TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Tolulope Emmanuel Ademilua
184584 IASM

OPTIMIZATION OF TEST DATA GENERATION FOR SOFTWARE-BASED SELF-TEST PROCESSORS

Master’s thesis

Supervisor: Professor, Raimund-Johanness Ubar

Co. Supervisor: Stephen Adeboye Oyeniran, M.Sc

Tallinn 2020

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 referred to. This thesis has not been presented for examination anywhere else.

Author: Tolulope Emmanuel Ademilua

11.05.2020

Tallinn 2020

Abstract

This paper develops a new paradigm for generating and optimizing test data for software-based tests of processors. The work aims to find the optimal amount of test data for testing processors with RISC architecture, which would ensure wide coverage of high- and low-level faults. The quality of software-based tests on processors depends on the test program, but the quality of the test program depends on the test data. The new concept is based on the separate generation of tests for the control and data processing parts of the processor modules, which are defined by a set of certain processor functions. The generation of test data is based on the division of the set of functions to be tested into groups, where a high-level decision diagram model is used to find the optimal distribution. The optimization criterion is to ensure maximum fault coverage with a minimum amount of test data at the minimum time required to generate tests.

An innovative approach has been developed to minimize the amount of memory required to store tests by minimizing test data. The novelty of the work is expressed in two aspects: (1) the savings of memory required for storing the test by minimizing the test data, and (2) the wider coverage of high- and low-level fault classes achieved by generating high-quality test data.

Experimental studies have been performed with a miniMIPS microprocessor. The experiments were able to demonstrate that the developed method provides high fault coverage at both high and low levels, but with fewer test data than the previous method. The result ensures high reliability and dependability of the processors.

Annotatsioon

Testandmete optimeerimine protsessorite tarkvarapõhistele testidele

Käesolevas töös arendatakse uut paradigmat testandmete genereerimiseks ja optimeerimiseks protsessorite tarkvarapõhistele testidele. Töö eesmärgiks on leida optimaalne testandmete hulk RISC-arhitektuuriga protsessorite testimiseks, mis tagaksid laia kõrg- ja madalatasandi rikete katte. Protsessorite tarkvarapõhiste testide kvaliteet sõltub testprogrammist, aga testprogrammi kvaliteet sõltub omakorda testandmetest. Uus kontseptsioon põhineb testide eraldi genereerimisel protsessori moodulite juht- ja andmetöötlusosadele, mis on defineeritud teatavate protsessori funktsioonide hulgaga. Testandmete genereerimine põhineb testitava funktsioonide hulga jaotamisel gruppideks, kus optimaalse jaotuse leidmiseks kasutatakse kõrgtaseme otsustusdiagrammide mudelit. Optimeerimiskriteeriumiks on tagada maksimaalne rikete kate minimaalse testandmete hulgaga minimaalse testide genereerimiseks kuluva aja juures.

Töös on arendatud uudne lähenemine testide salvestamiseks vajaliku mälumahu minimeerimiseks testandmete kokku pakkimise teel. Töö uudsus väljendub kahes aspektis: (1) testandmete minimeerimise teel saavutatav testi salvestamiseks vajaliku mälumahu kokkuhoid, ja (2) kõrge kvaliteediga testandmete genereerimisel saavutatav laiem kõrg- ja madalataseme rikete klassi kate.

Eksperimentaaluuringud on töös läbiviidud mikroprotsessoriga MiniMIPS. Eksperimentidega õnnestus demonstreerida, et väljatöötatud meetod tagab kõrge rikete katte nii kõrg- kui ka madalal tasandil, kuid seejuures väiksema testandmete hulgaga, kui senise meetodi puhul. Saadud tulemusega on tagatud protsessorite kõrge usaldusväärsus ja töökindlus.

List of abbreviations and terms

ATG	Automated Test Generation
ATPG	Automated Test Pattern Generation
SBST	Software-Based Self-Test
MP	Microprocessor
SAF	Stuck at Fault
BDD	Binary Decision Diagram
SSBDDs	Structurally Synthesized Binary Decision Diagrams
ROBDD	Reduced Ordered Binary Decision Diagram
HLDD	High Level Decision Diagram
CUT	Circuit Under Test
SLL	Shift Word Left Logical
SRL	Shift Word Right Logical
SRA	Shift Word Right Arithmetic
SLLV	Shift Word Left Logical Variable
MFHI	Move from HI Register
MFLO	Move from LO Register
HDL	Hardware Description Language
ATPG	Automated Test Pattern Generator
ISA	Instruction Set Architecture
MUT	Modules Under Test
HL	High Level
S-A-0	Stuck-At-0
S-A-1	Stuck-At-1
RISC	Reduced Instruction Set Computer
RAM	Random Access Memory
MSF	Multiple Stuck-At Fault
CPU	Central Processing Unit

FRF	Fanout-Free Region
SOC	System-On-Chip

Table of contents

List of figures

List of tables

1 Introduction

This thesis addresses how to generate an efficient test data for testing processor. The main emphasis is that the quality of test data determines the quality of the test program used by SBST in testing the processor. Hence, the level of fault coverage in the processor depends on the quality of the test program used. Based on this fact, we developed an optimal method to optimize test data generation by an algorithmic partitioning of the instruction set architecture of the processor under test. A good ordering of instructions set for test data generation is presented. A novel concept of HLDD synthesis was used to generate the test program.

This introductory chapter presents the background and problems of SBST leading to this research, followed by a more detailed objective. Finally, an overview of the thesis structure is provided.

1.1 Background and problem of SBST

The realization of SBST started in 1980 after the semiconductor industry was challenged to develop a novel testing method that can be integrated into MP test flow, due to the increase in technology advances [1]. Today even more technology advances in digital systems with massively parallel computing now exists, which open gaps into more way of reasoning how systems could be tested correctly without delays in system release to the market. 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 [2] [3]. The major problem of SBST when considering high-level faults is the difficulty of proving that the model covers all the low-level detectable faults [4]. The SBST approach is based on software programs that are designed to test the functionality of the processor cores [5]. The key idea of SBST is to make full use of on-chip programmable resources to run normal programs that test the processor itself [6]. RISC processor is a system with a judicious restriction to a small set of often used instructions with an architecture tailored to fast execution of all the instructions in this set [7]. The use of SBST techniques for testing of a modern processor cannot be underestimated because of the ease of synthesis using functional approaches, coverage for difficult to test faults, non-intrusive nature, low hardware overhead [8]. However, the test synthesis time required by SBST is high and it is a problem that relates to the test program used. The test program is an assembly program devised to extract information that reveals the correctness or valid operation of the machine that executes it, rather than calculating a function or performing a task. Test programs may be used to validate the correctness of a processor design or to check the correct functionality of a device after production [9]. The complexity of the present digital systems has rendered gate-level test generation impractical. However, functional testing has been developed as an alternative by researchers [10].

The quality of the SBST is mainly affected by the test data used in test programs [4]. In [11], divide and conquer approaches were considered for modules under test to generate high-quality test data by ATPG. However, the difficulties of this method set from the fact that functional constraints are needed to guide the ATPGs in order to produce functionally feasible test patterns. As stated in [12], the random test patterns are an alternative way to generate test data for MUTs. Researchers in [13] proposed two constraints in generating test data for detecting control faults based on partitioning the instructions of MP on an HLDD. These constraints are control constraints to activate the desired working modes of the processor and the data constraints to test if the selected working modes were correctly selected [13] [14]. This thesis will also make use of the two constraints to generate the test with HLDD. However, we have optimized the process of partitioning the instructions of the MP on HLDD with an automated program that generates the HLDD, which is opposed to the previous ways of manually generating the HLDD. We will further elaborate on the automation of the HLDD generation under the contribution section in this thesis. In [15], the synthesized test program proposed for µGP techniques could not realize a gate-level fault coverage more than 90% because it could not detect the hard-to-detect faults. The inadequate fault coverage gave rise to greedy based evolutionary approaches by Suriasarma which detects 40% of the hard-to-detect faults but the synthesis time is longer [16]. This brought researcher in [4] [14] into the limelight that testing MP at a high level of abstraction using HLDD to generate the test program improves the test program quality and it ensures a better fault coverage that covers the gate level faults inclusively. Raimund et al. emphasized in [17] that fault model for digital circuits have been developed for a different type of failure mechanisms like signal line bridges, transistor stuck-opens or failures due to increasing circuit delays. However, the oldest general fault modelling mechanisms that can effectively analyse arbitrary fault types is called the D-Calculus [18]. In [13], a novel method for implementation-independent test generation

for modules of RISC type microprocessors was proposed. According to the method it covers a larger class of faults than the traditional single SAF. This implementation independence of tests was achieved by testing separately the control and data path of the module explicitly.

In summary, we conclude that the challenges of SBST are the compacted test program generation which depends on the quality of the test data, since the test program is the determinant of the quality of the fault coverage.

According to [14] the proper testing of the MP after manufacturing process guaranteed and enhanced the reliability of the MP during the operational stage. This test remains crucial for the safety purpose of the safety-critical systems like MP.

This leads to the goal of this thesis.

1.2 Objectives

The objective of this thesis is to optimize the test data generation for MP in order to achieve high fault coverages both at high and low-level. Thus, different experiments have been carried out by automated transforming the given instruction set of the MP into different HLDDs. The goal of this thesis is divided into three phases each with a goal of optimizing test data for testing processor. Hence, this thesis presents the following goals:

I. To develop a mathematical model for partitioning the set of instructions under test.
II. To optimize the test data generation for testing of the MP at high-level.
III. To prove that the high-level faults model covers all the low-level detectable (non-redundant) faults.

1.3 Thesis organization

The thesis is consists of 7 chapters.

Chapter 1 introduces the thesis, which includes the background and problem of SBST, and objectives.

Chapter 2 presents the background information about the processor, digital systems, defect, faults, fault model and level of abstraction in developing digital systems.

Chapter 3 outlines the state of the art for testing of the processor. It covers in-depth ongoing research for testing of the processor. It also contains some literature review of related work of this topic. The summary of the state-of-the-art is then presented.

Chapter 4 describes the methodology flow for generating a fault model. First, the correlation between the DDs is studied, followed by the High-level decision diagram used for grouping functions together for a simulation-based test. Furthermore. It also contains the literature review of the related works of this topic.

Chapter 5 presents the test generation approaches; it covers the test data generation and test program generation, and it presents the algorithm proposed for HLDD generation in this thesis.

Chapter 6 describes the research environment for the SBST, and the experimental results are presented.

Chapter 7 summarizes the conclusion and presents the future research direction.

1.4 Overview of work

A novel approach is proposed for generating test data for SBST at high-level with regards to the followings. The approach is based on automating the partitioning set of instruction on HLDDs. An optimal test-data is then generated from the partitioned set of instructions. The HLDD is synthesized to generate the test program for testing the processor. The algorithmic partitioned approach of instruction set improves the test program quality, reduces memory size usage and ensures a better fault coverage as it considered the needed patterns for a certain test based on the test data generation constraints. Our approach on test generation for the processor using high-level decision diagram is to represent the functional behaviour of the instruction set of the processor in a way that it is easy to observe and traverse its paths in the HLDD to the terminal node. The processor is then tested at high-level and the test covers the gate-level faults inclusively. There are two approaches in testing at high-level, thus, in this thesis, the functional approach is used. The functional approaches use instruction set architecture, whereas the structural approaches are based on test generation using information from the lower level of design (gate-level or RTL-level description) of the processor under test [11, 19] [20, 9]. Due to the increasing complexity of digital circuits that has renders classical gate-level test generation impractical, high-level fault models are used widely in the field of SBST [11, 9].

The good way through which the instruction sets can be partitioned to generate high-quality test data for the test program is presented.

The processor is tested at high-level without resorting to its implementation details. The results for the experiments cover both high-level and gate-level inclusively.

The method for achieving an optimum test data generation is based on the algorithmic partitioning of the instruction set on HLDD, which reduces the memory size usage of the MP for storing the test program and ensure high fault coverage for both high-level(functional) and gate-level (lower-level) faults.

2 Processor

In this chapter, I present briefly the fundamental concepts of RISC processor design and its subcomponents used for designing the processor before diving into the state-of-the-art of testing the processor. Special emphasis is placed on testing of the processor, which is the main goal of this thesis. The modern technology advances are imposing new challenges on processor testing with billions of transistors which can operate at gigahertz frequencies [4]. The first RISC was developed with a high volume of software with hardware support for only the most time-consuming events [7]. The reason then was based on the limited transistors that can be integrated onto a single chip [7]. The transistor is the central component of the processor's circuitry [21] However, the reliability of the transistor is becoming enervated since its geometry dimension approaches further down the nanometre dimension [22]. According to Moore's law in every 18 months the number of transistors on integrated circuits doubles [23], [14]. The development in semiconductor technology and nanotechnology are strengthening the existence of this law. Nowadays advances in VLSI technology make it possible to realize the minicomputers of before on a single chip of silicon [7]. Hence, testing at the gate level is becoming more difficult. Most of the RISC instructions are "register-to-register" and take place entirely inside the chip and the data memory is restricted to the LOAD and STORE instructions [7]. The overhead of an MP is reduced with banks of registers equipped during architectural design of the MP, this simplifies the process of procedure call by changing a hardware pointer, thus avoiding the overhead of saving registers in memory [7].

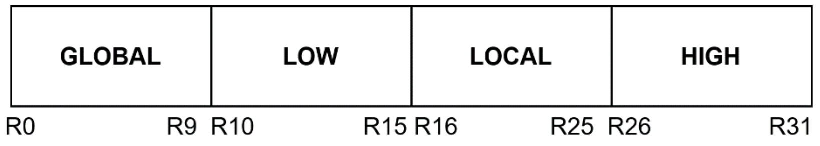


Figure 1. RISC Register Window

The instructions in the processor can be grouped into four categories [7] such as:

I. Arithmetic-logical

II. Memory access

III. Branch

IV. Miscellaneous

The execution time of the RISC processor is given by the cycle it takes to read a register, perform an ALU operation, and store the result back into a register [16]. I further present the foundation of MP as a digital system.

2.1 Digital System

Digital system is a system that stores data with 0 and 1, this system could be sequential or combinational. There might be combinations of 0s and 1s for a system to be operative in some conditions. The 0 refers to as OFF and 1 as ON. As a result, digital circuits are the foundation for computers and digital communication. It was invented in twenty centuries [20]. The complexity of digital circuits has rendered gate-level test generation impractical [21]. This has led to the development of SBST to reduce test generation complexity for a complex digital system like a processor. The goal is for the new approach to incorporates the benefits of functional (high-level) testing and still retains the accuracy of gate-level fault models.

2.1.1 Development life cycle of a digital system

The developmental approach of the digital system starts with system requirements. What will the system do? Based on that fact, testing is not exceptional. Digital systems undergo three major stages [14]:

- Design
- Production
- Operation

Each stage is prone to error. Hence, each stage needs to undergo review along the design process. Several models have been used, such as vee model, waterfall model and linear model, in order to follow up design specification. This model guides the process. However, it doesn’t stop an error to occur. At the design stage, there could be misinterpretation or omission of the specification [14]. Fault in the production stage could be as a result of component defects or defects due to component assembly issues. When the system is operational, the system can suddenly fail due to the undetected defects during production or due to environmental factors. Testing then remains the main factor to assure us that the system is doing what is supposed to do.

2.2 Digital System Testing

Testing a digital system means checking if the system is working according to the specification of its design and production for operational use.

This process starts with passing sets of inputs known as stimuli into the CUT, which at that time is also referred to as a black box. Then the black box is checked to see if it is working in accordance to specifications, by observing its response at the output terminal.

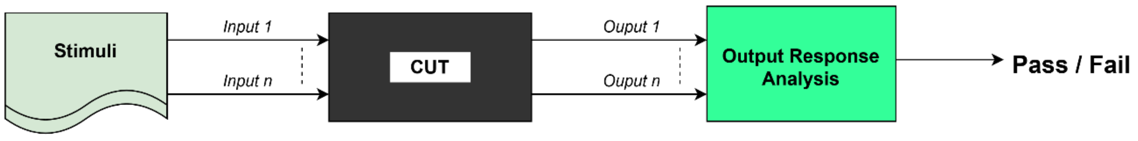


Figure 2: Process of Testing Digital System

The main reason for testing is to find out if the device is free of defects. A defect can lead to faults. Fault can lead to an error. An error can lead to failure.

2.3 Faults

2.3.1 Defects, Error, Failure

A defect is a failure mechanism in an electronic system which is the unintended physical difference between the implemented hardware and its intended design [24], [25]. Defects may therefore, cause deviation in system specifications".

A fault is the representation of defects at the abstracted functional level (electrical, Boolean or functional malfunction) [24]".

Table 1: Difference between a defect and a fault

Defect	Fault
Imperfection in hardware	Imperfection in function

Multiple faults (functional deviations) could occur due to a single physical defect (deviation from intended specifications) on a chip and these multiple faults my not been detected by a single test type.

An error is an unexpected output signal produced by a defective system".
Failure is the result of an error, e.g. A bad processor can usually cause system failure if the processor is not functioning properly.
Also, defects, fault and error are examined with *Figure. 3* below:

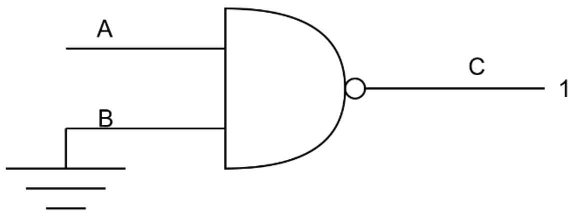


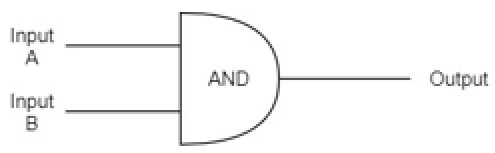
Figure 3: A NAND Gate with one input shorted to ground

- A defect, in this case, is the connection of input B to ground, since the output of a NAND gate produces only 0 when its inputs are 1. Therefore, the defect in its specification will facilitate an error. In the presence of the short, we will only have input combination 10 and 00 there will be no 11 to put of the led at the output.
- Fault, in this case, means that input B will always have a stack at 0 because this digit will never change.
- Error is the deviated output result generates by the gate due to the defect in the gate. Note, the error in the NAND gate is not permanent. The reason is that we will have the correct output as 1 whenever we have 00 or 10 as inputs patterns.
- Failure, in this case, is when the NAND gate is not performing as expected.

