THE LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Salam Harfoush

candidate for the Master of Science
degree in Computer Science.*

Signed: [redacted]

[redacted]

Date April 3, 97

*We also certify that written approval has been obtained for any proprietary material contained therein.

[redacted]

RT
346
c.1

AN APPROACH TO REDESIGN FOR TESTABILITY AT THE RT LEVEL USING BIST TECHNIQUES

by

SALAM S. HARFOUCHE

Submitted in partial fulfillment of the requirements for the Degree of Master of Science

Thesis Advisor: Dr. Haidar M. Harmanani

Department of Computer Science
Lebanese American University—Byblos
June 1997

An Approach to Redesign for Testability at the RT Level Using BIST Techniques

ABSTRACT

by

SALAM S. HARFOUCHE

The increasing density in VLSI chips complicates the design as well as it complicates the testability problem. This thesis proposes a new approach to redesign for testability at the Register Transfer Level (RTL). Given an RTL description of a data path, the purpose of the redesign process is to improve its testability with a minimal cost by: 1) inserting additional registers, if necessary; 2) Converting already existing registers into test registers so that they can be configured as TPGRs, MISRs, or BILBOs during test mode. In order to reduce test penalty, and insure the data path structural testability, it is necessary to automate the BIST Insertion process. BIST registers are chosen so that to minimize test and time overhead, by using Randomness and Transparency metrics of the combinational logic.

To my parents

ACKNOWLEGMENTS

I would like to thank my thesis advisor Dr. Haidar Harmanani for his guidance. Also, I would like to thank Dr. W. Keirouz and Dr. G. E. Nasr for being on my thesis committee.

I would like to thank my friends at the Lebanese American University for all their support during my studies.

Finally, I would like to thank my family for their support, and to extand my gratitude to the Hariri Foundation and to the Lebanese American University who helped me financially to accomplish this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

The move to VLSI (Very Large Scale Integration) design has increased circuits complexity and led to a complexity in the design process. Thus, manual designs became very costly in terms of manufacturing and turnaround time. Synthesis tools were proposed in order to shorten the design cycle, and facilitate the design process.

On the other hand, the increase in chips density made the access to chips' internals a harder task. This in turn complicated the testing process as well. Therefore, testing had to be taken into consideration early in the design cycle in order to create testable chips with a minimal area and time overhead. Different techniques were proposed to solve the testing problem in an efficient manner, mainly Design For Testability techniques (DFT).

In discussing the future of Digital Signal Processing (DSP) CAD tools, Paulin [Paul92] indicates the need for high level synthesis tools and clarifies the real importance of verification and test tools. His observations were a result of a survey conducted at Bell Northern Research, summarizing the opinions of DSP d - signers.

1.1 The Abstraction Levels

The design process can be modeled at different levels of abstraction, from the layout level to the system level (Fig 1.1).

System level designs are described as algorithms written in a High Level Language (HLL) such as C or any Hardware Description Language (HDL) such as VHDL (behavioral) or Verilog. Register Transfer (RT) level designs are described as independent components connected together through buses and other interconnections. Gate level designs are specified as an interconnection of gates specified in a given technology. Finally, the layout level designs are specified as a network of transistors.

A variety of CAD tools have been proposed at the layout and gate levels. Layout level CAD tools are synthesis tools that synthesize the gate level descri tion and generate a corresponding layout description. Logic Level CAD tools translate an RTL description into a gate level description.

Silicon Compilers are tools that transform a design through the abstraction level to get the final layout of the chip.

