TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Adeolu Samuel Osidibo 172624IASM

TEST DATA GENERATION FOR SOFTWARE-BASED SELF TEST OF MICROPROCESSORS

Master's thesis

Supervisor: Raimund-Johannes Ubar
Professor
Co. Supervisor: Adeboye Stephen Oyeniran
Early Stage Researcher

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been cited. This thesis has not been presented for examination anywhere else.

Author: Adeolu Samuel Osidibo

07.01.20

Abstract

Test program generation has been a dominant challenge of software based self-test in a microprocessor. It was previously generated manually, and this process inflates the cost on test and reduces the fault coverage efficiency. However, a novel approach to automate some of the test processes was proposed in order to expedite a faster delivery of well tested devices to the market, minimize the cost for testing and obtaining the topmost fault coverage. The approach was to generate and organize a test program for a Microprocessor using HLDD [9].

Generating a test program for a microprocessor (MP) requires a test data and selecting a test data is as important as the test program itself [9]. In [1], it was stated that the test data plays a very important role in determining the quality of a test. From previous works, it has been proven that in using HLDD concept, control faults can be detected using conformity test and the data path fault can be detected using a scanning test [1][2]. The HLDD consists of the terminal and non-terminal nodes. The terminal nodes serves as the operations for processing data while the non-terminal nodes represent the control variables given in the MP. Basically, it has been previously proven that control test can be used to detect control faults and pseudo-exhaustive test can be used to exhaustively test the data processing operations for faults.

We propose a new approach for testing for fault coverage in a MP using random patterns, and a combination of random and control test, and random and pseudo-exhaustive test to detect faults in specific modules of a given microprocessor (miniMIPS). In the thesis, a lot of different scenarios of combining different test data for exercising control and data parts of microprocessor modules with the goal of trading off different quality measures like test length (memory space needed for storing test information), test quality (high- and low-level fault coverage), and testing time (by running test programs in a simulation environment).

We demonstrated with experiments which of the methods or combination of methods is more efficient in offering a high fault coverage that will eventually assure the performance and safety of MPs post manufacturing. A low-level fault simulator was used to calculate the fault coverage obtained from our experiments.

Annotatsioon

Testandmete genereerimine mikroprotsessorite isetestimiseks

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 90 leheküljel, 6 peatükki, 30 joonist, 17 tabelit.

Testprogrammide genereerimine mikroprotsessorite enesetestimiseks on tõsineväljakutse protsessorite usaldusväärse töö tagamiseks. Mikroprotsessorite testprogramme koostatakse käsitsi, mis aga muudab selle töö kalliks, ega suuda garanteerida ka piisavat kvaliteeti. Üheks perspektiivseks lähenemissuunaks testprogrammide sünteesi automatiseerimisel on kõrgtaseme otsustusdiagrammide (HLDD) kasutamine [9].

Testprogrammide genereerimine põhineb testandmete kasutamisel, kusjuures andmete valikust oleneb oluliselt testimise kvaliteet. HLDD formalismi abil saab eristada kahte kontseptsiooni: mikroprotsessori juhtosa jaoks kasutada nn. konformseid (conformity) teste ning andmeosa jaoks nn. skaneerimisteste (scanning tests) [1-2]. HLDD graafid koosnevad terminaal- ja mitteterminaaltippudest. Terminaaltippude abil modelleeritakse andmetöötlusoperatsioone ja mitteterminaaltippude abil juhtsignaale. Vastavalt võib jaotada ka rikkeid mikroprotsessorites – juhtseadmete ja andmetöötlusseadmete riketeks.

Töös esitatakse uudne lähenemisviis automatiseeritud mikroprotsessorite testprogrammide sünteesiks, mis põhineb kolme tüüpi testandmete kasutamisel: testandmed juhtosa testiks ja andmeosa testiks, ning stohhastilised andmed. Töös on välja pakutud ning analüüsitud terve rida erinevaid stsenaariumeid testandmete kombineerimisel, mille eesmärgiks oli leida kompromisse kogutesti pikkuse (testide salvestamiseks vajaliku mälumahu), testimise kvaliteedi ja testimise aja vahel.

Töös on läbi viidud põhjalikud eksperimendid erinevate testprogrammide struktuuridega, mis võimaldas analüüsida ja kindlaks teha parimad lahendused, tagamaks mikroprotsessorite testimisel kõrget rikete katet ja suuremat usaldusväärsust mikroprotsessorite töös. Testprogrammide kvaliteedi määramiseks sai kasutatud Euroopa mikroelektroonika tipptööstusest pärit professionaalset rikete simulaatorit.

Acknowledgments

Firstly, I give glory to God, my loving father in heaven. His love, empowerment and guidance encouraged me to complete this thesis.

Secondly, I will like to thank my supervisor, Prof. Raimund-Johannes Ubar, for his supervision, advice and support throughout the course of this research. It was a pleasure working with you. I appreciate my co-supervisor Adeboye Oyeniran Stephen, for his technical explanations, motivation and the personal sacrifices he made to make sure I am on track with this Thesis. I am very grateful, and your efforts remains in my heart always.

To my lovely girlfriend, Oluwabunmi Temitayo Awe, I am blessed to have you. Your support, motivation, sacrifice and help have a permanent stamp in me. Thank you for the long nights of studying together for moral support, your patience, and contribution to this thesis. We did it baby!

My deep and sincere appreciation goes to friends, who have become family – Gbenga Niyi-Leigh, Adeniyi Adekoya, and Oluwajoba Adekoya. Your calls, your support, your sacrifice and counsel, and your immense contributions to the completion of this project is immensely recognized. You guys are the best!

Oluwatoorera Kayode-Isola, my amazing friend who is far but near. I recognize your love, prayers and encouragement, thank you so much. It is a wrap now.

I must say a very big thank you to my friends and family who prayed for me, checked on me and continually encouraged me. George Ayankojo, Bolaji and Ebunmide Ladokun, and Oluwaseyi Dada, I appreciate you guys a lot.

God bless you all

List of abbreviations and terms

ATG	Automated Test Generation
ATPG	Automated Test Pattern Generation
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CUT	Circuit Under Test
DUT	Design Under Test
FC	Fault Coverage
HLDD	High Level Decision Diagram
ILA	Iterative Logic Arrays
ISA	Instruction Set Architecture
MP	Microprocessor
MSF	Multiple Stuck-At Fault
NOC	Network-On-Chip
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROBDD	Reduced Ordered Binary Decision Diagram
ROM	Read-Only Memory

RTL	Register Transfer Level
S-A-0	Stuck-At-0
S-A-1	Stuck-At-1
SBST	Software Based Self-Test
SOC	System-on-Chip
VHDL	VHSIC Hardware Description Language

Table of contents

Abstract ... 3

Acknowledgments ... 5

1 Introduction ... 11

1.1 Background and problem ... 12

1.2 Objectives ... 13

1.3 Organization of thesis ... 14

2 Digital systems ... 15

2.1 Development life cycle of digital systems ... 15

2.2 Testing in digital systems ... 16

2.2.1 Defects, Faults, Error and Failure ... 17

2.2.2 Levels of Abstraction in Digital System Testing ... 17

2.3 Fault modelling ... 18

2.3.1 Stuck-At-faults Model ... 19

2.3.2 Conditional fault model ... 20

2.3.3 Open and Short Faults ... 20

2.3.4 Transistor Faults ... 21

2.4 HLDD based fault models ... 21

2.4.1 Decision diagrams ... 21

2.4.2 Structural Synthesized Binary Decision diagram (SSBDD) ... 21

2.4.3 High level Decision Diagram (HLLD) ... 22

2.4.4 HLDD Based Fault Models ... 23

2.5 Low level and high-level fault models ... 24

2.5.1 Behavioural Bit Stuck-At Fault Models ... 24

2.5.2 Branch and Condition Stuck-At Faults ... 24

3 Software-based self-test ... 25

3.1 Development of SBST ... 25

3.1.1 MiniMIPS ISA ... 26

3.1.2 HLDD Synthesis ... 28

3.1.3 Test Synthesis from HLDD ... 31

3.2 Test program generation with HLDD ... 32

3.2.1 High Level Test Data Generation ... 33

3.3 Test program generation 46
3.4 Fault simulation 47
3.5 Conclusions 47
4 Development and investigations of the methods 47
4.1 Test templates 48
4.2 Set-up of the Experiments 52
4.3 Combination of different methods 53
5 Implementation and investigations 56
5.1 Goals of the experiments 57
5.2 Investigations #1 58
5.3 Investigations #2 60
5.4 Investigations #3 61
5.5 Investigations #4 62
5.6 Investigations #5 62
5.7 Investigations #6 63
5.8 Investigations #7 64
6 Conclusion 67
References 68
Appendix 1 – Program Description and Manual 71
Appendix 2 – Structure of the miniMIPS processor 75
Appendix 3 – CPU specification for the experiments 76
Appendix 4 – Source Code 77
A Pseudo_template.py 77
B Random_Template.py 81
C Random_data_generator.py 85
D Parameter.txt 86
E TestProgramGenerator.py 89

List of figures

Figure 1: Testing Process of a digital circuit under test 16
Figure 2: Levels of abstraction of digital systems [11] 18
Figure 3 AND gate with two inputs 19
Figure 4: SA1 in an AND gate 19
Figure 5: Representing an RTL data path with HLDD [4] 22
Figure 6: SBST generation framework 26
Figure 7: Structural representation of miniMIPS registers [36] 27
Figure 8: Types of instruction formats 28
Figure 9: AND instruction architecture 28
Figure 10: MiniMIPS HLDD model using 4 instruction sets 30
Figure 11: Synthesis of HLDD for miniMIPS [9] 31
Figure 12: Example of test generation 32
Figure 13: High-Level test data and test program generation 33
Figure 14: Mapping of miniMIPS instruction formats and the HLDD functional variable 38
Figure 15: Algorithm for conformity test 38
Figure 16: Structure of Conformity test 39
Figure 17: Structure of Scanning Test 43
Figure 18: Algorithm for scanning test 44
Figure 19: PET combination to All PET Test Patterns 45
Figure 20: Test Program Generation process with four templates 46
Figure 21: HLDD Structure for method 2, 3, and 4 49
Figure 22: ADD structure in miniMIPS ISA 49
Figure 23: Set-up of the experiments 53
Figure 24: Experiment structure for experiments 1 - 7 54
Figure 25: Experiment structure for experiments 8 – 11 54
Figure 26: Structure of miniMIPS execute module 55
Figure 27: Combination of different methods 56
Figure 28: Test program generator response from Linux terminal 72
Figure 29: Generating dump file for fault coverage calculation 73
Figure 30: Structure of the miniMIPS processor 75

List of tables

Table 1: Truth table for an AND gate (no faults) 19
Table 2: Expansion of the miniMIPS ISA [9] 29
Table 3: Generation of PET data for adder [25] 41
Table 4: Generation of PET data for Subtractor [25] 41
Table 5: List of instructions under template 1 50
Table 6: List of instructions under template 2 51
Table 7: List of instructions under template 3 51
Table 8: List of instructions under template 4 52
Table 9: Results of Experiment 1 - 4 58
Table 10: Number of faults in MUTs of miniMIPS processor 59
Table 11: Test length and simulation time for experiments 1 - 4 59
Table 12: Comparison of method 2 and 3 60
Table 13: Comparison of experiments 1, 6 and 9 61
Table 14: Comparison of experiment 2, 5 and 8 62
Table 15: Comparison of experiment 4 and 7 62
Table 16: Comparison of experiments 8, 9, 10 and 11 63
Table 17: Observation of experiment 3, 8 and 9 64
Table 18: Observation of experiments 6 and 7 65
Table 19: Comparison for the significance of Random data 65

1 Introduction

This thesis focuses on a new approach of testing microprocessors with software based self-test by combining different test data generation methods to detect faults in microprocessor modules. A novel concept of HLDD synthesis was used to generate the test program while implementing the combination of test data.

This chapter discusses the background and problem, subsequently, the goal of the thesis, and lastly, the overall structure of this work.

1.1 Background and problem

The increase in technological advances has enabled much more complex digital systems (DS) to be built. Massive parallel computing and new design paradigms like System-on-Chip (SoC) and Network-on-Chip (NoC) now exists and needed in-depth research to develop new algorithms and design and test methods, based on microprocessors. As more complex digital systems are being developed, it becomes more evident that Moore's law is continually being proven. Moore's law emphasize that the number of transistors on integrated circuits doubles every 18 months [5] [6]. Fragility of transistors becomes more rampant as the microprocessors undergo rigorous manufacturing processes. Defects becomes inevitable in the transistors of MPs, which could lead to faults in the MPs and could bear severe consequences, especially when the complex system is a critical system. The failure of such system could cause loss of sensitive data, lives and properties. The severity of a fault in the transistors on a MP helps to emphasize the importance of testing to guarantee and improve the reliability of any MP during the operational stage.

In the recent decade, the semiconductor industry was challenged to develop novel testing methods that can be integrated in MP test flow. Without a humongous budget, the testing methods to be developed are targeted at high quality product development. A test method that suits the description was first proposed in 1980 [3], and it is called Software-Based Self-Test (SBST).

For the main purpose of testing the processor, the operational approach of SBST is to execute the test program on processor itself and its surrounding resources [4]. As mentioned earlier, this method eradicates the need for external hardware, which may be expensive, and the time of the test is limited with the performance of the processor. The

main subject in the SBST methodology is the test program generation, which must comply with the high-quality fault coverage standards imposed by the industry [4].

Self-test programs for microprocessors have emerged from been written manually, as a novel formal approach for modelling the high-level functionality and possible faulty behaviours was developed; High-Level Decision Diagram (HLDD). HLDDs can be considered as a generalization of logic level Binary Decision Diagrams (BDD) [4].

From previous works, it has been proven that in using HLDD concept, control faults can be detected using conformity test and the data path fault can be detected using a scanning test [1]-[2]. The HLDD consists of the terminal and non-terminal nodes. The terminal nodes serve as the data path while the non-terminal nodes represent the control variables given in the MP. Using HLDD, control test can be used to detect control faults and pseudo-exhaustive test can be used to exhaustively test the data processing operations for faults. However, as varieties of approaches of MP tests spikes up the interest in this topic of academia and industry as well, a combination of approaches may contribute immensely to the effectiveness of MP testing and improve testability.

1.2 Objectives

The goal of the thesis is to develop different methods for the combinations of test data for microprocessor software based self-testing. The Execute module of the MIPS microprocessor was partitioned into three sub- modules: ALU (arithmetic and logic operations), MULT1 and MULT2 (multiplication operations).

The section above identified that previous works have been done to improve SBST, develop test program for SBST, and develop different approaches in testing the control part and the data path of a microprocessor. The approaches applied in testing the control part and data path of a MP fulfils the constraints for test data generation. To further deduce the possibilities of these approaches, this thesis presents the following goals:

- Develop different combinations of test data for microprocessor software base self-testing.
- Develop test templates that enables the test program generator to handle various combination of test data.

- Carrying out simulation experiments through the developed methods to evaluate the quality (SAF coverage) of four basic test algorithms separately and evaluate the possible contribution of each test approach.

1.3 Organization of thesis

The thesis is organized as follows:

In chapter 2, an overview of the digital system is surveyed. The area to testing digital systems and various fault models were discussed, alongside the concept of high-level decision diagram (HLDD).

Chapter 3 covers the overview of software based self-test and its development. An in-depth view of HLDD was covered, including how it is used to generate test programs for the miniMIPS processor. The chapter also contains the test data generation stage, preceding the test program generation. Chapter 4 entails the development of our proposed methods in stages and the approach we applied in order to combine the 4 methods. We implemented the proposed methods by performing various experiments and analysed the results.

Lastly, the summary and conclusion of the thesis is presented in chapter 6.

2 Digital systems

In today's world the word digital is more common than sliced bread, so are its techniques are widely known and utilized in all sectors of life. According to [27] which described digital systems as a combination of devices designed and manipulate logical information or physical quantities that are represented in digital form that is the quantities can take on only discrete values. Arguably digital systems applications in the world of electronics, as well as other major technologies, have performed better than any other systems in any other era.

2.1 Development life cycle of digital systems

Every complex system goes through a development life cycle, i.e. the detailed plan for how to develop, alter, maintain, and replace a system to produce a system with the highest quality and lowest cost in the shortest time.

Digital system undergoes 3 stages which includes

- Design
- Production
- Operation

These stages are set up to mitigate the possible misconceptions that may or may not occur, every digital systems development is prone to human and system error at every stage, Each stage has sublevels with the Design stage consisting of specification, implementation, realization for Production possesses pilot and full while the Operation has the Installation and Maintenance sub levels [28]. With a wide ranch of possible errors manifesting any possibly every stage of development it is mandatory to undergo reviews and checks with every component and stages associated with the development life cycle of digital systems. Each stage undergoes reviews, as for the Operation stage requires possible repairs for faults, dividing the systems into level, with the considerations of various factors responsible for the faults encountered. System Designs need Verifications in a bid to check the correctness for each step employed, production stage also undergoes

testing which is where our focus will be centred upon in the next couple of headings in this chapter.

2.2 Testing in digital systems

Testing is a concept widely practiced in all sectors of life. Although the approaches may differ slightly, the fundamental achievement remains absolute and is made manifest in the process which is an endeavour in determining the overall correctness of a system with little to no doubt by exposing it to the hardest levels of scrutiny it can possibly handle.

This philosophy also rings through in the aspects of digital systems, which is often referred to as a black block experiment at every level of development to determine correct functionality with the application of stimuli at the input and observing the response on the output [28].

The investigation of the output includes the comparison of its expected reaction with the yield presented amid the introduction of stimuli, this process is known as circuit under test (CUT) as we will see in the figure below elaborating the testing process of a digital circuit under test.

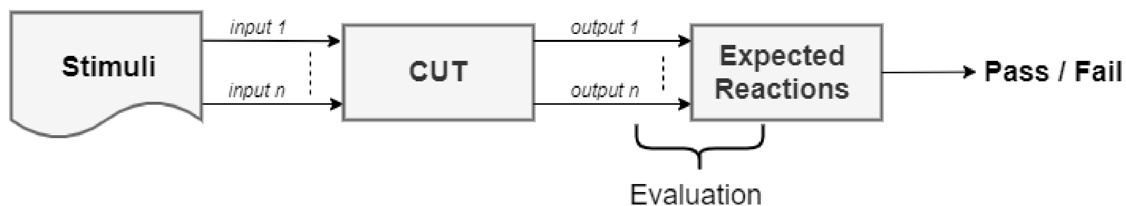


Figure 1: Testing Process of a digital circuit under test

Digital systems become more complex over the years as a result of technological advancements, components become less testable which makes development complex, with this growing complexity and technological advancement in digital systems, problems tend to arise during testing phases since testing is required to encompass multiple activities in the life of a system. These possible occurrences will be discussed extensively in the next topic.

2.2.1 Defects, Faults, Error and Failure

Common terms that are familiar with researchers who undergo testing with digital systems include errors, faults defects and flat out failures. These amongst other phrases are the possible red flags expressing incorrectness of a system.

A defect in an electronic system is the unintended difference between the implemented hardware and its intended design [33].

An error is the manifestation of a fault or multiple faults expressing the deviation from the appropriate behaviour in a system.

Failure indicates a fatal issue in a system or in its module which is making the system inoperative or unresponsive.

A fault depicts the presence of defects which could either reflect a temporary or irreversible change in hardware [15]. It could either be structural or physical forms.

