)成為了非侵入性、功能性以及全速測試的替代方案。應用軟體自我測試的技術可彌補傳統結構性測試的不足，並且能在客戶使用階段提升硬體可靠性(reliability)。在論文中，我們建立了一套完整的應用軟體測試流程，其中包含非功能性測試限制提取、測試圖騰指令轉換器、測試程式產生器、電路錯誤模擬器。此外，為了確保所產生測試程式之測試品質，我們亦提供了隨機程式評估比較結果於文末。

我們所提出的應用於軟體測試流程目的為在程式或應用執行的過程中，偵測出可能發生的電路老化缺陷(aging defect)及錯誤。所使用的錯誤模型為電路老化效應(aging effect)所造成的硬體缺陷，我們將模擬因電路老化效應所造成的路徑延遲錯誤(path delay fault)及轉態延遲錯誤(transition delay fault)，以偵測電路老化效應的初期現象。

關鍵字：應用軟體自我測試、電路老化效應、系統可靠性、積體電路系統測試

ABSTRACT



Since the insufficiency of conventional structural test, software-based self-test becomes the alternative solution for a non-intrusive, functional and at-speed testing. The use of software-based self-test could compensate the shortages of conventional structural test and enhance the hardware in-field reliability. In the thesis, we have provided a complete test flow for software-based self-test including constraint extraction, pattern-to-instruction converter, test program generator and fault simulator. Besides, in order to confirm the quality of test program generated by our methodology, the results of random program evaluation have been displayed in the last part of this thesis.

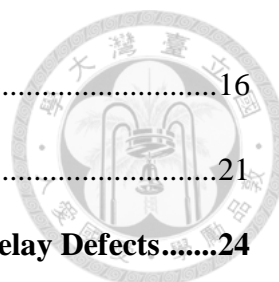
The proposed software-based self-test methodology aims to detect the possible hardware faults during the execution of test programs or applications. The target fault model is the hardware fault caused by aging effect. We model the fault behavior as the path delay fault and transition delay fault models for aging fault simulation.

Keywords: Software-Based Self-Test, Aging Effect, Reliability, VLSI System Testing

CONTENTS



口試委員會審定書	ii
誌謝	i
中文摘要	ii
ABSTRACT	iii
CONTENTS	iv
LIST OF FIGURES.....	vi
LIST OF TABLES.....	viii
Chapter 1 Introduction	1
1.1 Motivation	2
1.1.1 Challenges of manufacturing testing	2
1.1.2 Software-based self-test.....	3
1.1.3 Challenge of software-based self-test.....	5
1.2 Review of Previous Techniques	6
1.2.1 Test program generation approaches	6
1.2.2 Fault injection approaches	7
1.3 Contribution.....	9
1.4 Organizations of the Thesis	10
Chapter 2 Aging Effect and Delay Fault Testing	11
2.1 Aging Effect	12
2.2 Delay Fault Testing.....	12
2.3 Software-Based Delay Fault Testing.....	15



2.3.1	Path activation monitoring	16
2.3.2	Fault injection and detection	21
Chapter 3 Proposed Method for Software-Based Self-Test on Delay Defects.....		24
3.1	Proposed Methodology.....	25
3.2	Pre-Processing	25
3.3	Test generation	28
3.3.1	Static timing analysis.....	29
3.3.2	Automatic test pattern generation (ATPG).....	30
3.3.3	Pattern-to-instruction converter.....	32
3.4	Fault Simulation	37
Chapter 4 Experiment Result.....		39
4.1	Experiment Setup	40
4.2	Result Statistics	43
4.2.1	Transition delay fault testing	43
4.2.2	Path delay fault testing	44
4.2.3	Random program evaluation	46
Chapter 5 Conclusion.....		54
REFERENCE		56

LIST OF FIGURES



Figure 2-1	Transition delay fault model	14
Figure 2-2	Pseudo code of path activation monitoring testbench	16
Figure 2-3	Example of non-robust test conditions for AND gate	17
Figure 2-4	Example of robust test conditions for AND gate	17
Figure 2-5	Problem of non-robust test	18
Figure 2-6	Example of a non-robust path	20
Figure 2-7	Example of a robust path	20
Figure 2-8	Example of a robust* path	20
Figure 2-9	Testable path coverage	21
Figure 2-10	Fault behaviors in gate-level and RT-level simulation	21
Figure 2-11	Pseudo code of fault injection testbench	22
Figure 3-1	Arithmetic logic unit (ALU)	26
Figure 3-2	Flowchart of test generation	28
Figure 3-3	Critical path example reported by Synopsys Primetime	31
Figure 3-4	A test pattern generated by Synopsys TetraMAX	34
Figure 3-5	Classification of pattern information	34
Figure 3-6	Example of test program generated by our methodology	34
Figure 3-7	Example of two vectors from the test program	36
Figure 3-8	Instructions distribution of the test program	37
Figure 3-9	Flowchart of fault simulation	38



Figure 4-1	MIPS32 processor architecture	40
Figure 4-2	Single-issue in-order 5-stage pipeline	41
Figure 4-3	Four-way handshake mechanism	42
Figure 4-4	Equations of coverage calculation	44
Figure 4-5	Venn diagram of fault detection	46
Figure 4-6	Random program evaluation in robust 1-vector mode	48
Figure 4-7	Random program evaluation in robust 2-vector mode	48
Figure 4-8	Random program evaluation in robust 3-vector mode	49
Figure 4-9	Mean fault coverage of different vectors in robust mode	49
Figure 4-10	Random program evaluation in non-robust 1-vector mode	50
Figure 4-11	Random program evaluation in non-robust 2-vector mode	50
Figure 4-12	Random program evaluation in non-robust 3-vector mode	51
Figure 4-13	Mean fault coverage of different vectors in non-robust mode	51
Figure 4-14	Comparison between different successive vectors	52

LIST OF TABLES



Table 3-1	Mapping of ALU operation signal and instruction	27
Table 3-2	Constraints of ALU input	27
Table 3-3	Mapping table for patterns to instructions convertor	35
Table 4-1	Transition delay fault testing by TetraMAX	44
Table 4-2	Transition delay fault testing by software-based self-test	45
Table 4-3	Path delay fault testing by software-based self-test	47
Table 4-4	Random program format	48



Chapter 1

Introduction



1.1 Motivation

1.1.1 Challenges of manufacturing testing

In modern VLSI systems, the level of integration keeps increasing due to the large advancement in IC fabrication technology. With the elevating operation frequencies and shrinking feature sizes, the conventional structural testing is insufficient to achieve the desired test quality. Furthermore, test escapes increase as the number of un-modeled faults grows. Therefore, the conventional fault models such as stuck-at fault model and bridging fault model are insufficient for maintaining the desired quality product [3].

Scan testing is the most commonly used design-for-testability (DFT) technique to address the fault coverage and test cost concerns. The problem is lacking self-test ability in the field. Hardware-based structural self-test techniques, such as logic built-in self-test (BIST), provide the feasible solution. Built-in self-test eliminates the need of high-speed testers and can more accurately apply and analyze at-speed test signals on chips. However, BIST still have some disadvantages, such as nontrivial area, performance, and design time overhead. Moreover, structural BIST's non-functional, high switching random patterns consume much more power than normal system operation. Finally, to apply at-speed tests to detect timing-related faults, existing structural BIST must resolve various complex timing issues related to multiple clock domains, multiple frequencies, and test clock skews that are unique in test mode.

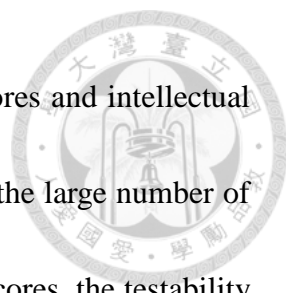
In order to carry out the at-speed, non-intrusive and functional testing, the new testing method called software-based self-test (SBST) came out.



1.1.2 Software-based self-test

The use of software-based self-test (SBST) might improve the reliability of electronic systems and overcome the shortcomings of the non-functional structural testing [4]. Safety-critical applications are usually equipped with a processor or a controller, requiring detecting possible faults in normal at-speed operational phase. The SBST technique consists in having the processor core execute the test program, activate the possible faults in the processor data paths by the instruction sequences, and eventually comparing the actual results of the computation against the expected ones, usually stored as a signature; any mismatch identifies a malfunctioning [5].

SBST can be adopted in end-of-manufacturing test and in-field test. With the external testers, the outputs of device are fully observed during the manufacturing test so that the fault controllability and observability can be increased. However, the increasing gap of the operation frequencies between external testers and the high-performance processors will increase test costs and lead to the escaped faults which might be detected only in the at-speed testing [6]. Thus, the importance of the at-speed testing may not be overemphasized.



Most of the SoC designs are built with embedded processor cores and intellectual property (IP) blocks which provide complex functionalities. Due to the large number of arithmetic and logic functional modules embedded in the processor cores, the testability of the processors becomes a crucial issue [7]. The in-field SBST can be based on the instruction set of the processor, any extra hardware such as DFT structures are unnecessary. It avoids the issues of area and performance overhead in the design.

The in-field SBST might provide at-speed testing with the limitation that data memory is observable; this may cause some faults to become functionally untestable [8] and reduce the fault coverage significantly. Generating the effective test program and avoiding over-testing the functionally untestable data paths in the processor cores might help retain the reliability and the fault coverage.

To sum up, SBST has the following advantages. First, it minimizes the addition of dedicated test circuitry. Second, it can also apply and analyze at-speed test signals on chip more accurately than external testers. Third, compared with the hardware-based self-test in nonfunctional BIST mode, SBST is executed in the design's normal operational mode. This can eliminate the excessive power consumption of structural BIST and avoid over-testing caused by the application of non-functional patterns during structural testing.



1.1.3 Challenge of software-based self-test

Although SBST conquers the disadvantages of BIST and facilitates at-speed, nonintrusive, functional testing, it still has some difficulties. The method for generating high quality test programs is the hardest part in SBST. Nowadays, fully automation approaches are not mature and still supported by the commercial EDA tools [9]. Especially in industry, manual development of test program is still adopted. However, the negative aspect of these approaches is that the effectiveness of manually developed test programs is highly influenced by the skills of the test engineers. The more complex the processors become, the harder it is to reach high fault coverage. Especially on those modules that are in charge of implementing multi-issue execution of instructions. Nowadays, the development of test programs for a complex processor usually follow the divide and conquer approach: the processor is segmented in several sub-modules, and a test program specialized on each of them is developed. By and large, the high quality test program generation is the most challenging part in SBST.

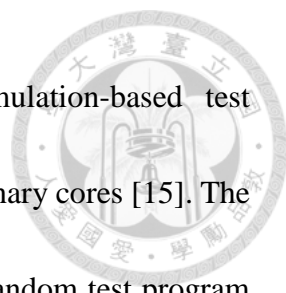


1.2 Review of Previous Techniques

1.2.1 Test program generation approaches

In general, the development of SBST test program consists of four steps. The first step is a creation and optimization of test pattern delivery templates. The second step is a functional constraint extraction. The third step is a test generation process for each module of the processor under test. The last step is basically a process of joining the test pattern with the test pattern delivery templates.

For the last decade, there has been an extensive research on SBST of embedded processors. The major focus of these research was on the method of the test program generation, since the quality of the SBST primarily depends on the test program. There are two main methods for test program generation, ATPG-aided test generation [9,10,11,12] and simulation-based [13,14,15,16] test generation. As to the ATPG-aided test generation, we might divide the processor into several module under tests (MUTs) and ease the tasks of ATPG for deriving the test patterns. However, without the ATPG constraint, some of the generated patterns are typically functionally infeasible. As a result, manually constraints collection is needed. Take today's complexity of the gate-level processor implementation under consideration, it is not feasible to have manual operations at gate-level obviously. Although some automatic constraint extraction methods have been proposed nowadays [17], the efficiency of these methods are still

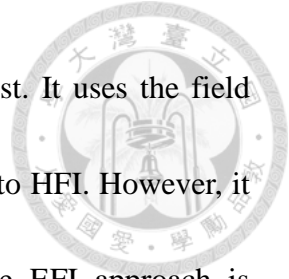


worth discussing. Unlike ATPG-aided test generation, the simulation-based test generation usually combined with the genetic algorithms or evolutionary cores [15]. The classic method of simulation-based test generation is applying the random test program to the processor and modifying the test program according to the response of the simulation results. That is to say, we do not have to consider the problem of constraint extraction when utilizing the simulation-based test generation methodology. However, the fragment program library design and the acceleration and optimization of simulation are the hardest issues to be solved.

1.2.2 Fault injection approaches

The fault injection approaches might be divided into four main categories, hardware-implemented fault injection (HFI), emulation-based fault injection (EFI), software-implemented fault injection (SWIFI) and simulation-based fault injection (SFI). These fault injection and simulation techniques are widely used for evaluating the reliability of the system and the quality of fault tolerance approaches in the presence of possible faults.

HFI techniques [18,19] are injecting the faults to the circuit with the external hardware sources. Because of the additional hardware, HFI is the fastest but the most expensive approach. Besides, the extra hardware might also cause the damage to the injected system. Stuck-at fault is the fault models that usually adopted in HFI.



EFI [20,21] is the alternative technique of HFI with lower cost. It uses the field programmable gate arrays (FPGAs) to achieve performance similar to HFI. However, it requires the synthesizable model of the system. Furthermore, the EFI approach is typically limited to the stuck-at fault model or needs circuit modification for soft errors.