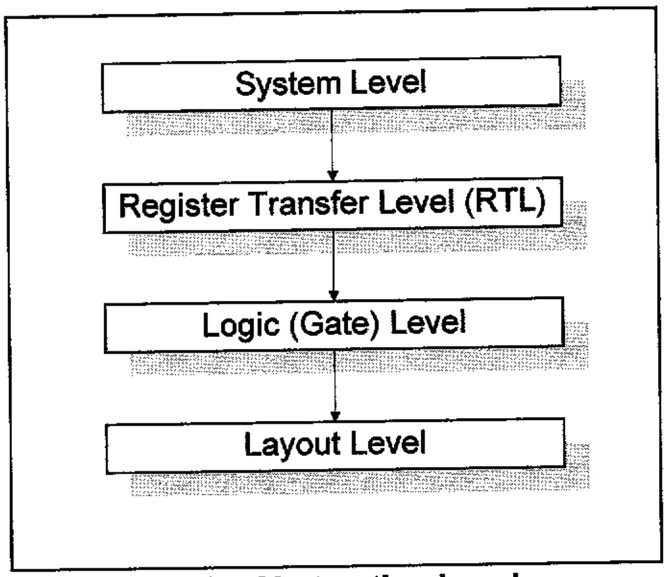


Figure 1.1: The Abstraction Levels

However, the increasing density of chips made gate and layout level CAD tools insufficient in a competitive and fast moving market. CAD tools had to address the RT and behavioral levels and thus High level synthesis tools emerged. High Level Synthesis transforms a behavior described into a data path described at the RT level.

1.2 Testing

Testing deals with revealing physical defects in circuits by applying test pa - terns to the circuit under test and verifying test responses. The VLSI testing problem has two facets: *Test Generation* and *Fault Simulation.*

Test Generation is the process of generating test patterns to test the work of the circuit under different inputs. As circuits grow in size, the test generation problem becomes more and more difficult.

The other facet of the testing problem is fault simulation that consists of determining the fault coverage for a specific set of input test patterns. The fault coverage shows the testability of the circuit through determining its capability to detect faults. At the end of the fault simulation process, faults detected by a specific pat - tern set are listed. Fault simulation is a very time consuming and expensive process.

1.3 High Level Synthesis

As was mentioned earlier, High Level Synthesis is the process of trans - forming a behavioral level design into an RTL design (a behavior into a data path and a controller) subject to a set of constraints. The controller is generated using a Finite State Machine (FSM) description. Along with the behavior, a set of constraints needs to be fulfilled. These constraints include, among others, the number of components, area, available resources, latency, and recently power consumption.

1.3.1 High Level Synthesis Steps

The High Level Synthesis steps as shown in Figure 1.2 starts with a front end tool that parses an algorithm to create an intermediate format, a Control Data Flow Graph (CDFG). The CDFG consists of nodes representing algorithmic operations such as additions and multiplications, and edges representing data transfers and indicate operation precedence.

The second step in High Level Synthesis is scheduling the CDFG into control steps to determine which operation should be executed at which time step. While preserving operator precedence, scheduling should meet the design constraints while minimizing the length of the schedule (latency) and minimizing the number of resources. A variety of scheduling techniques have been proposed; they all attempt to minimize costs such as area and delay costs.

The final step is allocation where the data path is created. During the allocation phase, the number of functional units, registers and busses is determined, and operations are mapped to FUs, variables to registers, and data transfers to and from busses are defined. Various allocation techniques have also been proposed and they all aim at minimizing area and delay costs. The output of this phase is a datapath and a controller, usually modeled in a hardware description language. Figure 1.3 gives an example illustrating the High level Synthesis task.