There is a bit of a comparison and contrast when we discuss errors and faults in a system, the appearance of an error automatically implies the presence of fault or some faults, however, faults do not necessarily cause error history has proven systems to work well for year even with the possibility of faults proven from stress test which have a slim chance of occurring in real life. It is important to detect faults that can lead to errors in systems so that they can be mitigated, guaranteeing systems functionality at optimum capacity for a long period of time.

2.2.2 Levels of Abstraction in Digital System Testing

This expresses the physical borders of digital systems, also known as series of abstraction, is the levels from the topmost to the bottom with which the digital system is designed. This principle is adopted to manage complexity and promote order when developing a system from the conceptual state even to the highest level.

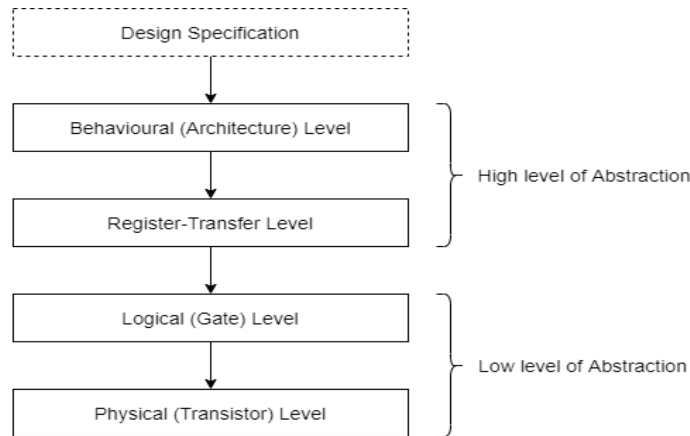


Figure 2: Levels of abstraction of digital systems [11]

The series are grouped together into two levels as we can see in the above figure, there is a low level of abstraction and the high level. The logic and the physical level accounts for the low level, the high level of abstraction is accounted for in the RTL and behavioural levels [29].

2.3 Fault modelling

In the area of test generation and fault simulation, integrals part of digital design, the diversities facing both centralize focus is enormous in respect to fault detection despite the similarities. Fault models are essential to the test generation and evaluation so much so that a wide range of fault models exist in determining the nature and behavioural defects in digital circuits.

The outcomes of test generation and fault simulations is highly predicated on the fault models, which usually faces a back and forth between cost and quality of test, sadly this is not enough to guarantee an accuracy in detecting faults in accordingly, a blend of various deficiency models at numerous cases are utilized in the age and assessment of test vectors.

A few faults have illustrated that numerous recognize test designs with high coverage give a high demonstrative resolution as well as can help boost the inclusion between nodes. This methodology makes the ATPG procedure increasingly troublesome and CPU-concentrated, yet it is quite simple to apply and doesn't require any adjustment in the test-pattern-generation flow.

2.3.1 Stuck-At-faults Model

Originally stuck-at fault is the widely proposed test for the logic circuit without the application of inputs because the fault suggests that faults will inevitably present itself when logic variables are stuck at 1s or 0s, relying on a percentile outcome in every sequence.

Consider an AND gate consisting of two inputs (A and B) and an output C

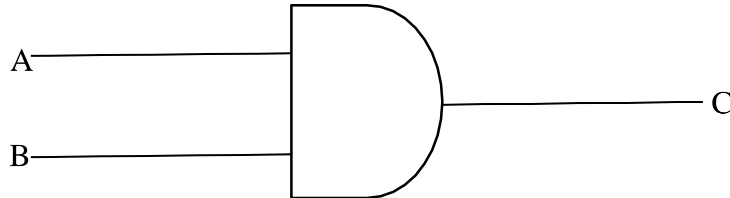


Figure 3 AND gate with two inputs

Table 1: Truth table for an AND gate (no faults)

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Let us assume that there is an SA1 at input A, if the logic value at A is 0 or 1, the logic value will remain as 1. Normally, if A=0 and B=1 then the output C=0 but due to the SA1 at input A then output C will be always be 1.

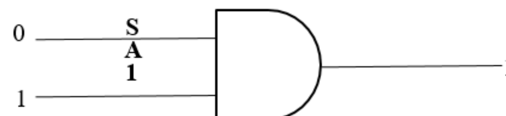


Figure 4: SA1 in an AND gate

Stuck-at fault has proved to be effective and efficient with a technique modelled with electromagnetic relays coils that become automatically stuck at the appearance of a fault [8]. However, its complexity becomes detrimental when handling test generation especially for a large number of faults present in a system, exposing its inaccuracy (over reliance on the percentage of sequence) in modern-day nanoelectronics technology.

2.3.2 Conditional fault model

This is a functional fault model that includes functional verification for every circuit either with partial or complete design level. This is also represented as an input pattern fault model possesses similar attributes to Stuck-at fault (SAF) model nevertheless, its level of accuracy proves valuable in applications to diminish complexity in test generation in modern-day nanotechnology and microprocessor [4].

Definition 1: A fault (l_i /α, $l_j = \beta$), where l_i and l_j are two lines in a circuit and $\alpha, \beta \in \{0,1\}$ is a conditional stuck-at (CSA) fault if l_i /α refers to the fault l_i stuck-at α and $l_j = \beta$ refers to the requirement that some test vector for the stuck fault l_i /α produces the value β on line l_j. This test vector is then said to detect the CSA fault ($l_i /\alpha\ l_j = \beta$).

The definition according to [10] includes the null condition possibility corresponding to a normal stuck fault, where ($l_i /\alpha\ l_j = \beta$) is simply (l_i /α) and no l_j or β is specified. This type of CSA faults is going to be called null condition CSA faults. The expression "completely specified CSA fault" will be used whenever it is necessary to emphasize the fact that both the condition line and the condition value have to be specified, as opposed to the null condition CSA faults.

In a bid to improve test generation and fault coverage numerous fault models have been created by researchers over the years, with unique components used for uncovering in respect to faults.

2.3.3 Open and Short Faults

Short faults can also be called bridging faults. This type of fault exists in the wire that interconnects the transistors that forms the circuit [11]. Also known as interconnects faults, it occurs due to the broken connections between different points that are expected to be connected in the circuit. Correspondingly, short faults exist whenever an accidental connection occurs between nodes that are not asserted to be connected.

2.3.4 Transistor Faults

The stuck-at fault cannot precisely emulate the behaviour of fault at the transistor level because of the multiple transistors that are used to construct CMOS logic gates [11]. Due to the occurrence of switching at the transistor level, there is a probability that a transistor would be stuck open or stuck short. Both possessing the idea that a single fault can affect different combinations of fan-out branches.

2.4 HLDD based fault models

2.4.1 Decision diagrams

Decision diagrams are methods of modelling digital systems at various level of abstraction. This can be modelled at both low and high levels of abstractions. The low level which is the logic level possesses binary features which are popularly referred to as BDDs and the other which deals with the behavioural and the RTL level is known as the HLDDs.

Binary decision diagram (BDD), a system for modelling digitally has been the standard in a data structure in computer-aided design (CAD) for manipulating Boolean functions at various levels of abstraction [9]. Over a jubilee ago when it was introduced, researchers have proposed other new data structures like the Reduced ordered BDD, Ternary decision diagram (ROBDD), Edge-Valued decision diagram (EVBDD), zero suppressed (ZBDD hybrid BDDs (HBDD) and a host of others, with each possessing a level of simplicity while retaining unique qualities which made BDDs one of the most popular representations of Boolean functions.

2.4.2 Structural Synthesized Binary Decision diagram (SSBDD)

SSBDDs are unique to other binary decision diagrams because they possess the ability to map logic circuits directly from the gate level structure. This functionality allows the modelling various objectives in testing like delays on paths, fault-masking, signal paths etc. a feature all other BDD do not possess.

2.4.3 High level Decision Diagram (HLLD)

High-Level Decision Diagram (HLDD), another alternative of Decision diagram that represents digital systems from the RTL to the behavioural levels of abstraction. The data processing operation of HLDD occurs using nodes, a technique which exhibits an extension of SSBDDs methods for test generation and fault simulation [21]. It comprises of terminal and non-terminal nodes representing boolean variables from structurally synthesized BDDs as boolean vectors or high level algebraic operations possessing not only the ability to describe the structure of a system usually synonymous to logic level circuits but also the working behaviour of the system thereby extending to the high level functions of the digital system.

Figure 5 expresses the functionality as well as the structural components of a circuit represented by an RTL data path using an HLDD. As you can see the data path circuit enumerates R1 and R2 registers with non-terminal nodes, internal nodes y1-y4 with intermediaries between the control unit and data with data buses.

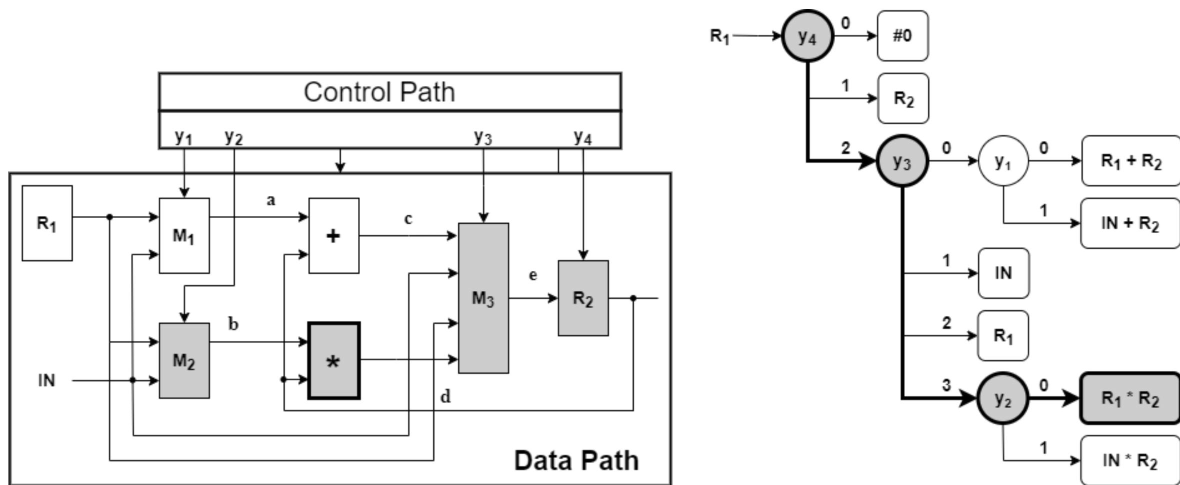


Figure 5: Representing an RTL data path with HLDD [4]

Seeing from the diagrams above the procedure for calculating the register variable R_2 assumed IN R_1, R_2 from left to right. Using HLDD, R_2 next state (direction) is calculated in each non-terminal node y_k of G_{R2}, predicated on the value of the R_2 using the expression shown by each terminal node, For instance, if $y_4 = 4$, $y_3 = 5$ and $y_2 = 3$, $R_2 = R_1 * R_2$ is activated, then the update is stated in R_2.

The HLDD nodes account for the structural components of the circuits as does the topology covers the behavioural aspect.

2.4.4 **HLDD Based Fault Models** [2]

In the definition of HLDD according to [2], it is a graphical representation of a given discrete function *F(Z)* and it is a directed acyclic graph that can be defined as a quadruple $Gz = (M, \Gamma, Z, F)$ with a set of nodes M, a mapping Γ from m to M. In this equation,

- M is a finite set of nodes
- Γ is a finite set of edges
- Z is a function which defines the variables labelling the node
- *F* is a function on *Γ*.

M is divided into two subsets of node: non-terminal M_N and terminal M_T nodes. $\Gamma(m) \subset M$ represents the set of all successors of the node $m \in M$ and $\Gamma^{-1}(m) \subset M$ denotes the set of all predecessors of *m*. The graph has a root node m_0 with $\Gamma\text{-}1\ (m) = \emptyset$. The nonterminal nodes $m \in M_N$ are labelled by variables $z(m) \in Z$. The terminal nodes $m_k \in M_T$ are labelled by sub-functions $z(m_k) = f_k\ (Z_k), fk(Z_k) \in F$, which may be as well variables $z_k \in Z$ or constants.

For each value *e* from a set *V(z(m))*, there is an existence of a corresponding output edge (m, m^e) from the node m into the successor node $m \in \Gamma(m), e \in V(z(m))$.

Z^t a vector of values assigned to Z at a time t. The edge (m, m^e), where $e \in V(z(m))$, is activated by Z^t if $z(m) = e$. A given path $l(Z^t) = (m, n)$ in the HLDD is called the activated if all edges on the path are activated. The activated by Z^t edges form a full activated path $l(Z^t)= l\ (m_0, m_k)$ which determines the value of the graph variable $f_k(Z_k)$ from the root node m_0 to one of the terminal node m_k.

The HLDD uses the cycle-based modelling theory for evaluating the behaviour of a digital system. The usage of this theory insinuates that the actuation of a circuit or system state at a particular cycle is possible, based on the exactness of the system behaviour modelling required.

2.5 Low level and high-level fault models

Growing complexities in digital systems have directly reduced the observability of internal components thereby narrowing down effective manipulation during testing, with this looming setback, adequate fault coverage may not be evaluated using certain models. Speed testing has become a commonly used approach to attain quality tests.

Test pattern generation in digital circuits has two critical approaches from its levels of abstraction [32]. First is identifying the appropriate model relatable to the physical fault and the other is inducing the respective models in generating patterns in identifying them. In most cases, researchers have often defined physical induced faults, which are also referred to as low-level faults which become evident on higher levels as convenient reasons why fault models from logic level such as stuck-at faults can be adaptable for fault modelling during test generation on a higher level.

In this section, we will discuss a few fault models at logic level testing which bear similarities to fault models at behavioural and R-T level giving rise to mapping low-level faults to High-level fault models.

2.5.1 Behavioural Bit Stuck-At Fault Models

It is common knowledge that Stuck -at fault models at logic level works when signals and variables is encoded in either stuck at 1 or 0 however when this low-level fault model is clearly mapped at behavioural and R-T level it becomes quite useful as well. Stuck- at fault models components at R-T level are synthesized to specific logic component thereby implementing input and outputs with that connection [32]. Although this approach can only model a subgroup of physical fault, it proves the potential physically induced fault possess even at higher levels in test generation.

2.5.2 Branch and Condition Stuck-At Faults

Branch stuck-at fault reflects a given section which behaviour is stuck at. These could be a chosen statement or a condition (if, else) statement whereby the condition is either suck at true or stuck at false.

A choice in a branch articulation might be founded on various conditions associated through consistent administrators. A condition may likewise be utilized in contingent assignments and watched practices

Generally, low-level abstraction is regarded in the physical subset of a circuit, but it really isn't the case all the time because the logic level and R-T level can be categorised as that level up abstraction as well [31] that is why this section discusses certain ways in which low-level faults can be mapped up to behavioural level after undergoing sensitization. These type of fault models are quite advantageous in delivering test vectors even to levels beyond the behavioural level of a circuit.

3 Software-based self-test

This chapter discusses the proposed formalised method used for SBST program synthesis for MPs. Using the HLDD model, the test program generation for microprocessor is in two levels: The system level and the module level of the microprocessor. Each HLDD presents the behaviour of a module, and the network of HLDDs presents the behaviour of the system as a whole. At the module level of the microprocessor, the nodes of the HLDDs are the target of test generation, while the HLDDs themselves are the targets at the system level. The HLDDs (module) tests T(*m*)that were generated locally are embedded into the system level test program template. This entails that the test stimuli for the modules will be made controllable and the results of the tests will be made observable at the system level [4].

3.1 Development of SBST

This section introduces the SBST generation framework. Figure 6 shows a generic overview of the framework. It consists of three main modules: HLDD synthesizer, test vector generator, and an SBST-generator synthesizer which converts test vectors into test programs using prepared test code templates [4]. The translation from a set of instructions into a test program is demonstrated on a 32-bit RISC MiniMIPS microprocessor [30] according to instruction set in MIPS architecture [30].

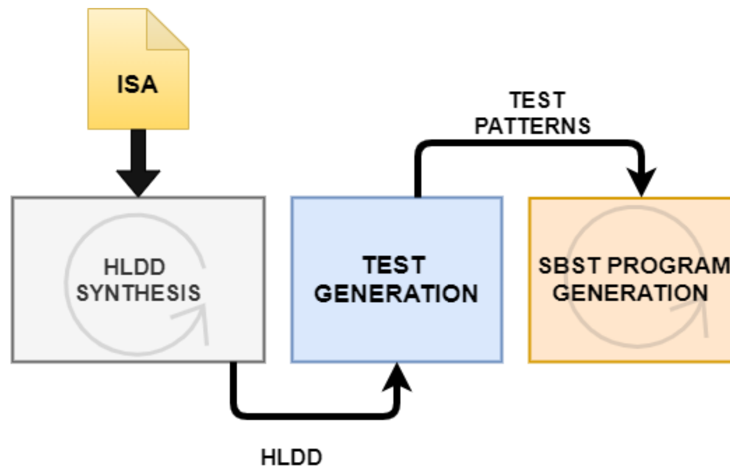


Figure 6: SBST generation framework

3.1.1 MiniMIPS ISA

An instruction set architecture, ISA is an abstract representation of a processor and its functionality provided in the architecture documentation. These includes the description of the general-purpose registers, flags, list of instructions, assembly language syntax and their binary representation [4]. These descriptions are presented in a specific format that can be transformed in High-level decision diagrams. Given this information, test programs can be created. In this section, the open-source microprocessor miniMIPS is considered. The miniMIPS has 32 registers that are 32 bits long. A structural representation is depicted in Figure 7.

Register	Assembly Reference	Usage
$0	$zero	$zero
$1	$at	Reserved for assembler use
$2 - $3	$v0 - $v1	Result storage
$4 -$7	$a0 - $a3	Argument storage
$8 - $15	$t0 - $t7	Temporary values
$16 - $23	$s0 - $s7	Saved registers for procedure calls
$24 - $25	$t8 - $t9	More temporary values
$26 - $27	$k0 - $k1	Reserved for OS
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

Figure 7: Structural representation of miniMIPS registers [36]

The instruction encoding contains different fields with specific encoding instructions that describe the function of the module. The rs, rt, rd fields hold the address of the registers where the operands and results of the functions are stored. OP1 encodes the type of instruction and OP2 encodes a type of registers for the instruction. Immediate defines the immediate value as an operand and the field address contains the address where to jump to [9]. The miniMIPS instruction format used can be categorized into three distinct types as shown in Figure 8.

Types	Bits	31 26	25 21	20 16	15 11	10 06	05 00
	R-Type	OP1	rs	rt	rd	sham	OP2
	I-Type	OP1	rs	rt	Immediate		
	J-Type	OP1	Address				

Figure 8: Types of instruction formats

Figure 9 shows an AND instruction description of the miniMIPS processor manual [30]

	Bits	31 26	25 21	20 16	15 11	10 06	05 00
AND		000000	rs	rt	rd	sham	100100
		OP1					*OP2*
ANDI		001100	rs	rt	Immediate		
		OP1					
J		000010	Address				
		OP1					

Figure 9: AND instruction architecture

3.1.2 HLDD Synthesis

High-level decision diagrams can be constructed from ISA. The HLDD can be constructed by representing the instructions given in the ISA in a structural format as shown in Table 2.