SWIFI [22,23] is based on the alteration of software states. The main advantage is that the simulation might run in near real time, lower costs and development effort. Typical SWIFI techniques intrude the software at machine code and assembly levels. Although it is easy to emulate the hardware when injecting faults at low level, it is hard to map the simulation results to the source code programs.

SFI [24,25] is a non-intrusive approach. Both of the behavior of hardware and software architectures could be modeled. Therefore, SFI can provide the flexibility of the target fault model and increase the controllability and observability. However, it is difficult to model the behavior of modern SoC design. Since many complex components such as processor cores, IPs and memory elements are applied to perform numerous functionalities and their complex interconnections. The efforts of implementing the simulator and low simulation performance are the main shortages of SFI approaches.

1.3 Contribution



Many software-based self-test approaches have been proposed recently. However, most of them target the stuck-at faults. With the operation frequencies keep increasing, the importance of aging defects might not be ignored. That is the main reason why we target the aging defects in the thesis. As to the fault simulation part, we choose to do the simulation at gate-level. Since our target faults are path delay faults (PDF) and transition delay faults (TDF), the complete information of gates and wires is needed. Besides, in order to handle the successive fault activation, the target wires might be under monitoring during the simulation which could only be realized at gate-level. Beside, in order to ensure the quality of the test program generated by our methodology, we have done the program evaluation part by doing random program evaluation.

The main contributions of this thesis are as follows:

- Generate the high quality test program in assembly language.
- Realize the fault simulator that could handle multiple fault activation.
- Evaluate the performance of the test program generated by our methodology.

1.4 Organizations of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, in order to best understand the proposed methodology, we provided some introduction of the basic knowledge of delay fault testing. In Chapter 3, we might describe our proposed methodology in detail. In Chapter 4, we might show some experiment results including fault coverage and test program evaluation. Finally, the conclusion of this thesis will be made in Chapter 5.





Chapter 2

Aging Effect and Delay Fault Testing



2.1 Aging Effect

Circuit aging effect refers to the deterioration of circuit performance over time. The length of time can be a few years to a few months under worst-case conditions. Circuits have always aged. The aging effect was not significant in the past. However, the simultaneous use of extremely small channel lengths, rapidly increasing operation frequencies and extending of IC lifetime, the circuit aging effect could no longer be ignored. All portions of the SoC, whether analog, digital or memory, will be affected. These negative impacts could include slower speeds, irregular-timing characteristics and increased power consumption. In extreme cases, circuit aging might even cause functional failures to occur over time. If we could provide the test for aging defect detection, we could also provide more reliable systems.

2.2 Delay Fault Testing

Delay fault models play a crucial role in the testing field. In order to ensure that the processor meets its performance specifications requires the application of delay test. These test should be applied at-speed and contain two-vector applied to the combinational portion of the circuit under test, to activate and propagate the fault effects to registers or other observation points. Different from the structural BIST, which needs to solve various complex timing issues such as multiple clock domains, multiple frequencies, and test

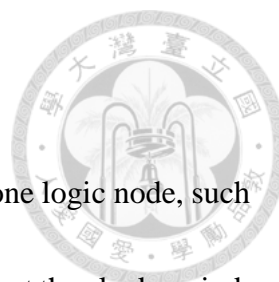


clock skews, using instruction sequence for processor delay fault testing is a more natural application of at-speed tests.

However, it is a difficult problem of detecting timing defects. Not only the test generation, but also the test application phase complicate the testing process. Delay defects are activated and observed by propagating the signal transitions through the circuit. The quality of the test patterns have a significantly impact on the testability of the target faults. Sequential circuits are especially hard to test since not all delay faults in the microprocessor could be tested in the functional mode by any instruction sequence. This is simply because no instruction sequence could produce the desired test sequence that could sensitize the path and capture the fault effect into the destination output of flip-flop at-speed. A fault is said to be functionally testable; otherwise, it is functionally untestable. These functionally untestable might cause the reduction of the fault coverage.

Manufacturing defects or process variations might affect distribution regions of a chip. The path delay fault model is better used for detecting small delay defect (SDD); nevertheless, it is a challenging problem that only a small part of paths in modern designs could be tested. Selecting critical paths requires accurate timing information and the noise factors might have great influence on the signal delay. It is acceptable to target a small subset of paths for test but the decision of target paths is still a hard problem.

Since the objective of this thesis is aging defects detection, we choose transition



delay fault (TDF) and path delay fault (PDF) as our fault models.

TDF model assumes that the large delay defect concentrated at one logic node, such that any signal transition passing through this node might be delayed past the clock period.

There are two types of TDFs at each input and output of a gate, slow-to-rise (STR) fault and slow-to-fall (STF) fault (Figure 2-1). STR fault illustrates the fault that the slow-to-rise transition happened too late and the output flip-flop might capture the wrong value.

On the contrary, STF fault illustrates the fault that the slow-to-fall transition happened too late. The advantage of adopting transition delay fault model is that the implementation of ATPG is much easier since there is no attention to paths. TDF may detect delay defects

such as shorts, opens and coupling defects that missed by stuck-at fault test. However, TDF might miss distributed and small delay defects. As to the fault size, the number of transition delay fault is linear to the circuit size.

TDF might miss distributed and small delay defects. As to the fault size, the number of transition delay fault is linear to the circuit size.

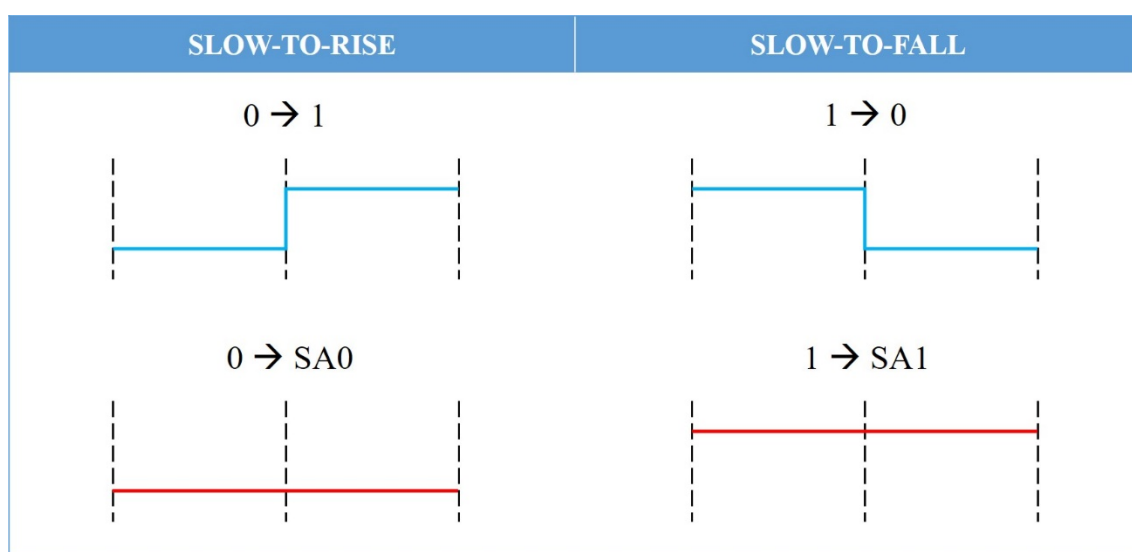
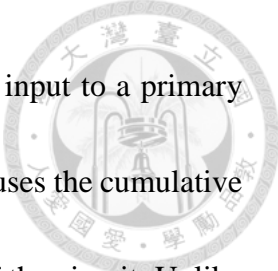


Figure 2-1 Transition delay fault model



A path is a sequence of connected gates from a circuit primary input to a primary output. And the PDF model assumes that a delay defect in a circuit causes the cumulative delay of a combinational path to exceed the specified clock period of the circuit. Unlike the transition delay tests that target the large delay defects, the path delay tests are more likely to detect small delay defects. Compare these two fault models, the PDF model is much more complex than the TDF model with lower fault coverage. Besides, the number of path delay fault is exponential to the circuit size. The large number of paths in modern designs is a major problem for path delay testing. It's really hard to test the whole paths. In practice, only a subset of paths will be tested. For most design, many path delay faults affect the circuit performance but cannot be tested easily. Moreover, there are a lot of structurally testable paths through scan-based testing might be functionally untestable paths. It cannot generate the desired test patterns to sensitize these functionally untestable paths and capture faults effects into the destination primary output of flip-flop in the functional test.

2.3 Software-Based Delay Fault Testing

There are three main tasks in software-based delay fault testing, that is path activation monitoring, fault injection and fault detection.



2.3.1 Path activation monitoring

The objective of path activation monitoring is to monitor whether the target path meets the pre-defined activation conditions. In this thesis, we provide three types of activation conditions: non-robust, robust and robust*. During the fault simulation, we would keep monitoring whether the target path be activated or not. If the target paths are activated, the correspondent fault injection testbench would be generated; otherwise, the target paths would be classified into the category of unactivated faults and no more fault simulation would be applied in the later processes. With the monitoring process, we could remove the unactivated faults from the fault list in advance. Without simulating the faults that would not be detected by the fault simulation, we could save the time and accelerate the whole simulation. Figure 2-2 is the pseudo code of path activation monitoring testbench. Then, after the program execution, we would compare the results in data memory with the golden one as the fault detection.

```
always @($target path output triggered) begin
    if (path successfully activated) begin
        generate the fault injection testbench
    end
end
end
```

Figure 2-2 Pseudo code of path activation monitoring testbench

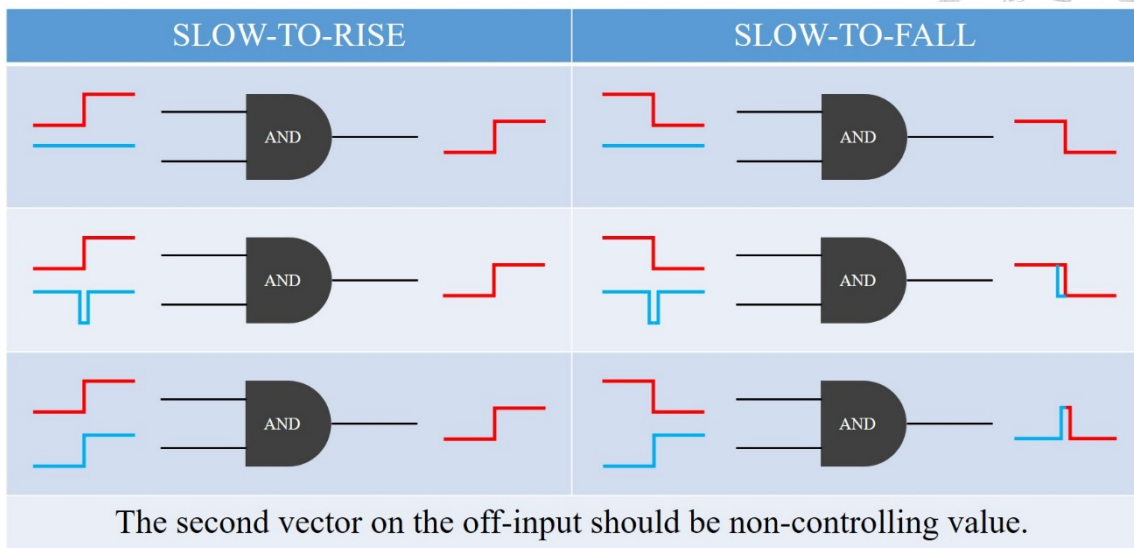


Figure 2-3 Example of non-robust test conditions for AND gate

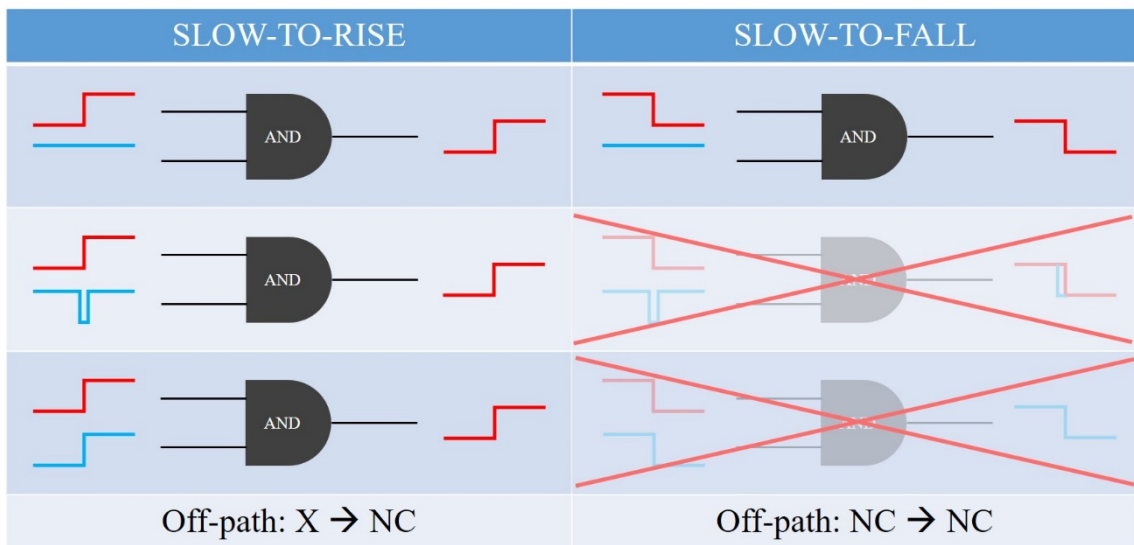


Figure 2-4 Example of robust test conditions for AND gate

Figure 2-3 and 2-4 are the examples of non-robust and robust test conditions for AND gate and the details of these three types of path activation monitoring would be discussed as follows.