New technologies bring new defects and must be modelled into faults. According to [17], the presence of fault *F* in a circuit *C* transforms the *C* into a new faulty circuit C^F. Let $y(x)$ be the logic function of a circuit *C* with perfect functionality. The presence of fault in *C y(x)*, changes the circuit and its function to C^F y^F*(x).* However, if a system functionality does not change in the present of faults, we call it redundant faults.

This raises the question of how do we address such faults? A combinational circuit that contains undetectable SAF is said to be redundant [24], [26]. Therefore, a redundant fault could be addressed by removing the unnecessary inputs that cause its occurrence. Let examine this statement with an AND gate of n inputs and *n+1* input respectively, where the extra input is redundant and remain always constant. This is depicted in *Figure.4*.

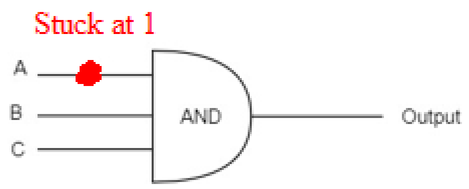
FAULT FREE ***AND*** GATE



AND GATE TRUTH TABLE

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

FAULTY ***AND*** GATE



REDUNDANT ***AND*** GATE TRUTH TABLE

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

Figure 4: Comparison of Fault free and Redundant Faulty gate

In the second table, the presence of the redundant input does not change the result of an AND gate. Hence, the two tables justify the correct output of an AND gate even in the presence of redundant inputs. Therefore, removing the redundant gate input of the faulty AND gate removes the redundancy.

2.3.2 Classification of faults

A fault can be classified into two types, soft and hard faults. A fault is mostly caused by a defect, this defect could be either shorts, open, improper manufacture, induced by thermal ageing or by environmental influences.

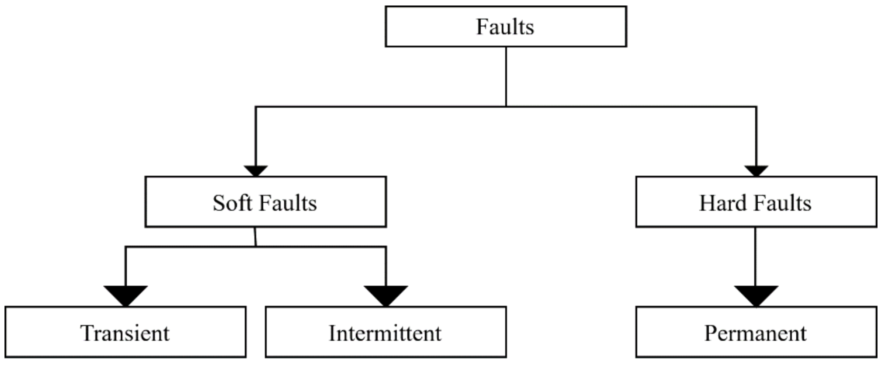


Figure 5: classification of faults

- Transient fault occurs randomly and remains active for a short period. This can be difficult to detect unless the result is seen at the propagation output as incorrect.
- Intermittent fault occurs at irregular intervals due to aging or unexpected environmental factors.
- Permanent faults are always active and will only disappear after the system has been repaired.

2.4 Fault models

A fault model is used for fault simulation, analysis and evaluation of the set of test vectors.

Depending on the level at which systems are handled fault models could be different and this type of inconsistency increase CAD system costs as there would be the need for tools available for each level explicitly considered. According to [27], the systems various levels of abstractions need different mathematical tools for each level. Defects must be modelled at the higher abstracted level as faults. Consequently, the main goal of the fault model is to reduce the infinite set of possible defect behaviours into a finite set of faults [24].

2.4.1 Stuck-at-faults

The most widely use fault model in digital testing is the single stack at fault model [26]. However, its limitations lie in its capability to model only a single fault. The characteristics of this model are given below:

- Only one circuit line is faulty
- Faulty line is permanently set to 0 or 1
- The fault can be at the input or output of a gate.

For the testing of processor, High-level fault model is adopted in this thesis.

2.5 Fault simulation

The ideal of fault simulation is nothing than simulating a digital circuit in the presence of fault [28]. The formula for fault coverage is given below.

$$\text{Fault coverage} = \frac{\text{Number of detected faults}}{\text{Total number of faults}}$$

Hence, a fault model is required in order to evaluate the fault coverage of the simulated circuit.

2.6 Levels of Abstraction in Digital System Testing

The modern technology advances are imposing new challenges on digital system testing with billions of transistors in a system which can operate at gigahertz frequencies [4]. Hence, design for testability must be considered during the design process, as the key to successful testing lies in the design process. Thus, the level of abstraction is used to manage the design process of the complex digital system [22]. It shows the design stage that should be implemented before the next. A high-level of abstraction focuses on most imperative data like the behavioural specification implementation. Figure 2.4 shows different levels of abstraction.

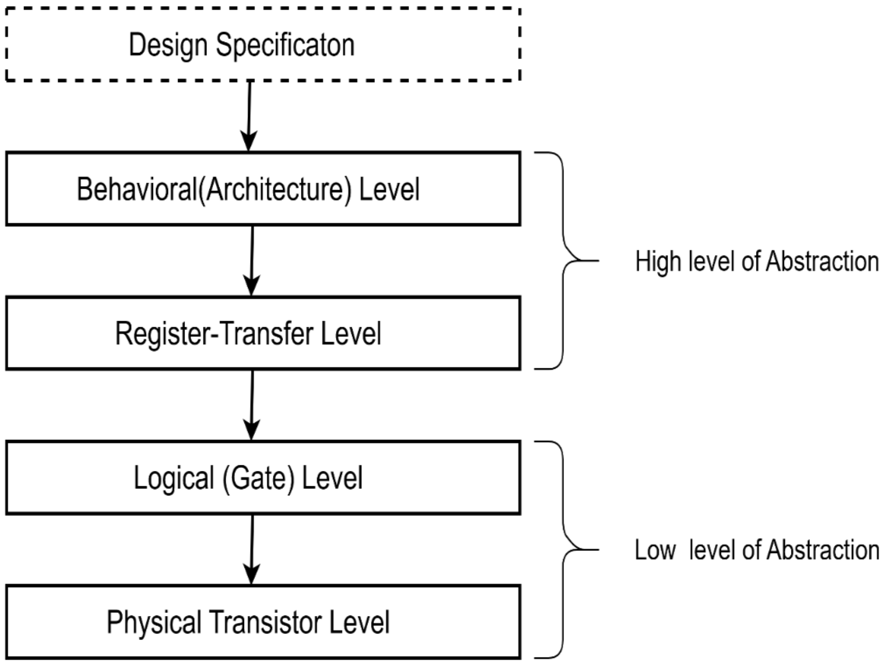


Figure 6: Level of Abstraction [29]

Before moving to test part let us briefly explain the level of abstraction.

The design specification is the requirements to follow for the design flow. One can say it is the plan for the system execution.

The behavioural level is the highest level of abstraction that describes the system in term of what it does or how it behaves. Hence, the behavioural level is the interpretation of the specification as a computer program.

An example is a circuit that warns car passenger when the door is open, or the seatbelt is not used [29].

warning = Ignition_on AND (Door_open OR Seatbelt_off)

As shown in *figure 6* the RTL with structural information only exist once the behavioural level is implemented. The RTL is synthesis to gate-level design in which sequential and combinational logics are represented in the form of interconnects of logic gates such as AND gate, OR gate, XOR gate etc [29], [14].

Finally, the gate-level design is synthesized to the physical level where the gates are represented by interconnections of transistors [14], [29].

Both the logic and physical level are represented as the structural level. The structural level describes a system as a collection of gates or components that are essential for the performance of the system.

The gate-level is the most common level of abstraction through which testing has been targeted. On the other hand, due to the high numbers of gates for the digital system, testing at gate level remains a bottleneck. However, for testing accuracy in VLSI, the physical layer has been targeted because of the direct contact to the layout and routing information of the manufactured system [14], [29].

However, testing a digital system such as the processor requires a very high level of abstraction. Therefore, this work, we will be focusing on testing at a high-level of abstraction.

2.7 Importance of Digital System Testing

The reliability of a digital system is enhanced through testing. Thus, testing could detect design errors, fabrication errors, fabrication defects and physical failures such as wear-out and environmental factors before the system goes to the operational use. An early test for fault detection before releasing to the operational environment reduces the cost of test, on the other hand, the cost of finding defects after the system has been released for operational use can be overwhelming. Therefore, quality and economy are two vital factors to be considered [5]. Failure cannot be tolerated in such a system since digital systems are widely used in safety-critical systems.

3 State-of-the-Art for testing processor

The approaches in the field of processor test can be organized into three major groups [17], [3]: structural, functional and software-based self-test methods. The structural approach is based on applying DFT techniques into a digital design. Hence, this process can change the design implementation of the system under test. Therefore, it is not safe for testing a safety-critical system like a processor.

The functional approach is used for testing chip after manufacturing. However, the functional automatic tester is expensive. Due to high cost of the functional approach, the industry raised interest in the structural approach which on the other hand is not efficient due to over-testing.

The current state-of-the-art is based on testing the processor using the gate-level information for SAF. However, the complexity of today’s processor that uses billions of transistors requires more high-level method in testing the processor rather than via gate-level. Due to this reason, a method that gives less overhead in a test budget was proposed by the semiconductor industry that can be incorporated in an established processor test flow [4]. This method is referred to as software-based self-test as mentioned in the background section.

Because of the growing density of integration in the semiconductor industry today’s chips are more sensitive to faults while the faults detection mechanisms of the latter become more complex [2]. The complexity of processors, and the limited accessibility of the internal logic, makes them very difficult to test [12], [19], [30], [13]. According to [31], the larger amount of logic for processor is based on the order of hundreds of thousands of gates which makes it very difficult to test using the classical approach of testing digital circuits. Hence, it is not enough to use the existing fault simulation and test generation software to derive tests for processors. One easy approach of solving this problem is to use pseudorandom test patterns [12], but its drawback is the lengthy test result which therefore remains impractical to test complex microprocessors [4]. However, some previous work in testing a complex microprocessor was based on functional models [5], [32], [4], [13], [11], [33], and has shown some method of success also it opens paths for the improvement of test generation for a complex microprocessor.

The result of the test generation method proposed in [34] by Saucier and Robach was not successful due to the results not directly observable. The reason is because of the unavailability of the basic control commands to the user which are necessary during the control state and its operands are not directly available to the functional units (like ALU, etc) of the microprocessor.

In [32] two different fault modules were considered in testing the bit-sliced microprocessor. The methods were based on the combinational and sequential separation of the microprocessor into modules. They altered changes in the truth table for the combinational module and in the state table for the sequential module. Based on these fault model they generate tests to verify the correct functioning of each component modules. The problem in [32] is that the approach tests only the data part and assume that the control part of the MP is fault-free. Also, it would be inefficient to generate tests when the modules are complex.

In [5] a microprocessor is represented by a set of functions such as 1) data transfer path, 2) data manipulation functions, 3) register decoding, 4) instruction sequencing.

A functional fault model is then developed for each of these functions, tests are generated as to detect faults in the fault model although the generated test was only able to test correctly the relatively simple functions except the instruction sequencing. The test generation for the instruction sequencing fails because the fault model was not based on the logical analysis of the instruction execution. However, in order to test correctly the instruction sequencing, each instruction is executed, and a test is generated to check for the correct execution of the instruction without any other instruction being executed simultaneously.

The problem in [5] was partly solved in [31] , where the researchers created a fault model that treated the presence of faults in each instruction individually. Thus, their result detects some faults not covered in [5], i.e. the instruction sequencing faults and the complexity of their test generation is only O($n_I * n_R + n_R^4$).

It was proposed in [35] and [10] that the processor can be divided into module under test to ease test data generation using ATPG. ATPG is one of the means of generating test data for SBST [36]. However, the drawback of using ATPG is the run-time for generating the test data for microprocessor testing [11]. In addition to the drawback, the traditional ATPGs target only single SAF.

In [13] a novel high-level implementation-independent test generation that covers high multiple SAF and fault simulation that evaluates the high-level fault coverage was proposed. The method is based on separate test generation for the control parts and data parts of the MP at high-level functional units. The control parts are tested with a high-level control fault model, whereas, they applied a pseudo-exhaustive test for the data parts. The application of the pseudo-exhaustive test to the data part is to keep the implementation-independent property of the proposed method. The approach in [13] can be used for easy identification of redundant gate-level faults in the control parts. Remind that, redundancy occurs when the function of gate or circuits does not change in the presence of a fault. Hence, a combinational circuit that contains undetectable SAF is said to be redundant [24], [26]. The novelties of the method which are based on 1) modelling the UUT by symbolic EDNF and 2) the translation of the traditional fault propagating task … to achieve full independence of the implementation details in test generation prove to be effective and cover multiple faults in high-level than the well-known traditional ATPGs. However, the drawback of this approach is based on the high numbers of patterns that the method uses for the test generation, which in returns increase computational time but on the other hands aid the fault coverage capability. *Thus, optimization is needed.*

In [4], [33], [37] a novel high-level functional control fault model was used to cover the high-level and gate-level faults of the MP. The functional fault model support hierarchical test approach, where the test pattern, which activates a low level fault at low level (gate-level), can be consider as the high-level constraint for the functional fault defined at the higher-level (functional or behavioural) [13], [38]. The mentioned functional fault models offer high flexibility in defect modelling beyond single SAF model [38]. The major problem of these approaches is the not sufficient fault coverage achieved when comparing with the gate level faults coverage approach. Perhaps, this is caused by the level of quality of the test data used in test program for testing the microprocessor.

In [2] extension to the class of functional faults model was proposed for the modules of RISC type MP. The goal of this approach is to cover large functional faults together with a large class of structural faults while using a high-level fault model. That is, the fault model implemented without the knowledge of the MP gate-level implementation details.

The research in [2] proves that the functional fault model could as well cover gate-level faults without using the gate-level implementation details.

This approach has a common implementation strategy with the already mentioned method in [32]. However, here both control and data parts were tested without assumptions in the correct functioning of any of its parts. Their method put together the structural faults and functional faults classes as a single measurable high-level functional fault model. Thus, High-Level-Decision Diagram was used for formalization of the high-level test generation. This method of testing enhanced fast testing period and reduce test cost. In *Table 2* experimental result for this approach gives high SAF coverage compared with other implementation-independent and the state-of-the-art approaches for testing microprocessor.

However, this approach has less coverage for the "Forward unit" (see table 2) of the MUT in compare with the ATIG that uses implementation detail. A drop-in fault coverage is one of the major challenges while covering low-level faults using high-level fault model.

3.1 Summary of state-of-the-art

Since the traditional low-level test methods for a complex system like MP has lost their importance, due to the complexity of today's sub-micron technologies, other approaches test methods based on the high level functional and behavioural method are gaining more popularity [39] [40]. Hierarchical mixed or multi-level approaches have also been used both for test generation and fault simulation [41] [42] [43].

These varieties of approaches have been used by researchers in order to improve fault coverages during SBST of processor. However, none have been able to generate high fault coverages for the gate level faults tested at high-level of abstraction (i.e. using functional fault model).

Many researchers target both the structural and functional approaches. The structural approaches make use of the processor lower-level details for test generation, whereas functional approaches use the processor high-level details [6].

To increase the speed of test generation, High-level fault model has been chosen, but this approach can be considered as “good” if the test generated using this model provide high SAF coverage or physical defect [4] [38].

In an attempt to improve these method, researchers in [33] have proposed different approaches in testing the MP at high-level, a good example is a proposed novel high-level implementation-independent of test program generation method for RISC processors in [13]. The high-level model of the processor is derived from the instruction set and its architectural features. The experimental result in [13] [33] gave high fault coverage for the given MP without the information about the gate-level implementation details. However, the experiment used high numbers of patterns for test data generation.

Thus, this thesis focuses on test data optimization. The instruction set architecture of the MP is used and a comprehensive functional fault model was developed. The approach is based on topological reasoning of the special SSBDDs that generalize to HLDDs. Hence, HLDD is generated automatically using the instruction set of the MP in order to improve the test data generation, which in return improve the faults coverage for both the high and low-level.

Experimental result obtained in this thesis is compared with the one proposed in [33] which has already been compared with other 3 methods from the state-of-the-art:

Table 2: Proposed submitted method Compared with other methods [33]

Module/unit	#faults	Gate-level implementation details exploited		Gate level implementation independent	
		ATIG	SBST	SBST	Proposed in [33]
ALU	203576	98.67%	n.a	97.85%	99.06%
PPS_EX	21136	97.62%	96.20%	84.12%	97.96%
Forward	3738	99.00%	99.68%	93.64%	98.03%

4 High Level Decision Diagrams

This chapter presents the approaches in the fault model procedure for the SBST to test a given processor (miniMIPS). The usefulness of Decision Diagram and its functionality in the field of SBST is shown. The process for test program generation through the concept of HLDD is analysed. It also, presents the ISA of the given MP and how the HLDD is synthesized for the given MP based on the ISA. The partitioning approach in this work is an optimized version and different from the work done in [2], [4], [13], [37], as a result of the algorithmic partitioning of processor instructions on HLDD for test data generation. Hence, the test program for SBST is generated via the synthesized HLDD.

Thus, the following sections explore the fundamental use of DDs, history of DDs, the transition reason from BDDs to HLDDs, the fault modelling method adopted in this work for test generation, the benefit of the HLDDs as fault model foundation for complex systems and in *chapter 5.4*, I present the algorithm for generating the HLDDs developed during the research of this work.

4.1 Overview of Decision Diagrams

In today's digital system gate-level test generation methods is obsolete as the complexity of the digital systems continue to increase [19], [4], [44], [24], [45], [6]. Promising approaches are high-level, or hierarchical methods that use functional descriptions of a system [24].
Due, to different system level of abstraction, there are no unique models as a uniform approach to generates test at a different level. However, DD has been choosing to serve as a base for uniform test generation, fault simulation and fault location for mixed-level representations of systems and the Boolean algebra as its plain logic level [24], [1].
Therefore, DD serves as manager of hierarchy in diagnostic modelling. Different types of DDs have been adopted and used for testing, such as the BDD, SSBDD and the generalize HLDD that handle the test generation problem at high-level.

Thus, when the DDs is used for describing the complexity of the digital system, we have to represent the system by a proper set of interconnected components (combinational or sequential sub-circuits) [43]. Then, the components by their corresponding functions which can be represented by DDs [46].