Table 2: Expansion of the miniMIPS ISA [9]

S/N	Instruction	OP1	OP2	Mnemonics	ISA Level Operation
1	ADD	000000 (0)	100000 (32)	ADD rd rs rt	rd= rs + rt
2	ADDI	001000 (8)	-	ADDI rt rs I	rt= rs + I
3	ADDIU	001001 (9)	-	ADDIU rt rs I	rt= rs + I
4	ADDU	000000 (0)	100001 (33)	ADDU rd rs rt	rd= rs + rt
5	AND	000000 (0)	100100 (36)	AND rd rs rt	rd= rs AND rt
6	ANDI	001100 (12)	-	ADDI rt rs I	rt= rs AND I
7	BEQ	000100 (4)	-	BEQ rs rt offset	If rs= rt then branch
8	BGEZ	000001 (1)	00001 (1)	BGEZ rs offset	If rs >=0 then branch
9	BGEZAL	000001 (1)	10001 (17)	BGEZAL rs offset	If rs >=0 then procedure
10	BGTZ	000111 (7)	-	BGTZ rs offset	If rs > 0 then branch
11	BLEZ	000110 (6)	-	BLEZ rs offset	If rs <=0 then branch
12	BLTZ	000001 (1)	00000 (0)	BLTZ rs offset	If rs < 0 then branch
13	BLTZAL	000001 (1)	10000 (16)	BLTZAL rs offset	If rs < 0 then procedure
14	BNE	000101 (5)	-	BNE rs offset	If rs != rt then branch
15	J	000010 (2)	-	J Target	rd= return_address
16	JALR	000000 (0)	001001 (9)	JALR rs JALR rd rs	rd =return_address
17	JR	000000 (0)	001000 (8)	JR rs	PC = rs
18	LUI	001111 (15)	-	LUI rt I	rt = I
19	LW	100011 (35)	-	LW rt offset (base)	rt = memory [base + offset]
20	MFHI	000000 (0)	010000 (16)	MFHI rd	rd= HI
21	MFLO	000000 (0)	010010 (18)	MFLO rd	rd= LO
22	MTHI	000000 (0)	010001 (17)	MTHI rs	HI = rs
23	MTLO	000000 (0)	010011 (19)	MTLO rs	LO = rs
24	MULT	000000 (0)	011000 (24)	MULT rs rt	[LO, HI] = rs X rt
25	MULTU	000000 (0)	011001 (25)	MULTU rs rt	[LO, HI] = rs X rt
26	NOR	000000 (0)	100111 (39)	NOR rd rs rt	rd= rs NOR rt
27	OR	000000 (0)	100101 (37)	OR rd rs rt	rd= rs OR rt
28	ORI	001101 (13)	-	ORI rt rs I	rt = rs OR I
29	SLL	000000 (0)	000000 (0)	SLL rd rt sa	rd = rt << sa
30	SLLV	000000 (0)	000100 (4)	SLLV rd rt rs	rd = rt << rs
31	SLT	000000 (0)	101010, (42)	SLT rd rs rt	rd = rs < rt
32	SLTI	001010 (10)	-	SLTI rt rs I	rt = rs < I
33	SLTIU	001011 (11)	-	SLTIU rt rs I	rt = rs < I
34	SLTU	000000 (0)	101011 (43)	SLTU rd rs rt	rd = rs < rt
35	SRA	000000 (0)	000011 (3)	SRA rd rt sa	rd = rt >> sa
36	SRAV	000000 (0)	000111 (7)	SRAV rd rt rs	rd = rt >> rs
37	SRL	000000 (0)	000010 (2)	SRL rd rt sa	rd = rt >> sa
38	SRLV	000000 (0)	000110 (6)	SRLV rd rt rs	rd = rt >>rs
39	SUB	000000 (0)	100010 (34)	SUB rd rs rt	rd= rs – rt
40	SUBU	000000 (0)	100011 (35)	SUBU rd rs rt	rd= rs – rt
41	SW	101011 (43)	-	SW rt offset(base)	Memory[base + offset]=rt
42	SYSCALL	000000 (0)	001100 (12)	SYSCALL	System call
43	XOR	000000 (0)	100110 (38)	XOR rd rs rt	rd= rs XOR rt
44	XORI	001110 (14)		XORI rt rs I	rt = rs XOR I
45	JAL	000011(3)	-	JAL target	rd=return_address
46	LWCO	110000	-	LWCO cs, offset(base)	cs=memory[base + offset]
47	MFCO	10000	0	MFCO rt, cs	rt = cs
48	MTCO	10000	100	MTC0 rt, cs	cs = rt

Using Table 2, an HLDD representing the behaviour of the system or the unit of the system under test can be created. From the table, it can be noted that OP1 and OP2 are control variables (which determines the path to be taken in the graph) and hence are non-terminal nodes in the HLDD. The combined states of OP1 and OP2 are however unique and will result in a terminal node defined by the instruction shown as ISA level operation.

Figure 10 shows an HLDD of 4 instructions, ADD, ADDU, ANDI, and J. The values of OP1 and OP2 have been converted to their decimal values for simplification.

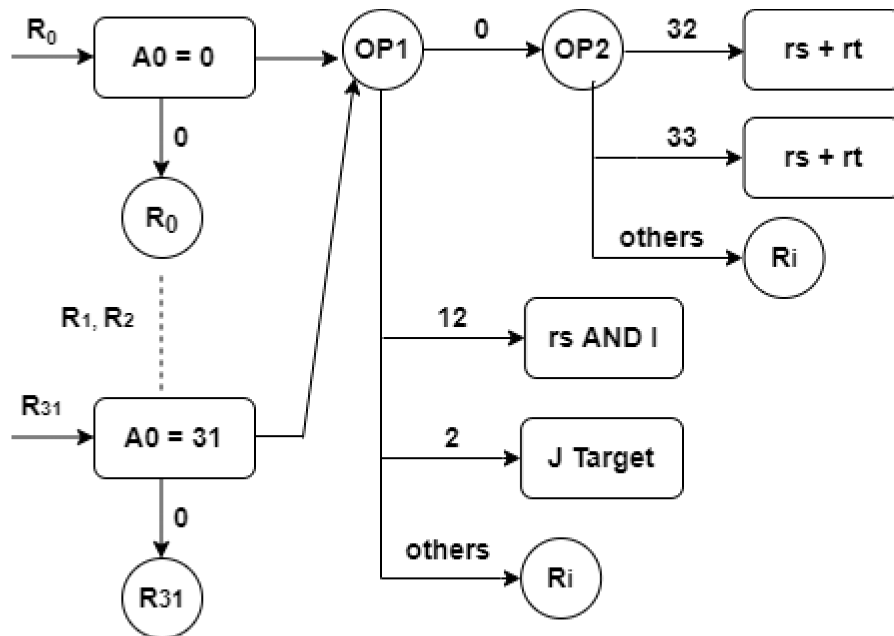


Figure 10: MiniMIPS HLDD model using 4 instruction sets

For the ADD instruction set, OP1 is 000000 and is shown in the diagram as the decimal value 0, likewise the OP2 is 100000 which is shown in the diagram as the decimal value 32. For the ADDU instruction, OP1 is 000000 and is shown in the diagram as the decimal value 0, likewise the OP2 is 100001 which is the decimal value 33. For the ANDI instruction, OP1 is 001100 shown as the decimal value 12 and for the J instruction, 000010 shown as the decimal value 2.

The system traverses to the ADD instruction when OP1 is 0 and OP2 is 32. Likewise, when OP1 is 0 and OP2 is 33 the system traverses to the ADDU terminal instruction. However, if OP1 is 12, then the system traverses to the AND instruction terminal and if OP1 is 2 then the system traverses to the J instruction terminal.

Figure 11 shows the complete HLDD representation of the miniMIPS instruction sets using the same concept.

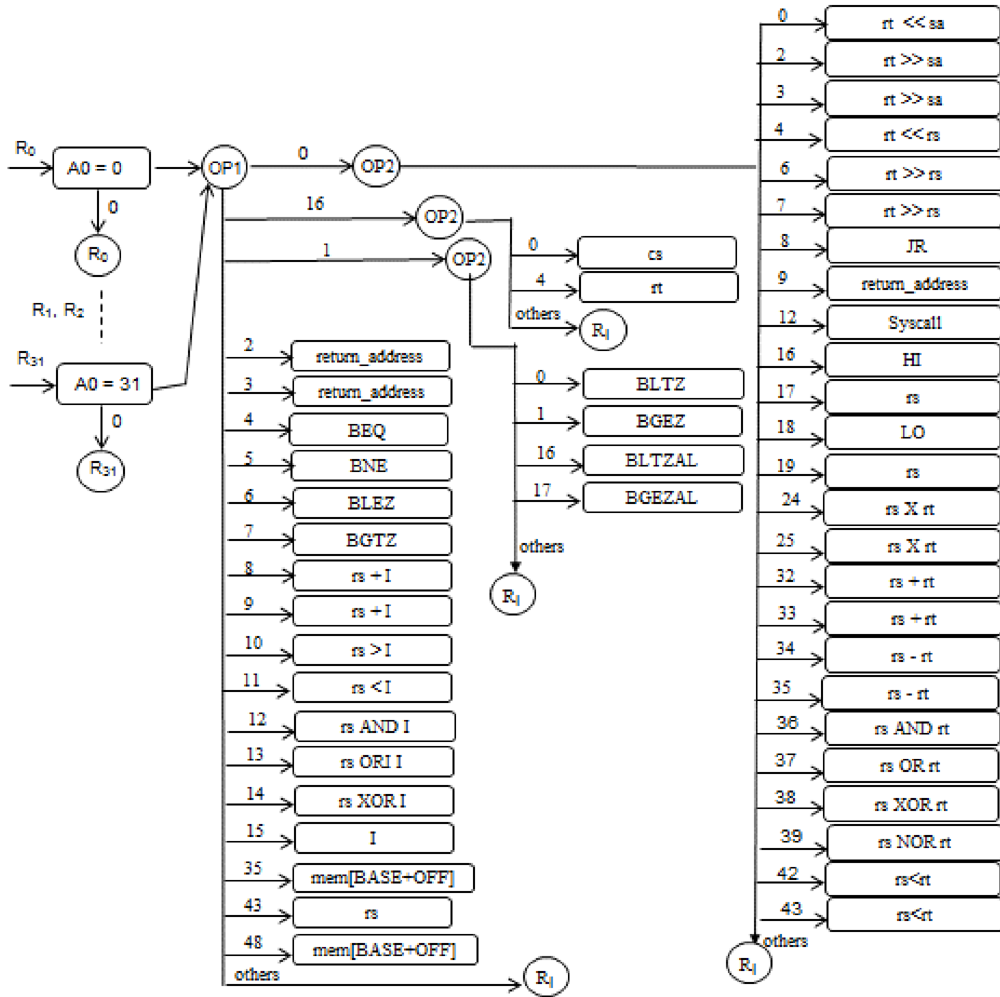


Figure 11: Synthesis of HLDD for miniMIPS [9]

3.1.3 Test Synthesis from HLDD

Using the HLDD graph model or the given processor, test generation can be performed. Test generation will result in a set of test patterns which can be used to test the structural entities of the processor [4]. The process of test generation involves traversing the graph by activating the nodes and consequently deriving a set of patterns. There are two types of nodes in the HLDD namely; control nodes and terminal nodes. The control nodes activate the path of the graph to a desired working mode or terminal node of the system. The terminal node contains nodes that activate the data path which can be used to test the different working modes of the processor.

During test generation, three sets of patterns are generated, the pathlist, the datalist and the testlist. The pathlist holds the patterns (control nodes variables values) that lead to the

terminal nodes. The datalist holds the patterns which will be loaded in the register during the execution of the test program. These patterns activate the datapath within the terminal nodes. The testlist contains the list of test patterns generated by walking through all the nodes. These contains the pathlist and the datalist.

Figure 12 as seen in [4] shows an example of test generation from HLDD model for a miniMIPS ADD instruction.

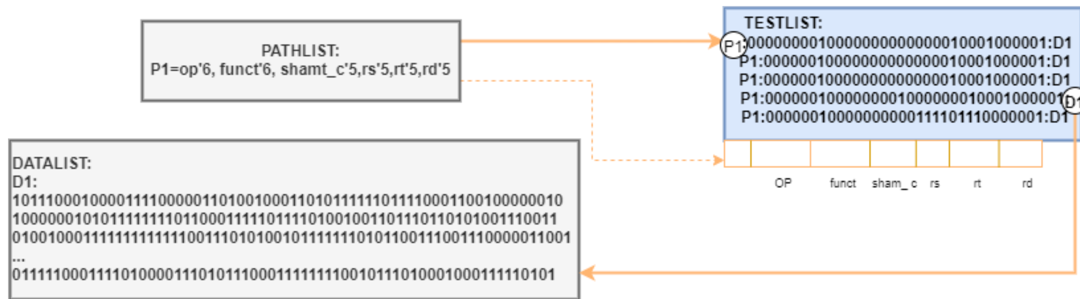


Figure 12: Example of test generation

The testlist has the following syntax P#:test:D# where # is the placeholder for enumeration, P represents the pathlist, D represents the datalist, and test is the binary representation of the node values.

The pathlist has the following syntax P#name1'width1,...,namen'widthn, # is the placeholder for the index, name is the name of the node, and width is the size of the node.

The datalist has the following syntax D#: binary list, where # is the placeholder for the index and binary list is a list of numeric values. This is generated using the methods in [4].

3.2 Test program generation with HLDD

The test program is an important part of SBST. It is divided into three stages. The first is the high-level test data generation, followed by the high-level test program generation and lastly, the fault coverage calculation. The visualization can be seen in Figure 13. High level test data generation was used to generate the control test data, and the pseudo-exhaustive data that were used in our experiments. High level test program generation was used with the test templates generated through HLDD synthesis in section 3.1.2.

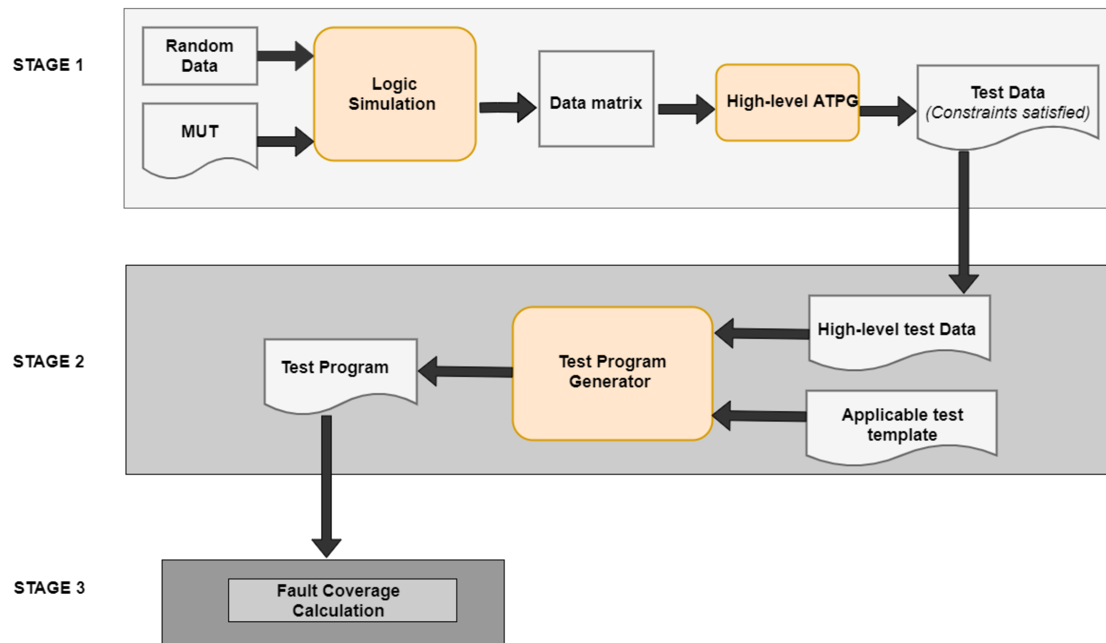


Figure 13: High-Level test data and test program generation

3.2.1 High Level Test Data Generation

Test generation is continually being improved every day in digital systems, as we are aware of technological advancements breeding new challenges in testing modern devices, we will like to discuss the complexities faced in test generation for microprocessors.

An essential quality in generating test in micro technologies is speed while focusing on faults two important data assignments is required giving rise to activation of the faults and the other for fault propagation [11], using some definitions relating to faults in microprocessors.

Test vectors are generated to imitate defects that may occur during the manufacturing process of chips, which may lead to the malfunctioning of the chips. Imitating physical defects means that the test vectors should be able to induce the faulty behaviour that matches the physical faults that may occur during the manufacturing process. The complexity of the system, the size of the tests to be taken and the factor of test quality are reasons why automatic test methods are used for generating test patterns for digital systems. In [1], the quality of a test is dependent on the test data, also the aim of any automatic test pattern generation (ATPG) is to produce efficient test patterns.

ATPG is the application of algorithmic based software for generating test vectors [12] and its need at the structural level is undisputed, because most, if not all the faults in a digital system has to be covered. In addition to the goal of producing efficient test patterns, ATPG aims to cover a high level of fault coverage during testing.

Test pattern generation algorithms can be accessed by the following indicators [13]

- Test effectiveness
- Fault coverage
- Length of the generated test
- Test generation time

In this thesis, the test data were generated using four basic methods (algorithms).

3.2.1.1 Method 1 – Conformity Test

Control Test Patterns– These are the test data patterns, generated to cover high-level functional fault model, and to be used by all instruction in the give processor (miniMIPS), so that the results of all instructions were distinguished pairwise in each bit of the data word. It is also known as **Conformity test**.

Test vectors are gotten from analysing the Circuit Under Test (CUT) and a specific type of fault is being targeted, followed by fault simulation. The targeted faults could be the defects that are in the structural part of a given CUT. After the test for the defects in the targeted area, and a fault is detected, fault simulation is carried out to find other faults that this generated test vector can detect [14]. The process of generating deterministic test patterns can be very extensive, and before fault simulation is carried out to detect other faults, the initial detected faults are noted.

Ideally, detecting all possible faults in a CUT is the aim of testing and we can conclude that detecting all possible faults in the CUT means 100% Fault Coverage (FC). Fault coverage is the percentage of fault that can be detected by the applied test vectors [15]. FC at 100% is desirable but is it not always reached in most tests due to some undetected faults. Undetected faults can also occur even when deterministic test patterns are being used.

Generating Operands for Testing Control path

As part of the HLDD fault model, two constraints were introduced for testing the control part of the MP in [23]. They are as follows:

$$\forall m^T \in M^T (m): [f (m^T) \neq \Omega)] \quad (1)$$

$$\forall m_i, _{mj} \in M^T (m), i \neq j: \forall k [f_k(m_i) < (f_k(m_i) * f_k(m_j))] \quad (2)$$

Where Ω = ZERO or ONE and * = logic OR or logic AND, both depending on the technology implemented in the MP [22] [23].

The label ZERO means the binary vector (000...0) while ONE means the binary vector (111...1). Representing the bit number of the data word is the index k.

The test operands used in the later for testing the control part has to satisfy constraints 1 and 2 (*equation 1* and *equation 2)* stated above. In order to conform to these constraints, Algorithm 1 was developed. The algorithm generates the bits of the operands (data words) which are represented by D_1 and D_2, starting from the LSB, bit by bit, unto the MSB. The essence for this is so that constraints 1 and 2 will be solved for all pairs of the functions $f_k(m_i)$ and $f_k(m_j)$ [24].