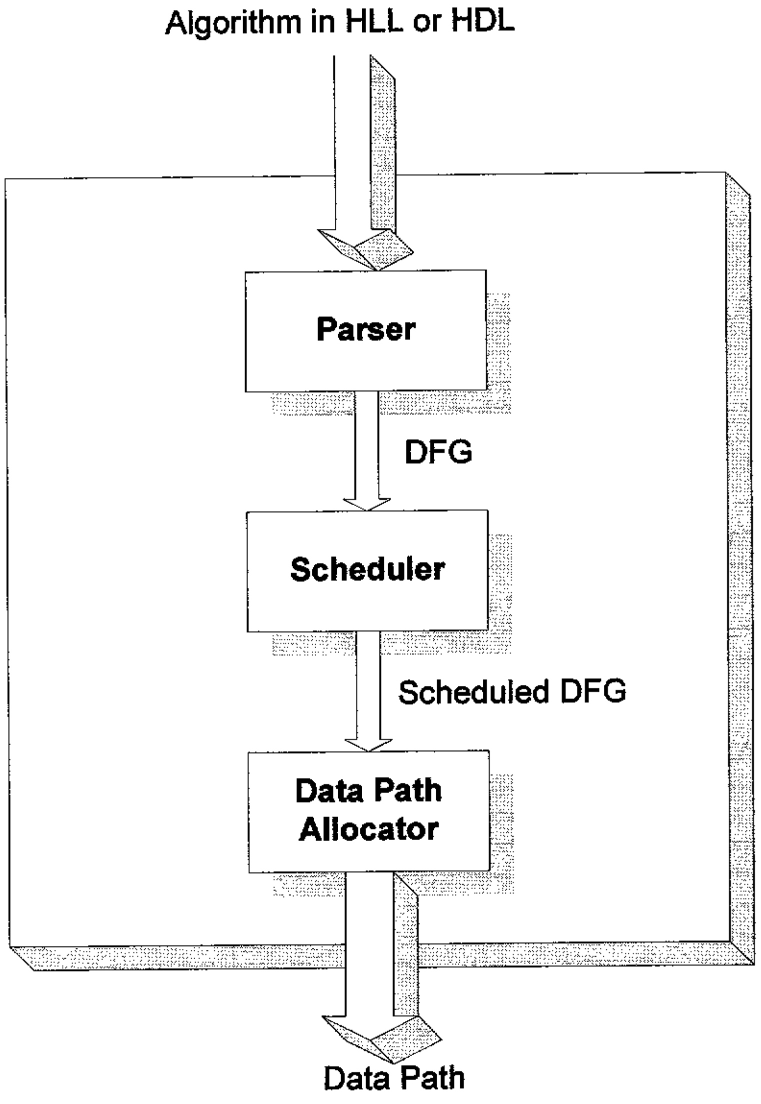


Figure 1.2: The Steps of High-level Synthesis

Initial Behavior

```
entity example is
        port( a,b:out bit;
              c,e,f: in bit;
              vdd,vss: in bit)
end example;
architecture behavior of example is
Begin
        b<= e + f;
        a<= b + c;
end;
```

Figure 1.3a: VHDL description of a simple circuit.