In summary, DDs are used for multi-level and hierarchical diagnostic modelling because of their uniform cover of different levels of abstraction, and because of their capability for uniform graph-based fault analysis and diagnostic reasoning [3], [43], [47], [48]. This means that instructions and faults can be modelled at gate-level, RTL and behaviour level using DD.

Hence, DD can be categorised into two major parts.

1) Logic level - (BDD)
2) Behavioural high-level DD - (HLDD)

This work focuses on modelling the MP at the behavioural level using the HLDD.

4.1.1 History of Decision Diagram

BDDs have become the state-of-the-art data structure in very-large-scale integration computer-aided design for representation and manipulation of Boolean functions [49], [50].

In 1959, C.Y. Lee introduced a method for representing digital circuits by binary decision programs [51].

The same model was introduced but with a different name as alternative graphs for test generation purposes in 1976 by R. Ubar, [1].

Bryant proposed ROBDDs as a new data structure [43] in 1986.

In [28], [48], [27], [52] SSBDDs was introduced as a special class of BDDs to represent the topology of the gate-level circuit in terms of signal paths. The advantage of SSBDD based approach is that the library of component is not needed for structural path activation and as clearly explained in [24] the, SSBDD based test generation procedures do not depend on whether the circuit is represented on the gate level or the whole circuit. It is a novel fact that the SBDD test generation procedures can be easily generalized for the high-level DDs to handle digital systems represented at higher levels [27], [53], [54].

4.2 HLDD

The most important impact of the HLDD is the possibility of generalization and extension of the methods for test generation, fault simulation and diagnoses [2], [52].
In order to use HLDD to describe complex systems, the system must be partitioned into a suitable set of interconnected components (combinational or sequential sub-circuits) [17]. HLDDs model allows mapping the control functions of systems into non-terminal nodes, and the data manipulation functions into terminal nodes [17].
The non-terminal nodes of the HLDDs represent the control variables [19], [4], [17]. Both terminal and non-terminal nodes of the HLDDs should be tested [10], [55], [24], [37]. We verify the behaviour of the circuit by testing the non-terminal nodes of an HLDD, on the other hand, we verify the working mode of a circuit, when testing the terminal nodes of an HLDD [37], [56].
The main purpose of modelling fault at high-level is to speed up fault coverage evaluation without reasoning about the gate-level implementation details [57] and the main idea of this kind of fault modelling is to obtain from the high-level (functional or behavioural) description of the system an incorrect version by introducing a fault into the description, [37], [24], [57], [26]. This approach is also referred to as **model perturbation** [24].
The traditional gate-level approach has lost their importance since the physical defects that may occur in digital systems often do not manifest themselves as stuck at fault [58]. Hence, high-level faults represent the effect of a physical defect on the system functionality. Therefore, to improve the test quality, there is a need to replace the traditional fault models like SAF with a realistic defect model [58]. That is, the defect model that can handle the defect orientation and the high-level behaviour of the system. Hence, HLDD in combination with a multi-level approach is the appropriate way of resolving this issue. It was proved in [13] and [26] that a high-level fault model can be explicit or implicit. An explicit model identifies each fault individually and every fault in the model is treated as a target for test generation [57] , [26].
However, implicit fault identifies faults that belong to the same classes, sharing “similar” properties, so that all fault in the same class can be detected by the same procedure [13], [26], [57], [59].
In [13] different fault models have been developed for digital circuits as to detect its failure mechanisms like transistors stack-open [60], signal line bridges [61], or failure due to delays in circuit [22]. High-level fault model is widely used in the field of SBST

[11], [59] for test generation at high-level of abstraction without resorting to the implementation details of the gate-level.

Its drawback is its low fault coverage for gate-level faults. Thus, the research in SBST focuses on the drawback as to detect high gate-level fault coverage based on test generated from the high-level of abstraction while test data is less. On the other hand, the advantage of high-level fault model cannot be underestimated due to its independent character on the system implementation details. Thus, it can be used not to check physical faults of microprocessor only, but also as a verification tool with which we check whether the implementation is free of design errors [24].

In a general case beyond the Boolean algebra a decision diagram can be defined as a non-cyclic directed graph:

Definition 4.1:

$G = (M, \Gamma, X)$ with a set of nodes M, a set of variables X , and a relation Γ in M where $\Gamma(m) \subseteq M$ denotes the set of successors of the node $m \epsilon M$ [17], [3], [27].

The nodes $m \epsilon M$ are labelled by variables x(m) ϵ X (constants or algebraic expressions of x ϵ X). For each value e from a set of possible predefined values e ϵ V(x(m)) of a non-terminal node variable x(m), there exists a corresponding output edge from the node m into a successor node m^e ϵ $\Gamma(m)$ [3], [56].

Note, that the SSBDD model can be regarded as a special case of HLDD based on the above-given definition [48]. While the above definition is the *graphical representation of a system* using graph theory, the formal definition of HLDD regarding a system is given below.

Definition 4.2:

Considered S as a given digital system, consist of different components (subsystems), which is denoted by a function $z = f(z_1, z_2, z_3, \dots, z_n) = f(Z)$ where Z is the set of variables(can be Boolean, vectors or integers) and $V(z_k)$ is the set of possible values for $z_k \in Z$ which are finite [3], [14].

The cycle-based modelling concept for analysing the behaviour of a digital circuit is adopted by HLDD. This means that HLDD allows system state determination at a cycle. Thus, since the MP is been modelled at high-level using the ISA, the instruction cycle-based will be considered. However, the following presents the list of faults affecting the operation of MP.

4.2.1 List of Faults affecting the Operation of MP

Before, presenting the fault model approaches for this work. It is imperative to list the faults affecting the operation of MP. These faults can be divided into the following classes [24]:

I. addressing faults affecting the register decoding function;

II. addressing faults affecting the instruction decoding and instruction sequencing functions;

III. faults in the data-storage function;

IV. faults in the data-transfer function;

V. faults in the data-manipulation function.

4.2.2 High-Level Fault Modelling for MP with HLDDs

Knowing which fault model is right to choose is the central problem of test generation in fault simulation. However, many approaches like fault tuple model [62] [3], pattern fault model [63] [3], input pattern fault model [64] [3], and functional fault model [65] have been shown and used by researchers. Thus, in order to generate high-quality test a good fault model should be adopted. Test generation needs high-level fault modelling because of its high complexity [19], [16], [4], [17]. The main and general problem of the fault model is the difficulty of proving that the model covers all the low-level detectable (non-redundant) faults [38]. In this thesis, attempt is made to prove that by generating a separate test for the control and data part of the MP, using high-level fault model.
According to [17], there are two opposite criteria [that] should be followed while developing tools for synthesis of tests:

1. Efficiency (the cost of test generation)
2. Quality of generated test (fault coverage).

These criteria are highly dependent on the type of fault model used during test generation and fault simulation for test quality ascension. Based on these obvious facts, a good fault model reflects the accurate physical mechanism of the real defect in a system its modelled for, which then support excellent test quality.

4.2.3 Complexity and Accuracy in Fault Model

Complexity and accuracy are other issues that the HL fault model solved while reasoning about test generation at high and low-level with a trade-off in consideration. The multi-level approach should be considered as accuracy requires low-level fault models and fault simulation [33] . On the other hand, high-level fault model solves the issue of complexity in test generation for the CUT and gives high fault coverage. Although the representation of defect with high-level (less complex) fault model hasn't achieved high accuracy in test quality for the low-level fault coverage like the fault coverage by the traditional ATPGs.

However, due to the complexity of today's electronic system, modelling its defect with low-level fault models is obsolete. The continuous research in this area has exposed the trade-off for balancing the accuracy and complexity concepts. The result in ***Table 2*** above is a good comparison example, where three modules of MP are tested with and without the knowledge of its implementation details and the result from different researchers were tabled and compared.

Therefore, increasing the accuracy of representing defects also increases the complexity of the fault model [30] [17], [66]. Since the new process of semiconductor technology arises in today's digital world, knowledge of know-how will always be needed to understand the new type of defects that could cause a failure mechanism in today's and in future nanoelectronics.

Also, HLDDs support functional fault model where the system architecture is first described at the system level and the system is partitioned into several functional blocks or modules, whereas, structural fault model applies to gates, flip-flops and interconnection between them [17]. HLDD-based high-level fault model is well suitable to support the development of a uniform and straightforward high-level test generation and fault simulation algorithm [58], [3].
Other different high-level models have been presented in [5], [67], [68], but none of these techniques established the relationship between high-level fault coverage and gate-level fault coverage. Thus, these models were only able to contribute to test generation but are not suitable for the gate-level test quality assessments. However, the approach in this thesis is different from these previously mentioned as this work considered both the high-

level and low-level faults coverage while modelling with HLDDs. Hence, this work optimized the work in [2] and [33] .

4.3 MiniMIPS ISA

The miniMIPS has a 32-bit core based on a von Neumann architecture, with a 5-stage pipeline, instruction extraction, instruction decoding, execution memory access and update registers [69]. The following table shows the full instructions of a miniMIPS processor used for the experiment in this thesis. The description includes flags, list of instructions, general purpose-register, assembly language syntax and their binary representation [7]. The ISA (*Table 3*) serves as an abstract representation of the microprocessor itself. High-level decision diagrams can be constructed from the instruction set architecture of an MP. Therefore, for testing the microprocessor we must test that the instructions are executing correctly. Twenty instructions are chosen for the experiment carried out in this thesis (Table 4).

Table 3: Full miniMIPS instruction set [14], [69].

MiniMIPS Instruction Set Architecture

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	rd = return_address
				JALR rd rs	
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	MTCO rt, cs	cs = rt

The following table shows the set of instructions from the list of miniMIPS instructions partitioned for testing the processor in this thesis.

4.4 HLDD Generation for MP

Consider the behavioural level structure of a microprocessor in *Figure. 7*, and the HLDD in *Figure. 8* generated as the source of functional fault model for the MP with a set of instructions in *Table 4*.

Note, the HLDD generation has been optimised in order to achieve high test data generation for SBST. The optimization is based on the logic behind the position of the terminal node and the connection between each function while less test data is used during the test. Hence, the ordering is referred to as "good". Chapter 5 of this thesis will present the test generation approach in detail. However, this section serves as the foundation for the test generation.

Table 4: Instructions selected for Experiment

Number	**Instruction**	**Code**
19	SLL	000000 = 0
28	OP2	000001 = 1
24	MFLO	010010 = 18
26	MTLO	010011= 19
17	MULT	011000 = 24
18	MULTU	011001 = 25
2	ADDU	100001 = 33
3	SUB	100010 = 34
1	ADD	100000 = 32
4	SUBU	100011=35
5	AND	100100 = 36
6	OR	100101 = 37
7	XOR	100110 = 38
8	NOR	100111 = 39
9	SLT	101010 = 42
10	SLTU	101011= 43
20	SRL	000010 = 2
21	SRA	000011 = 3
23	MFHI	010000 = 16
25	MTHI	010001= 17

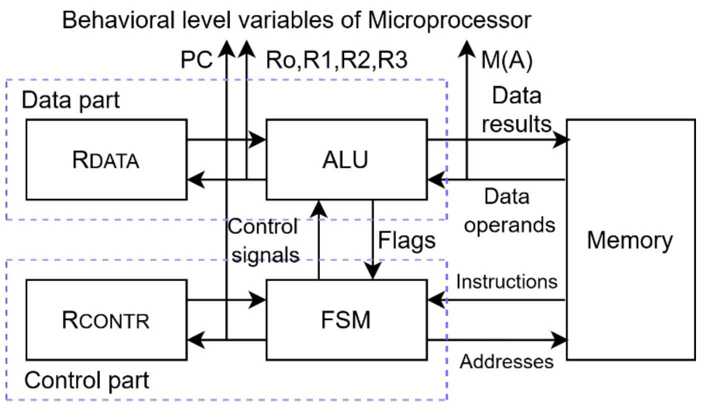


Figure 7*:* Behavioural Level Structure of the Microprocessor [4]

Consider MP functionality as a set of the following behavioural level functions [4], [14]:

1) $R_i = f_i(I, S(R_i)) = f_i(OP, B, S(R_i)\ where\ R_i \in R_{DATA}, i$=0,1,2,3, ... 19, $and\ S(R_i) = \{ R_{DATA},, M(A)\}$ is a set of data arguments for the functions f_i (a set of the source registers over all the instructions);
2) PC = $f_{pc}\ (I, C, PC) = f_i\ (OP, B\ , PC)$ where C is the flag variable serving as a condition for the branch operation;
3) $M(A)$=$f_M\left(I, S\big(M(A)\big)\right)$=$f_i\left(OP, B\ , S\big(M(A)\big)\right)$ where S(M(A)) = { $R_{DATA},, M(A)\}$

The functionality of the MP can now be represented by a set of behavioural level variables Z = $R_{DATA} \cup R_{CONTR} \cup M(A)$ and by a set of functions F ={$f_0, f_1\ , f_{2,}\ f_3, \dots , f_{19}, f_{PC}, f_M$} [17], [4], [14]. In this work, we are testing the data path and the control path of the MP. Which means we will model the behavioural level variables Z = $R_{DATA} \cup R_{CONTR}$ and a set of functions F ={$f_0, f_1\ , f_{2,}\ f_3, \dots , f_{19}$}. The behaviour of MP can be modelled by the functional basis F and monitored through the variables Z. For modelling of F the behavioural level HLDD model is used [4].

The following HLDD is automatically generated, by the algorithm developed for HLDD generation in this work. The algorithm is discussed in ***section 5.4***.

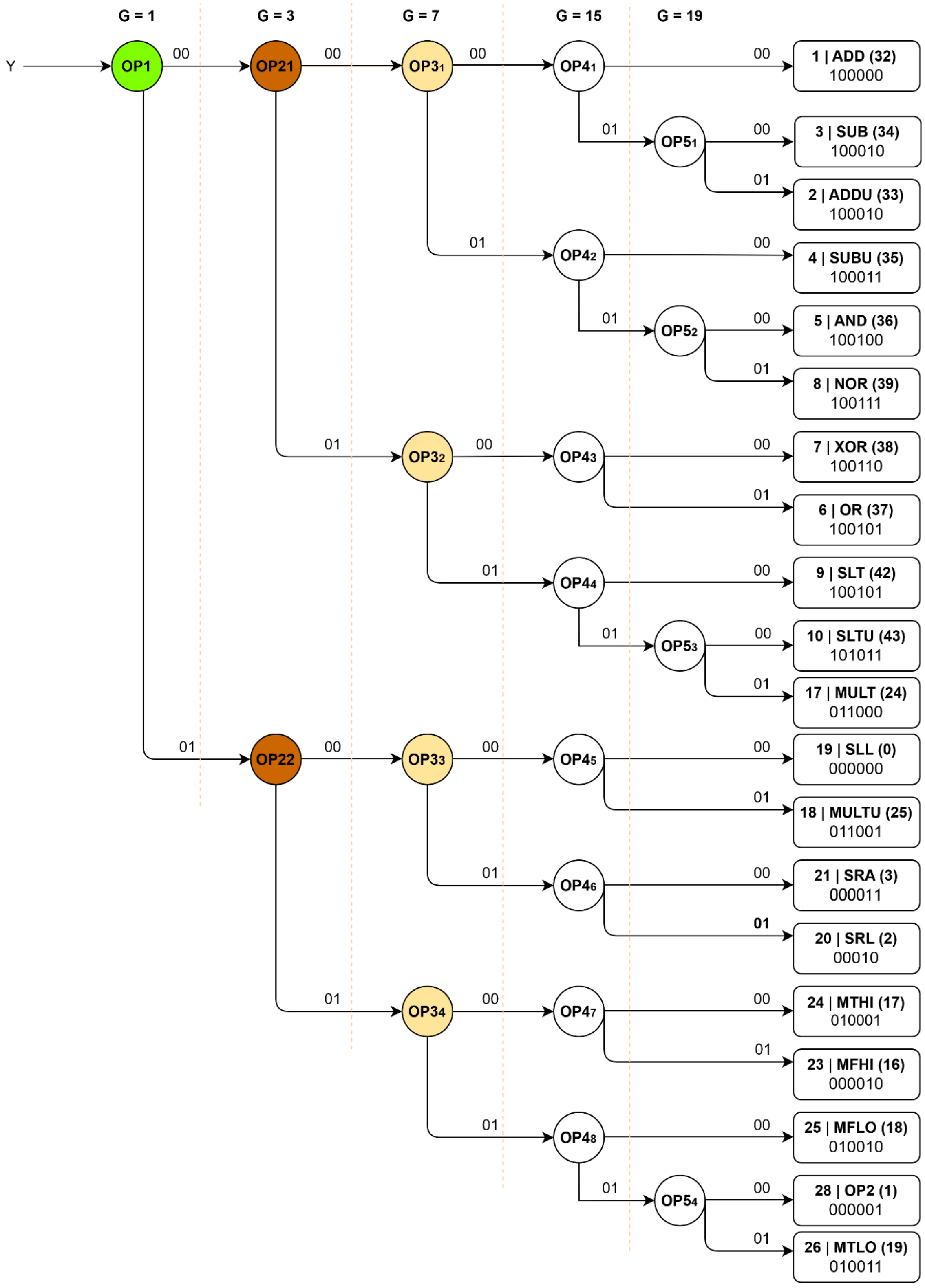


Figure 8:Generated HLDD for a subset of instructions of MP with 19 decision nodes.

In this thesis, I have developed a program to generate the HLDD which reduces the complexity of the processor for test generation. This type of approach is different from the manual approach adopted before. Thus, an approach to test data optimization for high fault coverage.

The generation of the HLDD for the selected instruction set allows calculating both the high – and low-level fault coverages. A fault of the nodes causes an incorrect leaving the path activated by a test [24].

The behaviour of the system is described in a specific working mode by each path of the HLDD [17] [3] Each of the non-terminal nodes can be referred to as a superposition(subsystem) of the entire HLDD. The following definition gives a functional meaning of the HLDD graphical representation of a system.