Algorithm 1: Test data generation for control part - RANDOM [25]

Input: Instruction set of the processor

Output: Sets of test operands *OPi* for each instruction, including a fault table D

Notations: *n* – represents the number of functions F_j,

op – test operand,

OP – current set of selected random test operands,

$f_i(op)$ – result of the instructions I_j for the operand(s) *op*,

D – Fault table,

D_{ij} – w-bit entry in D,

w – Length of the data word)

```
1. Initialize OP = ∅
2. Generate a set of random operands (R)
3. for i = 1, ...., n
4. Initialize OPi =  ∅,
5.    for j = 1, .....,n (j ≠ i)
***operands for solving constraints fi,k < fj,k
6.              Initialize Dij = 0
7.              for all op ∈ R while Dij ≠ 0
                ***adding new operands for covering Dij
8.                    Dij(op) = fj(op) ∧ (fi(op) ⊕ fj(op))
                *** calculating fault coverage for op
9.                    if (Dij (op) ∨ Dij) ⊕ Dij ≠  0 then
                  *** check for the coverage increment
10.                    begin
11.                    Dij = Dij ∨ Dij(op)
                    *** update of the coverage vector
12.                    include op into OPi
                    *** new operand is selected
13.                    end
14.              endfor op
15.    endfor j
16. endfor i
```

This Algorithm 1 will produce a set of operands for every instruction in the MP and a fault table that satisfies the constraint $f_{i,k} < f_{j,k}$. $D^k_{ij} = 1$ if the constraint is satisfied, and is covered by a minimum of one operand, otherwise $D^k_{ij} = 0$. Finally, the percentage of 1s in D is the high-level functional fault coverage for the test for control path [25].

Algorithm 1 is called RANDOM. This is because for each step of line 7, the random operand that came first (op ∈ R) will be selected with a goal of increasing the fault coverage. Another algorithm called GREEDY was established in order to reduce the test length.

The difference between the GREEDY algorithm and RANDOM is that at line 7, where the best operand is being searched for maximum fault coverage, the subsequent operand is selected and the algorithm proceeds with the search until target $D^k_{ij} = 1$ is reached or no further operands can satisfy the constraint $f_{i,k} < f_{j,k}$.

It is notable that the constraint $f_{i,k} < f_{j,k}$ may not be solved if the related functional fault is redundant, or the search spare R is not large enough [25].

According to [2], Conformity test is a test for a non-terminal node of the HLDD, and its goal is to test the control part of the microprocessor. The conformity test is generally generated according to constraints 1 and 2 that were set up for testing non-terminal nodes.

- **Generating Conformity Test Program for Control part of Microprocessor**

The generation of conformity test for the control part of the microprocessor was developed in [4] and [6]. According to [4] and [6], conformity test was explained as such:

Consider an HLDD $G^Y = (M, \Gamma, X)$ with Y = F(X) as a functional model of the instruction set of a given MP.

$X = C \cup D$ (which represents the instruction format of the MP)

Y = destination data

C = opcode of the instruction format, and is divided into sub-fields $C_k \in C$

D = source data of the instruction format and is divided into $D_k \in D$.

It is notable that the source and destination data variables may refer directly to the registers of may refer to the addressable memory locations. Examples of mapping between instruction formats and the HLDD functional variables are illustrated for three instruction formats below:

I. Instruction format with 1 opcode subfield (C), 1 source subfield (D) and one destination subfield (Y).

II. Instruction format with 1 opcode subfields, 2 source subfields (D1 and D2) and one destination subfield (Y).

III. Instruction format with 2 opcode subfields (C1 and C2), 2 source subfields (D1 and D2) and one destination subfield (Y).

Figure 14: Mapping of miniMIPS instruction formats and the HLDD functional variable

The main targets of the conformity tests are not the instructions as a whole, as per the instruction format, and it involves both the control and data functions. This depicts that if the opcode C is divided into subfields $C_k \in C$, then the control tests will target all the subfield one after the other. To test if all the sub-functions that relates to C_k were rightly selected, the node m in the HLDD module test T(m) for all the values of $x(m) \in V(x(m))$ has to be tested.

In [4], generating a test instruction for testing a fault $r \in R(m, v)$, it is essential to find a test pattern *Xt* which activates a path $l(m_0, m^{T,v})$ from the root node $m_0 \in M^N$ to a terminal node $m^{T,v} \in M^T$, so that $x(m) = v$, and $m \in l(m_0, m^{T,v})$; the pattern X^t corresponds to a full opcode C of instruction, which includes the needed value of C_k. It is also essential to complete the pattern X^t by generating the test data *D*, so that the constraints 1 was satisfied. The result for generating a test instruction for testing the fault model $R(m,v) \subset R(m)$ includes a control pattern (instruction) *C(m,v)*, and a set of data pattern *D(m,v)*.

The algorithm for conformity test program according to [4] is:

```
1. for all m ∈ M^N do
2.         for all v ∈ V (x(m)) do
3.         for all r do
4.            execute C(m,v).D(m,v,r)
5.         end for
6.        end for
7. end for
```

Figure 15: Algorithm for conformity test

- **Explanation of the algorithm for conformity test**

In line 1, $m \in M^N$ represents the nonterminal nodes, and line 1 is testing $T(M^N)$ for the fault model R. Lines 2 – 5, firstly initializes all registers involved in operations $f(m^{Tv})$ at every terminal nodes $m^{Tv} \in M^T(m) \subseteq M^T$ with values satisfying *constraints 2.* Secondly, it executes the instruction that assigns the value or v to $x(m)$, activates a path that leads to node m in G_Y, and the paths that transits from m to $m^{T,v} \in M^T(m)$; Thirdly, the algorithm observes the value of Y.

Line 6 ends the testing for nonterminal node m $\in M^N$ and line 7 ends the conformity test of the HLDD G^Y.

The conformity test is used to generate a template that will be used with the control test patterns. The functional variables in this test loops through all the instructions, while the other variables remain constants [9]. A test template was created for the conformity test as seen in Figure 16.

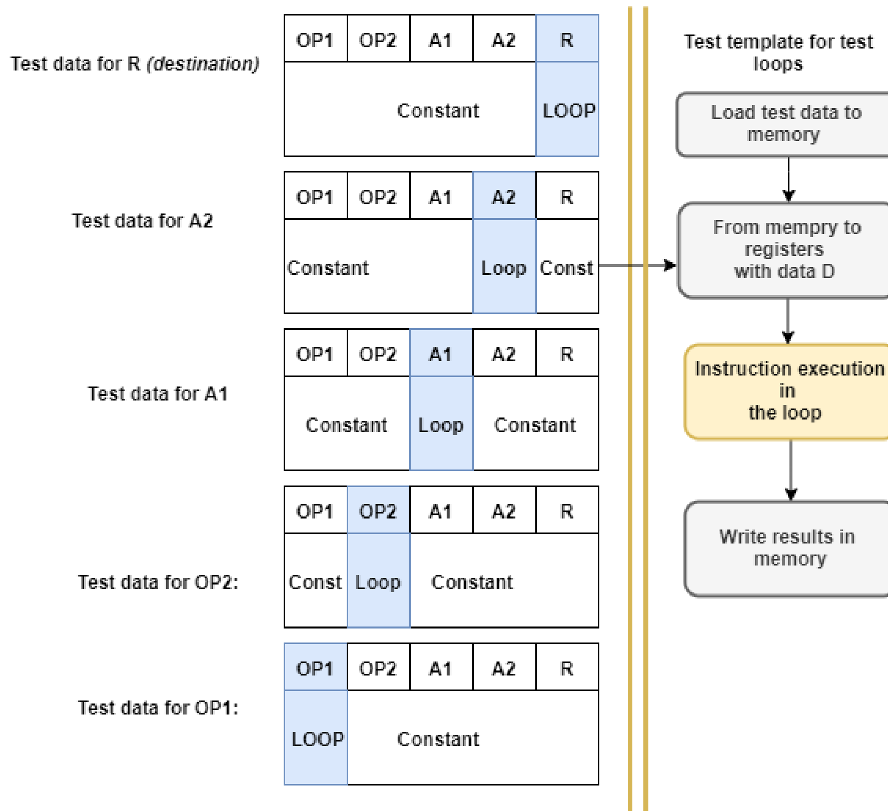


Figure 16: Structure of Conformity test

In Figure 16, OP1, OP2, A1, A2 and R serves as the control nodes (non-terminal nodes). Each of them has a test template that consists of instructions that leads to the path of a particular node. The test program that will be generated also consists of these instructions, which will be used test the control nodes. However, the control test data generated in with Algorithm 1 that satisfy the constraint 1 and 2 will be passed into the test program as described in Figure 13.

3.2.1.2 Method 2 – Short Scanning Test

Dedicated PET – this is the pseudo-exhaustive test data for testing each instruction with so called dedicated data, generated separately for each instruction based on its functionality, and to guarantee exhaustive test of each bit of the data word. It is also known as the **Short scanning test**.

For understanding the concept of pseudo-exhaustive test pattern, the concept of exhaustive test pattern should be understood.

Exhaustive Test Pattern

Exhaustive test patterns detect all the possible faults, either gate-level SAF faults, wired AND/OR faults, and bridging faults in a combinational circuit. A combinational CUT with N-input, exhaustive testing will require applying 2N exhaustive patterns [16]. This approach will not detect all possible transistor-level faults or delay faults because these kinds of faults needs a specific order at which the vectors needs to be arranged, if possible, the potential to repeat certain test vectors within the vector set [12]. If a combinational circuit has few primary inputs, exhaustive testing may be a viable option, where every possible input vector is considered [17]. However, in circuits with large amount of primary inputs, exhaustive testing might not be the viable approach. Due to this drawback, pseudo-exhaustive test makes it possible to partition the circuit and only exhaust the input vectors within each cone for each primary output [17].

Pseudo-Exhaustive Patterns alternatively, have lesser number of test patterns [18]. As stated above, the circuit is partitioned and is exhaustively tested. This means that a better FC is achieved. In [17], a circuit with three primary inputs *n1, n2*, and *n3*, with a

corresponding primary output cone each will have a total number of $2^{n1} + 2^{n2} + 2^{n3}$ pseudo-exhaustive vectors at most.

Generating Operands for Testing the Data Path

A significance of pseudo-exhaustive data is that remaining test generation procedure will not depend on the implementation details of the processor cores under test [25]. Ideally, logic operations are independent in all bits, hereby enabling the operations in all bits to be tested independently. In cases of unary operations, two exhaustive patterns will be enough, while for logic operations, we need to use four exhaustive patterns {(0,0), (0,1), (1,0), (1,1)} per bit [24].

Table 3: Generation of PET data for adder [25]

No	**. . . .**	**4-bit**	**3-bit**	**2-bit**	**1-bit**	**0-bit**
		$a_4\ b_4\ c_4$	$a_3\ b_3\ c_3$	$a_2\ b_2\ c_2$	$a_1\ b_1\ c_1$	$a_0\ b_0\ c_0$
1	. . . .	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	. . . .	0 1 0	0 1 0	0 1 0	0 1 0	0 0 1
3	. . . .	1 0 0	1 0 0	1 0 0	1 0 0	1 1 0
4	. . . .	1 1 0	0 0 1	1 1 0	0 0 1	0 1 1
5	. . . .	0 0 1	1 1 0	0 0 1	1 1 0	1 0 0
6	. . . .	0 1 1	0 1 1	0 1 1	0 1 1	1 0 1
7	. . . .	1 0 1	1 0 1	1 0 1	1 0 1	1 1 0
8	. . . .	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

Table 4: Generation of PET data for Subtractor [25]

No	**. . . .**	**4-bit**	**3-bit**	**2-bit**	**1-bit**	**0-bit**
		$a_4\ b_4\ c_4$	$a_3\ b_3\ c_3$	$a_2\ b_2\ c_2$	$a_1\ b_1\ c_1$	$a_0\ b_0\ c_0$
1	. . . .	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	. . . .	1 1 0	0 1 1	1 1 0	0 1 1	0 0 1
3	. . . .	0 0 1	1 0 0	0 0 1	1 0 0	0 1 0
4	. . . .	1 0 0	1 1 0	1 0 0	1 1 0	0 1 1
5	. . . .	0 1 1	0 0 1	0 1 1	0 0 1	1 0 0
6	. . . .	1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
7	. . . .	0 1 0	0 1 0	0 1 0	0 1 0	1 1 0
8	. . . .	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

In Table 3 and Table 4 above, ripple-carry is used for generating the PET data for addition and ripple-carry is implemented for generating the PET data for subtraction. ADD and SUB stands as operators in the miniMIPS ISA and the data generated are dedicated for these operators. The same applies to other operators used in our experiment – AND, XOR,

SLL, SRL, etc. To cover every combinations of the input operands a_0, b_0 and c_0 of each bit of the adder, 8 pairs of data were needed, as seen in Table 3 and Table 4. C_0 may be the carry bit in the case of addition, or the borrow bit in the case of subtraction. PET patterns are generated from the LSB, after calculating the carry for the c_n for the next bit, and the right values which will fit into the operands a_n and b_n. Through this previous step, all pseudo-exhaustive combinations for the bit section would be achieved [25]. Additionally, the columns titled "2-bit" and "1-bit" can be copy pasted for the next two-bit-pairs to the right [24].

Scanning Test Definition: According to [4], Scanning test is a test for a terminal node of the HLDD, and its goal is to test the data path of the microprocessor. It focuses on testing the correctness of the terminal nodes in the HLDD by making use of the same instruction with different test data.

Generating Scanning Test for the Data Path of MP

The generation of scanning test for the data path of the microprocessor was developed in [4] and [6]. According to [4] and [6], scanning test was explained as such:

Consider an HLDD $G^Y = (M, \Gamma, X)$ with Y = F(X) as a functional model of the instruction set of a given MP.

$X = C \cup D$ (which represents the instruction format of the MP)

Y = destination data

C = opcode of the instruction format, and is divided into sub-fields $C_k \in C$

D = source data of the instruction format and is divided into $D_k \in D$.

D and Y could be the address of a register or the address of a memory location. The source D could also be an immediate data which could be part of the miniMIPS instruction format.

The source and destination data variables may be the address of the registers, or to the addressable memory locations. The immediate data which will be part of the instruction format, may be represented by the source variable.

A test will be generated for every terminal nodes $m^T \in M^T \subset M$ in each HLDD $G^Y = \{G\}$, $Y \in U$ (|U| = number of HLDDs), for the purpose of testing the complete data path of the provided microprocessor which consists of various HLDDs.

In [4], the term of data functional fault model (DFFM) of the HLDD G^Y was introduced. It is denoted as $R(m^T)$ and a union of all functional fault models in the terminal nodes $m^T \in M^T \subset M$, and it represents the working nodes of the microprocessor $Y = f(m^T)$. Each functional fault $r \in R(m^T)$ is similar to the conditional SAF model developed for gate-level testing [7].

In order to test the faults $r \in R(m^T)$, we need to execute a test using the set of instruction of the microprocessor.

$$T(mT)\ i = \{C(m^T).\ D\ (m^T, r)\}$$

In the test above, $C(m^T)$ = Instruction code and it remains constant

$D\ (m^T, r)$ = Data. It is dynamic with the values from the set of constraints $R(m^T)$.

The general point of the scanning test is to reiterate the same instruction with data fetched by scanning a given data array.

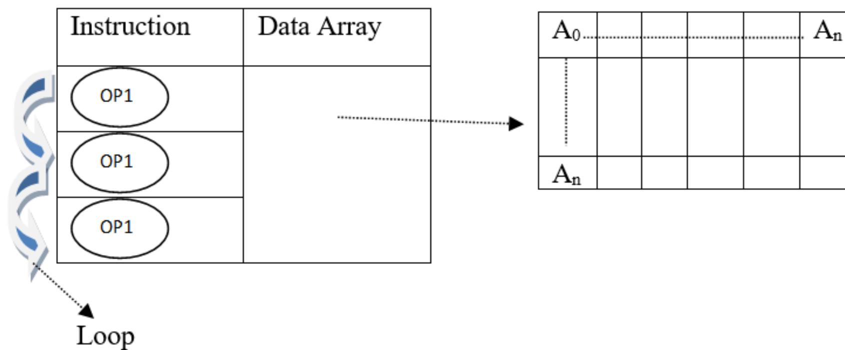


Figure 17: Structure of Scanning Test

In the figure above, all the registers are loaded with the data fetched from the array, the instructions that activates in the HLDD are activated and in a loop, lastly, the result of each operation is written into memory.

The algorithm for scanning test according to [4] is:

```
1. for all m ∈ M^N do
2.        for all r do
3.           execute C(m^T).D(m^T, r)
4.        end for
5.       end for
```

Figure 18: Algorithm for scanning test

Explanation of the algorithm for Scanning Test

In line 1, $m^T \in M^N$ represents the nonterminal nodes, and line 1 is testing $T(M^N)$ for the fault model R. Lines 2 – 3, firstly initializes all registers involved in the function $f(m^{Tv})$ at every terminal nodes $m^{Tv} \in M^T$ with the test data d. Secondly, it implements the instruction that activates in G_Y a path to the node m *in* M^T. Thirdly, the algorithm observes the value of Y.

Line 4 ends the testing for terminal node $m \in M^N$ and line 5 ends the scanning test of the HLDD G_Y.

The scanning test described in this section is called short scanning test.

3.2.1.3 Method 3 – Long Scanning Test

All PET Test Patterns – These are the pseudorandom test data for testing each instruction with a sum of all test patterns generated pseudo-exhaustively for the data part. It uses the combination of all the dedicated PET data to test for each instruction based on its functionality. It can be referred to as Long scanning test.

All PET represents combination of the all the PET data generated through scanning test in section 3.2.1.2. For our experiment in chapter 5, 9 patterns were dedicated for the ADD and SUB instructions, 4 patterns for the logic instructions AND, OR, XOR and NOR, while 2 different sets of patterns were dedicated for the shift, load and branch instructions. Lastly, a total of 310 patterns were dedicated for the MULT instruction.

In [26], a method to transform the *"paper and pencil"* 2-dimensional ILA of n-bit array into a set of *(n – 1)* 1-dimensional ILAs of n cells, which can be tested pseudo-exhaustively nearly as easily as ripple-carry adders. In order for such modification to occur, the concept of data-controlled segmentation of the circuit was introduced.

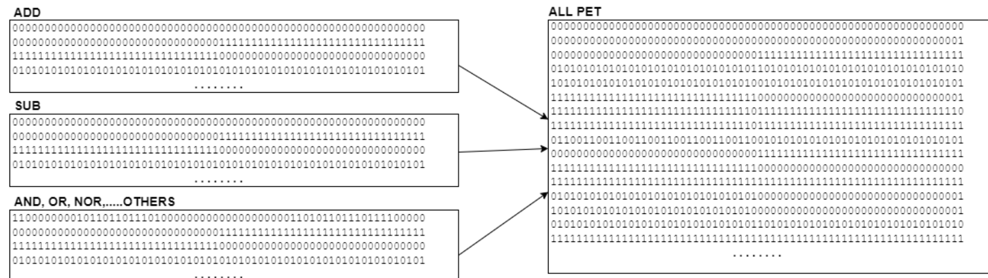


Figure 19: PET combination to All PET Test Patterns

3.2.1.4 Method 4 – Random Test

Random Test Patterns: Random test patterns are very easy to generate via Random Test Generation (RTG). Unlike deterministic and pseudo-exhaustive test patterns, no specific faults are targeted for the random test generation. Additionally, exhaustive test may be superior to RTG because RTG can produce duplicated vectors and may miss certain ones [17]. RTG stands out because it is easy to generate the random vectors, it does not satisfy any constraints and the complexity is low. However, the detrimental effect of random test patterns is that it can detect a set of faults that is up to 10 times larger than a deterministic test patterns for the same set of faults [12]. Due to this, determining the quality of a test set becomes difficult, because conventional methods based on fault simulation becomes costly [19]. In [20], some of the disadvantages of random test generation is that it can have very long test application time, love coverage, area overhead and additional delay.