Non-robust and robust test are two typical conditions of delay fault testing. Non-robust test guarantees to detect the delay fault only if no other path delay is increased. According to Figure 2-3, we could observe that if all of the on-path signals have the transitions, it could be defined as non-robust conditions. However, in the case of the off-path signals have the transitions before the on-path signals, the conditions might be more complex such as Figure 2-5. Though all of the on-path signals still have the transitions and could be defined as non-robust conditions, the output transition is actually caused by the off-path signals rather than the on-path signals and could not be defined as robust conditions such as the banned conditions in Figure 2-4. Compare these two types of delay path conditions, the robust conditions are stricter than the non-robust conditions. Unlike the non-robust test, the robust test guarantees to detect the delay fault independent of all other delays in the circuit.

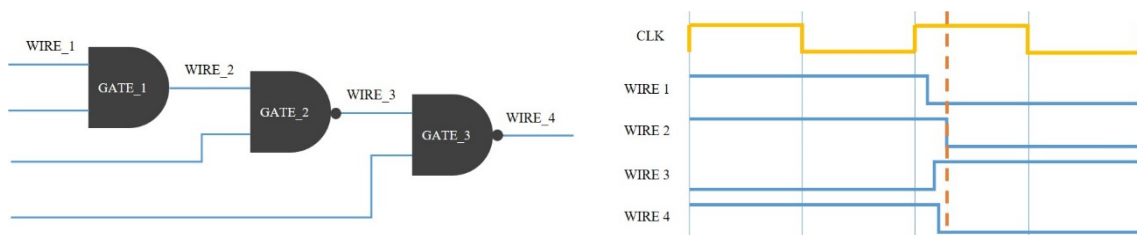
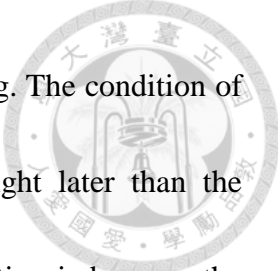


Figure 2-5 Problem of non-robust test

In addition to the robust method, another monitoring method called robust* is adopted for path activation monitoring. The condition of robust* monitoring is stricter



than the robust monitoring, but looser than the non-robust monitoring. The condition of robust* monitoring is that the transitions of the gate outputs might later than the transitions of the gate inputs. The reason why we define this condition is because the transitions of the gate outputs might later than the transitions of the gate inputs in the time simulation. Since the robust monitoring needs to check whether the whole off-path signals meet the conditions of robust, it needs much effort for monitoring the whole off-path signals and takes much time comparing to the robust* monitoring. With the robust* monitoring, we could only focus on the on-path signals and solve the problem of non-robust monitoring as Figure 2-5 shows.

To sum up, Figure 2-6, Figure 2-7 and Figure 2-8 illustrate the three types of path activation monitoring. Each of them represents the different conditions of the path activation. If the whole on-path wires have transitions, the target path might satisfy the non-robust condition. If the whole on-path wires have transitions and the whole off-path wires are non-controlling, the target path might satisfy the robust condition. Non-robust monitoring is the looser conditions with higher fault injection rate. On the contrary, robust monitoring is the stricter condition with lower fault injection rate.

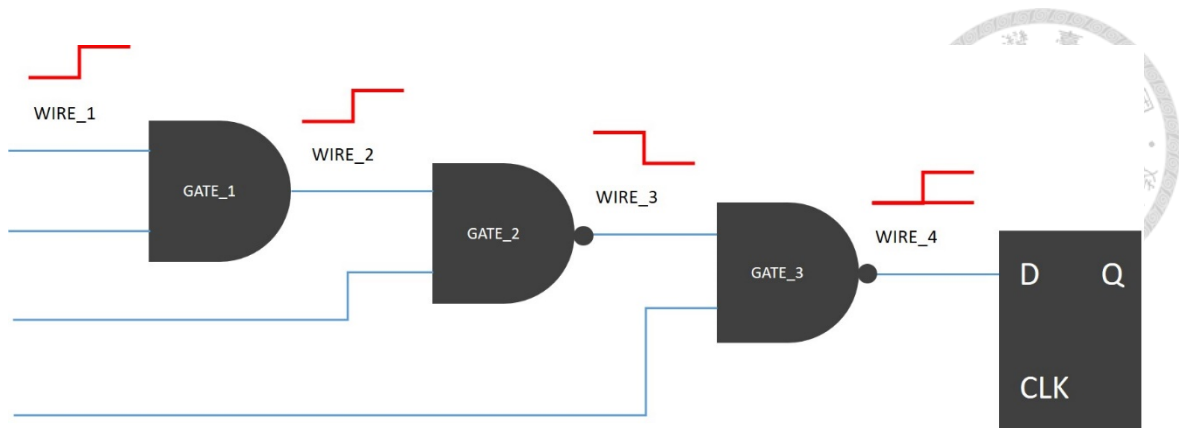


Figure 2-6 Example of a non-robust path

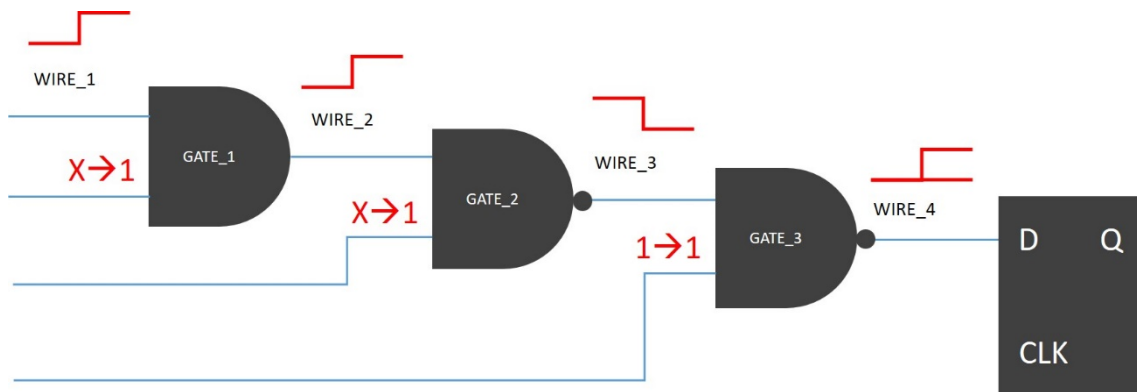


Figure 2-7 Example of a robust path

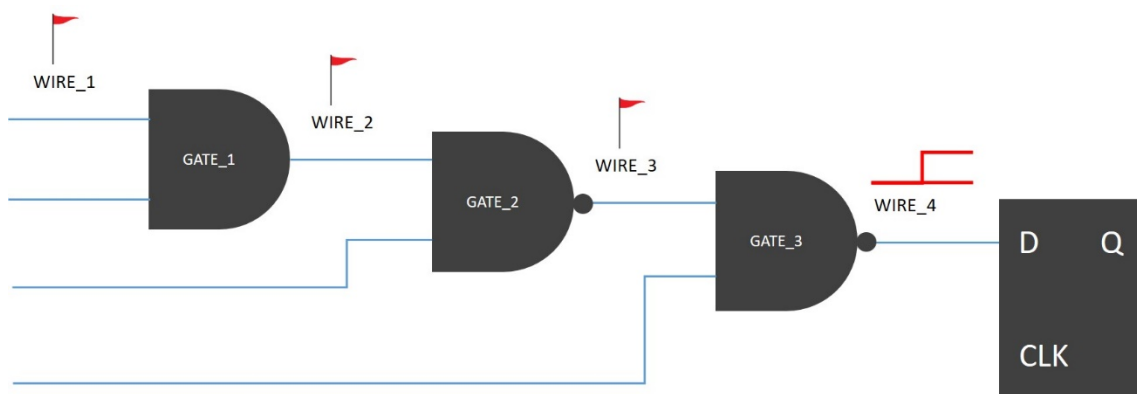


Figure 2-8 Example of a robust* path

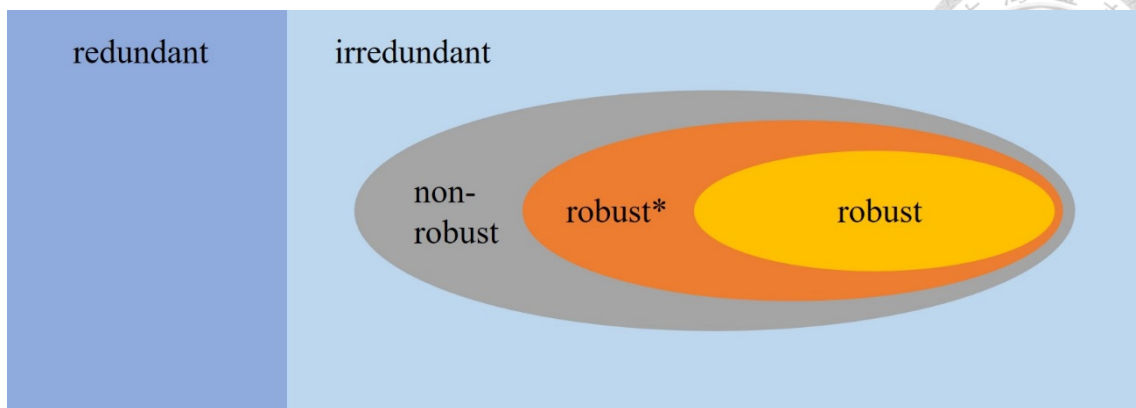


Figure 2-9 Testable path coverage

2.3.2 Fault injection and detection

The cause of path delay fault is that the target transition occurs later than the specified clock period and the target register might catch the wrong value. As shown in Figure 2-10, we could observe the fault behaviors in gate-level and RT-level simulation.

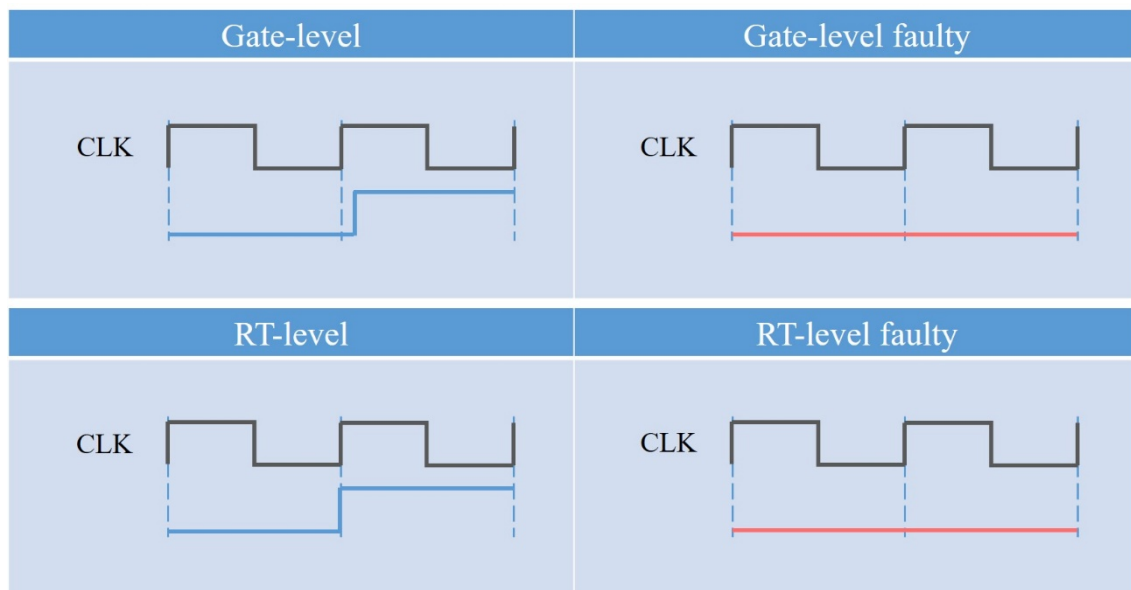


Figure 2-10 Fault behaviors in gate-level and RT-level simulation

```

always @($target path output triggered) begin
  if (path successfully activated) begin
    $deposit ($target register, ~$target register )
  end
end
end

```

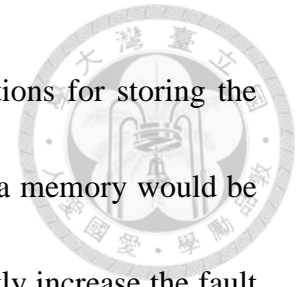


Figure 2-11 Pseudo code of fault injection testbench

Figure 2-11 is the pseudo code of fault injection testbench. While the target path is successfully activated, the value of the target register might be bit-flipped just as the behavior of path delay fault.