Definition 4.3 [3] ,[17], [70]:

A DD which represents a digital function $z = f(Z)$ is a directed acyclic graph $G_z = (M, \Gamma, X)$, where the set of nodes $M = M^N \cup M^T$ is partitioned into the subsets of non-terminal nodes M^N and terminal nodes M^T, and the set of variables $X = C \cup D$ is partitioned into the subsets of control variables C (e.g. instruction variables and data variables D (operands). A terminal node $m_T \in M_T = \{m_{T,0}, m_{T,1}\}$ is labelled by a constant $e \in \{0,1\}$ and is called a *leaf*, while all the non-terminal nodes $m \in M_N$ are labelled by variables nodes $x \in X$ and have successors whose number may be $2 \leq |\Gamma(m)| \leq |V(x(m))|$. Let us denote the associated with node *m* variable as *x(m),*

then m^0 is the successor of m for the value $x(m) = 0$ and m^1 is the successor of m for $x(m) = 1$. $\Gamma(m) \subset M$ denotes the set of all successors of $m \in M$, and $\Gamma^{-1}(m) \subset M$ denotes the set of all predecessors of $m \in M$. For terminal nodes $m_T \in M_T$ we have $\Gamma(m_T) = \varnothing$. There is a single node $m0 \in M$ where $\Gamma^{-1}(m) = \varnothing$ called root node.

The HLDD-based faults are classified into two general classes [24], [3]:

I. Control faults – These are related to the non-terminal nodes M^N.
II. Data faults – These are related to the terminal nodes M^T.

The approach in this work to test the microprocessor uses the HLDD as a fault model foundation for ISA partitioning.

The high-level fault location in the HLDD model is represented by the internal nodes in the graph [24].

The HLDD is synthesized, and since each node in the graph represents the high-level fault location, a certain structural fault collapsing resides in each HLDD nodes during the synthesis [4]. The HLDD in *Figure 8* shows the partitioned(simplified) version of a complex processor into a subset that consists of the control and data part.

As pointed out in other's work, the control path is corresponding to the non-terminal nodes in HLDD, and this is labelled by state or output variables of the control part serving as addresses or control words [43], [46]. Terminal nodes are the data path, which is labelled by the functions of data words or data words. [34] , [24], [46].
The high-level functionality of the MP is derived from the ISA.

HLDD can have just one non-terminal node with multiple terminal nodes. On the other hand, this same HLDD with one non-terminal node can be simplified to reduce the complexity of the system by partitioning the non-terminal node into multiple non-terminal nodes. Each non-terminal node represents a sub-system of the whole system. Thus, the simplification for complexity reduction was achieved in Figure 8 where we have 18 internal (non-terminal) nodes and 1 root (non-terminal) node.
It is good to also point out that various DDs, may occur where nodes have different interpretations and relationships to the system structure [3]. This kind of DDs may exist based on the representation level of the system [43]. However, in the following, relationship between SSBDDs and HLDDs are presented.

4.5 The Relationship of SSBDDs and HLDDs

SSBDDs represent structural aspects of combinational circuits [56]. Its usefulness ranges from the representation of possible structural faults in circuits, for test generation purposes.

The main goal of HLDDs was to generalize the diagnostic algorithms based on Boolean differential calculus and transformed to the graph language of BDDs for using them at higher levels of system abstraction [56] [71]. HLDDs was introduced to overcome the difficulties of high-level diagnostic reasoning of complex digital systems when using traditional hardware description languages [56].

The relationship between the SSBDDs and the HLDDs lies in the topological properties of graphs for the generalization of diagnosis algorithms from logic level SSBDDs to HLDDs. Consequently, SSBDDs that are used for representing logic circuits can be regarded as a special case of HLDDs for representing digital systems on higher abstraction levels [55], [56]. However, the difference between SSBDDs and the HLDDs could be measured from the fact that each SSBDDs node has two output edges, with two terminal nodes on the graph represented by constants 0 and 1, HLDDs differ as in the number of edges a node could have is not limited to two also it has more terminal nodes [56].

SSBDD is a model with several critical features, it uses the equivalent parenthesis form (EPF), that is, describes a digital circuit structurally [20]. [55] clearly define and compare SSBDD model with another mathematical model such as ROBDD. Consequently, the linearity of SSBDDs model in respect to the number of logic gates makes it more efficient for application use [55]. Therefore, it preserves the actual structure of the circuit, while other BDD models lack this feature [43]. Considering this fact, SSBDDs is generalized as a special DD for HLDDs [48]. Hence, these features make HLDD useful for testing at the behavioural level. This implies that testing the functionality of an MP at high-level to cover the gate-level faults inclusively without resorting to the knowledge of the implementation-details requires an HLDDs.

4.6 Benefit of HLDDs to generate test for MP

HLDD is serve as a uniform model for both the gate and RT or behavioural level simulation. Research has shown in [43] that high SAF coverage cannot guarantee high quality of testing since the types of faults that can be observed in a real gate doesn't depend only on the logic function of the gate, but also on its physical design. Thus, HLDD allows functional fault modelling of the system that helps to map faults from lower to higher levels in multi-level diagnostic modelling of systems [4].

HLDDs helps to develop uniform approaches and generalize the concepts of solving different test problems like test generation, fault simulation, fault diagnosis, testability analysis etc [24].

HLDDs fault model helps in testing exhaustively each node(sub-system) of a system. Remind that each node in the HLDD represents a fault location.

In this work, HLDD serves as a base for high-quality test generation. This means quality test cannot be generated by the test data generator without a good selection of functions under test. Hence, HLDD generalizes the base for that by allowing the logic connection of functions in non-terminal nodes which give fewer (optimized) test data. However, this does not apply to all HLDD structures. The level (quality) of test data depends on the HLDD character: good or bad. That is, if the HLDD partitioned is good then test data generation will also be good else test data generation will be bad. This raises a question which HLDD is good and which is bad? The algorithm for the good ordering of the instruction sets handles this question and it is depicted in section 5.4 of this thesis.

4.7 Synthesis of HLDDs

For the creation of the HLDD model, the functional variables of the digital system should be determined. This functional variable represents the states of the system in both, data and control units. Hence, the behaviour of the system is described as a state transfer functions and for each functional variable in the set of variables, a separate state transfer function corresponds [17]. Therefore, in this work, each transfer function is represented by an HLDD.

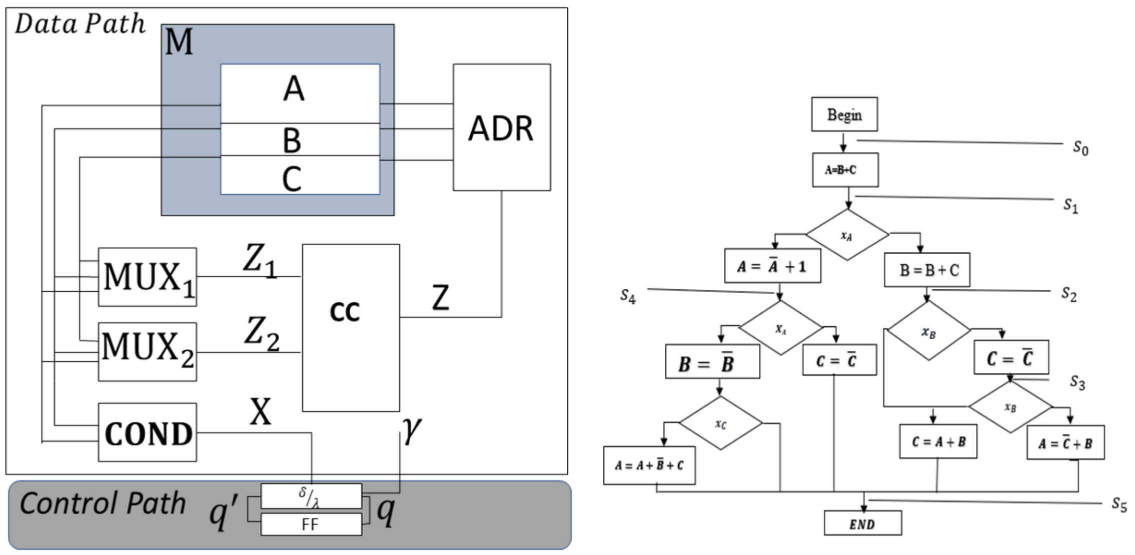


Figure 9. Digital system as a network of components and a flowchart of its behaviour.

An example of a simple digital system (a processor) is presented in *Figure 9.A*. It describes as a structural RTL circuit, consisting of control and data paths and a traditional data flow graph *Figure 9.A*. This digital system has a universe of functional variables U, which are divided into two parts. The first part represents data functional variables U_D

and the second represents the control functional variables U_C. Hence, the universe U of the functional variables represents the level of granularity through which the behaviour of the system is described. Due to this reason the universe U represents the state of the system at each step of simulating the behaviour of the system. In the data path, the universe of functional variables contains a set of data register $U_D = \{A, B, C\}$ which represent the data unit state of the system, whereas the control part has a state variable q that represents the state of the control unit. Each S_i stands as a state of the control unit as shown in the data-flow graph of the system (*Figure 9.B*).

This control and data part can be related to the HLDD generated for the work of this thesis (*Figure 8*), where the nodes of the HLDDs represent the modules of the MP, and the faults related to the HLDD nodes form the set of the representative fault of the MP under test.

There are three methods for the synthesis of HLDDs for representing digital systems [43].

I. Iterative superposition of HLDDs – this method can be used when there are different structural networks of subsystems within a system. An example of such representation of a system is depicted in *Figure 9.A.*

II. Symbolic execution of procedural descriptions – this is the functional representation of a system at higher behavioural levels. System functionality can be given inform of HDL or in a flowchart form. An example of such representation of a system is depicted in *Figure 9.B*.

III. VHLDD – this is based on using a shared HLDDs for representing the HLDD.

In this work, the iterative superposition of HLDDs and symbolic execution of procedural descriptions are considered as the hierarchical or multi-level approaches use for the synthesis of HLDDs. Hence, A certain structural fault collapsing has been performed during the HLDD synthesis, since the high-level fault locations in the HLDD model are represented by nodes in the HLDD. For instance, when testing the non-terminal nodes of the HLDD *Figure 8*, the low-level structural faults in the related control buses, multiplexers and decoders are collapsed, hence faults can be determined only in relation to the HLDD nodes.

Each non-terminal node of the HLDD represents a sub-module, and in each node resides the MP functions $f_i \in F$ which then spread at the terminal node N^T.

4.8 HLDD Conclusions

In this chapter, the usefulness of HLDD was presented. The reduction in the complexity of the MP for fault modelling and test generation was achieved based on the superposition character of the HLDD when partitioned. That means, each module in the MP performs its unique function, and each of this unique function makes the whole MP when synthesized.

Each module was presented as a node from the HLDDs. These nodes are referred to as faults location in the MP. The nodes are classified as non-terminal - where the control path of the MP flows and the terminal node - which is the data manipulation unit.

It was also shown in this chapter, the fault model method adopted for the MP test generation in the next chapter. In order to synthesis the HLDDs, two different system representation was shown in figure 9. Hence, Figure 9 explains clearly the two methods for HLDDs synthesis.

The first two methods, 1) the iterative superposition of HLDDs, which correspond to a high-level structural representation of the system and 2) the symbolic execution of procedural descriptions, which correspond to the functional representation of the system, are the hierarchical or multi-level approaches used for HLDDs synthesis. The faults influencing the behaviour of MP can be associated with nodes along the given path whereas each non-terminal node represents a structural unit or sub-circuit of the system.

The main motivation to introduce HLDDs was to improve the efficiency of test generation methods for combinational circuits by exploiting the possibility to reduce the complexity of the model compared to the traditional gate-level approaches [56]. According to [49], the one-to-one mapping techniques of the SSBDDs solve a lot of test and diagnosis related problems of digital circuits.

Hence, this chapter will serve as a guideline for test generation in the next chapter.

5 Test generation for microprocessor

The main assumption of the current approach to test generation is that the implementation of the circuits of microprocessor is not given. Hence, the approach is fully functional, and it is called implementation-independent test generation approach.

We divide the process of test generation into two sub-processes:

- test generation for the control part of MUT, and
- test generation for the data part of MUT.

This work is devoted for test generation for the control part of the MUT, where the target is to verify functionally that the instruction is correctly selected. This test is called also **conformity test**. For such a verification, one must show by suitable test data selection that the expected result of the instruction under test (IUT) is not corrupted in the process of instruction decoding by any faults in the instruction decoder. The process of test data generation will be explained in the next chapter.

For testing the data part of MUT, pseudo-exhaustive tests are generated. It is not possible to apply all possible operands exhaustively to the adder, therefore the adder is partitioned for testing purposes into bit-slices, where each bit will be tested exhaustively. For example, for testing the adder, to each bit of the adder, all possible 8 single-bit patterns are applied (exhaustive test for this bit). All bit slices are tested in parallel. From that comes the term – pseudo-exhaustive test.

5.1 Test data generation concept

During the experiments, test is generated for each MUT of the processor. A sub-circuit of the processor involved in executing the selected group of functions under test was considered, that is, the UUT within the MUT. The UUT has two high-level inputs which are data D and control signals C, and an observable output Y.
Based on the functional test approach, all functions within the MUT are well comparable due to the same observation node. What makes this approach to be high-level is because the logic structure of the UUT is unknown, and based on this fact, the traditional methods of path activation from the fault site to the observation point is obsolete. On the under

hand, since all functions of the group (MUT) under test are well-comparable, the traditional fault propagation technique is substituted by the direct generation of data needed to differ all tested faulty behaviours from the correct behaviour, that is, to activate all the high-level functional faults.

The concept is illustrated in Figure.10.

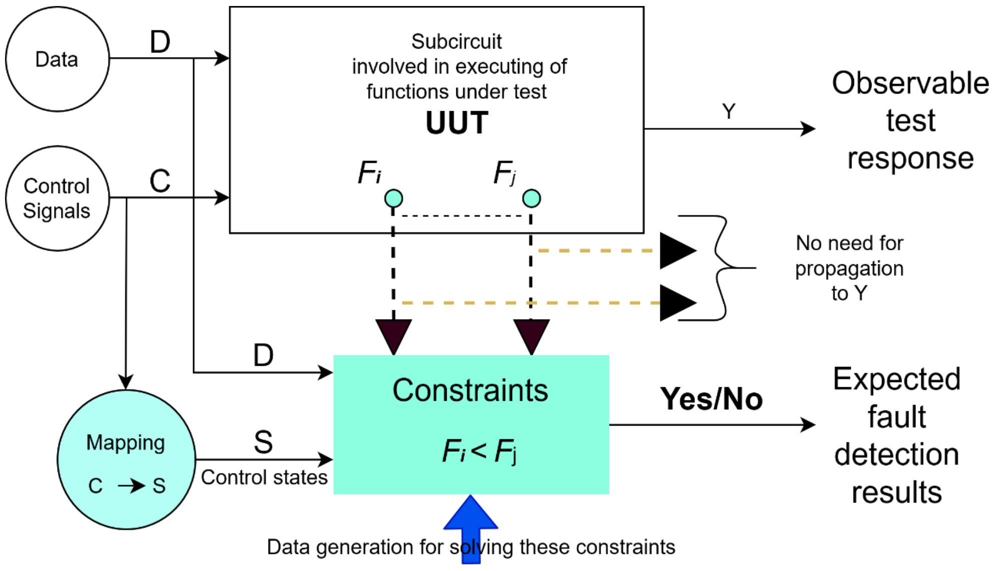

Data
D
Control Signals
C
Subcircuit involved in executing of functions under test
UUT
F_i
F_j
Y
Observable test response
No need for propagation to Y
D
Constraints
$F_i < F_j$
Yes/No
Expected fault detection results
Mapping
C → S
S
Control states
Data generation for solving these constraints

Figure 10. Illustration of Unit under Test

Consider an HLDD G^Y *(figure 8)* representing a function Y = F(X), as defined in Definition 3.2. Let the test pattern X^t, for testing the node $m \in M^N$, activates in G^Y a path $l(m_0, m^{T,v})$ from the root node m_0 to a terminal node $m^{T,v} \in M^T$, so that $x(m) = v$, and $m \in l(m_0, m^{T,v})$.

If there is a functional fault $r(m, v) \in R(m, v)$ related to the node $m \in M^N$, which causes an incorrect exit from the path $l(m_0, m^{T,v})$ and activates a wrong path $l(m, m^{T,v*})$ instead of the correct path $l(m, m^{T,v})$. Hence, at the observation point the correct path $l(m, m^{T,v})$ should produce the value $Y = f(m^{T,v})$, whereas the wrong path has change the value to value $Y = f(m^{T,v*})$. To detect this fault $r(m, v)$, the condition

$$f(m^{T,v}) \neq f(m^{T,v*})$$

must hold.

Hence, the main idea of test data generation for the control test is to develop a set of constraints, which is satisfied and guarantees that independently of the real structural implementation of the instruction decoder the instruction is correctly selected.

According to [4], test data is the determinant of the quality of the test program. The two constraints to be satisfied by a set of data operands for test data generation that detects control faults in an HL fault model are listed below [13]

$$\forall m^{T} \in M^{T}(m) \colon \exists\, \mathrm{x}^{\mathrm{t}} \rightarrow \forall\, k\, [f_{k}(m^{T}) \neq \psi)] \quad \ldots\ldots\ldots\ldots (1)$$

$$\forall m^{T,i}, m^{T,j} \in M^{T}(m) \colon \exists\, \mathrm{x}^{\mathrm{t}} \rightarrow \forall\, k\, [f_{k}(m^{T,i}) * f_{k}(m^{T,j})] \;\ldots\ldots\ldots (2)$$

Where:

Ψ means zero (0) and $a * b$ means a < b for logic OR

Ψ means one (1) and $a * b$ means a > b for logic AND

K is the number of data word bits

x^{t} is the test pattern

$f_{k}(m^{T,j})$ is the system initial state.

The constraints are summarised below:

The constraint 1: states that HLDD terminal nodes must not be equal to zero if the implementation technology of the MP is logic OR or it must not be equal to one if the logic implementation is AND.

The constraint 2: states that if it is AND logic then the result of the terminal node under test must be greater than other terminal nodes, but if it is OR logic the result of the terminal node under test must be less than the result of other terminal nodes in the HLDD.

Each constraint represents a particular high-level control fault [72].

In the following, the algorithm [17] , [73] for generating the set T(c) of test data for testing the control path is given.