RTG makes it possible for the random vectors to be evenly distributed in the pattern set. This means that the random patterns will eventually have equal numbers of logic 1s and 0s in the set as a whole. The method used to generate the random patterns for the experiments carried out in this thesis is not totally random. A pseudo-random number was used so that the random patterns can remain the same in cases where they need to be re-generated. For RTG, we cannot totally be confident in the kind of result or FC, we can

only be certain that the random patterns used in a test will detect all possible Single Stuck Fault (SSF) [12].

Similar to line 2 of Algorithm 1, a fewer set of random data is generated with a python script and loaded directly as a high-level test data into the test program generator, alongside the test templates, as illustrated in Figure 13. It is notable to mention that the randomly generated patterns do not satisfy any constraints as compared to the control test patterns.

3.3 Test program generation

As discussed in section 3.1.2, the test program is generated from the HLDD synthesis. The synthesis is implemented through prepared test code templates, used in generating the test program. As miniMIPS is a processor with 32-bits registers, the initial target of the test template is to reset all the 31 registers in Figure 7, to make sure that the current test program is not affected by the previously generated program with a different data in the registers. After the registers are set to null, the test data is loaded into the memory. The test data could be the control, PET, All PET or random test data. The final process is the generation of the test program based on the conformity test template or scanning test template or random data test program. Section 4.1 will provide a more elaborate explanation on the test template creation.

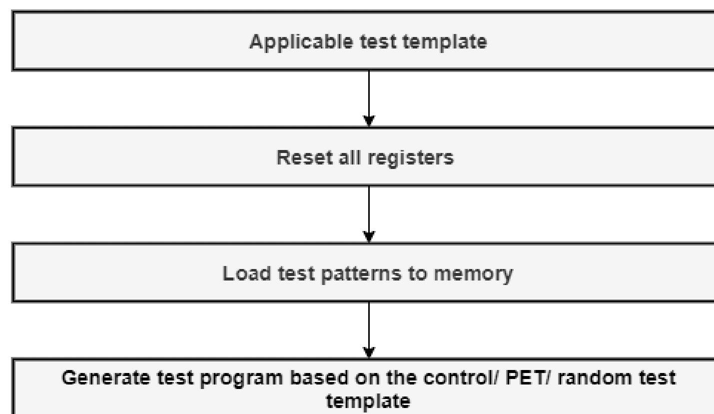


Figure 20: Test Program Generation process with four templates

3.4 Fault simulation

The fault simulation is the final stage of the automated SBST. It comprises two fault simulators:

1. "Home-made" high-level fault simulator for measuring the quality of conformity test, whereas the PET-based scanning test by definition is considered a full (100%) high-level test.

2. Professional low-level fault simulator called TetraMax for the final evaluation of the test quality in terms of standard SAF coverage. From the simulation of the test program in assembly language, with the test bench of the MP within ModelSim, a vcd file is generated and used to calculate the fault coverage in the selected MUT of the processor.

3.5 Conclusions

1. In this chapter, basic approaches of testing microprocessors were considered: conformity test with control test patterns and scanning test with two versions of using PET test data (short and long scanning test).

2. The test data used in these basic approaches are divided into 4 classes: Control test data, PET, all-PET and random test data.

3. Based on these two types of tests (conformity and scanning), and 4 types of test data, in the following chapters several combinations of test structures using different test data are investigated and compared.

4 Development and investigations of the methods

The Execute module was partitioned also into two parts: control part and data part. The test program was developed in two parts: for testing the control part (conformity test), and for testing the data part (scanning test). This chapter covers the test program generation with the data generated with the four methods or classes: Control test data,

PET, all-PET and random test data, and different combinations of the methods. We created a new template for PET, all PET and Random data for maximum fault coverage. The goal of the experimental research was to evaluate the quality (SAF coverage) of all the four basic test methods separately, to compare the two PET approaches and to evaluate the possible contribution of the random test approach by investigating the quality of different combinations of the basic test methods.

4.1 Test templates

Prior to the test program generation is the manual creation of the test template according to Figure 20. Contrary to the SBST program generation in [9], our experiment generates test program based on scanning tests and random test data, including conformity test.

As discussed in 3.1.2, HLDD was used to generate templates and test data for the test program. The work done in [9] was to generate the HLDD for the control part of miniMIPS processor and test template for the control test program. For us to create a test program for method 2, 3 and 4, the HLDD graph has to be synthesized to generate a test template and test data for the tests. Four test templates were created for the generated PET and random data.

For our experiments, method 2 and 3 uses the same test template, with 23 miniMIPS instructions, from the ISA. The figure below illustrates the HLDD graph that was used for the random, PET and all PET templates.

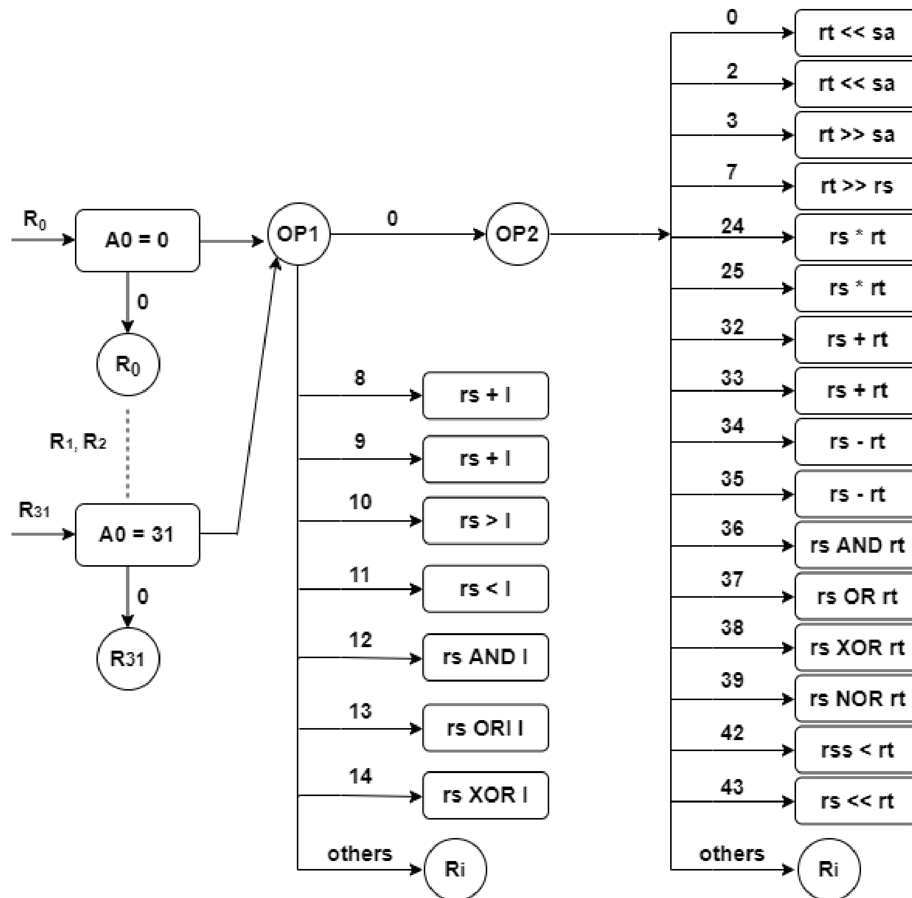
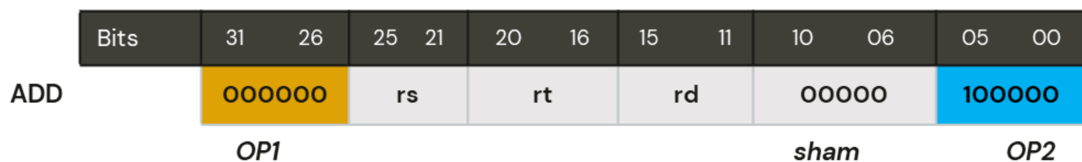


Figure 21: HLDD Structure for method 2, 3, and 4

Figure 21 describes how the instructions were sub-divided based on the number of operands with data that needs to be loaded into the registers. An example is the ADD instructions, which needs two registers to load operands. According to the miniMIPs ISA, the ADD instruction has the structure below:



	Bits	31 26	25 21	20 16	15 11	10 06	05 00
ADD		000000	rs	rt	rd	00000	100000
		OP1				*sham*	*OP2*

Figure 22: ADD structure in miniMIPS ISA

The ADD instruction is represented by the operation $rs + rt$. The address of the register with the loaded value of operand 1 is *rs*, while *rt* holds the value of operand 2.

Test template 1

```
Operation_Instruction_method:
Load patterns
Instruction rd, O1, O2
Sw rd, offset(M)
Jal increment
```

Instruction represents the instructions that are listed in Table 5. Method represents the type of method being used for the experiment, which could be pseudo or random. Pseudo was used to represent PET and all PET. The result register is represented by *rd*, O_1, O_2 are the operand registers, offset value is 4 and M represents the memory address where the result is stored.

Table 5: List of instructions under template 1

S/N	Instructions
1	ADD
2	ADDU
3	SUB
4	SUBU
5	OR
6	XOR
7	NOR
8	AND
9	SLT
10	SLTU
11	SRAV

Test template 2

```
Operation_Instruction_method:
Load patterns
Instruction rd, O1, I
Sw rd, offset(M)
Jal increment
```

Instruction represents the instructions that are listed in Table 6. Method represents the type of method being used for the experiment, which could be pseudo or random. Pseudo was used to represent PET and all PET. The result register is represented by *rd*, O_1 is the

operand register, I stand for the immediate value, offset value is 4 and M represents the memory address where the result is stored.

Table 6: List of instructions under template 2

S/N	Instructions
1	ADDI
2	ADDIU
3	ANDI
4	ORI
5	XORI
6	SLTI
7	SLTIU

Test template 3

```
Operation_Instruction_method:
Load patterns
Instruction rd, O1, SA
Sw rd, offset(M)
Jal increment
```

Instruction represents the instructions that are listed in Table 7. SA represents the shift amount, the result register is represented by *rd*, O_1 is the operand register, offset value is 4 and M represents the memory address where the result is stored.

Table 7: List of instructions under template 3

S/N	Instructions
1	SLL
2	SRA
3	SRL
4	LUI

Test template 4

```
Operation_Instruction_method:
    Load patterns
    Instruction rd, O1, O2
    MFLO rd
    Sw rd, offset(M)
    MFHI rd
    Sw rd, offset(M)
    Jal increment
```

Instruction represents the instructions that are listed in Table 8. SA represents the shift amount, the result register is represented by *rd*, O_1 and O_2 are the operand registers, offset value is 4 and M represents the memory address where the result is stored.

Table 8: List of instructions under template 4

S/N	Instructions
1	MULT
2	MULTU

4.2 Set-up of the Experiments

The set-up of the experiment can be visualized in Figure 23. The experiments were performed on a Linux based computer and modelSim simulator was used to simulate the environment of the experiment. In order to simulate the behaviour of the miniMIPS MP, the HDL of miniMIPS was implored. The MP uses its RAM and ROM as memory. In order to load the test program, the ROM is used, and the RAM is used for storing and later accessing the test data and results. As discussed in 4.1, the test templates were developed manually from the HLDD and going forward, the test program is automatically generated with the help of a python script. As illustrated in Figure 23, after the test program is generated, the assembler that comes with the miniMIPS MP converts the test program written in assemble language, into an executable binary file (machine code). The executable binary file is used by the ROM via the test bench. After the simulation of the MP with ModelSim, it automatically executes the test program and provides a test response, which is later loaded into the memory. The test responses are stored into dump file, in .vcd format. The dump file is passed into the low-level fault simulator called TetraMax and the result of the fault coverage calculation is provided.

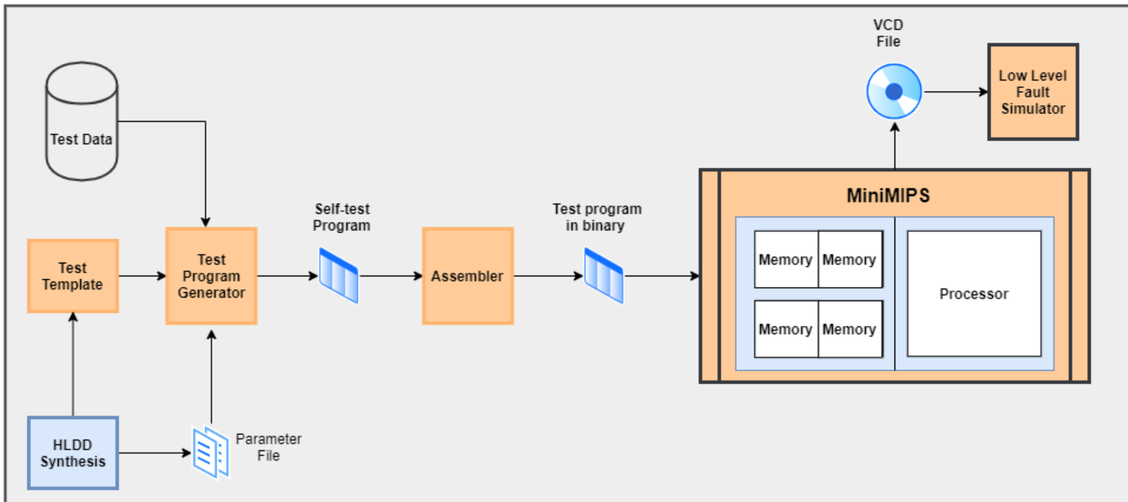


Figure 23: Set-up of the experiments

4.3 Combination of different methods

The goal of the experimental research was to evaluate the quality (SAF coverage) of all the four basic test methods separately, to compare the two PET approaches and to evaluate the possible contribution of the random test approach by investigating the quality of different combinations of the basic test methods. The experimental research consisted in carrying out 11 experiments. They are illustrated in the figures below:

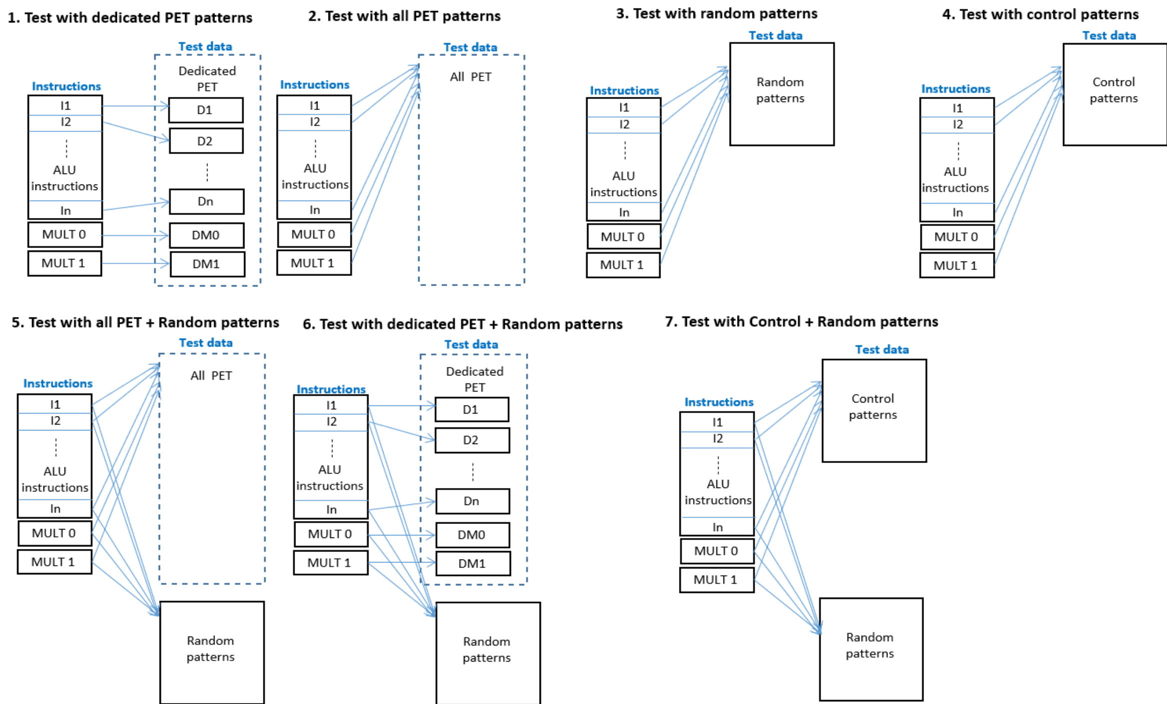


Figure 24: Experiment structure for experiments 1 - 7

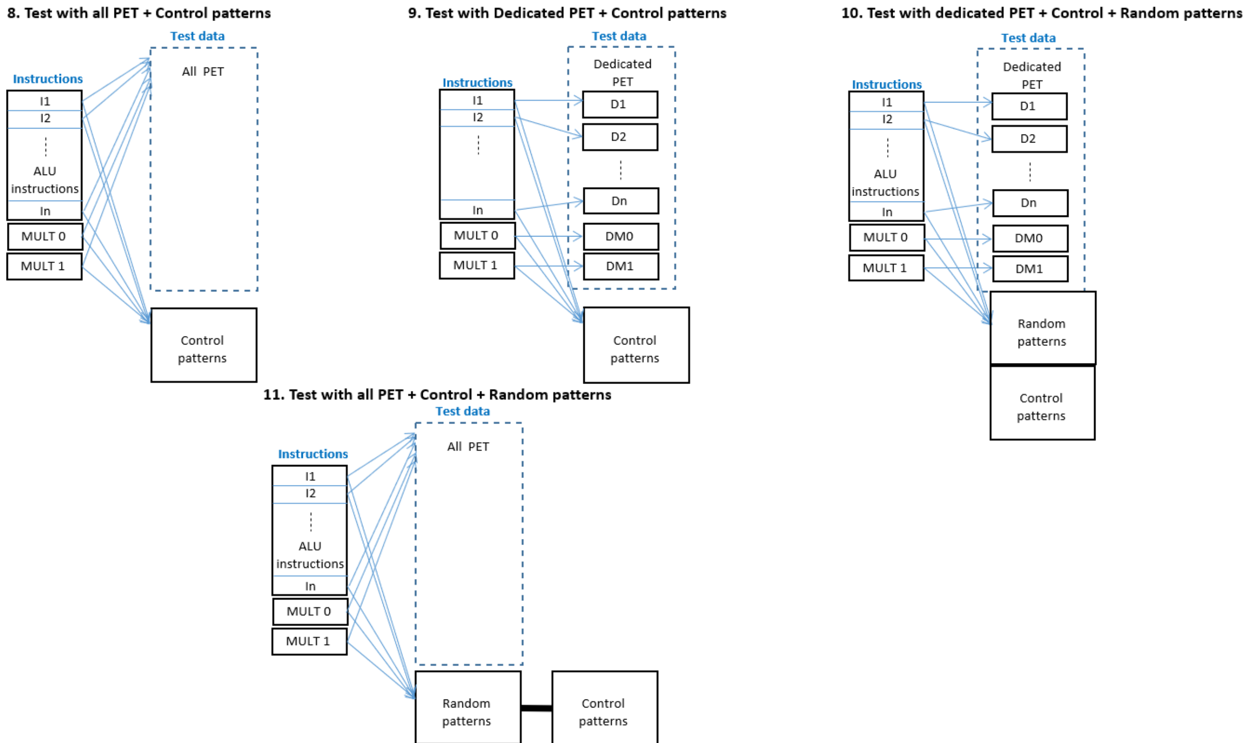


Figure 25: Experiment structure for experiments 8 – 11

In Figure 24 and Figure 25, the instructions represent the ALU instructions, from I_1 to I_n. MULT 0 and MULT 1 are the multipliers in the ALU of the miniMIPS processor. The ALU of the miniMIPS processor can consists of the executable module *PPS_EX*. The PPS_EX module consists of the ALU, which has the ADD, MULT 0 and MULT 1 modules in it.