The observability solution for the fault detection is comparing the results in the data memory with the golden ones after the program execution. The fault is defined as the detected fault (DT) if the fault effect could be propagated to the data memory and the computational results are different from the golden ones. Otherwise, the fault is an undetected fault (UD) if the fault effect is masked during the simulation and the results in the data memory are same as the golden ones. However, since the limitation of the observability, only check the execution results in the data memory might reduce the fault coverage significantly. Some faults are verified to be undetected faults because they do not have any influence on the computational results in the data memory but they actually exist in the circuit. The reason why these faults could not be observed is that there are no instructions could store the results such as overflow flag and stall signal to the data

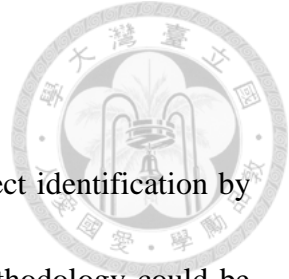
memory directly. As a result, the development of creating instructions for storing the computational results that could not be observed directly in the data memory would be the future work. With these observation instructions, we could greatly increase the fault coverage.





Chapter 3

Proposed Methodology for Software-Based Self-Test on Delay Defects



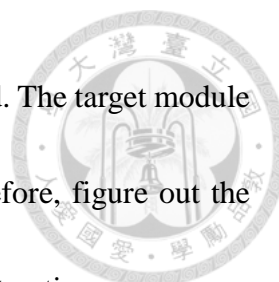
3.1 Proposed Methodology

The objective of the proposed methodology is early aging defect identification by software-based self-test approach for processors. The proposed methodology could be separated into three main steps, pre-processing, test generation and fault simulation. The detail of each step might be explained respectively in the following sections.

3.2 Pre-Processing

The main goal of pre-processing is to do the constraint extraction. Since there are some states that cannot be reached through any instruction sequence. If we could figure out the constraint of the processor previously, we might avoid the unreachable states and generate the patterns that could be converted into instructions. Nowadays, fully automation of constraint extraction is still not mature. Although some automatic methodology of constraint extraction has been proposed recently, they cannot meet the sufficient accuracy. As a result, manually constraint extraction is still adopted especially in industry. In this thesis, we do the constraint extraction manually. However, with the complexity of processors keep increasing, the importance of the automation of constraint extraction cannot be overemphasized.

The development of the test program for a complex processor usually follows the divide and conquer approach. That is to say, the processor is segmented in several sub-



modules, and a test program specialized on each of them is developed. The target module that we would like to test is the arithmetic logic unit (ALU). Therefore, figure out the input and output of the ALU is the previous step before constraint extraction.

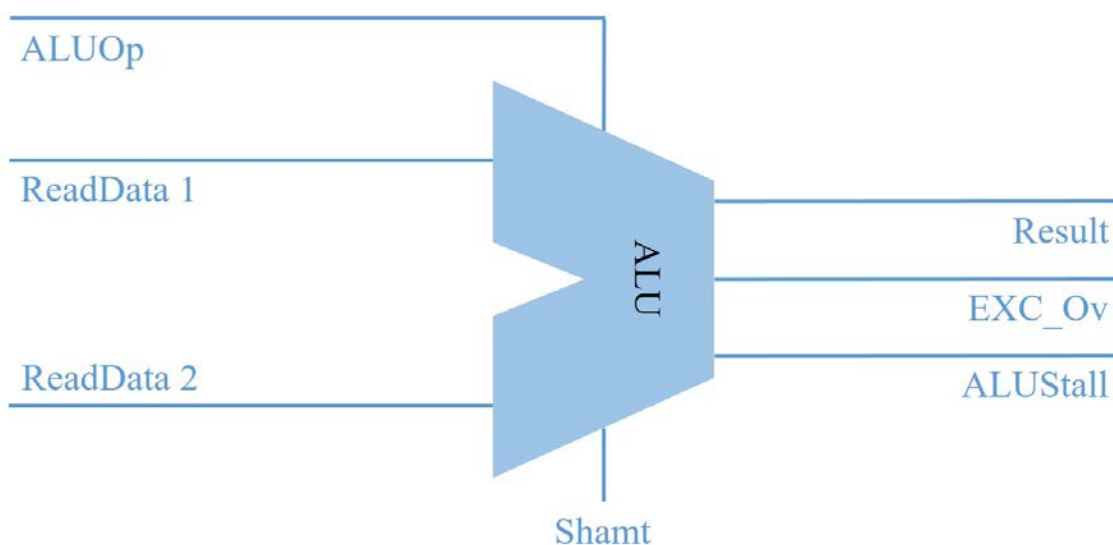
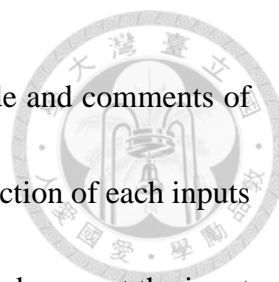


Figure 3-1 Arithmetic logic unit (ALU)

Figure 3-1 is the ALU module that we are going to test. The ALUOp wire is a 5 bits wire that control the ALU operation. Table 3-1 shows the relation between the ALU operation signal and the instruction. The ReadData1 wire and ReadData2 wire are 32 bits wires which are the operands of the ALU. The Shamt wire is a 5 bits wire that be used for shift operations. The result wire is a 32 bits wire which is the ALU computation result. The EXC_Ov is the overflow flag and the ALUStall is the stall signal for long ALU operation such as divide.



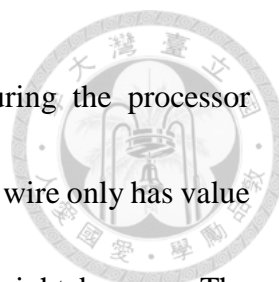
The method for deriving the constraints is reading the RTL code and comments of our processor test. By reading the source code, I could realize the function of each inputs and how they work. Then, I would apply some random programs and dump out the input signals per cycle. The constraints could be confirmed by analyzing the simulation results.

Table 3-1 Mapping of ALU operation signal and instruction

ALU_Op	Inst.	ALU_Op	Inst.	ALU_Op	Inst.	ALU_Op	Inst.
0	ADDU	8	MADDU	16	MULT	24	SLTU
1	ADD	9	MFHI	17	MULTU	25	SRA
2	AND	10	MFLO	18	NOR	26	SRAV
3	CLO	11	MTHI	19	OR	27	SRL
4	CLZ	12	MTLO	20	SLL	28	SRLV
5	DIV	13	MSUB	21		29	SUB
6	DIVU	14	MSUBU	22	SLLV	30	SUBU
7	MADD	15	MUL	23	SLT	31	XOR

Table 3-2 Constraints of ALU input

ALU input	constraint
clock	no constraint
reset	zero
EX_Stall	zero
EX_Flush	zero
A [31:0]	no constraint
B [31:0]	no constraint
Shamt [4:0]	only has value with “SLL”, “SRA”, “SRL”; otherwise zero
Operation [4:0]	some values are unavailable, show in the following pages



The reset, EX_Stall and EX_Flush wires should be zero during the processor execution. There are no constraint of the 32 bits operands. The Shamt wire only has value while executing the SLL, SRA ad SRL instruction; otherwise it might be zero. The Operation wire has three constraint values, 5, 6 and 21. According to Table 3-1, 5 and 6 could be mapped to the instruction DIV and DIVU. Since DIV and DIVU instructions might execute 32 cycles, we could not control them with two successive patterns. Therefore, other methods should be adopted for testing the divider module. And 21 is the reserved value for MIPS32 release 2 instruction.

3.3 Test Generation

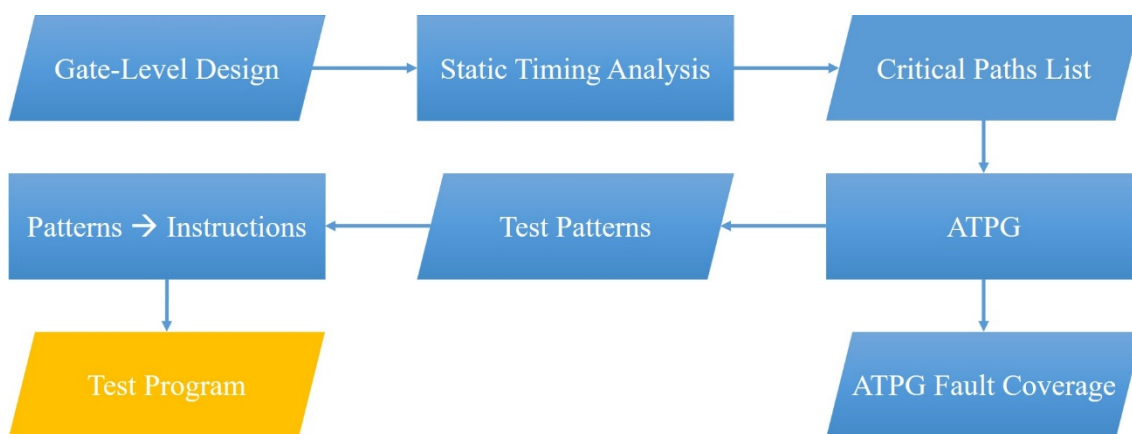



Figure 3-2 Flowchart of test generation

Figure 3-2 shows the proposed methodology flow for test generation. The test generation processes path delay fault and transition delay fault quite similar. The only

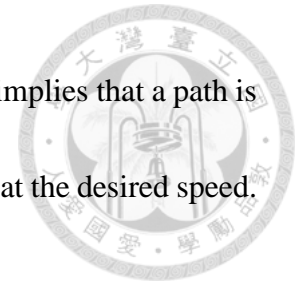


difference is the process of critical paths extraction by static timing analysis. The reason why the path delay fault testing needs the process of static timing analysis and the detail of each steps in the methodology flow will be explained in the following section

3.3.1 Static timing analysis

Static timing analysis (STA) is a fast and reasonable measurement for computing the circuit timing without simulating the entire circuit by input patterns. Unlike the number of transition delay fault which is linear to the circuit size, the number of path delay fault is exponential to the circuit size. It is impracticable to test the whole path delay faults in the circuit. As a result, we tend to do the static timing analysis to figure out the critical paths. The critical path is defined as the serially combinational gates of a path with maximum delay which may have higher probability having the timing violation. The process of critical paths extraction might greatly reduce the fault list size. We consider the slack of setup time violation to find out critical paths in the circuit. If the required signal arrives too late, it may cause the setup time violation. The transition of input signals, different operating environment and manufacturing variations contribute to the delay of signal arrival time, as well as the aging defect. The slack is defined as the difference between the required time and the arrival time. A positive slack implies that the arrival time is earlier than the required time. That is to say, the path with positive slack might not

affect the overall delay of the circuit. Conversely, the negative slack implies that a path is too slow, and the path must be sped up if the whole circuit is to work at the desired speed.



3.3.2 Automatic test pattern generation (ATPG)


As stated above, there are two major method for test program generation, ATPG-aided test generation and simulation-based test generation. ATPG-aided test generation is the deterministic, stable and efficient method with acceptable fault coverage. It stands on the view point of circuit analysis and generates the high quality test patterns. Nevertheless, the difficulties is the constraint extraction and the mapping between test patterns and instructions.



```
$path {  
  // from: B[26]  
  // to: Result[0]  
  $name "IO_1" ;  
  $cycle 0 ;  
  $slack -0.455934 ;  
  $transition {  
    "U3905/B" v ; // (MUX2X1)  
    "U3907/B" ^ ; // (MUX2X1)  
    "U3911/B" v ; // (MUX2X1)  
    "U3920/A" v ; // (MUX2X1)  
    "U2635/A" ^ ; // (IN VX1)  
    "U2880/A" v ; // (AOI22X1)  
    "U2167/A" ^ ; // (BUFX2)  
    "U1679/A" ^ ; // (AND2X1)  
    "U1677/A" ^ ; // (IN VX1)  
    "U155/A" v ; // (OR2X1)  
    "U154/B" v ; // (OR2X1)  
    "U2881/B" v ; // (OAI21X1)  
    "U1595/B" ^ ; // (AND2X1)  
    "U1596/A" ^ ; // (IN VX1)  
  }  
}
```

Figure 3-3 Critical path example reported by Synopsys Primetime


In the steps for setting desired ATPG options, we might set the constraints that we extracted previously. With the constraints, we could assure that the test patterns might be converted to the instructions successfully. There are two types of constraint option in TetraMAX, `add_atpg_constraints` and `add_atpg` primitives. `Add_atpg_constraints` command defines on nets that must be satisfied during pattern generation. In this



command, we should specify a name to identify the constraint, the constraint value (0, 1, Z), and the place in the design to apply the constraint. Add_atpg_primitives command creates a primitive that is added to the design and has its inputs connected to specified nets. When you constrain the output of the added primitive, it forces the pattern generation algorithm to conform to specified logical conditions at the connection points. In this command, we should specify a name for the added primitive, its logical function, and its input connections.