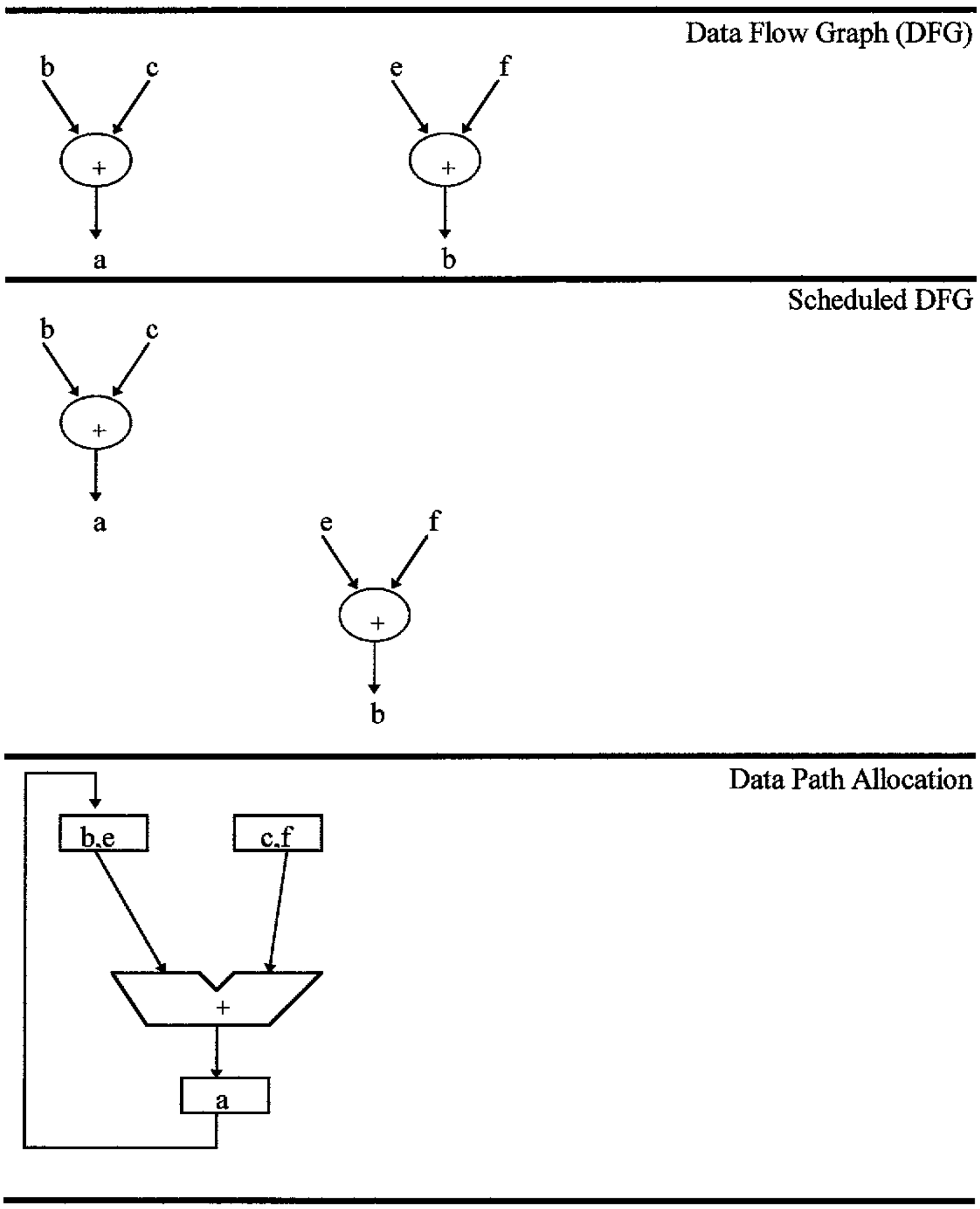


Figure 1.3b: The High Level Synthesis Task

1.4 Design For Testability

There are two main issues in Design For Testability (DFT): controllability and observability.

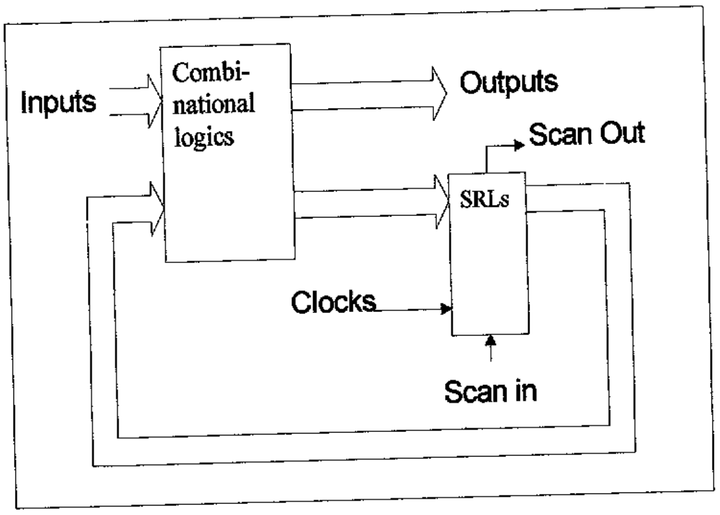
Controllability means that the input terminals or devices of the component under test can control the output, while observability means that the circuit under test can be observed at some output terminals or devices.

Design For Testability techniques present ways of approaching test generation and fault simulation in order to reduce the overall testing cost and to produce a high quality product. A lot of different approaches are followed by DFT techniques. These approaches fall into two major categories: the *adhoc* techniques solve a problem for a given design and are not generally applicable. The *structured techniques* are generally applicable and better automated since they aim at reducing the sequential complexity of a system to aid the test generation and fault simulation processes.

The two most commonly employed structured techniques are *Scan* and *BIST.*

1.4.1 Scan Technique

In the scan method, individual latches are connected to form a big chain of shift registers. The chip has two modes: a *normal* and a *test* mode controlled by an input/output pin. In normal mode, the circuit performs its intended function. However, in scan mode, the latches are chained together to form shift registers, and thus control as well as observe circuits' internals. The chip contains two extra pins: a scan-in pin is used to shift in test patterns. With the patterns shifted in, the circuit is then converted to normal mode and tested. Afterwards, the circuit is switched back to scan mode and the resulting values are shifted out through the second extra pin: the scan-out pin (Figure 1.4 [WiPa83]).