Algorithm 5-1 GREEDY- test data generation for ALU

```
Input: Instruction set of the processor
Output: Sets of test operands $D_i$ for each instruction, and fault table $R$
Notations: n – number of instructions (functions $f_i$),
$d$ – test operand
$D$ – current set of selected random test operands,
$f_j(op)$ – the result of the instruction $I_j$ for the operand(s) $op$,
$R$ – fault table, $R_{ij}$ – $w$-bit entry in $R$ ($w$ – length of the data word).
1.  Initialize OP = $\emptyset$
2.  Generate a set of R random data operands
3.  for $i = 1, \ldots, n$
        ***generation of operands for instruction $I_i$
4.      Initialize $OP_i = \emptyset$,
5.      for $j = 1, \ldots, n$ ($j \neq i$)
            ***operands for solving constraints $f_{i,k} < f_{j,k}$
6.          Initialize $D_{ij} = 0$
7.          for all op $\in$ $R$ while $D_{ij} \neq 0$
                ***adding new operands for covering $D_{ij}$
8.              $D_{ij}(op) = f_j(op) \wedge f_j((op) \oplus f_j(op))$
                *** calculating fault coverage for op
9.              if ($D_{ij}$(op) $\vee$ $D_{ij}$) $\oplus$ $D_{ij}$ $\neq 0$
                    ***check for the coverage increment
10.                 begin
11.                 $D_{ij} = D_{ij} \vee D_{ij}(op)$
                    *** update of the coverage vector
12.                 Include op into $OP_i$
                    *** new operand is selected
13.                 end
14.         endfor op
15.     endfor j
16. endfor i
```

The algorithm result will be a set of operands OP_i for each I_{ij}, and the fault table D = || D^k_{ij} || where D^k_{ij} = 1 means that the functional fault described by the constraints $f_{i,k} < f_{j,k}$ is covered at least by one operand $op \in OP_i$, otherwise D^k_{ij} = 0.

The greedy algorithm solves constraints for each function by selecting the best pattern from the entire search space while random selects data sequentially.

However, two reasons could make the constrains $f_{i,k} < f_{j,k}$ not solvable: 1) due to functional fault redundancy or 2) limited search space *R*.

5.2 The roles of HLDDs in test generation for microprocessors

In this work, a new view is introduced to the test program generation using HLDDs. So far, the role of HLDDs has been to represent the high-level structure of the microprocessor or its MUT, whereas in this work, a fully implementation-independent approach to test generation is considered, and the role of HLDD will be an exploration of possible partitioning of the instruction set into subsets as targets for testing.

Let us call this approach as functional use of HLDDs for Instruction set partitioning.

So far HLDDs has been used as high-level structural model where HLDD itself represents a system, whereas the nodes represent a Module Under Test (MUT. The terminal nodes represent the data part of MUT, in more details, operational sub-modules corresponding to the instructions of the microprocessor. The internal (non-terminal) nodes, in their turn, represent the sub-modules of the control part of MUT, for example, separate decoder blocks for decoding the subfields of the instruction format. Hence, in this approach, the nodes are the target of test generation at the module level, whereas HLDD itself is the target at the system level. The test stimuli are generated for nodes (representing the submodules), but the nodes are made controllable and the responses to test stimuli will be observed at the system level.

Let us call this approach as structural use of HLDDs for node-based high-level structural testing.

For both applications of HLDDs, the concept of constructing the test program remains the same. Test programs are classified into two types [3]: conformity and scanning test programs. The conformity test program is used for testing the control part of the processor, whereas the scanning test program is used for testing the data path of the processor. The term “conformity” comes from the goal of testing the control part for compliance to specification, whereas the term “scanning” comes from the idea of testing the data part by checking its functionality with all pseudo-exhaustive patterns one-by-one.

The thesis targets the conformity test program generation and uses as the functional model of the microprocessor instruction set.

5.2.1 Conformity Test for processor

Conformity test is used for testing the control path of the processor [3], [14].

The theory and algorithm of conformity test was given in [17]. Thus, it is stated below as:

Consider an HLDD $G^{y} = (M, \Gamma, X)$ with Y = F(X), as a functional model of the instruction set of a given processor. The HLDD used in this work is depicted in Figure 8.

Where $X = C \cup D$ denotes instruction format of the processor

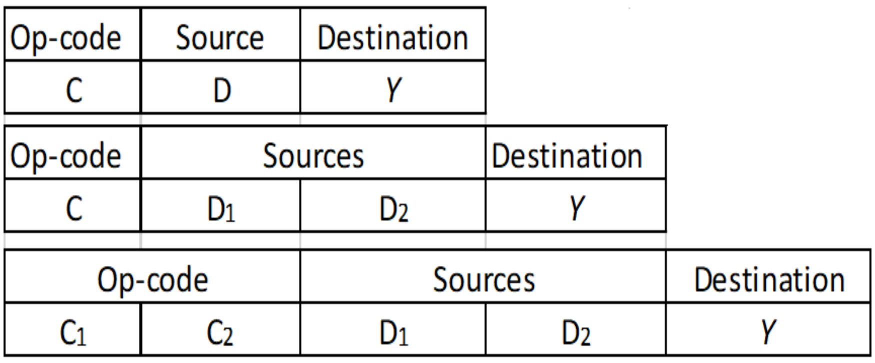
Y represents the destination,

C denotes the opcode which may be partitioned into sub-fields $C_K \in C$ of the instruction format

D denotes source which may also be partitioned into several sources $D_K \in D$

The source and the destination data variables can be referred to as the address of a register or addressable memory locations.

The following figure 11 shows an example of mapping between the processor instruction formats and the HLDD functional variables [3].



Op-code	Source	Destination
C	D	Y

Op-code	Sources		Destination
C	D_1	D_2	Y

Op-code		Sources		Destination
C_1	C_2	D_1	D_2	Y

Figure 11: Mapping between the instruction formats and the HLDD functional variable.

This implies that the target of the control test are not the instructions of the processor, because the instruction present both the control path and the data path, hence, the target of the control test is the parts of the instruction format. That is if the opcode C is partitioned into sub codes $C_K \in C$, then the control test will test each of the C_K explicitly. The general goal of this is to check if the control sub-functions are correctly selected [17], [3], [14].

Therefore, node m in the HLDD for all values $x(m) \in V(x(m))$ must be tested for checking if all control sub-functions related to C_K are selected correctly. According to [17] [3], the procedure for generating a fault $r \in R(m,v)$ is by finding a test pattern X^t that 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 ∈ *l(*$m_{0,}$ $m^{T,v}$*);* the pattern X^t corresponds to a full opcode C of instruction, which includes the value of C_K. In addition, it is also imperative to complete the pattern X^t by generating the test data *D*, so that the constraints *1 and 2* were satisfied.

The whole test with loops, can be represented as the following conformity test algorithm [17].

```
1:  for all $m \in M^N$ do
2:     for all $v \in V(x(m))$ 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 12:Algorithm for conformity test

Explanation of the algorithm is as follows:

Line 1, represents a test *T(*M^N*)* for non-terminal nodes $m \in M^N$ for the fault mode R.

Lines 2-5, At first, it initializes all registers that are involved in operation at every terminal node with data that satisfying constraints 2. Secondly, it implements the instruction to assign the value of *v* to *x(m)*, and also activates a path in G^y to the node *m, and the paths from m to* $m^{T,v} \in M^T$*(m)*; Thirdly, the value of Y is observed.

Hence, at line 6 the test for non-terminal nodes m ∈ M^{N} ends while line 7 ends the conformity test for the HLDD.

The usefulness of the conformity test is to generate a test template that will be used with the control test patterns. The conformity test created a test template as seen in figure 14.

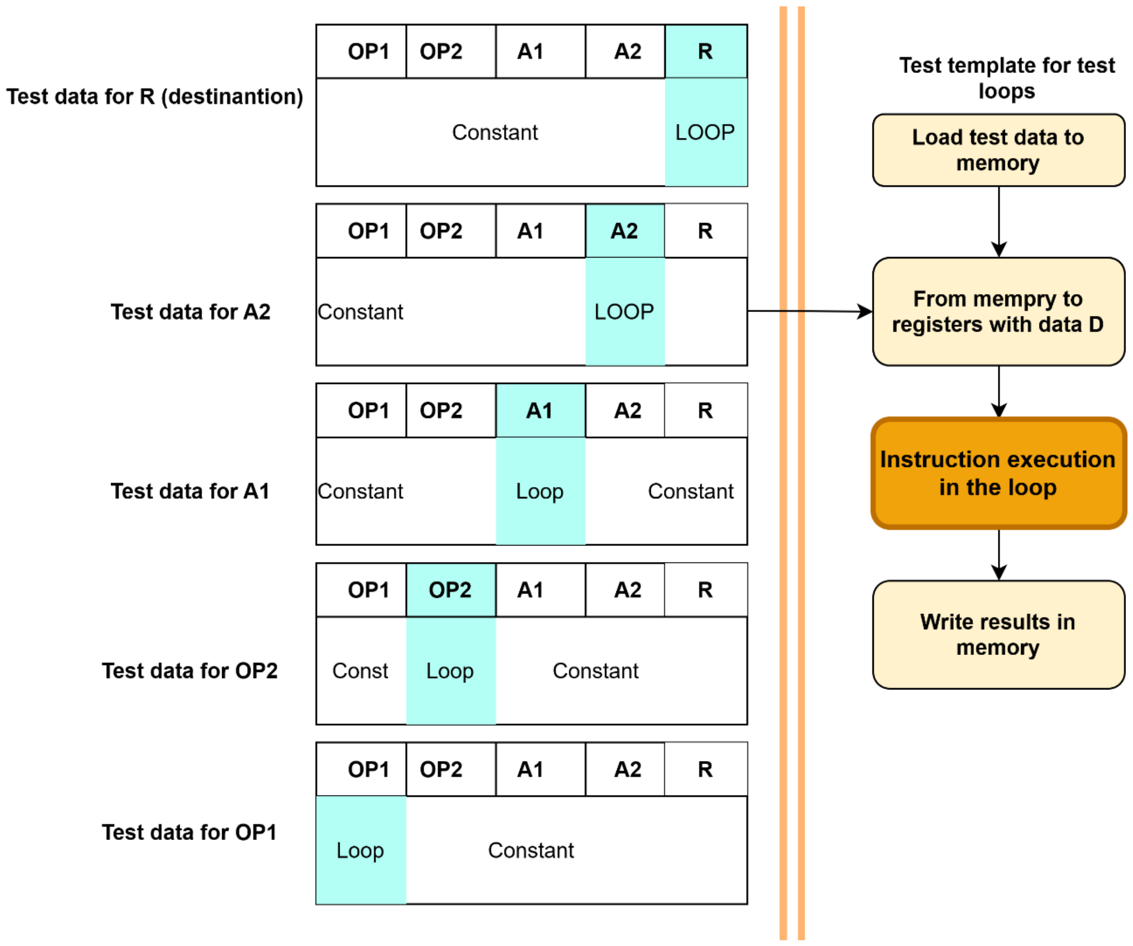


Figure 13: Structure of Conformity test

The OP1, OP2, A1, A2 and R represents the non-terminal node (control node) of the HLDD. Each control node has a test template that is made up of instructions which enables a node to be tested. Hence, the HLDD in figure 8 categorizes the instructions based on the non-terminal nodes they are connected to. Therefore, the instructions connected to the node are used for testing a node of the HLDD [14].

5.3 Test program generation

In accordance with the described method of generating test data for testing a processor, test is organize based on the architecture shown in figure 14. The full test is organized into different parts, where the target of each part is to test a subset of functions F and same template is used for testing each function $f_i \in F$.

The full test is structured into three embedded loops. The same test template is use for the first outer loop. Ideally, the number of loops and the number of different test templates are equal, hence, it is a subroutine with a uniform structure that has three parts: 1) initialization of the processor, 2) execution of the instruction so that for each function $f_i \in F$ the response are propagating to the node of observation. Since the same test template is use for each function, the second middle loop handles the selected test template by repeating it for all functions $f_i \in F$ for a list of related Instructions I_i to be inserted into the current template. To address A_1, A_2, the data operands d_1, d_2 is reference by each instruction pattern $I_i = (\text{opcode}_i, A_1, A_2)$. Two consecutive loops will be executed for testing the control part and the data part of each instruction $I_i \in F$ under test. The test *data* $d = (d_1, d_2) \in D_i^*$ is use for testing the control part, whereas for testing the data part the number of test patterns use is determined by the length of pseudo-random test sequence.

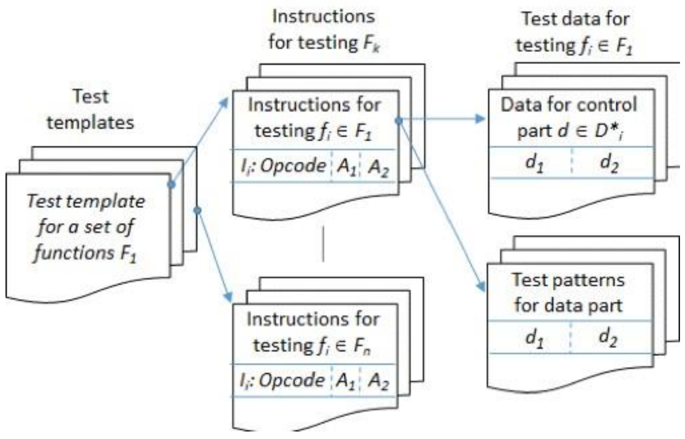

Test templates
Test template for a set of functions F_1
Instructions for testing F_k
Instructions for testing $f_i \in F_1$
I_i: Opcode A_1 A_2
Instructions for testing $f_i \in F_n$
I_i: Opcode A_1 A_2
Test data for testing $f_i \in F_1$
Data for control part $d \in D^*_i$
d_1 d_2
Test patterns for data part
d_1 d_2

Figure 14. Architecture of the test program [74]

The novelty of the proposed test method stands in on-line test generation based on unrolling on the fly the stored in compact way of all needed test information in the form of the sets of test program templates, test instructions and related test data lists.

According to [17] the test program generation for a processor using the HLDD can be achieved in two levels. These are system and module levels. Nodes are the target of test generation at the module level, whereas HLDD itself is the target at the system level. This means that modules will be combined to form the system level since each non-terminal node on the HLDD represents a module. Hence, the results of tests will be observed at the system level, once the test stimuli for the modules have been made controllable.

Test program are divided into two types: conformity and scanning test programs. The conformity test program is use for testing the control part of the processor, whereas the scanning test program is use for testing the data path of the processor.

5.4 Using HLDDs for instruction set partitioning

A new paradigm for generating the HLDD was developed during the research work of this thesis. The program is used for the partitioning instruction set of an MP on an HLDD. The picture is depicted in *Figure 8*. Thus, this program is referred to as HLDD generator.

.

The inputs for the HLDD generator are:

I. number of instructions.

II. data patterns $D = d^k{}_{ij}$.

The output for the HLDD generator is:

I. an HLDDs.

High-level test data generator uses deterministic test data generation algorithm to generates data patterns $d^k{}_{ij}$ which distinguish each instruction, and the HLDD is generated by the HLDD generator.

The steps for the HLDD generator Algorithm are as follows:

1. I started creating a 2D matrix of n(n) size
   a. Inside each matrix cell, resides the data pattern $d^k{}_{ij}$.
      i. The data $d^k{}_{ij}$ are calculated by the high-level fault simulator for all instruction f_i executed by the processor.
   b. Each data pattern $d^k{}_{ij}$ is assigned weight $d^k{}_{ij}(w)$
2. A loop is created that iterates through the matrix to hold each cell as a 2-dimensional array except $when\ i = j$;
   a. the diagonal cells when $i = j$ from the list of indices were not considered as it serves as redundancies of the functional faults,
   b. where $i \neq j$ indexes of f as $f_i, f_j \in F$ so that for each $f_i\ and\ f_j$ are concatenated together to hold each data pattern $d^k{}_{ij}$ and weight $d^k{}_{ij}(w)$ for each f_{ij}.
3. A list L is created to get all the data pattern $d^k{}_{ij}$ matrix with respect to weight $d^k{}_{ij}(w)$, then
   a. the data pattern $d^k{}_{ij}$ for each $f_i = f_j$ are taken as a list l_i, where $l_i \in L$;
   b. the weight $d^k{}_{ij}(w)$ are taken for each l_i,
4. The sum of weights S^w for each l_i in L selected for partitioning is calculated as $S^{W(l_i)}$.
5. An algorithm is then developed for the generation of HLDD, taking in the list of instruction based on the sum of weight for each list $I_i(S^{W(l_i)})$ as an input.

Assume, we have the following matrix where size = n(n) where the f_i and f_j have n size each. Let n be 4. Note n denotes the numbers of instructions selected for partitioning. Analysis of the Algorithm for HLDDs generation:

Table 5: Sample of Matrix table for partitioning 4 instructions.

	1	2	3	4
1	■	25	20	5
2	10	■	10	12
3	5	4	■	6
4	2	6	5	■

Assume, each element in the matrix is the number of useful patterns generated by the high-level test generator. Note, that the diagonal cells are not considered.
Weight is assigned to these data patterns based on their hierarchy. See *Table 6*.

Table 6. Shows how weights are assigned to patterns

Patterns	25	20	5	10	10	12	5	4	6	2	6	5
Weights	8	7	3	5	5	6	3	2	4	1	4	3

It is possible to have two or more instructions using the same numbers of patterns with equal test length or different numbers of patterns with equal test length. Therefore, using the same weight value. For instance, pattern 10 occur twice and have the same weight value of 5. *Table 7* shows how each row and column is divided into four parts with each having three unique matrix slots for the case of four instructions under partitioning. For each cell in the matrix, weight is assigned to a pattern.

Table 7. *Expansion of Table 6*

no	Matrix Slot	no of pattern	Weight
1	[1 < 2]	25	8
2	[1 < 3]	20	7
3	[1 < 4]	5	3
4	[2 < 1]	10	5
5	[2 < 3]	10	5
6	[2 < 4]	12	6
7	[3 < 1]	5	3
8	[3 < 2]	4	2
9	[3 < 4]	6	4
10	[4 < 1]	2	1
11	[4 < 2]	6	4
12	[4 < 3]	5	3

Note that for four instructions selected for partitioning we have 12 possible slots. Hence, it requires 12 data patterns to be generated for distinguishing those functions. Logically, the number of length of patterns grows exponentially regarding the number of instructions set under consideration for partitioning. In our experiment, we have used 20 ISA for testing the miniMIPS ALU. Therefore, the number of data patterns used for distinguishing those functions is 380 and that can be calculated with the following formula.

$$Number\ of\ generated\ data\ patterns = \Big(\big((n * C_r) * (n * C_c)\big) - n\Big)$$

Note, the formula is also the size of the entire location under consideration in the matrix. Where:

n - denotes the size

C_r - denotes the row cell

C_c -denotes the column cell.

With more ISA under consideration to be partitioned on HLDD, we can see that we are prone to errors if things are done manually. Thus, it can make the test generation and fault coverage inefficient.

Each cell was combined where there is the same sequence of occurrence in indexes into a unique list, i.e. we create a list to get the corresponding instruction patterns its weights. This statement is depicted in *Table 8.*

Table 8: Shows the list of patterns and weights based on relationship.

no	Corresponding Patterns	no	Corresponding Weights
1	[25, 20, 5, 10, 5, 2]	1	[8, 7, 3, 5, 3, 1] = 27
2	[25, 10, 10, 12, 4, 6]	2	[8, 5, 5, 6, 2, 4] = 30
3	[20, 10, 5, 4, 6, 5]	3	[7, 5, 3, 2, 4, 3] = 24
4	[5, 12, 6, 2, 6, 5]	4	[3, 6, 4, 1, 4, 3] = 21
Total sum of weights for 4 instructions			102

The sum from the list was then taken for all the indexes where each instruction resides. The following matrix tables show the graphical selection cases for Table 8.

Table 9: Selection case 1

	1	2	3	4
1	■	**25**	**20**	**5**
2	**10**	■	10	12
3	**5**	4	■	6
4	**2**	6	5	■

Table 10. Selection case 2

	1	2	3	4
1	■	**25**	20	5
2	**10**	■	**10**	**12**
3	5	**4**	■	6
4	2	**6**	5	■

Table 11. Selection case 3

	1	2	3	4
1	■	25	**20**	5
2	10	■	**10**	12
3	**5**	**4**	■	**6**
4	2	6	**5**	■

Table 12. Selection case 4

	1	2	3	4
1	■	25	20	**5**
2	10	■	10	**12**
3	5	4	■	**6**
4	**2**	**6**	**5**	■

Algorithms for Cells Selection