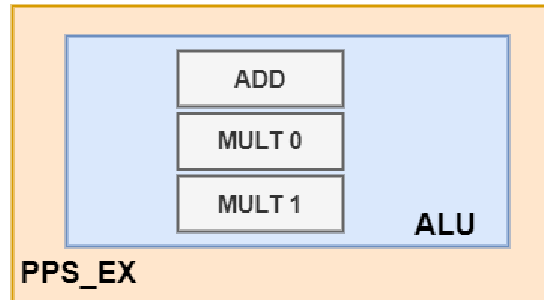


Figure 26: Structure of miniMIPS execute module

The test data D1, D2...Dn, DM0 and DM1 for dedicated PET represents the dedicated data used for each instruction in the test template. D1, D2...Dn are the patterns as illustrated in Figure 19. DM0 and DM1 represents the 310 data dedicated for the MULT instruction.

The results of the experiments will be discussed and analysed in chapter 5. However, experiments 1 – 4 explored the 4 basic methods of data for testing the MP. The combination of the 4 methods commences from experiment 5 until 11. Let us represent each method with *M. M1, M2, M3* and *M4* represents methods 1 – 4.

Where Method 1 (M1) – Control patterns for testing

Method 2 (M2) – Dedicated PET patterns for testing

Method 3 (M3) – All PET patterns for testing

Method 4 (M4) – Random patterns for testing

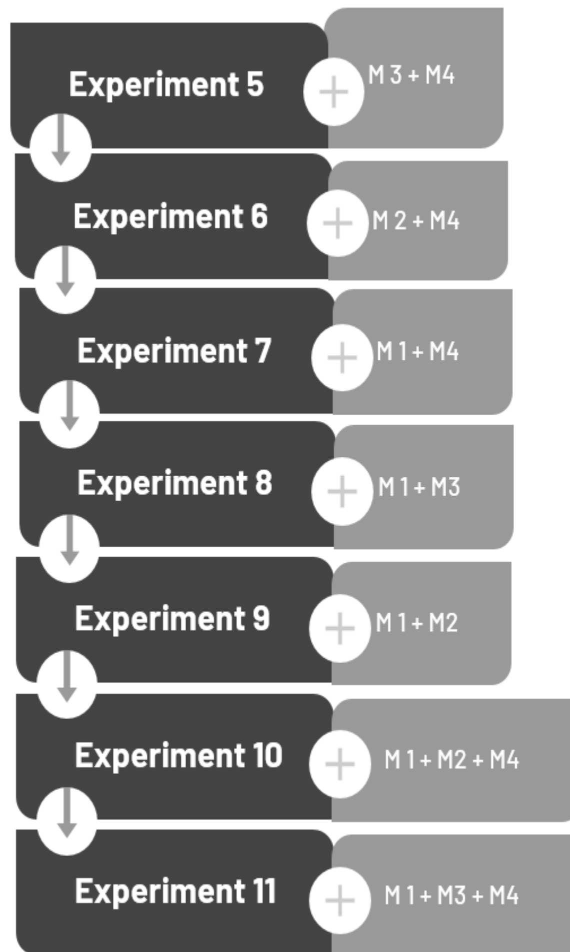


Figure 27: Combination of different methods

5 Implementation and investigations

The goal of generation of different test structures using different test data was to establish the impacts of different test data to the quality of test programs in terms of low-level fault coverage, test length and simulation time that is in correlation with testing time in real world. Absolute testing times were not the goals, rather relations between different modifications of test structures.

The results of the experiments are presented in the 5.2 – 5.8. We will discuss the fault coverage, the test lengths and the simulation times. Ideally, the advantage of methods 1-4 is in the fact, that they do not need information of the real gate-level structure of the

units under test, and hence are classified as implementation-independent test generation approach.

The quality of the scanning and conformity tests can be characterized as follows:

1. The control test guarantees 100% of SAF coverage in the control part of the unit, and as well the coverage of larger class of faults than SAF, including also the conditional SAF, multiple SAF and bridging faults, due to the exhaustive functional testing conception.

2. The scanning test, using either dedicated PET or all PET patterns, guarantees full fault coverage of SAF faults only for logic instructions and for ripple carry adders, but not for more complex adders like carry-look-ahead adders and carry save adders, also for different types of multipliers. Hence, the PET approach can be considered as a heuristic approach, which however is expected to give a high SAF coverage. An added value of the PET approach is in the coverage of larger class of faults than SAF only, including the conditional SAF, multiple SAF and bridging faults due to the exhaustive test conception.

3. The random test gives no any guarantee, and its fault coverage can be calculated only afterwards by fault simulation. The advantages of the random test are in the ease of test generation, but the disadvantage is in the longer test compared to the algorithmic tests PET or control test.

Note, the PET and control tests have mutual effects, in the sense that PET, targeting the faults in the data part, covers also the faults in the control part, and the control test, vice versa, covers also the faults in both parts of the module. From that it follows, when applying both, control and PET tests, then the possible deficiency of PET should be removed or at least reduced. The same purpose of improving the PET quality is also in applying random patterns.

5.1 Goals of the experiments

Referencing Figure 24, Figure 25 and Figure 27, the research targets was to illustrate the following:

1) The first 4 experiments show the fault coverage of the basic algorithms.

2) The experiments 1 and 2 compare the both PET methods.

3) The experiments 1, 6 and 9 show the contributions of the Control and Random tests to the Dedicated PET test.

4) The experiments 2, 5 and 8 show the contributions of the Control and Random tests to the All PET test.

5) The experiments 4 and 7 show the contribution of the Random Test to the Control test

6) The experiments 8, 9, 10 and 11 show the contribution of the Random Test to the two versions of full deterministic tests consisting of the control test and either Dedicated PET (experiments 9 and 10) or All PET (experiments 8 and 11) tests.

7) The experiments 3, 8 and 9 show the comparison of the deterministic high-level implementation-independent approach vs. pure random approach (trade-off problem).

5.2 Investigations #1

In this section, experiment 1 represents the test with dedicated PET patterns, experiment 2 represents the test with all PET patterns, experiment 3 is for random patterns and experiment 4 implies the test for the control patterns. For each experiment four sub-experiments were performed for the four different MUTs – PPS_EX, ADD, MULT 1 and MULT 0.

Table 9: Results of Experiment 1 - 4

MUT	Experiment 1 (M2)	Experiment 2 (M3)	Experiment 3 (M4)	Experiment 4 (M1)
	Fault coverage (%)			
PPS_EX	93.05	93.82	96.58	98.35
ADD	90.62	96.58	99.8	99.96
MULT1	94.51	94.51	98.84	99.37
MULT0	96.52	96.52	98.94	99.33

Each MUTs are the modules in the ALU of the miniMIPs processor. Each of them has different number of faults present in them, but the number of faults is the same for every experiment.

Table 10: Number of faults in MUTs of miniMIPS processor

MUT	Number of faults
PPS_EX	211832
ADD	2516
MULT 1	95188
MULT 0	91810

Additionally, the larger the MUT, the more time it takes for the low-level fault simulator to measure the fault coverage. However, the test length for the experiments is a huge contributing factor. The test length is the amount of test data used. This emphasizes that the amount of test data used for the experiment influences the time it takes to measure the fault coverage. The test length and simulation time used in experiments 1-4 are shown in Table 11.

In experiment 1 the combination of all dedicated PET patterns is a total of 28, in addition to the 310 patterns dedicated to the MULT function. A total of 338 patterns were used in experiment 2. The length of the test program increases in proportion to the number of patterns used in generating it. The miniMIPS processor has a limited memory, therefore, if the test program is too long, the vcd file will not be generated when simulating the test bench of the MP. Hence, a total of 150 random patterns were used in experiment 3. Lastly, experiment 4 uses the 166 patterns that were generated via conformity test.

Table 11: Test length and simulation time for experiments 1 - 4

	MUT	Test length (data)	Fault simulation time (s)	Simulation time (s)
Experiment 1 (M2)	PPS_EX	338	1155.24	0.005
	ADD	9	8.74	0.003
	MULT 1	310	469.29	0.005
	MULT 0	310	302.11	0.003
Experiment 2 (M3)	PPS_EX	338	1920.31	0.009
	ADD	28	9.50	0.003
	MULT 1	310	768.03	0.008
	MULT 0	310	599.57	0.006
Experiment 3 (M4)	PPS_EX		5356.57	0.025
	ADD		27.63	0.011
	MULT 1		1913.71	0.02
	MULT 0	150	1764.62	0.019
Experiment 4 (M1)	PPS_EX		7703.38	0.036
	ADD		40.52	0.016
	MULT 1		2826.60	0.029
	MULT 0	166	2825.95	0.031

The fault simulation time is the time taken by the low-level fault simulator (TetraMAX) to complete the FC calculation. Hence, in order to get the simulation time which correlates with the testing time, the following formula was applied:

$$Simulation\ time(s) = \frac{Fault\ simulationtime(s)}{Number\ of\ faults\ in\ MUT} \quad (3)$$

It is notable that the time taken to perform each experiment is not always directly proportional to the test length. This is evident as experiment 2 has a longer test length that experiment 3 and 4, however, the time taken to measure the fault coverage in experiment 3 and 4 are more than double of the time in experiment 1 and 2. This observation also proves that PET technique is more effective than random test [26].

Table 9 shows that experiment 4, which uses the control patterns has the best FC in each MUT. However, experiment 3 produced a better FC than experiment 1 & 2 because random patterns are purely random and the type of result it produces are unprecedented and uncertain. The number of random patterns must be large in order to produce an excellent FC.

5.3 Investigations #2

In this section, we will compare the results of both PET methods in experiment 1 and 2. The execute module (PPS_EX) will be evaluated as it contains other modules and is sufficient for comparison.

Table 12: Comparison of method 2 and 3

		Results		
	MUT	**FC (%)**	**Test length**	**Simulation time (s)**
Experiment 1 (M2)	PPS_EX	93.05	338	0.005
	ADD	90.62	9	0.003
	MULT 1	94.51	310	0.005
	MULT 0	96.52	310	0.003
Experiment 2 (M3)	PPS_EX	93.82	338	0.009
	ADD	96.58	28	0.003
	MULT 1	94.51	310	0.008
	MULT 0	96.52	310	0.006

From the result above, it is evident that M3 – using all PET patterns produces a better FC than using only the dedicated PET patterns. The stipulated reason is because M3 makes

sure that the combined dedicated PET patterns is used to test the instructions specified in the template. Instead of 9 dedicated patterns for the ADD instruction, 28 combined patterns + 310 patterns for the multiplier are used. It becomes more evident in this comparison that the more the test data, the larger the test length and test program, the more time spent by the low-level fault simulator to calculate the FC.

5.4 Investigations #3

Experiments 1, 6 and 9 are evaluated to investigate the contribution of the control (M1) and random test (M4) to the dedicated PET test (M2).

Table 13: Comparison of experiments 1, 6 and 9

	MUT	Results		
		FC (%)	Test length	Simulation time (s)
Experiment 1 (M2)	PPS_EX	93.05	338	0.005
	ADD	90.62	9	0.003
	MULT 1	94.51	310	0.005
	MULT 0	96.52	310	0.003
Experiment 6 (M2 + M4)	PPS_EX	96.48	428	0.011
	ADD	99.56	99	0.010
	MULT 1	98.56	400	0.010
	MULT 0	98.5	400	0.008
Experiment 9 (M2 + M1)	PPS_EX	98.66	504	0.040
	ADD	99.96	175	0.017
	MULT 1	99.42	476	0.032
	MULT 0	99.67	476	0.029

The contribution of the control test data is more prominent than the random test data. The control test data and the dedicated PET data is the same as a full conformity test and scanning test. Meaning that the test covers all the control parts and the data paths of the MP. A total of 90 random patterns were used, in addition to the dedicated PET patterns. We can conclude that the random patterns increased the test length of experiment 6, while displacing efficiency and quality. Contrary to that, the control test data covered the non-terminal nodes, while the PET data covered the data-path. Theoretically, both methods M2 and M1 are meant to be the best data set for the data path and control part respectively. However, the trade-off for a better FC % is the time taken to calculate the FC. Experiment 9 took more than 3 times more seconds than experiment 6 and more than 7 time more seconds than experiment 1.

5.5 Investigations #4

In this section, experiments 2, 5 and 8 are evaluated to investigate the contribution of the control (M1) and random test (M4) to all PET test (M3).

Table 14: Comparison of experiment 2, 5 and 8

	MUT	Results		
		FC (%)	**Test length**	**Simulation time (s)**
Experiment 2 (M3)	PPS_EX	93.82	338	0.009
	ADD	96.58	28	0.003
	MULT 1	94.51	310	0.008
	MULT 0	96.52	310	0.006
Experiment 5 (M3 + M4)	PPS_EX	96.39	408	0.013
	ADD	99.56	98	0.006
	MULT 1	98.29	380	0.011
	MULT 0	98.92	380	0.009
Experiment 8 (M3 + M1)	PPS_EX	98.66	504	0.040
	ADD	99.96	194	0.017
	MULT 1	99.42	476	0.033
	MULT 0	99.67	476	0.029

A very similar result was obtained in section 5.3. The trade-off of time versus quality is the same, however, we can observe that FC% of experiment 8 and 9 are the same – 98.66%. This means that the combination of the control pattern with either dedicated PET or all PET provides the same result. The question about this observation is, could 98.66% be the best FC in the PPS_EX module, since the control and PET patterns are both covering the full non-terminal and terminal nodes?

5.6 Investigations #5

The control and random test data were combined in experiment 7. This will be compared to the FC obtained from using only the control test data.

Table 15: Comparison of experiment 4 and 7

<table>
<tr><th rowspan="2"></th><th rowspan="2">MUT</th><th colspan="3">Results</th></tr>
<tr><th>FC (%)</th><th>Test length</th><th>Simulation time (s)</th></tr>
<tr><td rowspan="4">Experiment 4 (M1)</td><td>PPS_EX</td><td>98.35</td><td rowspan="4">166</td><td>0.036</td></tr>
<tr><td>ADD</td><td>99.96</td><td>0.016</td></tr>
<tr><td>MULT 1</td><td>99.37</td><td>0.029</td></tr>
<tr><td>MULT 0</td><td>99.33</td><td>0.031</td></tr>
<tr><td rowspan="4">Experiment 7 (M1 + M4)</td><td>PPS_EX</td><td>98.57</td><td>266</td><td>0.042</td></tr>
<tr><td>ADD</td><td>99.96</td><td>266</td><td>0.017</td></tr>
<tr><td>MULT 1</td><td>99.45</td><td>266</td><td>0.034</td></tr>
<tr><td>MULT 0</td><td>99.4</td><td>266</td><td>0.031</td></tr>
</table>

This comparison was evaluated to observe the impact of the 100 random data when combine with the control test. The contribution of the random patterns is to the minimal, with an increase of 0.22 % in the PPS_EX module. The test length in experiment 7 is significantly higher than in experiment 4, hence, it is safe to conclude that the impact of 100 random patterns on the 166-control data used as conformity test, is inversely proportional to the level of increase of FC in experiment 7.

5.7 Investigations #6

The experiments 8, 9, 10 and 11 show the contribution of the Random Test to the two versions of full deterministic tests consisting of the control test and either Dedicated PET (experiments 9 and 10) or All PET (experiments 8 and 11) tests.

Table 16: Comparison of experiments 8, 9, 10 and 11

		Results		
	MUT	**FC (%)**	**Test length**	**Simulation time (s)**
Experiment 8 (M1 + M3)	PPS_EX	98.66	504	0.040
	ADD	99.96	194	0.017
	MULT 1	99.42	476	0.033
	MULT 0	99.67	476	0.029
Experiment 9 (M1 + M2)	PPS_EX	98.66	504	0.040
	ADD	99.96	175	0.017
	MULT 1	99.42	476	0.032
	MULT 0	99.67	476	0.029
Experiment 10 (M1 + M2 + M4)	PPS_EX	98.7	549	0.042
	ADD	99.96	220	0.017
	MULT 1	99.46	521	0.034
	MULT 0	99.7	521	0.031
Experiment 11 (M1 + M3 + M4)	PPS_EX	98.69	531	0.042
	ADD	99.96	221	0.017
	MULT 1	99.45	503	0.034
	MULT 0	99.7	503	0.030

In 3.2.1 the process of generating deterministic test patterns can be very extensive, and before fault simulation is carried out to detect other faults, the initial detected faults are noted. Experiment 8 and 9 are full deterministic tests with the control patterns and PET patterns. The question posed in investigation 5 concerning 98.66% as the maximum FC was posed, and the theory is negated in Table 16. Notably, random patterns added 0.04 % in experiment 10 and 0.03% in experiment 11. The contribution of random patterns in

experiments 10 and 11 is very minute and we can deduce that its efficiency is to be questioned.

Experiment 10 had 166 control patterns, dedicated PET patterns and 45 random patterns, however, experiment 11 had 166 control patterns, 28 + 310 all PET patterns and 27 random patterns. The effect of the larger random patterns in experiment 10 is visible with an FC of 0.01% more than experiment 11. Random patterns increase the test length and the FC calculation time but does not pose a strong contribution to the FC %.

5.8 Investigations #7

The experiments 3, 8 and 9 show the comparison of the deterministic high-level implementation-independent approach vs. pure random approach (trade-off problem)

Table 17: Observation of experiment 3, 8 and 9

	MUT	Results		
		FC (%)	Test length	Simulation time (s)
Experiment 3 (M4)	PPS_EX	93.82	338	0.009
	ADD	96.58	28	0.003
	MULT 1	94.51	310	0.008
	MULT 0	96.52	310	0.006
Experiment 8 (M1 + M3)	PPS_EX	98.66	504	0.040
	ADD	99.96	194	0.017
	MULT 1	99.42	476	0.033
	MULT 0	99.67	476	0.029
Experiment 9 (M1 + M2)	PPS_EX	98.66	504	0.040
	ADD	99.96	175	0.017
	MULT 1	99.42	476	0.032
	MULT 0	99.67	476	0.029

The trade-off between M4 and either M1 + M3, or M1 + M2 is the FC percentage if we consider the execute module (PPS_EX) of the miniMIPS processor. The MUT in consideration is the execute module (PPS_EX) of the miniMIPS processor. The time taken to complete experiment 3 is more than 2 times lesser than experiment 8 and 9. This is particularly due to the complete conformity and scanning test performed in experiment 8 and 9. The results in Table 17 supports the claim that random test is good enough, but time is a trade-off for a better quality FC result in experiment 8 and 9. The combination of the control test and the PET test provided the best FC result of 98.66%, however, the dedicated PET test was more effective as a lesser time (8441.24s fault simulation time in

experiment 9 versus 8531.83s in experiment 8) was used to calculate the FC to get the same result of 98.66%.