3.3.3 Pattern-to-instruction converter

After we derive the test patterns, the next step is to generate the test program. The test program format could be separated into three main sections, operands preparation, two-vector and result store. In the rest of this section, an example of converting patterns to instructions might be showed and illustrated in detail. Figure 3-4 is a pattern that generated by Synopsys TetraMAX. First, we need to figure out the value of each input of ALU according to their position. Then, we might get the value that should be applied on the ALU inputs. In this step, we might also check whether the pattern is legal that meets the constraints we set. Figure 3-5 is the result after doing the test pattern classification. We could observe that the pattern meets the constraints and could be converted into instructions. Table 3-3 is the mapping table that records the relation between ALU inputs



and instructions. The mapping table is derived manually by reading the RTL description of the processor and doing some simulation. However, with the SoC design becoming more and more complex, the automation of generating the mapping table between test patterns and instructions should be developed in the future. According to the mapping table, we might convert the patterns into two-vector instructions. Although it seems that we finish the conversion patterns and instructions. There are still some tricky details about operand preparation and register usage that would be discussed as follows.

```

{ pattern 1 fast_sequential }
{ vector }
  vector("_default_WFT_") := [ 0001110111000110111001001
  101110101010100010101101101101001110001010
  10110000000];
{ capture }
  vector("_default_WFT_") := [ 0001010010011101010010100
  01010010110100011110111011011000000011111011
  100000000];

```

Figure 3-4 A test pattern generated by Synopsys TetraMAX

{ vector 1 }		{ vector 2 }	
clock	-	clock	-
reset	0	reset	0
EX_Stall	0	EX_Stall	0
A	EE3726EA	A	A4EA514B
B	A2B6D38A	B	47BB603E
Operation	16	Operation	1C
Shamt	0	Shamt	0
EX_Flush_BAR	0	EX_Flush_BAR	0

Figure 3-5 Classification of pattern information

Operand Preparation	Two-Vector	Result Store
lui \$t0, 60983 addi \$t0, 9962 lui \$t1, 41655 addi \$t1, 54154 lui \$t3, 42218 addi \$t3, 20811 lui \$t4, 18363 addi \$t4, 24638	sllv \$t2, \$t1, \$t0 srlv \$t5, \$t4, \$t3	sw \$t2, 8(\$zero) sw \$t5, 12(\$zero)

Figure 3-6 Example of test program generated by our methodology

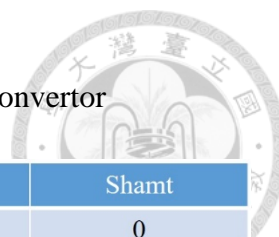
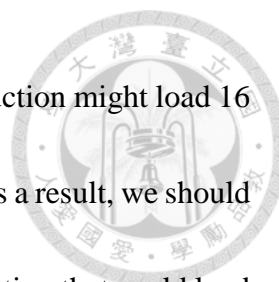


Table 3-3 Mapping table for patterns to instructions convertor

Instruction	ALUOp	ReadData1	ReadData2	Shamt
ADDU R_D, R_S, R_T	00	R_S	R_T	0
ADD R_D, R_S, R_T	01	R_S	R_T	0
AND R_D, R_S, R_T	02	R_S	R_T	0
CLO R_D, R_S	03	R_S	0	0
CLZ R_D, R_S	04	R_S	0	0
MADD R_S, R_T	07	R_S	R_T	0
MADDU R_S, R_T	08	R_S	R_T	0
MFHI R_D	09	0	0	0
MFLO R_D	10	0	0	0
MTHI R_S	11	R_S	0	0
MTLO R_S	12	R_S	0	0
MSUB R_S, R_T	13	R_S	R_T	0
MSUBU R_S, R_T	14	R_S	R_T	0
MUL R_D, R_S, R_T	15	R_S	R_T	0
MULT R_S, R_T	16	R_S	R_T	0
MULTU R_S, R_T	17	R_S	R_T	0
NOR R_D, R_S, R_T	18	R_S	R_T	0
OR R_D, R_S, R_T	19	R_S	R_T	0
SLL $R_D, R_S, \text{shift5}$	20	0	R_T	shift5
SLLV R_D, R_S, R_T	22	R_T	R_S	0
SLT R_D, R_S, R_T	23	R_S	R_T	0
SLTU R_D, R_S, R_T	24	R_S	R_T	0
SRA $R_D, R_S, \text{shift5}$	25	0	R_T	shift5
SRAV R_D, R_S, R_T	26	R_T	R_S	0
SRL $R_D, R_S, \text{shift5}$	27	0	R_T	shift5
SRLV R_D, R_S, R_T	28	R_T	R_S	0
SUB R_D, R_S, R_T	29	R_S	R_T	0
SUBU R_D, R_S, R_T	30	R_S	R_T	0
XOR R_D, R_S, R_T	31	R_S	R_T	0

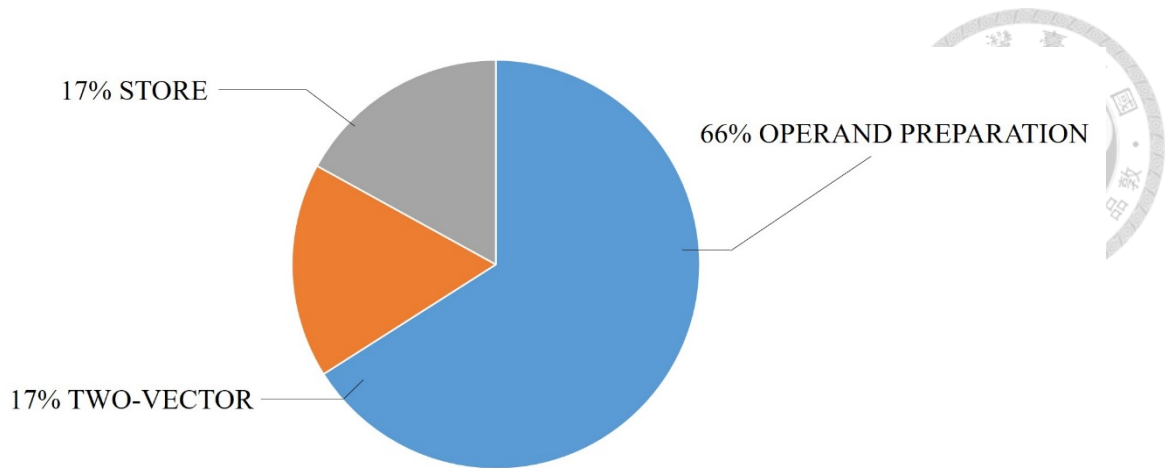
Since we need to execute the test pattern during two successive cycle, the operands should be prepared in advance. As shown in Figure 3-6, we could observe that the operands have been prepared before the two-vector part. Each operand needs the combination of two instructions LUI and ADDI to reach the target value. The reason why



we need two instructions to reach the target value is because one instruction might load 16 immediate value at most. However, the operands are 32 bits values. As a result, we should utilize two instructions to load the 32 bits operands. LUI is the instruction that could load upper 16 bits value to the target register. ADDI is the instruction that could add lower 16 bits value to the target register. There is a small detail we need to take care, the LUI instruction might load the upper 16 bits immediate value with the lower 16 bits value set to zero. Therefore, LUI instruction must place before the ADDI instruction, or the wrong value would be loaded to the register. On the other hand, in order to ensure that the two instructions might be executed during two successive cycles. We should choose different registers for storing operands. That is to say, we should avoid reusing the registers which are used in the first vector. Since the processor contains full data forwarding unit, the two instructions might not be executed during the two successive cycles if we do not avoid the problem of data dependency. As shown in Figure 3-7, we might observed that the registers that used in the second vector might not be used in the first vector.

```
sllv $t2, $t1, $t0 → vector 1
srlv $t5, $t4, $t3 → vector 2
```

Figure 3-7 Example of two vectors from the test program



50

Figure 3-8 Instructions distribution of the test program

Figure 3-8 is the pie chart that suggests the instructions distribution of the test program. According to the pie chart, we could observe that 66% of the instructions in the test program do the work for operands preparation. The number of instructions for operands preparation is four times as much as the number of instructions for two-vector or result store. In my opinion, if we could create the instruction that could load 32 bits value to the register at a time, we could greatly shrink the size of the test program.

3.4 Fault Simulation

After we generate the test program, the next step would be the fault simulation for confirming the effect of the test program that we generated. In the process of fault simulation, the simulation-based fault injection method would be adopted. Figure 3-9 is the flowchart of fault simulation.

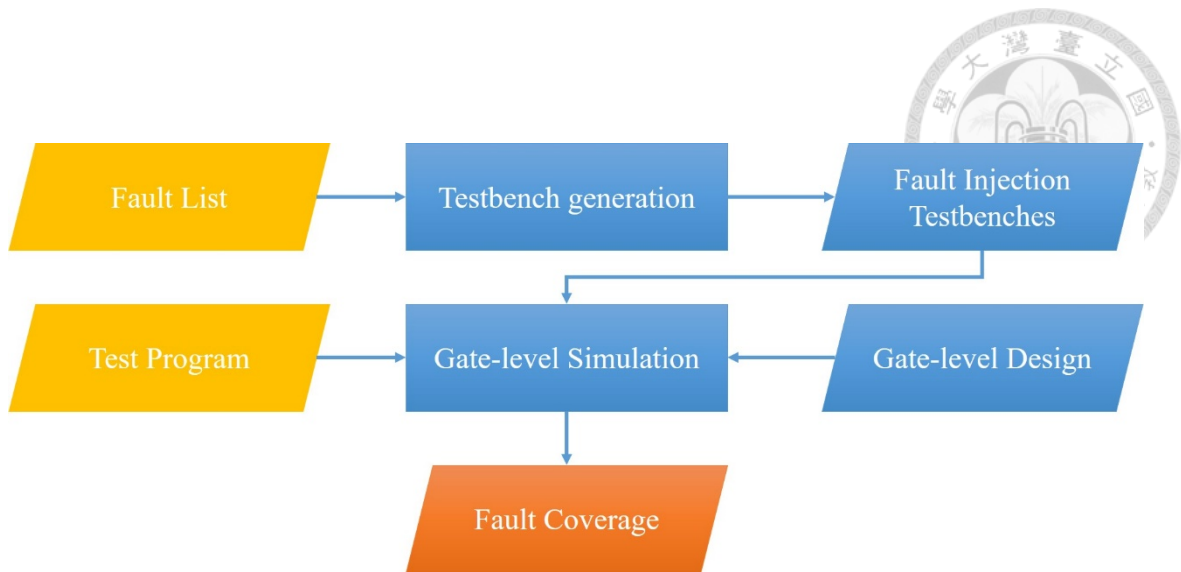


Figure 3-9 Flowchart of fault simulation



Chapter 4

Experiment Result



4.1 Experiment Setup

The target processor in our experiments is a MIPS32 processor. Figure 4-1 is the MIPS32 processor architecture from [31]. This processor is an open-source design and could be downloaded from Github. This design was created by Grant Ayers and funded by the eXtensible Utah Multicore (XUM) project at the University of Utah from 2010-2012.

It is a standalone MIPS32 processor, all required MIPS32 instructions are implemented, including hardware multiplication and division. This is a bare-metal processor, without memory management unit (MMU) and floating point unit (FPU). The hardware divider is small, multi-cycle and runs asynchronously from the pipeline allowing some masking of latency.

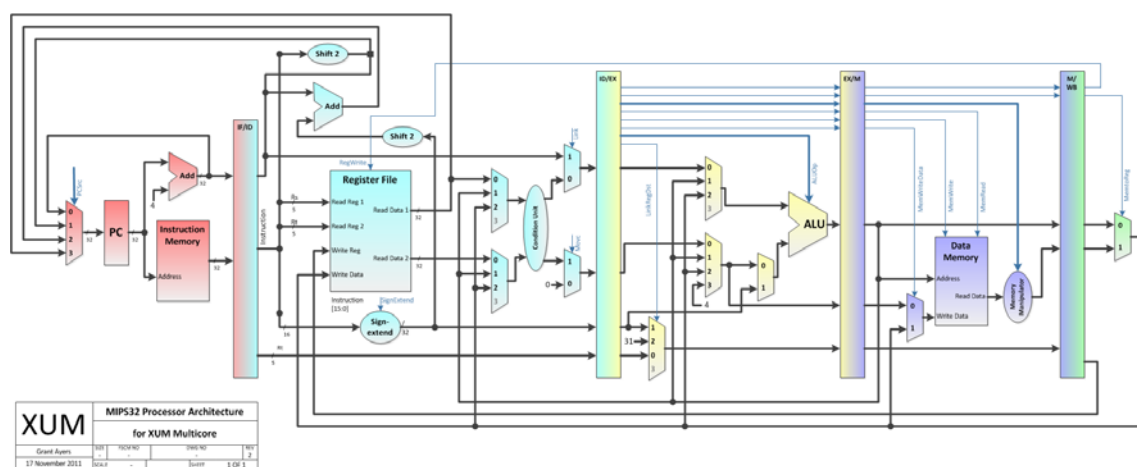


Figure 4-1 MIPS32 processor architecture

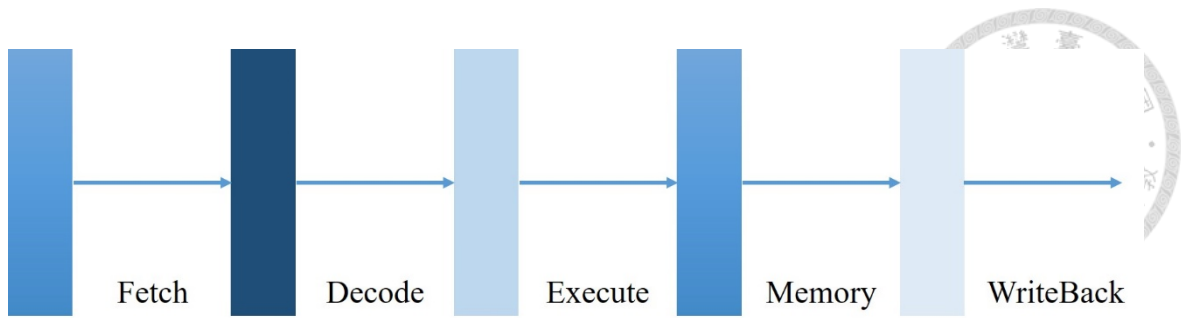


Figure 4-2 Single-issue in-order 5-stage pipeline

This MIPS32 processor architecture is the single-issue in-order 5-stage pipeline including Instruction fetch, Instruction decode, Execute, Memory and Write back stages. Figure 4-2 illustrates the pipeline stages of the architecture.