```
self.lis = []  ***creates list to get pattern matrix with respect to weights
for m in range(1, self.size):
    l = []
    for i in range(1, self.size):
        for j in range(1, self.size):
            if (i != j):
                if (i == m or j == m): ***if true append the index of that slot to a list
                    l.append(weights.index(self.slots[i][j]) + 1)
    self.lis.append(l)
```

The HLDDs is then generated recursively for the instruction set $I_i(S^{W(l_i)})$ based on the result in Table 8. The algorithm for the HLDDs generation is shown below:

Algorithm for HLDD generation

```
def hldd(self, instructions):  *** Inputs are the instructions of miniMIPS
    total = 0
    vals = []
    for i in Ins:
        total += self.sumlis[i - 1]  ***get Sum list of patterns for each ins.
        vals.append(self.sumlis[i - 1])
    half = total // 2
    s = 0
    firsthalf = []
    sechalf = []
    for i in range(len(vals)):
        if s + vals[i] <= half:
            s += vals[i]
            firsthalf.append(instructions[i])
        else:
            sechalf.append(instructions[i])
    print(instructions, '->', firsthalf, ' and ', sechalf)
    if (len(firsthalf) >= 2):
        self.hldd(firsthalf)
    if (len(sechalf) >= 2):
        self.hldd(sechalf)
```

Since there are one row and one column for every node in the HLDD, the number of edges required to fill the matrix is $|V|^2$. To this light, the topology of the instructions under test for the given MP is well interrelated. Therefore, a set of digital functions $y = F(X)$ of components or sub-circuits in digital systems may be represented as graph [4], [38].

The graphs generated through this program were synthesized directly from the topology of the gate-level network. Such a topological view allowed to generalize the knowledge and methods of test synthesis and fault analysis from the Boolean level to higher register-transfer and behaviour levels of digital systems by introducing High-level DDs (HLDD) [56], [49].

5.5 Conclusion

This chapter explained the concept of test generation use in this work. The algorithm for the HLDDs generation developed in this work was presented which is an optimized approach of the previous HLDDs partitioning approaches used in [2] and [33]. This model will be used for the Experimental section for the SBST in the next chapter.

6 Experimental part of the thesis

This chapter presents the research environment used during the research of this work, and the experimental results for the software-based tests of an MP.

6.1 SBST and Experimental Contributions

SBST is the process by which a microprocessor uses its resources to test itself. In this case, we do not rely on external testers. Remind that an instruction set architecture is an abstract representation of a processor and its functionality provided in the architecture documentation [4], [3]. Therefore, in order to test the processor, its instructions set are used. The test framework is divided mainly into three parts:

- test data generations.
- test program generation.
- test evaluation

Figure 15: The tasks of SBST generation flow in this thesis

The following shows the tool framework used for the SBST process.

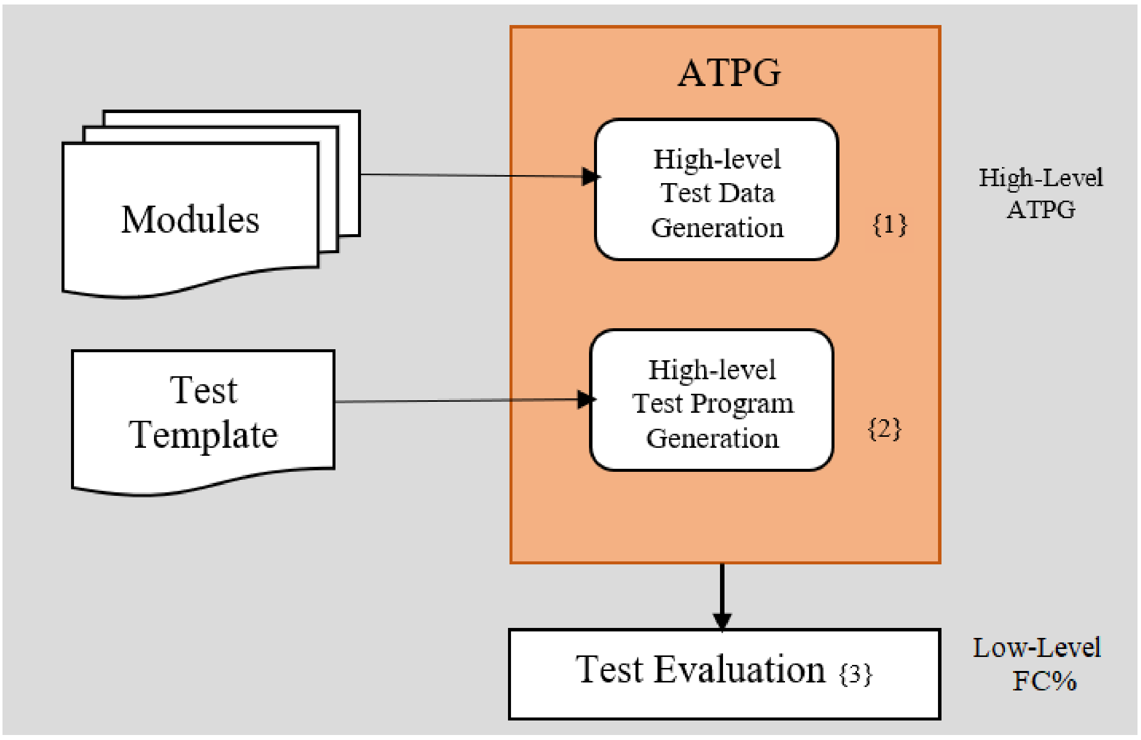


Figure 16: Tool framework for test generation.

The inputs to the tool are modules and the test templates. The modules are from the instruction set architecture of the processor when the complexity of the MP under test is reduced by partitioning the instructions set into modules. This we can see from the set of instructions on the non-terminal nodes of the HLDD in *Figure 8*. As each non-terminal node of the HLDD represents a module. The high-level test data generator for the control part of the processor, uses a novel constraint-based functional high-level (HL) control fault model in section *5.1*, and generates tests to verify that all functions of nodes are selected correctly. This is called as conformity test and is described in [13].

The SBST generation flow is listed below:

1. Data generation
2. Manual preparation
3. Program generation
4. Program compilation
5. Test execution
6. Fault grading/simulation

6.2 Research environment

The process begins with simulating the functions of modules with random data using logic simulator. The simulator *Questasim* generates for each data, a set of functional outputs $y_i = f_i(d_i)$ for every instruction I_i. Then the functional output from the logic simulator is passed as input to high-level ATPG for control test data generation (1), the HL ATPG is based on two algorithms, which are *greedy* and *random* HL test data generation. Step 2 is the manual preparation of test templates. Test templates and test data serve as inputs for the test program generator (3) that generates a test program. The test program then serves as input to the processor compiler (4). The compiler compiled the test program and outputs binary test program which then serves as input to the processor under test (5). Then a .vcd dump file contains all signal value changes as the processor executes is created. The dump file is then used for grading by *Tetramax* during fault simulation (6). The experiment set-up for SBST applications for the given processor is depicted in *Figure 17* below.

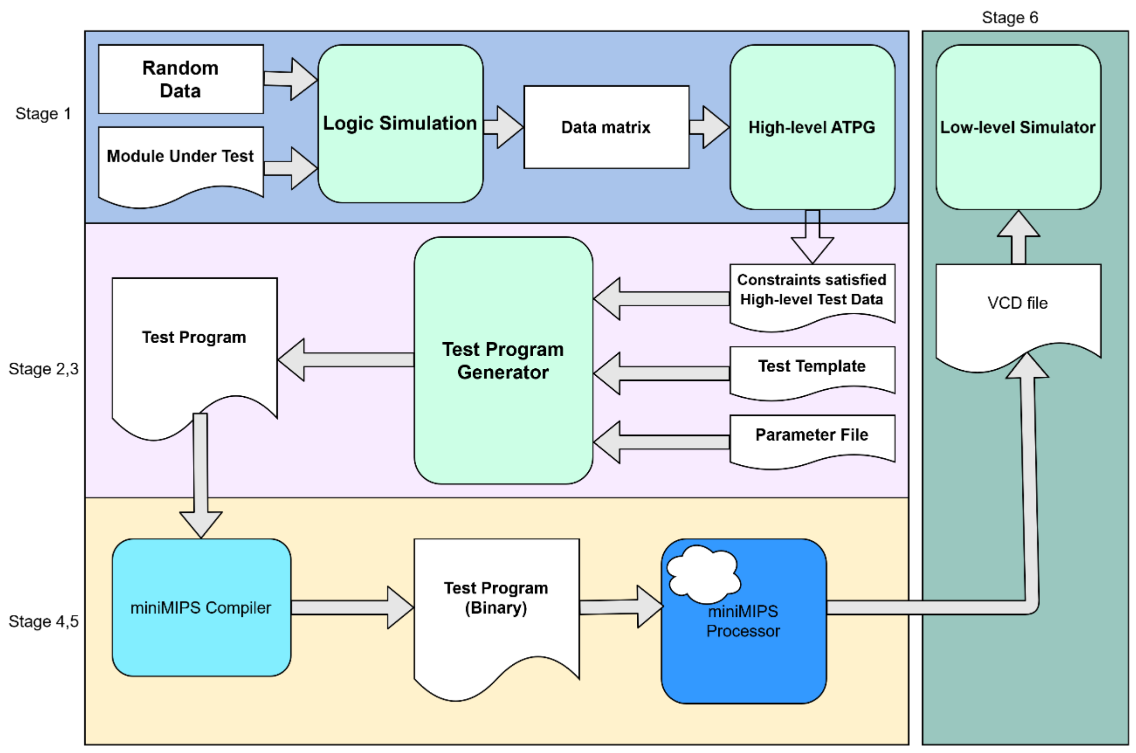


Figure 17: High-Level tool framework in action for test generation.

6.3 Experimental research on test generation for MiniMIPS

I carried out experiments to compare the HLDD-based structural test generation approach, which was the basis of the concept, recently published in JETTA paper [74] with the new HLDD-based functional test generation approach developed in this work.

In both approaches we consider for testing the control part of MUT for MiniMIPS microprocessor, representing a part of ALU, and given as a set of instructions listed in Table 4. The MUT is represented as the HLDD model in Figure 8.

As it was explained, the 19 internal nodes OP1, OP21, …, OP54 in Figure 8, represent the control part of MUT, and the 20 terminal nodes labelled by decimal numbers of opcodes and related instruction acronyms.

Note, the HLDDs can be represented in different modifications, using different numbers of decision nodes, however, having always the same number of terminal nodes, equal to the number of instructions of the MUT.

6.3.1 High-Level structural test generation with HLDDs

This approach can be called as a **node based** high-level functional test generation for the control part of MUT, where the targets of test generation are the HLDD nodes.

In this approach MUT represents by a single HLDD where for each node a high-level functional test is to be generated. For example, the HLDD in Figure 8 consists of 19 internal nodes, hence for all of them 19 test data must be generated.

Let us denote the HLDD model representing the MUT by notation G = n, where n – means the number of internal nodes in the HLDD. For example, the HLDD in Figure 8 will have a notation G = 19. In Figure 19, as an example two HLDDs with structures notated as G = 1 and G = 3, are depicted.

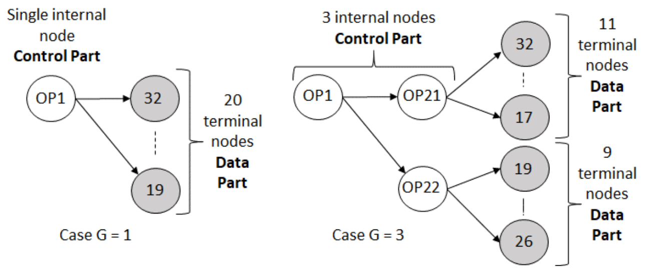


Figure 18: Structures notated as G=1 and G =3

A set of experiments were carried out for 5 different structures of the HLDD in Figure.8, highlighted by vertical levels, and representing the following models:

G=1: {OP1},

G = 3: {OP1, OP21, OP22},

G=7: {OP1, OP21, OP22, OP31, …, OP34},

G=15: {OP1, OP21, OP22, OP31,…,OP34, OP41,…,OP44, OP51,…,OP54}, and

G=19: {OP1, …, OP54}

As it was explained, for each HLDD node *m* with *n* output edges, *n* terminal nodes will be chosen, which determine a subset of functions $F(n) = \{f_i\}$. The subset $F(n) \subseteq F$, where *F* is the full set of functions of MUT, determines in its turn the set of constraints (1) and (2) to be satisfied by selecting proper test data. The number of these constraints is equal to $n(n\text{-}1)$, which is interpreted as the number of high-level functional faults of the HLDD node m under test.

The results of the test generation experiments for the 5 versions of HLDD models are depicted in Table 13.

The full set of high-level functional faults is equal to $N(N\text{-}1)$, where N – is the number of all functional faults of the MUT. For example, for the HLDD in Figure 8, N = 20, and the full number of high-level functional faults for the MUT is 10136, which corresponds to the HLDD version G = 1. For all other HLDD versions derived from the HLDD in Figure 8, i.e. For G=3, G=7, G=15, and G=19, the number of faults is less than for G=1, as it can be seen from Table 13.

Table 13. Node-Based Test Generation fault coverage result.

Test	**Partition**	**Pattern (Control) #**	**Time sec**	**Number of High-Level faults N**	**SAF Coverage(%)**					**High-Level FC(%)**			**Detected Faults**	**Real Global Detected Faults**
					Adder	Mult0	Mult1	ALU	PPS-EX	Local	Global	Real Global		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Control fault test data gene-ration	G=1	143	78.44	12160	100	99.39	99.37	99.03	98.02	100	100	100	10246	10246
	G=3	116	44.3	5888	100	99.2	99.23	98.87	97.85	100	47.56	100	4873	10246
	G=7	93	29.97	2816	100	99.06	99.03	98.72	97.71	100	22.36	99.91	2291	10237
	G=15	75	21.15	1472	99.96	99.11	99.01	98.73	97.71	100	11.24	99.93	1152	10239
	G=19	72	22.53	1216	99.92	99.01	98.87	98.6	97.59	100	10.5	99.9	1078	10236

The columns in Table 13 represent the following: 2 – type of the HLDD, 3 – number of test data generated, 4 – time used for test data generation, 5 – number of high-level functional faults represented in the model, columns 6-10 represent the gate-level SAF coverages achieved by high-level implementation-independent test generation, columns 11-13 represent the high-level fault coverages, where „Local coverage“ refers the set of covered targeted faults given in column 5, „Global coverage“ means the percentage of targeted and covered faults in relation to all 12160 faults, and „Real Global coverage“ means the percentage of all covered faults in relation to all 12160 faults as an added value found by fault simulation; the columns 14 and 15 are related to the columns 12 and 13, respectively.

Table 13 shows, that the bigger is the set of faults to be tested (column 5), the bigger will be the set of test data generated (3), the longer is the test generation time (4), the higher are the gate-level SAF coverage (6-10), and the high-level real global fault coverage (13, 15).

Let us define as the baseline (state-of-the-art) the case G=1, which coincides with the method developed in [74].

From above, the following conclusions follow:

Statement 1. From Table 13 it follows that compared with the baseline G=1, all other versions of HLDDs G=3, G=7, G=15, and G=19 outperform the baseline in test data volume and speed of test generation, whereas slightly loose in gate-level fault coverage.

Statement 2: Table 13 demonstrates a trade-off possibility between the test data volume and test generation time on one hand and fault coverage (test quality) on the other hand.

Statement 3: From the experimental results in Table 13, it follows that when trading off different parameters, the test data volume can be reduced two times, and test generation time can be reduced even 3-4 times at the maximum minor loss in gate-level fault coverage 0,43% for ALU.

6.3.2 High-Level functional test generation with HLDDs

This approach can be called as a **partition based** high-level functional test generation for the control part of MUT, where HLDDs are used for partitioning the sets of instructions of MUT, and the targets of test generation are the partitions of instructions.

In this approach MUT represents by a set of single internal node HLDDs, which determines a partition of instruction, where for each single internal node a high-level functional test is to be generated. Let us denote the set of single internal nodes HLDDs representing the MUT by notation Gn, where n – means the number of partitions of the instruction set of MUT. For example, the set of HLDDs G8 in *Figure 20* consists of 8 HLDDs, derived from the HLDD in Figure 8, which determine the following partition of instructions {(32,34,33), (35,36, 39), (38,37), (42,43,24),(0,25), (3,2), (17,16), (18,1,19)}.

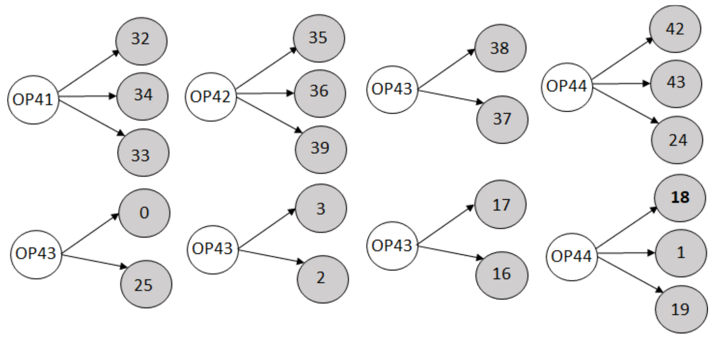


Figure 19: Set of HLDDs G8

A set of experiments were carried out for 4 different sets of the single internal node HLDDs, derived from Figure 8, representing the following partitions of instructions:

G1: {32, 34, …, 19} – single partition including all terminal nodes (1 HLDD),

G2: {(32,34, …, 17), (19,18,…, 26)} – 2 partitions (2 HLDDs),

G4: {(32,34,…,39), (38,37,…, 24), (0,25,3,2), (17,16,18,1,19)} - 4 partitions (4 HLDDs)

G8: {(32,34,33), (35,36, 39), (38,37), (42,43,24),(0,25), (3,2), (17,16), (18,1,19)} - 8 partitions.

The results of the test generation experiments for the 4 versions of partitions are depicted in Table 14.

Table 14. Partitioned-based approach fault coverage result.

Test	**Partition**	**Pattern (Control) #**	**Time**	**Number of High-Level faults N**	**SAF Coverage(%)**					**High-Level FC(%)**		**Detected Faults**	**Real Global Detected Faults**
					Adder	Mult0	Mult1	ALU	PPS-EX	Local	Real Global		
1	2	3	4	5	6	7	8	9	10	11	13	14	15
Control only	G1	143	78.44	12160	100	99.39	99.37	99.03	98.02	100	100	10246	10246
	G2	111	43.01	5824	100	99.18	99.21	98.86	97.84	100	100	4810	10246
	G4	83	25.9	2624	100	98.95	98.88	98.59	97.58	100	99.91	2100	10237
	G8	49	12.7	1024	99.88	98.65	98.46	98.24	97.24	100	99.49	703	10194

The columns in Table 14 are the same, so that the two experiments could be easily compared. Table 14 shows similarly as in Table 13, that the bigger is the set of faults to be tested (column 5), the bigger will be the set of test data generated (3), the longer is the test generation time (4), the higher are the gate-level SAF coverage (6-10) and the high-level real global fault coverage (13, 15).

From above, the following two conclusions follow in comparison with the baseline of state-of-the-art [74] which corresponds to the HLDD model G=1, and in comparison, between the two proposed in thesis approaches:

Statement 4. From Table 14, it follows that compared with the baseline G=1 (in Table 13), all test experiments with HLDD models G1, G2, G4, and G=8 outperform or are equal with the baseline in test data volume and speed of test generation, whereas slightly loose in gate-level fault coverage.

Statement 5: Table 14 demonstrates, similarly to Table 13, a trade-off possibility between the test data volumes and test generation time on one hand and fault coverage (test quality) on the other hand.

Statement 6: From the experimental results in Table 14, it follows that when trading off different parameters, the test data volume can be reduced 2.5 times, and test generation time can be reduced even 6 times at the maximum minor loss in gate-level fault coverage 0,8% for ALU.

7 Conclusion

This thesis has addressed a set of test issues in the field of design for testability, and particularly, in testing of RISC-based processors by developing a methodology for generating an efficient test data for Software-Based Self-Test (SBST).

In the thesis, a new method is developed for SBST of RISC type processors, using the High-Level Decision-Diagrams (HLDD) theory. The method follows the idea of implementation-independent test generation for microprocessors [JETTA] proposed recently, where as input information about the processor to be tested, only the instruction set is given, and no knowledge about the low-level details about the gate-level structure of the processor is needed.