An observation to take note of is the effect of the random test on the dedicated PET and control test. This can be evaluated in experiments 6 and 7.

Table 18: Observation of experiments 6 and 7

	MUT	Results		
		FC (%)	Test length	Simulation time (s)
Experiment 6 (M2 + M4)	PPS_EX	96.48	428	0.011
	ADD	99.56	99	0.010
	MULT 1	98.56	400	0.010
	MULT 0	98.5	400	0.008
Experiment 7 (M1 + M4)	PPS_EX	98.57	266	0.042
	ADD	99.96	266	0.017
	MULT 1	99.45	266	0.034
	MULT 0	99.4	266	0.031

As compared to the results in investigation 1, for M2 and M1, the random data had a huge impact on the dedicated PET data in experiment 6. The increase between experiment 1 (dedicated PET) and experiment 6 is 3.43% and there is a significant time difference due to the additional 90 random patterns.

Table 19: Comparison for the significance of Random data

	MUT	Results			
		FC (%)	Test length	Fault simulation time (s)	Simulation time (s)
Experiment 1 (M2)	PPS_EX	93.05	338	1155.24	0.005
	ADD	90.62	9	8.74	0.003
	MULT 1	94.51	310	469.29	0.005
	MULT 0	96.52	310	302.11	0.003
Experiment 6 (M2 + M4)	PPS_EX	96.48	428	2407.36	0.011
	ADD	99.56	99	25.6	0.010
	MULT 1	98.56	400	921.48	0.010
	MULT 0	98.5	400	721.5	0.008
Experiment 4 (M1)	PPS_EX	98.35	166	7703.38	0.036
	ADD	99.96		40.52	0.016
	MULT 1	99.37		2826.6	0.029
	MULT 0	99.33		2825.95	0.031
Experiment 7 (M1 + M4)	PPS_EX	98.57	266	9002.67	0.042
	ADD	99.96	266	42.48	0.017
	MULT 1	99.45	266	3251.83	0.034
	MULT 0	99.4	266	2861.73	0.031

From Table 19, it becomes evident that in the PPS_EX module, the control test is dominant, and the random data contributes a minute improvement to the FC in experiment 7. The time difference in experiment 7 makes the effect of the random patterns almost negligible if compared to the proportion of FC increase. The FC in experiment 1 increased by increased by 3.68% and fault simulation time increased by 108%. On the contrary, the FC in experiment 4 increased by 0.22% and time by 16.9%.

It is worth noting that the maximum FC achieved for the selected MUTs during our experiment, are 98.7% for PPS_EX, 99.96% for ADD, 99.46% for MULT 1 and 99.7% for MULT 0 module. In the experiments where the FC % for the ADD module is 99.96%, we observed that there is always 1 undetectable fault, hence, our inability to reach 100% FC in the ADD module of the miniMIPS processor.

6 Conclusion

This thesis focused on developing different combinations of test structures and test data for microprocessor software based self-testing. The test objective was Execute module of the MIPS microprocessor partitioned into three sub-modules: ALU (arithmetic and logic operations), MULT1 and MULT2 (multiplication operations). The Execute module was partitioned also into two parts: control part and data part. The test program was developed in two parts: for testing the control part (conformity test), and for testing the data part (scanning test). We demonstrated how to generate the different test data – Dedicated PET, all PET, control test and random test data. Test templates were used to organize the test program from the HLDD synthesis.

The contribution of this thesis is to propose the best method or combination of methods to be used in SBST, while ensuring high quality test at a minimal cost.

The goal of the research was actualized as we observed that the dedicated PET test in combination with the control test is recommended for the best FC and time effectiveness. However, when the dedicated PET and all PET test are performed individually, all PET test provides a better FC but at a huge time cost. Additionally, the random patterns served a purpose of improving the quality of the PET test by reducing the possible deficiency during scanning test, while its contribution to the control test is minimal and can be negligible. Our experimental research also revealed a new discovery that there is always one undetectable fault in the ADD module of the miniMIPS processor, hence, 100% FC is not achievable in this module.

References

[1] A. S. Oyeniran, U. E. Odozi and R. Ubar, “A New Measure for Calculating Multiple Fault Coverage of Microprocessor Self-Test,” *15th Biennial Baltic Electronics Conference,* pp. 75-78, 2016.

[2] A. Jasnetski, R. Ubar, A. Tsertov and M. Brik, “Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams,” *Proceedings of the Estonian Academy of Sciences,* vol. 63, no. 1, pp. 48-61, 2014.

[3] S. M. Thatte and J. A. Abraham, “Test Generation for Microprocessors,” *IEEE Transactions on Computers,* no. 29, pp. 429 - 411, 1980.

[4] R. Ubar, A. Jasnetski, A. Tsertov and A. S. Oyeniran, Software-Based Self-Test with Decision Diagrams for Microprocessors, Lambert Academic Publishing, 2018.

[5] G. E. Moore, “Cramming More Components Onto Integrated Circuits, Electronics,” *Electronics,* vol. 38, no. 8, pp. 82-85, 1965.

[6] D. House, G. E. Moore and I. T. Roadmap, “Moore's law,” 2015.

[7] S. Holst and H. J. Wunderlich, “Adaptice debug and diagnosis without fault dictionaries,” *J. Electron Test,* vol. 25, no. 4-5, pp. 259-268, 2009.

[8] N. Burgess, R. I. Damper, S. J. Shaw and D. R. J. Wikins, “Faults and fault effects in NMOS circuits-impact on design for testability,” *Electron. Circuit System. IEEE Proc. G,* vol. 132, no. 3, pp. 82-89, 1985.

[9] O. O. Medaiyese, A Method for Synthesis of Self-Test Software for Microprocessors, Tallinn University of Technology, 2018.

[10] T. Bengtsson and S. Kumar, “A Survey of High-Level Test Generation Methodologies and Fault Models,” School of Engineering Jönköping University.

[11] W. Laung-Terng, W. Cheng-Wen and X. Wen, “VLSI Test Principles and Architectures,” in *Design for Testability*, San Francisco, 2006.

[12] U. E. Odozi, “High-Level Synthesis and Analysis of Test Data for Software Based Self-Test in Microprocessors,” 2016.

[13] O. Novak, E. Gramatova and R. Ubar, “Handbook of Testing Electronic Systems,” 2005.

[14] Z. Navabi, “Digital System Test and Testable Design: Using HDL Models,” LLC 2011.

[15] V. S. Bagad, VLSI Design, Technical Publications Pune, 2008.

[16] L.-T. Wang, Y.-W. Chang and K.-T. Cheng, Electronic Design Automation, 2009.

[17] J. C.-M. Li and M. S. Hsiao, Electronic Design Automation, 2009.

[18] G. Sudhagar and S. S. Kumar, “VLSI Design of Efficient Architecture in Recursive Pseudo-Exhaustive Two-Pattern Generation,” *Journal of Theoretical and Applied Information Technology,* 2013.

[19] M. Abramovici, M. A. Breuer and A. D. Friedman, Digital Systems Testing and Testable Design, IEEE Press, 1999.

[20] R. Ubar, “Lecture slide on Built-In-Self-Test”.

[21] A. Jassnetski, A. S. Oyeniran, A. Tsertov, M. Schölzel and R. Ubar, “High-Level Modelling and Testing of Multiple Control Faults in Digital Systems,” in *Proc. of DDECS*, April 20-22, 2016.

[22] S. M. Thatte and J. A. Abraham, “Test Generation for Microprocessors,” *IEEE Trans. On Computers,* no. 6, pp. 429-441, June 1980.

[23] D. Brahme and J. A. Abraham, “Funtional Testing of Micro-Processors,” *IEEE Trans. on Comp,* no. 6, pp. 4755-485, 1984.

[24] A. S. Oyeniran, A. Jasnetski, A. Tsertov and R. Ubar, “High-Level Dta Generation for Software-Based Self-Test in Microprocessors,” in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, Bar, Montenegro, 11-15 June 2017.

[25] A. S. Oyeniran, R. Ubar, S. P. Azad and J. Raik, “High-Level Test Generation for Processing Elements in Many-Core Systems,” 2017.

[26] S. A. Oyeniran, P. S. Azad and R. Ubar, “Parallel Pseudo-Exhaustive Testing of Array Multipliers with Data-Controlled Segmentation,” 2018.

[27] D. P. Siewiorek and L. K.-W. Lai, “Testing of Digital Systems,” *Proceeding of the IEEE,* vol. 69, no. 10, pp. 1321-1333, October 1981.

[28] V. Agrwal and M. Bushnell, Essentials of Electronics Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Boston: Kluwer Academic Publishers, 2000.

[29] R. J. Tocci, N. S. Widmer and G. L. Moss, Digital Systems Principles and Applications, 10 ed., Pearson Education International, 1977.

[30] OpenCores, “MiniMIPS ISA”.

[31] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3 ed., San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007.

[32] “Laboratory Exercise 1: Preliminary Fault Simulation Concepts,” October 2015.

[33] Reason tutorial, BEC. (2002, October 9). Defect Oriented Test. Tallinn, Estonia.

Appendix 1 – Program Description and Manual

This section describes how our experiments were performed. It contains a step-by-step illustration of how the test program is generated and FC calculation for each experiment. Each experiment has its folder and they have similar steps.

1. The folders and files for the experiments can be downloaded through this link: https://github.com/Coded99/Test_Data_Generation_for_SBST_of_Microprocessors

2. Linux operating system is recommended to perform the experiment. There are 11 folders representing experiments 1 – 11. Each of the folder contains the Test program generator, fault coverage calculation and other folders for compiling the miniMIPS processor.

3. Open the **Test Program Generator** folder, located here are 6 folders and 10 files. For our experiment, only the Test_Program and input folder, clean.sh, compile_minimips.sh, logic_sim.sh, vsim_gui.tcl and tst.src are needed for navigation, other files and folders are dependencies. The description of the needed folders and files are as follows:

I. **Test program folder**: This contains the python script for the test program generation and the applicable template specific to the experiment.

- **Test template**: Depending on the experiment being performed, the applicable template is in this folder. If control test, then the control template (op1_template.py and op2_template.py), if random or PET test, then random (random_template.py) or PET (pseudo_template.py) template are found here.

- **Parameter.txt**: The parameter.txt file dictates the instructions to be included in the test program. These instructions are catalogued depending on the HLDD synthesis specific to the experiment.

- **load_memory.py**: This file is used to load the control test data (data.txt located in the input folder) into the processor's memory.

- **TestProgramGenerator.py**: This is the master file in this folder. It uses all the scripts in the folder to generate the test program.

- **Outputme.py**: This file stores the generated test program temporarily, so that it can be copied to tst.src and used to generate the dumpports_ex.vcd file.

II. **Input folder**: This folder contains the test data to be used for the experiment. For control test, data.txt and branch.txt files are used as inputs. This folder could also contain pseudo_input or random_input folders.

   - **Pseudo_input**: This folder could contain 1 or 5 .txt files. When running an experiment that contains the dedicated PET data, 5 files will be located – add.txt, sub.txt, logic.txt, shift.txt, and mult.txt. Otherwise, only 1 file named all.txt will be located for an experiment with all PET data.

   - **Random_input**: This folder contains the random data to be used for an experiment that uses the random data.

For clarity, the purpose of the following files will be described as we proceed with the manual - clean.sh, compile_minimips.sh , logic_sim.sh, vsim_gui.tcl and tst.src

4. Navigate to the test program folder and open via linux terminal. In order to generate the test program, type in the following command: `python TestProgramGenerator.py`. The script will generate an output file as seen in Figure 28. If the terminal does not report an error, it means the program was executed successfully.

```
adosid@lx33:~/miniMIPS_Test_Program/TG/Test_program> python TestProgramGenerator.py
...........parameters............
iterator = 27
pattern_count = 28
store_result_address = 29
test_pattern_address = 30
result_register = 18
jump_address = 25
branch_count = 26
source_register1 = 15
source_register2 = 16
...................................
```

Figure 28: Test program generator response from Linux terminal

The test program is automatically stored in the outputme.txt file. Open this file and copy the content into the tst.src file located in the Test_Program_Generator folder.

5. After the test program is pasted into tst.src, the command `./clean.sh` is executed in the Test_Program_Generator folder, in order to clean the miniMIPS folder by removing any .bin, .lst, .vcd and .wlf files, which were created by the assembler. These files could be present in the folder as a result of a previously executed test program.

6. Go back to the terminal of the linux OS and enable the logic simulator environment – ModelSim. I used a computer in the TalTech University laboratory which has the logic simulator installed. From the terminal, enter the `CAD` command twice (once per instance). A list of the installed `CAD` software will be seen on the screen. To enable the ModelSim environment, enter command “2”.

7. Next is to compile the miniMIPS HDL from the terminal. The complile_minimips.sh file is to be used for this purpose. Once entered in the terminal with `./compile_minimips.sh`, the miniMIPS HDL is compiled.

8. The test program in tst.src is in assembly code and needs to be converted into a binary code for the miniMIPS processor. Enter the command `./logic_sim.sh` for this purpose.

   The ModelSim environment will be opened after a few seconds and a waveform as in Figure 29 is displayed. Depending on the amount of test data or combination of methods being used for the particular experiment, it might take up to 5 minutes for ModelSim to finalize the execution of the test program.

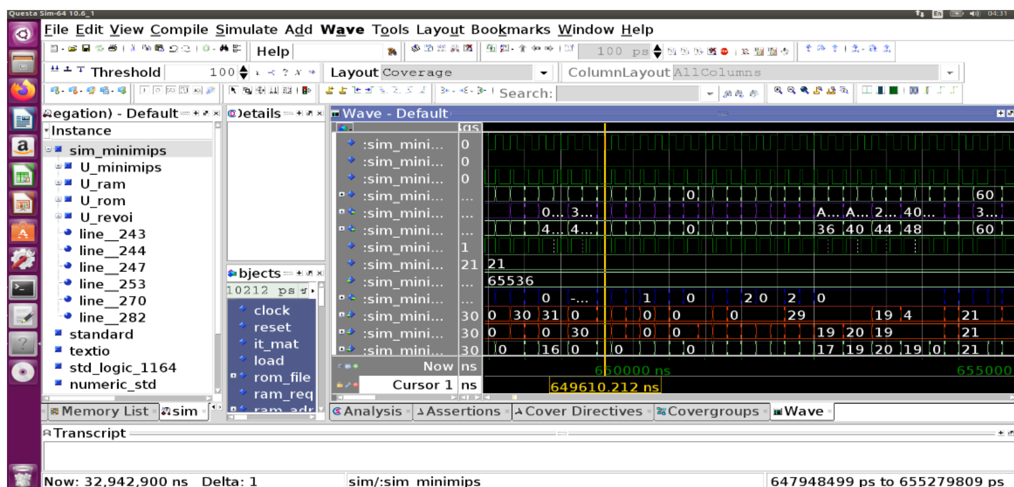


Figure 29: Generating dump file for fault coverage calculation

After the execution of the test program is completed, ModelSim window is automatically closed as specified in the ./logic_sim.sh file.

9. The previous step generated various .vcd files, rom.bin, rom.lst, out.txt, and transcript. Since the focus of our experiments was on the execute module of the miniMIPS processor, we will use only the dummports_ex.vcd file. This file contains the primary input and output values of the executed instructions and is used for the fault coverage calculation.

10. Copy the dummports_ex.vcd file and paste is in the Fault_coverage_calculator folder. The folder should contain 3 files and 1 folder after the dummports_ex.vcd file has been pasted. The folder is named gate_level and the files are dummports_ex.vcd, run_tst_fsim.sh and tmax.tcl.

11. Tmax.tcl is TetraMAX script to be executed for the fault coverage calculation. This file contains a command to induce the expected faults in the PPS_EX MUT of the miniMIPS processor. This means that all possible faults in the module will be created, including not detected faults [32]. Navigate to the Fault_coverage_calculator folder via the terminal in order to configure the environment for TetraMAX.

12. Once again, enter the command `CAD`. If the terminal was not closed from step 4 until now, the command "3" should be entered to configure the environment for TetraMAX. However, if the terminal was previously closed, then `CAD` will be entered twice before the command "3".

13. Enter the command `./run_tst_fsim.sh` to commence the FC calculation. This process can take up to 4 hours, depending on the size of the MUT and the test program.

14. The result of the FC calculation will be displayed on the terminal or found in a file named report.txt, with the details of the time taken for FC calculation.