Besides, the memory interface is separated from the processor. The original design of the memory utilizes four-way handshake to exchange the data. Figure 4-3 explains the mechanism of the four-way handshake. This interface is simple and robust but the performance of the system is limited. The minimum CPI is increased from 1 to between 3 and 4. It is not practical in SoC designs nowadays. In addition, this handshake mechanism causes the pipeline stages to be stalled. The stalled pipeline stage would lead to the untestable faults. In order to prevent over-testing the functionally untestable faults, we modify the design to make CPI be close to 1. However, accessing the data memory still needs two cycles to exchange the data.

The experiments run on a Intel(R) Xeon(R) CPU E3-1230 v3 @ 3.30 GHz with 32 GB RAM.

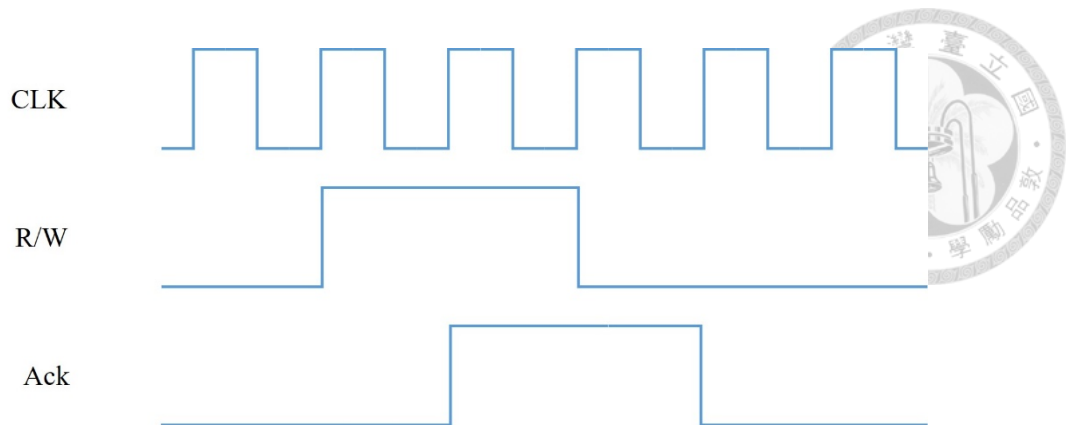


Figure 4-3 Four-way handshake mechanism

The EDA tools we used in the methodology flow are described as follows:

Synopsys Design Compiler

- Synthesize the RTL Verilog design to get the gate-level circuit. There are totally 1,885 flip-flops in the synthesized design.

Synopsys Primetime

- Static timing analysis tool for figuring out the vulnerable paths with lower slack. The critical paths list could be used not only for generating the fault list but also for test pattern generation by TetraMAX.

Synopsys TetraMAX

- ATPG tool for generating the test patterns which would be used to be converted to assembly test program.

Cadence NC-Verilog

- Run timed gate-level simulation.



4.2 Result Statistics

In this section, we would display some experimental results and evaluate the quality of the test program generated by our methodology.

4.2.1 Transition delay fault testing

Table 4-1 and Table 4-2 are the results of transition delay fault testing by TetraMAX and software-based self-test respectively. Figure 4-4 are the equations of coverage calculation. In Table 4-1, we could observe that the ATPG-untestable faults account for around 40% of the total faults. The main reasons that cause the ATPG-untestable faults could be conclude into two points. The first reason is the lack of design-for-testability (DFT). Without the DFT insertion, the observability of the circuit would be decreased and cause the poor performance of the fault coverage. The second reason is the addition of ATPG constraints. Since we need to ensure that the test patterns could be converted into instructions, some unreachable states or illegal conditions should be confined previously. Compare Table 4-1 and Table 4-2, the fault coverage of the transition delay fault testing by software-based self-test is a bit higher than the fault coverage of the testing by TetraMAX. This result implies that we convert the test patterns into the instructions completely. Besides, some faults are detected by software-based self-test accidentally.

Table 4-1 Transition delay fault testing by TetraMAX

	TetraMax	SBST
Detected	8513	8525
ATPG-untestable	6558	6558
Not-detected	1447	1435
Fault coverage	51.54%	51.62%
Test coverage	85.58%	85.60%

$$\text{Fault coverage} = \frac{\text{Detected faults}}{\text{Total faults}} \times 100\%$$

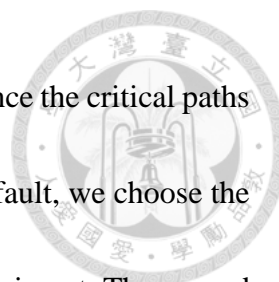
$$\text{Test coverage} = \frac{\text{Detected faults}}{\text{Total faults} - \text{ATPG untestable faults}} \times 100\%$$

Figure 4-4 Equations of coverage calculation

4.2.2 Path delay fault testing

Except for the transition delay fault which represents the large delay defect, the path delay fault which represents the small delay defect is also a hot issue when it comes to delay fault testing.

Table 4-2 are the experimental results by different monitoring conditions respectively. As mentioned before, the fault size of the path delay fault is exponential to the circuit size. It is impractical to test the whole path delay faults in the circuit. Actually,



we only test the subset of the whole path delay faults in the circuit. Since the critical paths with lower slack might have higher probability to get the path delay fault, we choose the top thousand critical paths for the path delay fault testing in our experiment. The second row is the fault coverage by the constrained ATPG and non-constrained ATPG. The fault coverage by the constrained ATPG is around 40% less than the non-constrained ones. That is to say, whether the constraints be applied or not might have great influence on the fault coverage. The third row is the number of activated paths according to the three different methods. As we mentioned in subsection 3.4.1, the non-robust monitoring is the method with loosen condition and could activate more paths than the other methods. On the contrary, the robust monitoring with stricter condition could active less paths than the other methods.

Table 4-2 Path delay fault testing by software-based self-test

	Non-robust	Robust	Robust*
Total faults	1002		
ATPG fault coverage	22.48% (constrained)	61.35% (non-constrained)	
Activated paths	357	245	273
Detected faults	307	237	266
Fault coverage	30.64%	23.65%	26.55%
Test coverage	41.21%	31.81%	35.70%
Activated / Detected	85.99%	96.73%	97.44%

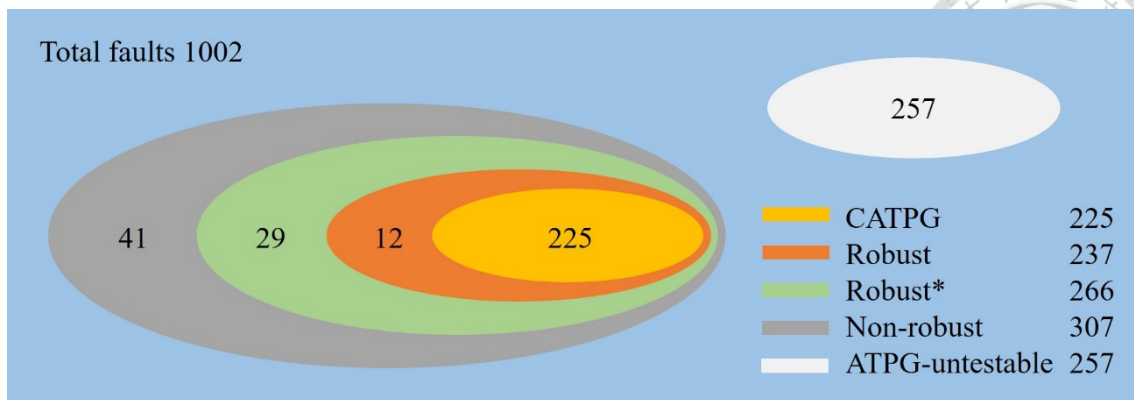
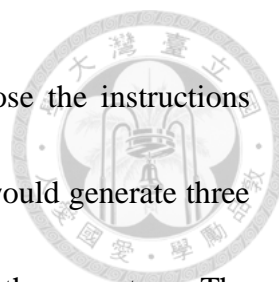


Figure 4-5 Venn diagram of fault detection

Figure 4-5 is the Venn diagram of the fault detection. The faults that could be detected by the non-robust monitoring might be detected by the robust, robust* and constrained ATPG methods. The faults that could be detected by the robust* monitoring might be detected by the robust and constrained ATPG methods. To sum up, the faults that could be detected by the monitoring with strict condition might also be detected by the monitoring with loose condition.

4.2.3 Random program evaluation

The major objective of the random program evaluation to evaluate the quality of test program generated by our methodology. The processes are generating random test programs, doing fault simulation by our simulator and finally comparing the results. The random program is not totally random. Table 4-3 illustrates the format of the random program. First, we need to randomly generate the operands that would be used by the



instructions in the second step. Second, we would randomly choose the instructions according to the MIPS32 instruction set reference. In this step, we would generate three different programs with successive one vector, two vectors and three vectors. The difference of these three types of program would be discussed afterwards. Third, we should store the result to the data memory for checking the fault effect.

Table 4-3 Random program format

Format	Example
1. operand preparation	lui \$t0, 13579 addi \$t0, 13579 lui \$t1, 24680 addi \$t1, 24680
2. random instruction - 1 vector - 2 vectors - 3 vectors	sub \$t2, \$t1, \$t0
3. store result	sw \$t2, 0(\$zero)

The test programs would be simulated by our simulator with non-robust and robust mode separately. The fault coverage would be recorded per hundred instructions. Finally, we would draw the line graph for observing and analyzing the results.