The target of the current work was to reduce the complexity of the model by partitioning the instruction set into subsets and generating the test data for tese subsets separately with following joining all test patterns together. For partitioning, the HLDD theory was used.

In the thesis, a method, algorithm, and a software tool for automated synthesis of the optimized HLDD model were developed. It was also shown that such a HLDD model allows always to derive a set of different partitionings of the instruction set. This set could be used for trading off between three different characteristics of the test data to be used for SBST: test quality, test generation time, and test length.

The synthesized from the instruction set HLDD allows straightforwardly to derive the best solution either, on one hand, regarding the test quality at not minimized time cost and test length, or on the other hand, regard minimum time cost and test length at the expense of slightly reduced (not more than 0,8%) fault coverage (as the test quality paramether).

As special case, the synthesized HLDD model involves also the best solution that can be generated by the recently developed method in [74].

Three main contributions of this thesis are summarised as follows:

- An algorithmic method, and a tool for optimized partitioning of the instruction sets of microprocessors for test data generation were developed, which help to reduce the complexity of test generation, compared with known similar methods.

- The feasibility and efficiency were experimentally tested by generating test data for testing the control part of the Execute Unit of a real RISC-type microprocessor MiniMIPS.
- Optimized test data were generated, using the developed HLDD synthesis tool in combination with other professional and home-made tools available in the lab.
- High quality test patterns were achieved, which outperform regarding the test length, and are better or equal to the results of state-of-the-art similar implementation-independent SBST generation methods.

The results of the thesis were submitted as a research paper to the Conference of European Association for Education in Electrical and Information Engineering (EAEEIA).

8 References

[1] Ubar, R , “Test Generation for Digital Circuits with Alternative Graphs.,” *Proceedings of Tallinn Technical University No 409,* pp. 75-81, 1976.

[2] A.S. Oyeniran, R. Ubar, M. Jenihhin, J. Raik, “On Test Generation for Microprocessors for Extended Class of Functional Faults,” *Tallinn University of Technology, Estonia.*

[3] Artjom Jasnetski, “Software-Based Self-Test for Microprocessors with High-Level Decision Diagrams,” in *Tallinn University of Technology,* April 2018.

[4] A. Jasnetski, R. Ubar, A. Tsertov, and M. Brik, Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams, Proc. Est. Acad. Sci., vol. 63, no. 1, pp. 48–61, 2014.

[5] S. M. Thatte and J. A. Abraham, , “Test Generation for Microprocessors,” *Transactions on Computers ,* Vols. C-29, pp. 429-441, 1980.

[6] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, Matteo Sonza Reorda, Microprocessor Software-Based Self-Testing.

[7] C.H Sequin and D.A Patterson, “Design and Implementation of RISC 1”.

[8] I.Bayraktaroglu, J.Hunt, D.Watkins., “Cache resident functional microprocessor testing:Avoiding high speed io issues.,” *IEEE Int.Test Conference,* 2006.

[9] E. Sanchez and M. S. Reorda, "On the Functional Test of Branch Prediction Units," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 9, pp. 1675 -1688, Sept. 2015.

[10] A.Reifert et al., “On the Automatic Generation of SBST Test Program for In-Field Test,” pp. 1186-1191, 2015.

[11] N.Kranitis, A.Paschalis, D. Gizopoulos and G. Xenoulis, Software-based self-testing of embedded processors, in IEEE Transactions on Computers, vol. 54, no. 4, pp. 461- 475, April 2005.

[12] P.Thevenod-Fosse and R.David, “Random testing of the control section of a microprocessor,” *in Proc. Fault-Tolerant Computing Symp., Milan, Italy, June,* pp. 366-373, 1983.

[13] A.S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors.

[14] M.O. Oloruntobi, A Method for Synthesis of Self Test Software for Microprocessors, Tallinn University of Technology, 2018.

[15] G. Squillero, MicroGP evolutionary assembly program generator, Genetic Programming and Evolvable Machines, vol. 6, no. 3, pp. 247– 263,, 2005.

[16] V.M. Suryasarman, S.Biswas, and A. Sahu, Automation of test program synthesis for processor post-silicon validation, Journal of Electronic Testing, vol. 34, no. 1, pp. 83–103, Feb 2018.

[17] R.Ubar, A.Jasnetski, A.Tsertov, A.Oyeniran, Software-Based Self-Test with Decision Diagrams for Microprocessors.

[18] J.P.Roth., Diagnosis of Automata Failures: A Calculus and a method. IBM J.Res. Develop., Vol. 10, No. 4, pp. 278-291, July 1966.

[19] C. H -. Wen, L. C. Wang and Kwang-Ting Cheng, "Simulation-based functional test generation for embedded processors," Proc. Tenth IEEE International High-Level Design Validation and Test Workshop, pp. 3-10, Napa Valley, CA, 2005.

[20] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, Automatic test program generation: A case study, IEEE Design & Test of Computers, vol. 21, no. 2, pp. 102–109, 2004.

[21] G. E. Moore, , "Cramming More Components Onto Integrated Circuits, Electronics, April 19, 1965," Electronics, vol. 38, no. 8, pp. 82–85, 1965.

[22] A.Kristic, K.T.Cheng. , "Delay Fault Testing for VLSI Circuits.," *Dordrecht, The Netherlands, Kluwer Academic Publishers,* p. 1998, Oct.

[23] G. E. Moore, ""Cramming More Components onto Integrated Circuits, Electronics, April 19, 1965," Electronics, vol. 38, no. 8, pp. 82–85, 1965".

[24] R. Ubar, Ondrej Novak, Elena Gramatova, "Handbook of Testing Electronic Systems".

[25] Taltech digital Library, "https://digi.lib.ttu.ee/i/file.php?DLID=803&t=1".

[26] Abramovici, M., Breuer, M.A., Friedman, A. D, "Digital Ssytems Testing and Testable Designs," *Computer Science Press,* p. 653, 1995.

[27] R. Ubar, "Testing with alternative graph, design and test of computer 1996".

[28] Plakk, M., Ubar, R, "Digital Circuit Test Design Using Alternative Graph Model," *Automation and Remote Control, no. 5, part 2, Plenium Publishing Corporation, USA,* vol. 41, pp. 714-722, 1980.

[29] W. Laung-Terng, W. Cheng-Wen, and X. Wen, VLSI Test Principles and Architectures: Design For Testability. San Francisco:, 2006.

[30] A. Krstic and L. Chen, "Embedded Software-Based Self-Test for Programmable Core-Based Designs," IEEE Des. Test Comput., vol. 19, no. 4, pp. 18–27, 2002.

[31] D. Brahme, J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Transactions on Computers,* Vols. C-33, p. 475, June 1984.

[32] T.Sridhar and J.P. Hayes, "A functional approach to testing bit-sliced microprocessors," *IEEE Trans. Comput.,,* Vols. C-30, pp. 563-571, Aug.1981.

[33] R. Ubar, M. Jenihhin, A.S Oyeiran, J. Raik, "Implementation-Independent Functional Test Generation for RISC Microprocessors," *Tallinn University of Technology, Estonia.*

[34] C. Robach and G. Saucier, "Dynamic testing of control units," *IEEE Trans Comput., ,* Vols. C-27, pp. 617-623, July 1978.

[35] L.Chen, et al. , "A scalable software based self-test methodology for programmable processors," *in Proc. of DAC, 2003,* pp. 548-553.

[36] U. E. Odozi, ""High-Level Synthesis and Analysis of Test Data for Software Based Self-Test in Microprocessors," Tallinn University of Technology, 2016.".

[37] A.S Oyeniran, R.Ubar, High-Level Functional Test Generation for Microprocessor Modules.

[38] A.S Oyeniran, R. Ubar, M. Jenihhin, C.C. Gursoy, J. Raik, “High-Level Implementation-Free Functional SBST for RISC Processors,” *Tallinn University of Technology, Estonia.*

[39] J. Lee, J. H. Patel., “Hierarchical test generation under architectural level functional constraints.,” *IEEE Trans. Computer-Aided Design,* vol. 15, p. 1144–1151, 1997.

[40] S.Misera, H.-T.Vierhaus, L.Breitenfeld, A.Sieber., “ A Mixed Language Fault Simulation of VHDL and SystemC,” *Proc. of 9th EUROMICRO Conf on Digital System Design (DSD’06).*

[41] H.Ichihara, N.Okamoto, T.Inoue, T.Hosokawa, H.Fujiwara., “An Effective Design for Hierarchical Test Generation Based on Strong Testability,” *Proc. of ATS,* pp. 288-293, 2005.

[42] S.Mirkhani, M.Lavasani, Z.Navabi, “Hierarchical Fault Simulation Using Behavioral and Gate Level Hardware Models,” *Proc. of ATS,* pp. 374-379, 2002.

[43] R.Ubar, J.Raik, M. Jenihhin, “Diagnostic Modeling of Digital Systems with Multi-Level Decision Diagrams,” *Tallinn University of Technology, Raja 15, 12618, Tallinn, Estonia.*

[44] “COPERNICUS JEP 9624 FUTEG Functional Test Generation and Diagnosis,” 1994-97.

[45] Dey, L. Chen and S, SW-based self-test methodology for processor cores, IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, pp. 369 - 380, March 2001.

[46] E. Sanchez, M. S. Reorda, and G. Squillero, Efficient techniques for automatic verification-oriented test set optimization, International Journal of Parallel Programming, vol. 34, no. 1, pp. 93–109, 2006.

[47] C.Y.Lee, “Representation of Switching Circuits by Binary Decision Programs,” *The Bell System Technical Journal,* pp. pp.985-999, July 1959.

[48] J.Raik , R.Ubar, “Feasibility of Structurally Synthesized BDD Models for Test Generation,” *Proceedings of European Test Workshop, Spain,* pp. 145-146, 1998.

[49] R.Ubar, D.Mironov, J.Raik, A.Jutman, “Structural fault collapsing by superposition of BDDs for test generation in digital circuits,” in *11th International Symposium on Quality of Electronic Design,* San Jose, CA, USA, March 2010.

[50] R. Ubar, J. Raik, H. T. Vierhaus, “Design and Test Technology for Dependable Systems-on-chip,” *Tallinn University of Technology, Estonia.*

[51] Lee, C.Y, “Representation of Switching Circuits by Binary Decision Programs,” *The Bell System Technical Journal,* pp. 989-999, 1959.

[52] Ubar, R. , “Multi-Valued Simulation of Digital Circuits with Structurally Synthesized Binary Decision Diagrams.,” *OPA (Overseas Publishers Association) N.V Gordon and Breach Publishers, Multiple Value Logic,* vol. 4, pp. 141-157, 1998.

[53] Ubar, R. , “Vektorielle Alternative Graphen und Fehlerdiagnose fur Digitale Systeme,” *Nachrichtentechnik/Elektronik (31),* pp. 25-29, 1981.

[54] Ubar,R., “Test Generation for Digital Systems on the Vector Alternative Graph Model,” *Proceedings of the 13th Annual International Symposium on Fault Tolerant Computing* , pp. 374-377, 1983.

[55] R.Ubar, A.Jutman, J.Raik, “SSBDD Model: Advantageous Properties and Efficient Simulation Algorithms”.

[56] R. Ubar, “Overview about Low-Level and High-Level Decision Diagrams for Diagnostic Modeling of Digital Systems,” SER.: ELEC. ENERG. vol. 24, no. 3, December 2011, 303-324.

[57] Plekskacz, W.A., Kasprowicz, D., Oleszczak, T.,Kuzmiczz, W., “CMOS Standard Cells Characterization for Defect Based Testing.,” *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Ssytems,* pp. 384-392, 2001.

[58] R.Ubar, J.Raik, W.Pleslkacz, M.Blyzniuk, T.Cibakova, E.Gramtova, W.Kuzmicz, M.Lobur, “Hierarchical Defect-Oriented Fault Simulation for Digital Circuits,” *State University "Lviska Politechnika", Institute of informatics, Bratislavia, Warsaw University of Technology, Tallinn University of Technology.*

[59] E. Sanchez and M. S. Reorda, ““On the functional test of branch prediction units,",” *IEEE Trans. on VLSI Systems,* pp. vol. 23, no. 9, 2015.

[60] H.K.Lee, D.S.Ha. SOPRANO: , “An Efficent Automatic Test Pattern Generator for StuckOpen Faults in CMOS Combinational Circuits.,” *DAC,* 1990.

[61] L.Zhuo, et.al., “A Circuit Level Fault Model for Resistive Opens and Bridges.,” *Proc. VLSI Test Symp., Napa, CA,* pp. 379-384, Apr/May 2003.

[62] K.Dwarrakanath and R.Blaton, “"Universal Fault Simulation using fault tuples",” *in DAC,* 2000.

[63] K. Keller, “Hierarchical Pattern Faults For Describing Logic Circuit Failure Mechanisms,” *USA Patent 5546408,* 13 August 1994.

[64] R. Blanton and J. Hayes, “On the Propertiest of the Input Pattern Fault Model,” *ACM Transactions on Design Automation of Electronic Systems,* vol. 8, pp. no. 1 108-124, 2003.

[65] R. Ubar, “Detection of Suspected Faults in Combinational Circults by Solving Boolean Differential Equations,” *Automation and Remote Control,* vol. 40, pp. 1692-1703, 1980.

[66] L.Chen, et al, A scalable software based self-test methodology for programmable processors, in Proc. of DAC, 2003, pp. 548 - 553, 2003.

[67] P. Ward and J. Armstrong, “Behavioral Fault Simulation in VHDL,” *in 27th ACM/IEEE Design Automation Conference,* 1990.

[68] S. Ghosh and T. Cakraborty, “On Behavior Fault Modelling for Digital Designs,” *in Electronic testing: Theory and Applications, Kluwer Academic Publishers,* pp. 135-151, 1991.

[69] miniMIPS opensource, https://opencores.org/projects/minimips.

[70] Brik, Marina, “Software-based self-test generation for microprocessors with high-level decisiondiagrams/Korgtasemega otsustusdiagrammidel pohinev testprogrammide sunteesmikroprotsessoritele.(COMPUTER ENGINEERING)(Report),” 2014.

[71] ““Diagnostics of complex digital systems,” Ph.D. dissertation, Latvian Academy of Sciences, Riga, Latvia, 1986”.

[72] Adeboye Oyeniran, Tolulope Ademilua, Margus Kruus, Raimund Ubar, “Environment for Innovative University Research Training in the Field of Digital Test,” *EAEEIE,* 2020.

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

[74] A.S Oyeniran, R.Ubar, M. Jenihhin, J.Raik, High-Level Implementation-Independent Functional Software-Based Self-Test for Modules of RISC Processors.

Appendix 1 – Program Description and Manual

This section contains a step-by-step illustration on how the experiments were carried out.

1. The code folder for the experiment is available on GitHub and can be cloned using the following links https://github.com/ademilua/miniMIPS_Test_Data_Optimization.git and https://github.com/ademilua/High-level-Decision-Diagram-Generator-for-processor.git

2. The experiment is recommended to be carried out on Linux operating system.

3. Open the TG, which is the **Test Program Generator** folder. For the experiment in this thesis, Test-Program, clean.sh, input folder, compile_minimips.sh vsim_gui.tcl, logic_sim.sh, and tst.src are used. The remaining files in the folder are dependencies. The description of the used folders and files are given below:

   I. **Test program folder:** It contains the necessary files for the test program generation and the template for the experiment.

   - **Test template:** This file is used to generate codes to test non-terminal nodes.

   - **Parameter.txt:** This determines the needed instructions for test program generation. These instructions are presented based on the HLDD synthesis for the given experiment. Hence, parameter.txt file increases the flexibility of test program generation.

   - **Outputme.**py: The file serves as a temporary storage for the generated test program, so that it can be copied to the dumpports_ex.vcd file.

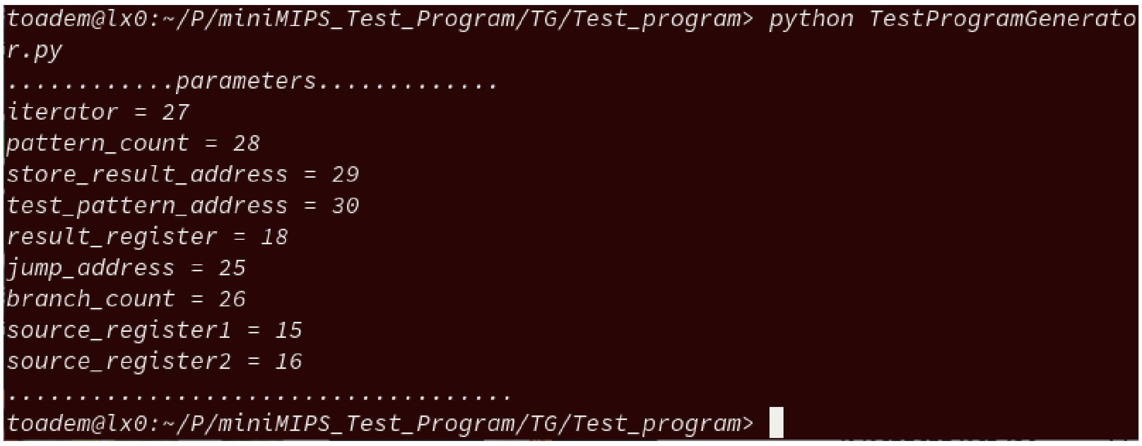
   - **load_memory.py**: The file loads the control test data (data.txt located in the input folder) into the processor's memory.

   - **TestProgramGenerator.py:** This file uses all the scripts in the test program folder to generate the test program. One can refer to it as the master file in the test program folder.

- **reset.py:** It generates code to reset all the registers available in the processor.

II. **Input folder:** This folder stores the test data such as data.txt and branch.txt to be used for the control test experiment.

4. To generate the test program, navigate to the test program folder and right click to open a linux terminal. Once the terminal is open, then type in the following command: `python TestProgramGenerator.py`. The script should generate the following response as shown in *figure 20* and a test program that is temporary store in outputme.txt:



```
toadem@lx0:~/P/miniMIPS_Test_Program/TG/Test_program> python TestProgramGenerato
r.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
..................................
toadem@lx0:~/P/miniMIPS_Test_Program/TG/Test_program> 
```

Figure 20: Test Program generator response from Linux terminal

Then copy the generated test program from ***outputme.txt*** into the ***tst.src*** file located in the Test_Program_Generator folder.

5. After the above process, then execute the command `./clean.sh` in the test program generator folder so as to remove the following: .bin, .lst, .vcd and .wlf files, that were generated by the assembler during the test program generation.

6. Navigate back and open the linux terminal from the miniMIPS folder and enable the logic simulator environment-MoldelSim. There are computers in Taltech lab with ModelSim installed, hence you can use any of the computer, then from the linux terminal, enter the `CAD` command. There should be a list of `CAD` command showing on the screen. To enable the logic simulator environment, enter command “**2**” .

7. Then the ./compile_minimips.sh is used for compiling the miniMIPS HDL from the terminal.

8. The generated test program is in assembly code, enter the command ./logic_sim.sh to convert the test program in tst.src into a binary code for the miniMIPS processor.

Then wait for the ModelSim environment to open, and a waveform is shown like in figure below. The time taken for the ModelSim to finalize the test program execution depends on the amount of test data. In some cases, you may need to wait 2-4 minutes before the execution of the test program.

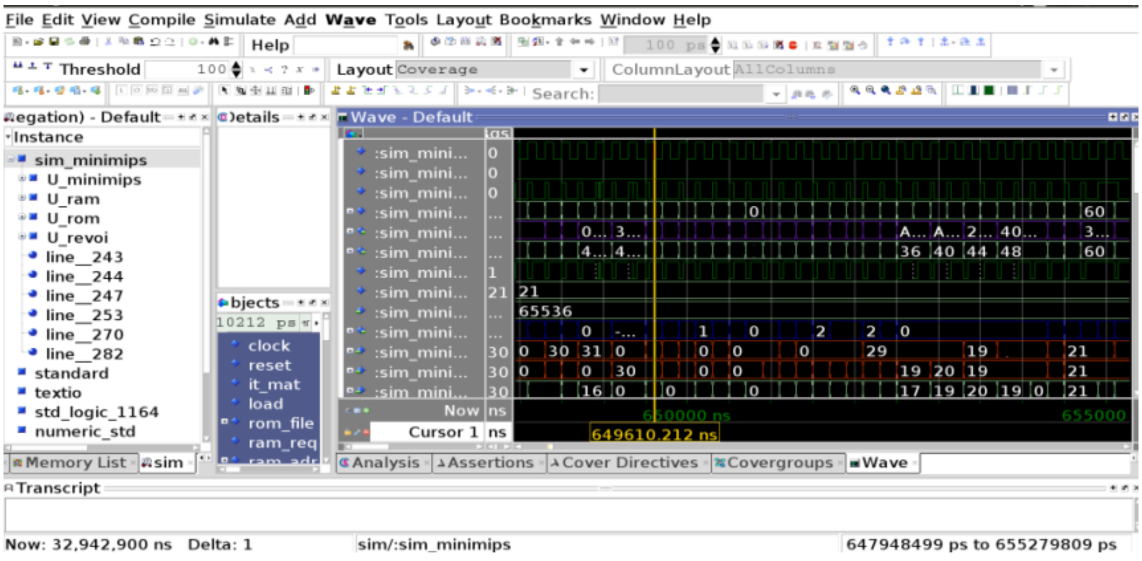


Figure 21: Generating dump file for fault coverage evaluation

This window closes automatically, once the test program is executed.

9. out.txt, rom.lst , .vcd files, and transcript were generated by the previous step, out of these files, the dummports_ex.vcd file is used for our experiment. The dummports_ex.vcd contains the input and output values of the executed instructions which is used for the fault coverage calculation.

10. Next is to copy the dummports_ex.vcd file and paste it in the Fault_coverage_calculator folder. After the dummports_ex.vcd file has been pasted, the folder should contain 1 folder and 3 files. The files are tmax.tcl, run_tst_fsim.sh and dummports_ex.vcd.

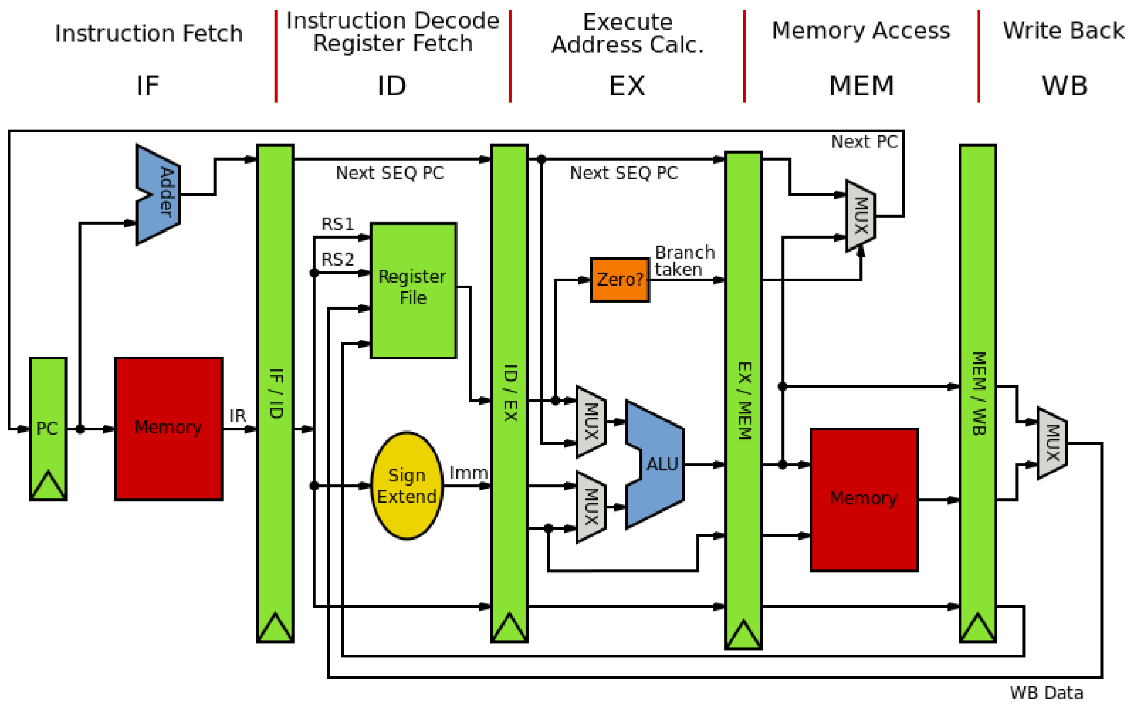
11. The tmax.tcl file must be executed for the fault coverage calculation. The file is a TetraMax script and it has a command to create all possible faults in the PPS_EX MUT including not detected faults. In order to configure the TetraMax environment, navigate to the Fault_coverage_calculator folder from the linux command terminal.

12. Once you are in the folder, enter the command CAD twice to configure the environment for TetraMax.

13. To start the fault calculation, enter the command ./run_tst_fsim.sh .

14. Finally, the fault calculation result and the details of time taken will be displayed on the command terminal or safe in a file named report.txt.

Appendix 2 – 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

hldd.py

```python
import sys
import numpy as np
import time
import random


class matrix(object):
    def __init__(self, size, pattern):
        start_time = time.time()
        self.size = size + 1  # Create the size of the array
        self.slots = np.arange(self.size * self.size).reshape(self.size, 
self.size)
        weights = []  # creates an empty list for weights
        for i in pattern:  # adds unique elements from pattern in weights
            if (i not in weights):
                weights.append(i)
        weights.sort()  # sorts the list
        count = 0  # Set a count variable that will be incremented for each 
cell that are not diagonal
        self.numbering = 0
        self.row = (np.array([0] * self.size))  # Create an array of 0 values 
with the array size
        self.col = (np.array([0] * self.size))
        row = (len(self.row))
        col = (len(self.col))
        for row_position in range(1, row):
            for col_position in range(1, col):
                self.map = np.concatenate(
                    [[row_position], [col_position]])  # Concatenate the 
index from row and col together to get indices

                if row_position == col_position:
                    self.slots[row_position][col_position] = 0
                else:
                    self.numbering += 1
                    print(f'{self.numbering} : For indices {self.map} ->')
                    self.slots[row_position][col_position] = pattern[count]
                    w = weights.index(pattern[count]) + 1
                    count += 1
                    print(f'Pattern is: 
{self.slots[row_position][col_position]}', end='|')
                    print('Weight is : {0}.'.format(w))
                    print("_" * 30)
        print('----------------Matrix Table-----------')
        print('Note Row 0 and column 0 are not used')
        print(self.slots)
        print('-----------------------------------------')
        self.lis = []  # creates list to get pattern matrix with respect to 
weights
        for m in range(1, self.size):
            l = []
            for i in range(1, self.size):
                for j in range(1, self.size):
                    if (i != j):
                        if (i == m or j == m):  # if true append the index of
```

```
that slot but not the slot pattern
                            l.append(self.slots[i][j])
            self.lis.append(l)
        print('--------------Corresponding Patterns ----------------')
        c = 1
        for i in self.lis:  # prints weights(indexes) of all patterns resides
in the slots
            print(f'{c} -> {i}')
            c += 1
        print('Note each row in the patterns list hold values from row and
column based on relationship')
        self.lis = []  # creates list to get pattern matrix with respect to
weights
        for m in range(1, self.size):
            l = []
            for i in range(1, self.size):
                for j in range(1, self.size):
                    if (i != j):
                        if (i == m or j == m):  # if true append the index of
that slot to a list
                            l.append(weights.index(self.slots[i][j]) + 1)
            self.lis.append(l)
        print('--------------Weights from the Corresponding patterns---------
-------')
        c = 1
        for i in self.lis:  # prints weights(indexes) of all patterns resides
in the slots
            print(f'{c} -> {i}')
            c += 1
        print('Note each row in the weights list hold values from row and
column based on relationship')
        self.sumlis = []  # creates list to get sum for all weights
        for i in self.lis:
            s = 0
            for k in i:
                s += k
            self.sumlis.append(s)
        print('---------------Sum of Weights for {0} functions---------------
- '.format(self.size - 1))  # str.format()
        c = 1
        for i in self.sumlis:  # calculates total sum
            print('{0} -> {1}'.format(c, i))
            c += 1
        self.totalsum = sum(self.sumlis)
        print(
            f'-----------Total sum of weights for {self.size - 1} functions =
: {self.totalsum}')  # string interpolation format
        print('Main Sum List :', self.sumlis)
        print('\n\nGraph\n')
        a = []
        for i in list(range(1, len(self.sumlis) + 1)):  # stores indexes of
sumlis in list for graph plotting
            a.append(i)
            # m = a[::-1]
            # random.shuffle(a)
        #a = [a[i] for i in unordered]

        self.graph(a)
        print("--- %s seconds ---" % (time.time() - start_time))
```

```
    def hldd(self, instructions):  # shows graph
        total = 0
        vals = []  # stores sum values of indexes
        for i in instructions:
            total += self.sumlis[i - 1]
            vals.append(self.sumlis[i - 1])
            # print(vals)
        half = total // 2  # calculates half
        s = 0  # stores total sum
        firsthalf = []
        sechalf = []  # saves rest of values
        for i in range(len(vals)):
            if s + vals[i] <= half:  # saves values that can be stored
together and have sum less or equal to half
                s += vals[i]
                firsthalf.append(instructions[i])
            else:  # saves rest of values
                sechalf.append(instructions[i])
        print(instructions, '->', firsthalf, ' and ', sechalf)
        if (len(firsthalf) >= 2):  # runs for first half
            self.hldd(firsthalf)
        if (len(sechalf) >= 2):  # runs of second half
            self.hldd(sechalf)
```

merge.py

```
import sys
# delete the first line of the output file
def delete_line(num):
  #num = 1

  with open('generated_file/out.txt', 'r') as fr:
      lines = fr.readlines()
      if num in range(len(lines)):
        del lines[num-1]

  with open('generated_file/out.txt', 'w') as fw:
      fw.writelines(lines)


# merge content of file, appending last column in each segment to the first
def mergefile(filen):
  try:
     f = open(filen,'r')
  except Exception as e:
    print "Could not copy file %s " % (filen)
    print e
    sys.exit(3)


  lines = []
  firstPass = True
  counter = 0

  for line in f:
      try:
          line = line.strip()
          columns = line.split()
          name = columns[3]

          if firstPass:
              lines.append(line)
          else:
              lines[counter] += ' ' + name
          counter += 1

      except IndexError:
          counter = 0
          firstPass = False

  f.close()
  out = open('sim_generated_file/output.txt', 'w')
  for line in lines:
      #print line[0:35]
      out.write(line[29:] + "\n")
  out.close()
```

simulate.py

```
import os
from randomGenrate import *
#import randomGenrate
from  merge import *

def main():

   if os.path.exists("sim_generated_file"):
      files = [file for file in os.listdir("sim_generated_file")]
      for file in files:
         os.remove("sim_generated_file"+"/"+file)
   else:
      os.mkdir("sim_generated_file")

   generate_pattern_t(test_l,bit_l)

   os.system("vsim -do " + "simulate.do")
   filename = "sim_generated_file/out.txt"
   mergefile(filename)

if __name__== "__main__":
  main()
```

random.py

```python
import random
import os

"""fixes the number of random number generated each time executed.
If removed random number changes each time"""
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')

if os.path.exists("sim_input"):
   files = [file for file in os.listdir("sim_input")]
   for file in files:
      os.remove("sim_input"+"/"+file)
else:
   os.mkdir("sim_input")


filename = "sim_input/input.txt"  # input data in format needed by simulation
filename2 = "sim_input/hex_input.txt"
filename3 = "sim_input/inputclone.txt" # clone of input with patterns
separated for clarity

target = open(filename, 'w')
target2 = open(filename2, 'w')
target3 = open(filename3, 'w')

test_l = raw_input("test length: ")
bit_l = raw_input("bit length: ")

"""method to generate random test patterns.
where pl is pattern length and bl is bit length"""
def generate_pattern(pl, bl):
   #print "Input A"
   target.write("Input A\n")
   for counter in range(1,int(pl)+1):
      ina = randbin2(int(bl))
      #print ina
      target.write(ina)
      target.write("\n")

   #print "Input B"
   target.write("\nInput B\n")
   for counter in range(1,int(pl)+1):
      inb = randbin2(int(bl))
      #print inb
      target.write(inb)
      target.write("\n")

   target.close()
```

```
# methods generate test patterns in binary and hexadecimal for 2 operands in
tabular form
def generate_pattern_t(pl, bl):
   #target.write("Input A"+ "             "+"Input B \n")
   #target2.write("Input A"+ "       "+"Input B \n")
   for counter in range(1,int(pl)+1):
      ina = randbin2(int(bl)) # call function that generates random binary
numbers
      inb = randbin2(int(bl)) # call function that generates random binary
numbers

      inah = hex(int(ina,2)) # convert binary to hex
      inbh = hex(int(inb,2)) # convert binary to hex

      target3.write(ina + "  "+ inb) # format readable. same as input file
but readable
      target.write(ina + ""+ inb) #format of data needed for simulation
      target2.write(inah + " "+ inbh)

      target.write("\n")
      target2.write("\n")
      target3.write("\n")

   #target.write("\n")
   #target.write("\n")  # just for simulation
   target.close()
   target2.close()
   target3.close()
   print "test pattern generated"
```