Appendix 2 – Structure of the miniMIPS processor

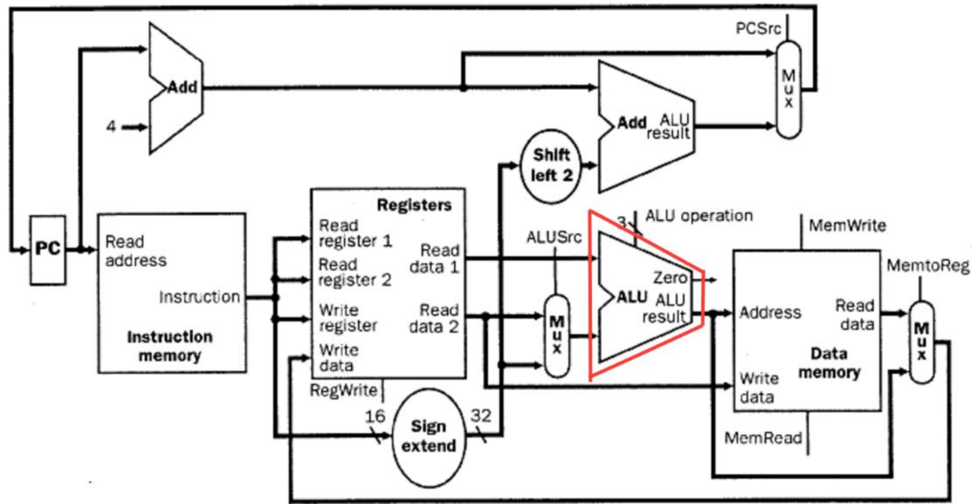


Figure 30: Structure of the miniMIPS processor

Appendix 3 – CPU specification for the experiments

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 4

On-line CPU(s) list: 0-3

Thread(s) per core: 1

Core(s) per socket: 4

Socket(s): 1

NUMA node(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 158

Model name: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz

Stepping: 9

CPU MHz: 3703.725

CPU max MHz: 3800.0000

CPU min MHz: 800.0000

BogoMIPS: 6815.85

Virtualization: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 6144K

NUMA node0 CPU(s): 0-3

Appendix 4 – Source Codes

A Pseudo_template.py

```
def pseudo_template(inputFile, out, instruct, result_register):
  f = open(inputFile, 'r')  # input file

  instruction = instruct

  out.write("jal reset_offsets\n")
  if(instruction[0:2] == "mf" or instruction[0:2] == "mt"):
      out.write("jal reset_hi_lo\n")
  if(instruction[0:] == "add" or instruction[0:] == "sub"):
      out.write("jal init_cp\n")
  out.write("operation_"+instruction+"_psuedo:\n")
  offset = 0
  register = 2
  outline = []

  line = f.readlines()

  def load_data(i, register):
  # selection by 16 bits
      bit_set1 = i[:16]
      bit_set2 = i[16:32]
      bit_set3 = i[32:48]
      bit_set4 = i[48:68]
      for x in range(2):
         if (x == 0):
             most_sig_bit = int(bit_set1, 2)
             least_sig_bit = int(bit_set2, 2)
         else:
             most_sig_bit = int(bit_set3, 2)
             least_sig_bit = int(bit_set4, 2)
         out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
         out.write("\tori $%d, $%d, %d\n" %
                    (register, register, least_sig_bit))
         if (register >= 3):
             register = 2
         else:
             register += 1

  def load_data_immediate(i, register, opcode):
      bit_set1 = i[:16]
      bit_set2 = i[16:32]
      bit_set3 = i[32:48]
      bit_set4 = i[48:68]
      for x in range(2):
         if (x == 0):
             most_sig_bit = int(bit_set1, 2)
             least_sig_bit = int(bit_set2, 2)
             out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
             out.write("\tori $%d, $%d, %d\n" %
                        (register, register, least_sig_bit))
         else:
             most_sig_bit = int(bit_set3, 2)
             least_sig_bit = int(bit_set4, 2)
             out.write("\t%s $%d, $%d, %d\n" %
```

```
                       (opcode,      result_register,      register+1,
least_sig_bit))
         if (register >= 2):
             register = 1
         else:
             register += 1

  def load_data_shift(i, register, opcode):
      bit_set1 = i[:16]
      bit_set2 = i[16:32]
      bit_set3 = i[32:48]
      bit_set4 = i[48:68]
      shift_amount = 5
      for x in range(2):
         if (x == 0):
             most_sig_bit = int(bit_set1, 2)
             least_sig_bit = int(bit_set2, 2)
         else:
          if(opcode == "lui"):
              out.write("\tlui $%d, %d\n" % (register+1, most_sig_bit))
          elif(str(opcode[0:2]) == "mf" or str(opcode[0:2]) == "mt"):
              out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
              out.write("\tori $%d, $%d, %d\n" %
                        (register, register, least_sig_bit))
              out.write("\t%s $%d\n" % (opcode, register))
          else:
              most_sig_bit = int(bit_set3, 2)
              least_sig_bit = int(bit_set4, 2)
              out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
              out.write("\tori $%d, $%d, %d\n" %
                        (register, register, least_sig_bit))
              out.write("\t%s $%s, $%d, %d\n" %
                        (opcode,      result_register,      register,
shift_amount))
         if (register >= 2):
             register = 2
         else:
             register += 1

  def alu_shifts(file_name, offset, opcode):
      for i in (line):
          load_data_shift(i, register, opcode)
          out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
          out.write("\tjal increment\n")
          # offset += 4

  def alu_immediate(file_name, offset, opcode):
      for i in (line):
          load_data_immediate(i, register, opcode)
          out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
          out.write("\tjal increment\n")
          # offset += 4

  def alu_op(file_name, offset):
      for i in (line):
          load_data(i, register)
          out.write("\t%s $%s, $%d, $%d\n" %
                    (instruction,      result_register,      register,
register+1))
          out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
          out.write("\tjal increment\n")
```

```
          # offset += 4

  def mult_op(file_name, offset):
      for i in (line):
          load_data(i, register)
          out.write("\t%s   $%d,   $%d\n"   %   (instruction,   register,
register+1))
          out.write("\tmflo $%s\n" % (result_register))
          out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
          out.write("\tmfhi $%s\n" % (result_register))
          out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
          out.write("\tjal increment\n")
          # offset += 4

  def hi_lo(file_name, offset, opcode):
      for i in (line):
          load_data_shift(i, register, opcode)
          if(opcode[2:4] == "hi"):
              out.write("\tmfhi $%d\n" % (register+2))
          else:
              out.write("\tmflo $%d\n" % (register+2))
             out.write("\tsw $%d, %d($29)\n" % (register+2, offset))
          out.write("\tjal increment\n")
          # offset += 4


  if ((instruction =="mult") or (instruction =="multu")):
      mult_op(f, offset)
  elif((instruction =="addiu") or (instruction =="addi") or (instruction
=="andi")or   (instruction   =="ori")or   (instruction   =="xori")   or
(instruction =="sltiu")or (instruction =="slti")):
      alu_immediate(f, offset, instruction)
  elif((instruction  =="sll")  or  (instruction  =="sra")or  (instruction
=="srl")or (instruction =="lui")):
      alu_shifts(f,offset,instruction)
  elif((instruction == "mtlo") or (instruction =="mthi")):
      hi_lo(f,offset,instruction)
  else:
      alu_op(f,offset)

  f.close()

def make_pseudo_template(para, outputFile, result_register):
  Ins = open(para,'r')
  data_lines = []
  firstPass = True

  for line in Ins:
    if "=" not in line:
      try:
        catergory = line[0:2]
        line = line.rstrip()
        instru = line[2:]
        instr = str.strip(instru)
        instruction = instr[0:]
        if (catergory == 'p_'):
          if  (instruction  ==  'addu'  or  instruction  ==  'add'  or
instruction == 'addi' or instruction == 'addiu'):
            inputFile = '../input/pseudo_input/add.txt'
            pseudo_template(inputFile,    outputFile,    instruction,
result_register)
```

```
            outputFile.write("\n")
          elif(instruction == 'subu'or instruction == 'sub'):
            inputFile = '../input/pseudo_input/sub.txt'
            pseudo_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
          elif(instruction   ==   'or'   or   instruction   ==   'xor'   or
instruction == 'nor' or instruction == 'and' or instruction == 'andi'
or instruction == 'ori' or instruction == 'xori'):
            inputFile = '../input/pseudo_input/logic.txt'
            pseudo_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
          elif(instruction   ==   'sll'   or   instruction   ==   'srl'   or
instruction == 'sra' or instruction == 'srav' or instruction == 'slt'
or instruction == 'sltu' or instruction == 'slti' or instruction ==
'sltiu'):
            inputFile = '../input/pseudo_input/shift.txt'
            pseudo_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
          elif(instruction == 'mult' or instruction == 'multu'):
            inputFile = '../input/pseudo_input/mult.txt'
            pseudo_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
      except IndexError:
        firstPass = False
    else:
      do='nothing'

  Ins.close()
```

B Random_Template.py

```
def random_template(inputFile, out, instruct, result_register):
  f = open(inputFile,'r')          #input file

  instruction = instruct

  out.write("jal reset_offsets\n")
  if(instruction[0:2] =="mf" or instruction[0:2] =="mt"):
      out.write("jal reset_hi_lo\n")
  if(instruction[0:] =="add" or instruction[0:] =="sub"):
      out.write("jal init_cp\n")
  #out.write("operation_"+instruction+":\n")
  out.write("operation_"+instruction+"_random:\n")
  offset = 0
  register = 2
  outline = []

  line=f.readlines()

  def load_data(i, register):
  #selection by 16 bits
      bit_set1 = i[:16]
      bit_set2 = i[16:32]
      bit_set3 = i[32:48]
      bit_set4 = i[48:68]
      for x in range(2):
         if (x == 0):
             most_sig_bit   = int(bit_set1,2)
             least_sig_bit  = int(bit_set2,2)
         else:
             most_sig_bit   = int(bit_set3,2)
             least_sig_bit  = int(bit_set4,2)
         out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
         out.write("\tori  $%d,  $%d,  %d\n"  %  (register,  register,
least_sig_bit))
         if (register >= 3):
             register = 2
         else:
             register += 1

  def load_data_immediate(i, register, opcode):
      bit_set1 = i[:16]
      bit_set2 = i[16:32]
      bit_set3 = i[32:48]
      bit_set4 = i[48:68]
      for x in range(2):
         if (x == 0):
             most_sig_bit   = int(bit_set1,2)
             least_sig_bit  = int(bit_set2,2)
             out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
             out.write("\tori  $%d,  $%d,  %d\n"  %  (register,  register,
least_sig_bit))
         else:
             most_sig_bit   = int(bit_set3,2)
             least_sig_bit  = int(bit_set4,2)
             out.write("\t%s $%s, $%d, %d\n" % (opcode, result_register,
register+1, least_sig_bit))
         if (register >= 2):
             register = 1
         else:
```

```
            register += 1

    def load_data_shift(i, register, opcode):
        bit_set1 = i[:16]
        bit_set2 = i[16:32]
        bit_set3 = i[32:48]
        bit_set4 = i[48:68]
        shift_amount = 5
        for x in range(2):
           if (x == 0):
               most_sig_bit   = int(bit_set1,2)
               least_sig_bit  = int(bit_set2,2)
           else:
            if(opcode =="lui"):
                out.write("\tlui $%d, %d\n" % (register+1, most_sig_bit))
            elif(str(opcode[0:2]) =="mf" or str(opcode[0:2]) =="mt"):
                out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
                out.write("\tori $%d, $%d, %d\n" % (register, register,
least_sig_bit))
                out.write("\t%s $%d\n" % (opcode, register))
            else:
                most_sig_bit   = int(bit_set3,2)
                least_sig_bit  = int(bit_set4,2)
                out.write("\tlui $%d, %d\n" % (register, most_sig_bit))
                out.write("\tori $%d, $%d, %d\n" % (register, register,
least_sig_bit))
                out.write("\t%s    $%s,    $%d,    %d\n"    %    (opcode,
result_register, register, shift_amount))
           if (register >= 2):
               register = 2
           else:
               register += 1

    def alu_shifts(file_name, offset, opcode):
        for i in (line):
            load_data_shift(i, register, opcode)
            out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
            out.write("\tjal increment\n")
            #offset += 4

    def alu_immediate(file_name, offset, opcode):
        for i in (line):
            load_data_immediate(i, register, opcode)
            out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
            out.write("\tjal increment\n")
            #offset += 4

    def alu_op(file_name, offset):
        for i in (line):
            load_data(i, register)
            out.write("\t%s    $%s,    $%d,    $%d\n"    %    (instruction,
result_register, register, register+1))
            out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
            out.write("\tjal increment\n")
            #offset += 4

    def mult_op(file_name, offset):
        for i in (line):
            load_data(i, register)
            out.write("\t%s   $%d,   $%d\n"   %   (instruction,   register,
register+1))
```

```
            out.write("\tmflo $%s\n" % (result_register))
            out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
            out.write("\tmfhi $%s\n" % (result_register))
            out.write("\tsw $%s, %d($29)\n" % (result_register, offset))
            out.write("\tjal increment\n")
            #offset += 4

    def hi_lo(file_name, offset,opcode):
        for i in (line):
            load_data_shift(i, register,opcode)
            if(opcode[2:4]=="hi"):
                out.write("\tmfhi $%d\n" % (register+2))
            else:
                out.write("\tmflo $%d\n" % (register+2))
            out.write("\tsw $%d, %d($29)\n" % (register+2, offset))
            out.write("\tjal increment\n")
            #offset += 4

    if ((instruction =="mult") or (instruction =="multu")):
        mult_op(f, offset)
    elif((instruction =="addiu") or (instruction =="addi") or (instruction
=="andi")or   (instruction   =="ori")or   (instruction   =="xori")   or
(instruction =="sltiu")or (instruction =="slti")):
        alu_immediate(f, offset, instruction)
    elif((instruction  =="sll")  or  (instruction  =="sra")or  (instruction
=="srl")or (instruction =="lui")):
        alu_shifts(f,offset,instruction)
    elif((instruction == "mtlo") or (instruction =="mthi")):
        hi_lo(f,offset,instruction)
    else:
        alu_op(f,offset)

    f.close()
def make_random_template(para, outputFile, result_register):
    Ins = open(para,'r')
    data_lines = []
    firstPass = True

    for line in Ins:
      if "=" not in line:
        try:
          catergory = line[0:2]
          line = line.rstrip()
          instru = line[2:]
          instr = str.strip(instru)
          instruction = instr[0:]
          if (catergory == 'r_'):
            if  (instruction  ==  'addu'  or  instruction  ==  'add'  or
instruction == 'addi' or instruction == 'addiu'):
              inputFile = '../input/random_input/random.txt'
              random_template(inputFile,     outputFile,     instruction,
result_register)
              outputFile.write("\n")
            elif(instruction == 'subu'or instruction == 'sub'):
              inputFile = '../input/random_input/random.txt'
              random_template(inputFile,     outputFile,     instruction,
result_register)
              outputFile.write("\n")
            elif(instruction   ==   'or'   or   instruction   ==   'xor'   or
instruction == 'nor' or instruction == 'and' or instruction == 'andi'
or instruction == 'ori' or instruction == 'xori'):
```

```
            inputFile = '../input/random_input/random.txt'
            random_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
          elif(instruction   ==   'sll'   or   instruction   ==   'srl'   or
instruction == 'sra' or instruction == 'srav' or instruction == 'slt'
or instruction == 'sltu' or instruction == 'slti' or instruction ==
'sltiu'):
            inputFile = '../input/random_input/random.txt'
            random_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")
          elif(instruction == 'mult' or instruction == 'multu'):
            inputFile = '../input/random_input/random.txt'
            random_template(inputFile,     outputFile,     instruction,
result_register)
            outputFile.write("\n")

      except IndexError:
        firstPass = False
    else:
      do='nothing'

 Ins.close()
```

C Random_data_generator.py

```
import random
import os

random.seed(10110010011010101100101100101001)
# function that generates random binary number

def randbin2(d):
    mx = (2 ** d) - 1
    # for counter in range(1,lenght+1):
    while True:
        b = bin(random.randint(0, mx))
        return b[2:].rjust(d, '0')

# create/open text file and write data into it.
f = open("Data_75.txt", 'w+')

for i in range(0, 75):
    f.write(randbin2(64))
    f.write("\n")

f.close()
```

D Parameter.txt

```
;parameters: Define parameter for test program generation.

iterator=27
pattern_count=28
branch_count=26
result_address=29
pattern_address=30
result_register=18
source_register1=15
source_register2=16
jump_address=25
shift_amount=5

;for testing OP1
y a_add
n a_addu
n a_and
n a _nor
n a_or
n a_subu
y a_sub
n a_xor
n a_sllv
n a_slt
n a_sltu
n a_srav
n a_srlv
n a_sll
n a_srl
n a_sra
;n a_mult
;n a_multu
;n a_mfhi
;n a_mflo
;n a_mthi
;n a_mtlo

;for testing HILO
n b_mult
n b_multu
n c_mthi
n c_mtlo

; for immediate
y i_addi
n i_addiu
n i_andi
n i_ori
n i_slti
n i_sltiu
n i_xori

#for coprocssor writing
n 1_mtc0
n 1_mfc0

#for load and store
n 2_lw_sw
```

```
# For co_processor load and store -- modify
n 3_lw0_sw0

#pseudo-exhaustive data
p_add
p_sub
p_addu
p_subu
p_and
p_or
p_xor
p_nor
p_sll
p_srl
p_srlv
p_sra
p_slt
p_sltu
p_mult
p_multu
p_addi
p_addiu
p_andi
p_ori
p_slti
p_sltiu
p_xori
p_srav

#Random data
r_add
r_sub
r_addu
r_subu
r_and
r_or
r_xor
r_nor
r_sll
r_srl
r_srlv
r_sra
r_slt
r_sltu
r_mult
r_multu
r_equ
r_nequ
r_beq
r_bne
r_bgez
r_bgezal
r_bgtz
r_blez
r_bltz
r_bltzal
r_addi
r_addiu
r_andi
r_ori
r_slti
```

```
r_sltiu
r_xori
r_srav

;Branches
n d_beq
n d_bne
n e_bgez
n e_bgezal
n e_bgtz
n e_blez
n e_bltz
n e_bltzal
```

E TestProgramGenerator.py

```
import reset
import load_memory
import op2_template
import op1_template
import pseudo_template
import random_template
import register_test
import pipeline
import op2_template_optimized
import op1_template_optimized

#test data file
parameter = "parameter.txt"
l = open(parameter,'r')
out = open('outputme.txt', 'w')


#interrupt
reset.interrupt_function(out)

#reset registers
reset.reset_function(out)

iterator = 0
pattern_count = 0
result_address = 0
pattern_address = 0
result_register = 0
shift_amount = 0
jump_address = 0
branch_count = 0
source_register1 = 0
source_register2 = 0

# fixes the parameter used for the program
print "...........parameters............"
firstPass = True
for line in l:
      if "=" in line:
            try:
                  check = line.split("=")
                  if (check[0] == 'iterator'):
                        iterator = check[1].rstrip()
                  elif (check[0] == 'pattern_count'):
                        pattern_count = check[1].rstrip()
                  elif (check[0] == 'result_address'):
                        result_address = check[1].rstrip()
                  elif (check[0] == 'pattern_address'):
                        pattern_address = check[1].rstrip()
                  elif (check[0] == 'result_register'):
                        result_register = check[1].rstrip()
                  elif (check[0] == 'shift_amount'):
                        shift_amount = check[1].rstrip()
                  elif (check[0] == 'jump_address'):
                        jump_address = check[1].rstrip()
                  elif (check[0] == 'branch_count'):
                        branch_count = check[1].rstrip()
                  elif (check[0] == 'source_register1'):
                        source_register1 = check[1].rstrip()
```

```
                elif (check[0] == 'source_register2'):
                        source_register2 = check[1].rstrip()
            except IndexError:
                firstPass = False

print 'iterator =', iterator
print 'pattern_count =', pattern_count
print 'store_result_address =', result_address
print 'test_pattern_address =', pattern_address
print 'result_register =', result_register
print 'jump_address =', jump_address
print 'branch_count =', branch_count
print 'source_register1 =', source_register1
print 'source_register2 =', source_register2
print "...................................."

out.write(" main:\n")

out.write(";........other test........;\n")
out.write(";........reset $26 back for branch loops........;\n")
out.write(";........reset $28 back for other test loops........;\n")
out.write(" lui $%s, %d\n" % (pattern_count, 0))
out.write(";........set memory location for signature........;\n")
out.write(" lui $%s, %d\n" % (result_address, 1))
out.write(" ori $%s, $%s, %d\n\n" % (result_address, result_address,
6000))

out.write(" jal reset_offsets\n")


#syscall
pipeline.syscall(out)

##template for psuedo-exhaustive data
out.write(";..........data-path test..........;\n")
out.write(" lui $%s, %d\n" % (result_address, 1))
out.write(" ori $%s, $%s, %d\n\n" % (result_address, result_address,
10000))
pseudo_template.make_pseudo_template(parameter,out, result_register)

##template for random data
out.write(";..........random-data-path test..........;\n")
out.write(" lui $%s, %d\n" % (result_address, 1))
out.write(" ori $%s, $%s, %d\n\n" % (result_address, result_address,
15000))
random_template.make_random_template(parameter,out, result_register)

#break
pipeline.breaks(out)

#pattern loading, reset offset module, increment offset
reset.end_program(out)
reset.load_pattern(out, pattern_address)
reset.reset_offsets(out, pattern_address,iterator,result_register)
reset.increment_offset(out, pattern_address,iterator, result_address)
reset.increment(out, pattern_address,iterator, result_address)
reset.store(out,result_register, result_address)
reset.init_cp(out)

out.write("end:\n")
out.write("\t j end\n")
```

```
out.close()
l.close()
```