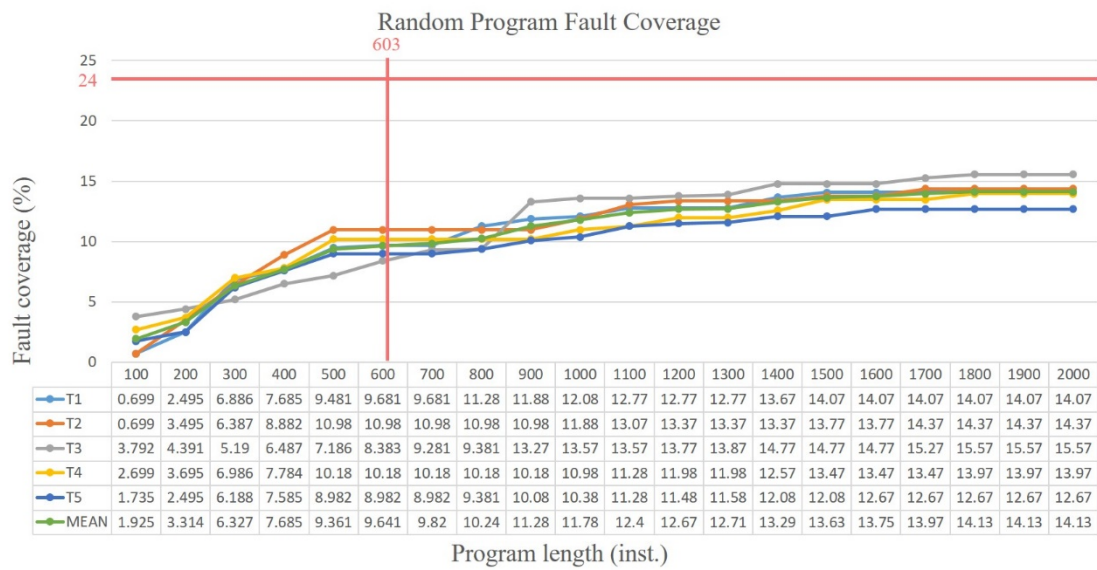


Figure 4-6 Random program evaluation in robust 1-vector mode

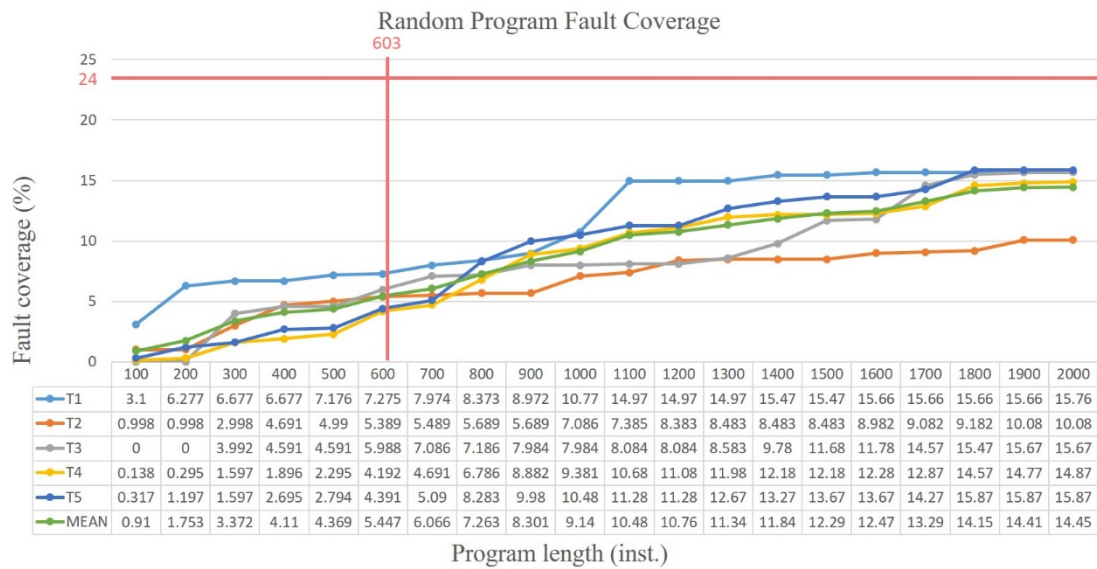


Figure 4-7 Random program evaluation in robust 2-vector mode

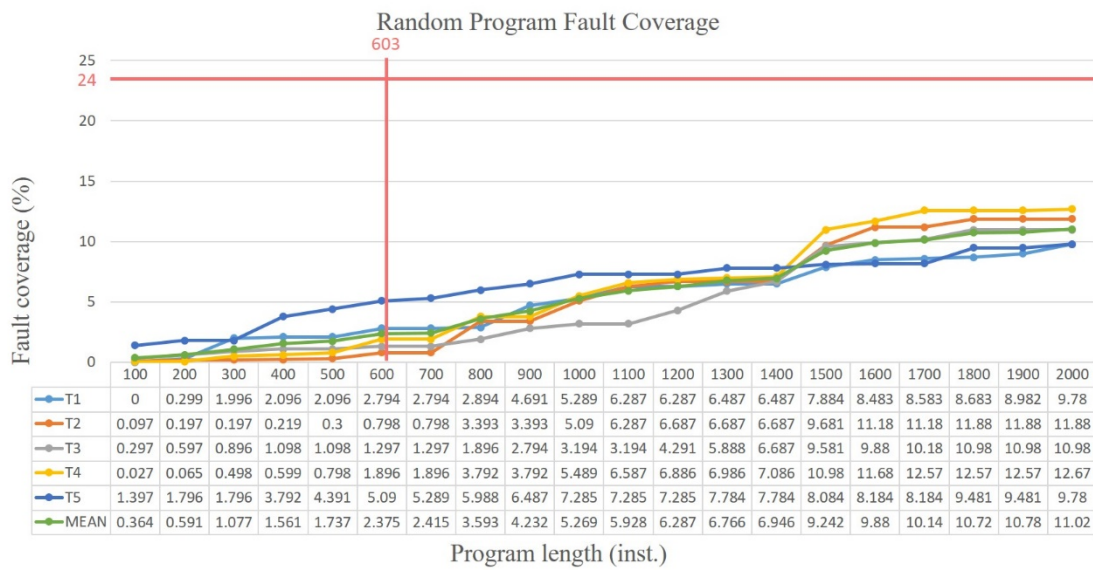


Figure 4-8 Random program evaluation in robust 3-vector mode

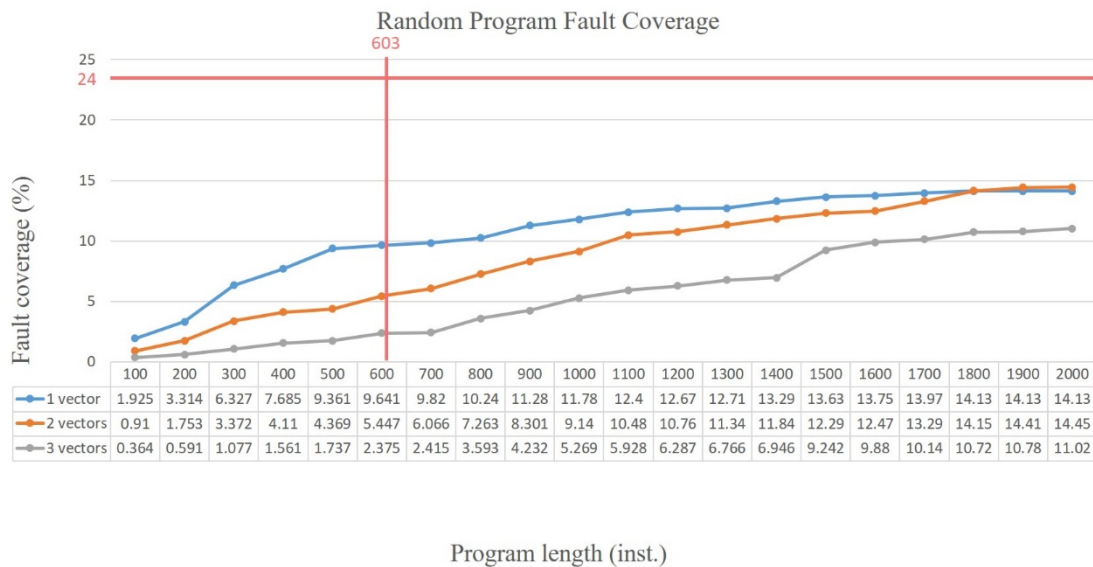
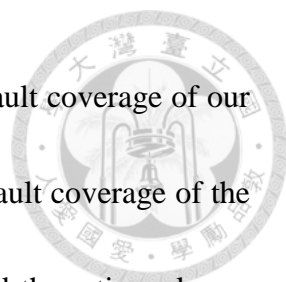


Figure 4-9 Mean fault coverage of different vectors in robust mode

Figure 4-6, Figure 4-7 and Figure 4-8 are the results of random program fault coverage. Figure 4-9 is the mean fault coverage of the random program with different vectors. The marking line in these figures is the fault coverage of the test program



generated by our methodology. According to the marking line, the fault coverage of our test program could reach 24% with 603 instructions. However, the fault coverage of the random programs could locate in between 10% to 15% with around three times larger than the program size of our test program.

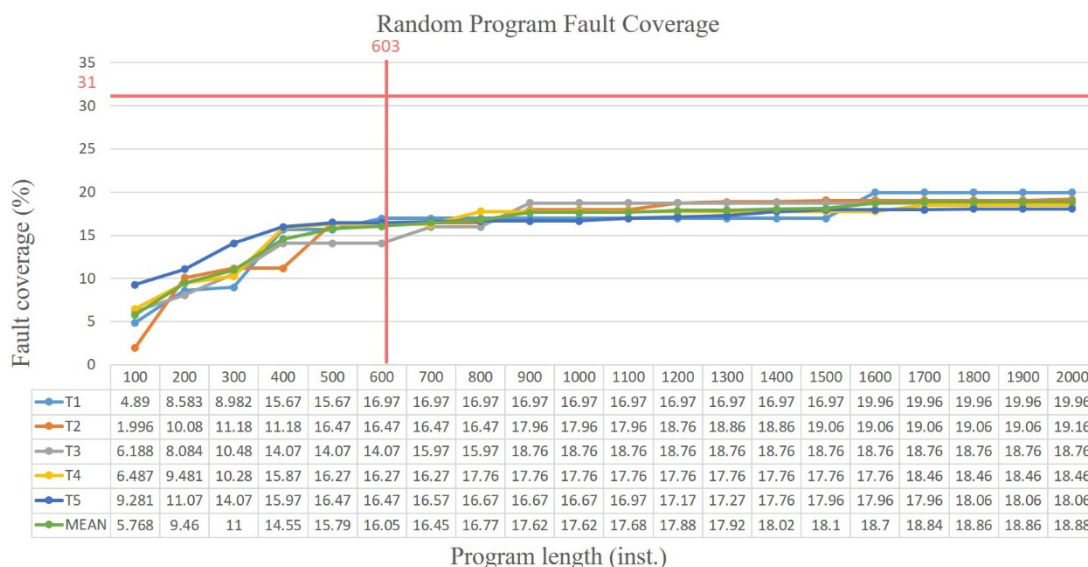


Figure 4-10 Random program evaluation in non-robust 1-vector mode

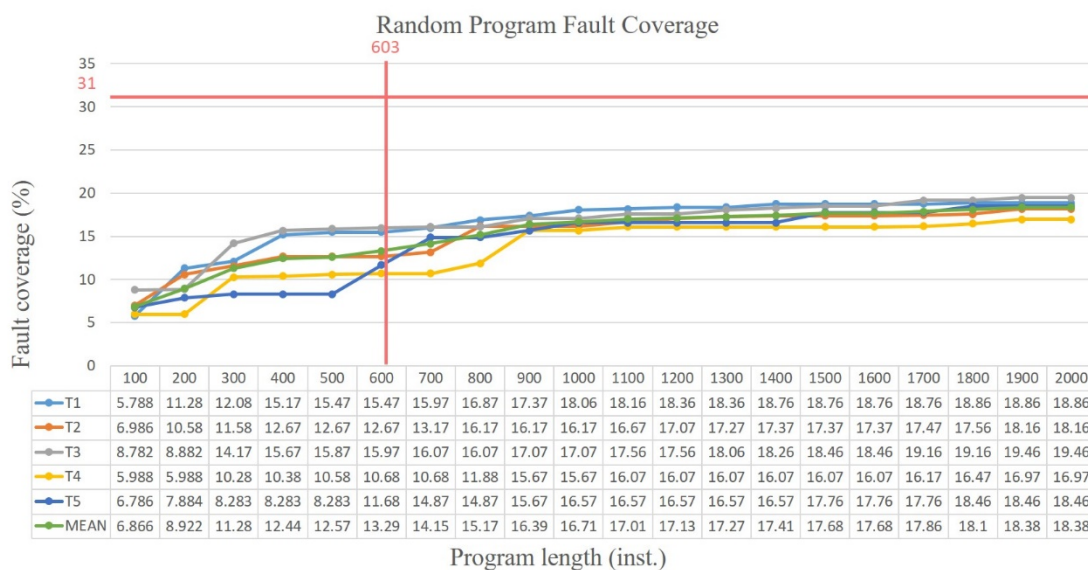


Figure 4-11 Random program evaluation in non-robust 2-vector mode

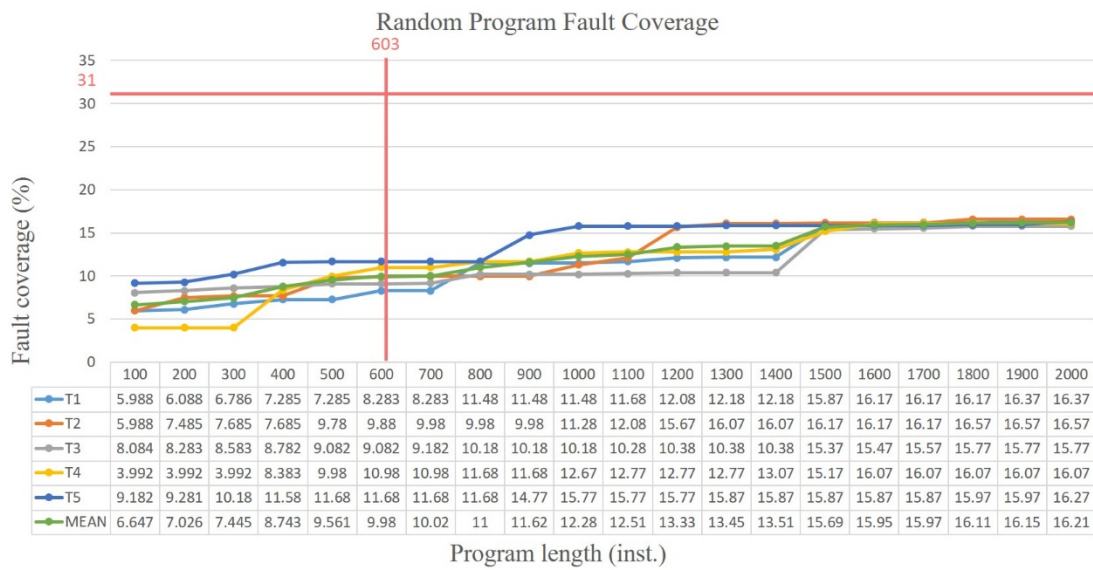


Figure 4-12 Random program evaluation in non-robust 3-vector mode

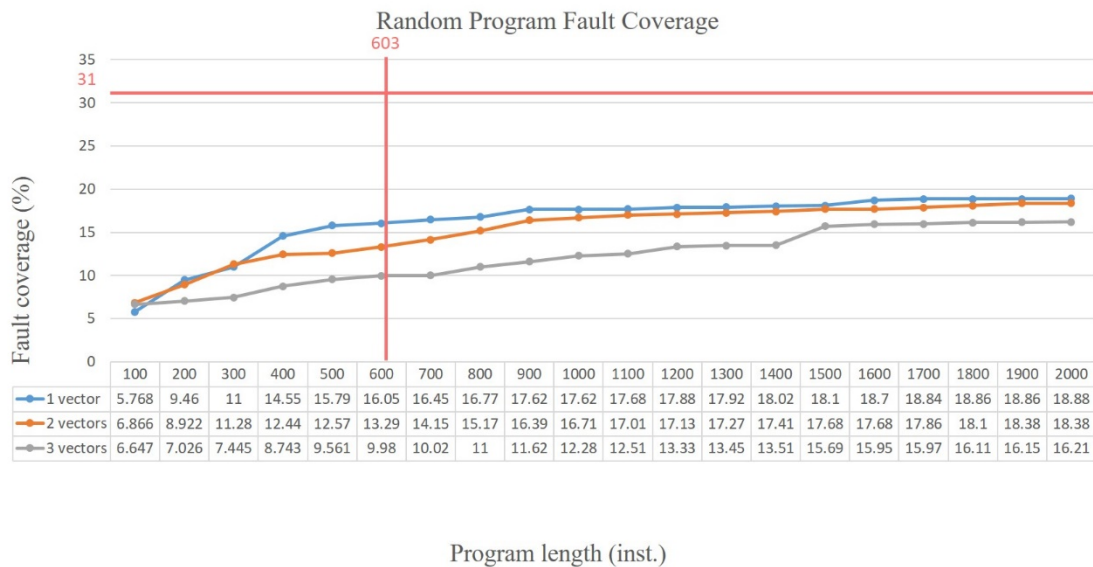
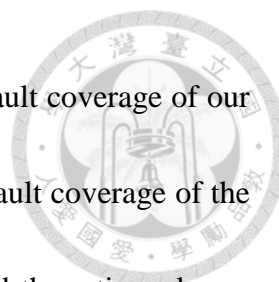


Figure 4-13 Mean fault coverage of different vectors in non-robust mode

Figure 4-10, Figure 4-11 and Figure 4-12 are the results of random program fault coverage. Figure 4-13 is the mean fault coverage of the random program with different vectors. The marking line in these figures is the fault coverage of the test program



generated by our methodology. According to the marking line, the fault coverage of our test program could reach 31% with 603 instructions. However, the fault coverage of the random programs could locate in between 15% to 20% with around three times larger than the program size of our test program.

Based on the results of fault coverage of random programs, we could say that our test program is more efficient and effective. The fault coverage of our test program could be two to eight times higher than the fault coverage of random programs in the same instruction length. Besides, the fault coverage threshold of the random test program is around half of our test program.

Compare the results of fault coverage in robust and non-robust mode, they have the similar tendency. The random program with less successive vectors might reach the fault coverage threshold earlier than the program with more successive vectors. The reason of this phenomenon would be discussed afterwards.

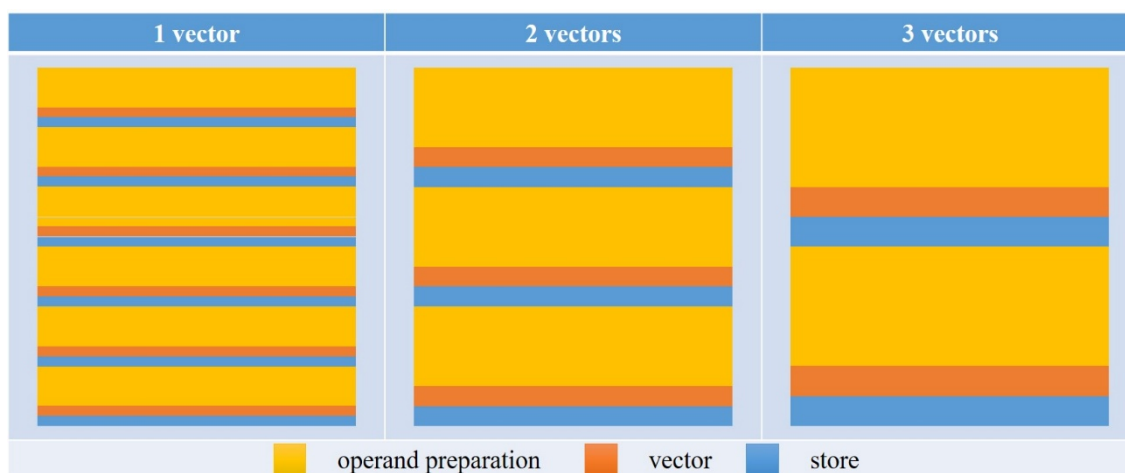


Figure 4-14 Comparison between different successive vectors

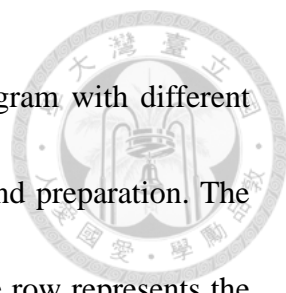
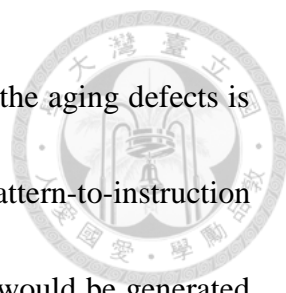


Figure 4-14 illustrates the distribution of the random test program with different successive vectors. The yellow row represents the process of operand preparation. The orange row represents the process of test vector execution. The blue row represents the process of result store. As we seen, in the same program length, the program with less successive vectors might execute the test vector and result store more times. That is to say, it might do the path activation and fault effect capture more times. As a result, is could reach the fault coverage threshold earlier than the program with more successive vectors. However, although the program with less successive vectors could reach the fault coverage threshold earlier, it might have the similar threshold than the other programs. To sum up, the number of successive vectors could only affect the time to reach the fault coverage threshold, it could not have any influence on the threshold.



Chapter 5

Conclusion



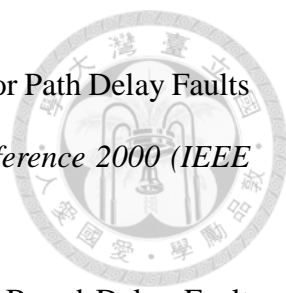
In this thesis, a software-based self-test methodology targeting the aging defects is proposed. The methodology includes ATPG-aid test generation, pattern-to-instruction converter, testbench generator and fault simulator. The test patterns would be generated by the ATPG tool, then the pattern-to-instruction converter would convert the patterns into instructions and synthesize the test program. The testbench generator might generate the testbench for doing fault simulation. It provides three types of path activation monitoring: non-robust, robust and robust*. Finally, in order to evaluate the quality of our test program, we adopt the method of random program evaluation. In this process, we compare the results of fault coverage of the random program with multiple successive vectors in robust and non-robust mode.

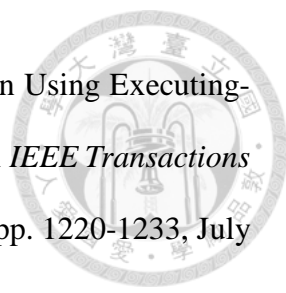
The future work includes: functionally path classification and automatically constraint extraction. The process of functionally path classification is to remove the nonfunctional paths previously. It could not only avoid over-testing nonfunctional paths but also increase the fault coverage. As to the automatically constraint extraction, since the processors become more and more complex, manually constraint extraction would become more difficult and insufficient. The importance of automatically constraint extraction would not be overemphasized.

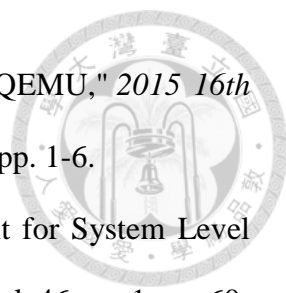
REFERENCE



- [1] T. H. Li. (2017). *A Flexible Hybrid Fault Simulator for Software-Based Self-Test* (Unpublished master's thesis). National Taiwan University, Taipei, Taiwan.
- [2] L. T. Wang, Charles E. Stroud, Nur A. Toubia, *System-on-Chip Test Architectures: Nanometer Design for Testability*. United States: Morgan Kaufmann, 2008, ch.11.
- [3] P. C. Maxwell, V. Johansen and I. Chiang, "Functional and Scan Tests: The Effectiveness of I_{DDQ} How Many Fault Coverages Do We Need?," *Proceedings International Test Conference 1992*, Baltimore, MD, 1992, pp. 168-177.
- [4] D. Gizopoulos *et al.*, "Systematic Software-Based Self-Test for Pipelined Processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1441-1453, Nov. 2008.
- [5] A. Krstic, W. C. Lai, K. T. Cheng, L. Chen and S. Dey, "Embedded Software-Based Self-Test for Programmable Core-Based Designs," in *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 18-27, Jul/Aug 2002.
- [6] L. Chen, S. Dey, P. Sanchez, K. Sekar and Y. Chen, "Embedded Hardware and Software Self-Testing Methodologies for Processor Cores," *Proceedings 37th Design Automation Conference*, 2000, pp. 625-630.
- [7] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis and Y. Zorian, "Deterministic Software-Based Self-Testing of Embedded Processor Cores," *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, Munich, 2001, pp. 92-96.
- [8] S. Almukhaizim, P. Petrov and A. Orailoglu, "Faults in Processor Control Subsystems: Testing Correctness and Performance Faults in the Data Prefetching Unit," *Proceedings 10th Asian Test Symposium*, Kyoto, 2001, pp. 319-324.

- 
- [9] W. C. Lai, A. Krstic and K. T. Cheng, "Test Program Synthesis for Path Delay Faults in Microprocessor Cores," *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, Atlantic City, NJ, 2000, pp. 1080-1089.
- [10] V. Singh, M. Inoue, K. K. Saluja and H. Fujiwara, "Software-Based Delay Fault Testing of Processor Cores," *2003 Test Symposium*, 2003, pp. 68-71.
- [11] C. H. P. Wen, L. C. Wang, K. T. Cheng, K. Yang, W. T. Liu and J. J. Chen, "On a Software-Based Self-Test Methodology and Its Application," *23rd IEEE VLSI Test Symposium (VTS'05)*, 2005, pp. 107-113.
- [12] S. Gurusurthy, R. Vemu, J. A. Abraham and D. G. Saab, "Automatic Generation of Instructions to Robustly Test Delay Defects in Processors," *12th IEEE European Test Symposium (ETS'07)*, Freiburg, 2007, pp. 173-178.
- [13] P. Bernardi, M. Grosso, E. Sanchez and M. Sonza Reorda, "On the Automatic Generation of Test Programs for Path-Delay Faults in Microprocessor Cores," *12th IEEE European Test Symposium (ETS'07)*, Freiburg, 2007, pp. 179-184.
- [14] K. Christou, M. K. Michael, P. Bernardi, M. Grosso, E. Sanchez and M. S. Reorda, "A Novel SBST Generation Technique for Path-Delay Faults in Microprocessors Exploiting Gate- and RT-Level Descriptions," *26th IEEE VLSI Test Symposium (vts 2008)*, San Diego, CA, 2008, pp. 389-394.
- [15] F. Corno, E. Sanchez, M. S. Reorda and G. Squillero, "Automatic Test Program Generation: A Case Study," in *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 102-109, Mar-Apr 2004.
- [16] D. Sabena, M. S. Reorda and L. Sterpone, "On the Automatic Generation of Optimized Software-Based Self-Test Programs for VLIW Processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 813-823, April 2014.

- 
- [17] Y. Zhang, H. Li and X. Li, "Automatic Test Program Generation Using Executing-Trace-Based Constraint Extraction for Embedded Processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 7, pp. 1220-1233, July 2013.
- [18] J. Arlat *et al.*, "Fault Injection for Dependability Validation: A Methodology and Some Applications," in *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166-182, Feb 1990.
- [19] U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation," *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, Chicago, IL, USA, 1989, pp. 340-347.
- [20] P. Kenterlis, N. Kranitis, A. Paschalis, D. Gizopoulos and M. Psarakis, "A Low-Cost SEU Fault Emulation Platform for SRAM-Based FPGAs," *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, Lake Como, 2006, pp. 7-13.
- [21] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda and M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation," in *IEEE Transactions on Nuclear Science*, vol. 48, no. 6, pp. 2210-2216, Dec 2001.
- [22] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk and R. K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, Chicago, IL, 2000, pp. 91-100.
- [23] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal and J. J. Beahan, "A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance," *2001 International Conference on Dependable Systems and Networks*, Goteborg, Sweden, 2001, pp. 501-506.

- 
- [24] D. Ferraretto and G. Pravadelli, "Efficient Fault Injection in QEMU," *2015 16th Latin-American Test Symposium (LATS)*, Puerto Vallarta, 2015, pp. 1-6.
- [25] K. K. Goswami, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," in *IEEE Transactions on Computers*, vol. 46, no. 1, pp. 60-74, Jan 1997.
- [26] P. Bernardi, M. Grosso, E. Sanchez and M. S. Reorda, "A Deterministic Methodology for Identifying Functionally Untestable Path-Delay Faults in Microprocessor Cores," *2008 Ninth International Workshop on Microprocessor Test and Verification*, Austin, TX, 2008, pp. 103-108.
- [27] D. Gizopoulos *et al.*, "Systematic Software-Based Self-Test for Pipelined Processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1441-1453, Nov. 2008.
- [28] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. S. Reorda and B. Becker, "An Effective Approach to Automatic Functional Processor Test Generation for Small-Delay Faults," *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2014, pp. 1-6.
- [29] A. U. R. Shaheen, F. A. Hussin, N. H. Hamid and N. B. Z. Ali, "Automatic Generation of Test Instructions for Path Delay Faults Based-On Stuck-At Fault in Processor Cores Using Assignment Decision Diagram," *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, Kuala Lumpur, 2014, pp. 1-5.
- [30] N. Hage, R. Gulve, M. Fujita and V. Singh, "On Testing of Superscalar Processors in Functional Mode for Delay Faults," *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, Hyderabad, 2017, pp. 397-402.

- [31] G. Ayers, A 32-bit MIPS processor which aims for conformance to the MIPS32 Release 1 ISA. (Old University of Utah XUM archive), 2014, from https://github.com/grantae/mips32r1_xum