SRL: Shift Register Latch

Figure 1.4 : A Sample Scan Based Design

Designers attempted to minimize the additional cost of transforming a circuit into a scan circuit. They introduced the idea of partial scan in which only a subset of the registers in the circuit are transformed into scan registers. Different methodologies were devised to achieve an optimum selection of scan registers in order to minimize the test overhead.

1.4.2 BIST Technique

The second technique is the Built-In Self Test (BIST) technique where each module is controlled by Test Pattern Generators (TPGRs) at its inputs, and a Multi-Input Signature Register (MISR) is placed at its output to observe output sequences. Many BIST techniques are known such as BILBO (Built-In Logic Block Observer), Syndrome testing, testing by verifying walsh testing coefficients and autonomous testing. Among all these techniques the BILBO technique has been widely used. A BILBO register is a register that can be configured as a TPGR and an MISR but in different test sessions. Designers aim at minimizing the number of BIST registers in the circuit while keeping a high fault coverage at a relatively low test time.

In complex systems, there are usually some conflicts between two modules' input/output registers for which different treatments are devised. In Figure 1.5, if all the functional units are to be tested at the same time, there will be a problem at

R5 and R6. These registers will have to be a TPGR and an MISR in the same test session (*i.e.* R5 has to be an MISR for FU1 and a TPGR for FU3 and R6 has to be an MISR for FU2 and a TPGR for FU3). BILBO cannot support this operation and the solution in this case is to configure the register as Concurrent BILBOs or CBILBOs. However, CBILBOs are very costly in terms of area and delay over - head.

Scheduling the test of operations into test sessions determines the test sequence and may, in some cases, decrease the need for CBILBOs. In Fig 1.4, two test sessions is the optimum number of test sessions, where FU1 and FU2 are tested during the first test session and FU3 is tested during the second. Thus, R5 and R6 are configured as BILBOs and used as MISRs in the first session, and as TPGRs in the second, while R7 is an MISR and the rest are TPGRs. The number of test sessions affects the test time: The smaller the number of test sessions, the shorter the test time.

One advantage of the BIST technique is that some FUs are tested in parallel (ex: FU1 and FU2), thus reducing the overall testing time. Another advantage is that BIST can run at normal operation clock. However one disadvantage to the BIST technique is due to the insertion of extra hardware that requires an overhead in area.

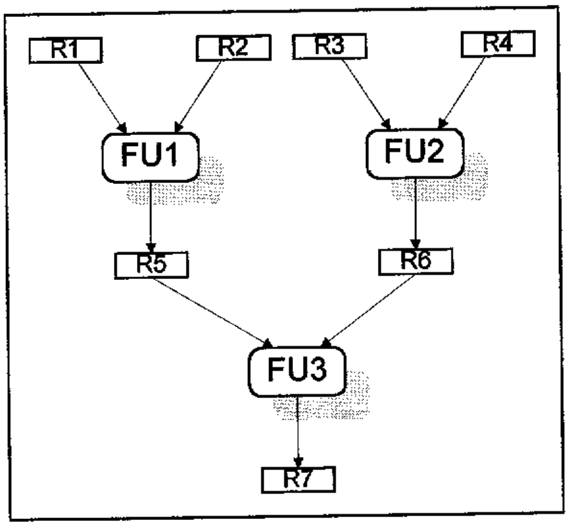


Figure 1.5: A Simple Data path

1.5 Problem Definition and Thesis Outline

The system implemented in this thesis addresses the problem of redesign for testability. It identifies parts of the design that are hard to test, and inserts registers (normal) where needed. Afterwards, a test selection process is applied to the resulting design to select TPGRs and MISRs of the different components. During the selection process, the randomness and transparency metrics are considered to minimize the number of test points selected. The final step is the scheduling process in which the test of the components is distributed to different test sessions. The scheduling process tries to minimize the test time by minimizing the number of test sessions needed. The input to the system is a VHDL description which is parsed and a simple flattened netlist is generated. The output of the system is a VHDL description of the testable data path. The output can be fed to any layout synthesis tool in order to generate the necessary chip layout.

Chapter 2 presents a literature review, while chapter 3 introduces the problem and discusses the system implementation. Chapter 4 presents results and concludes outlining future directions.

Chapter 2

Literature Review

A lot of High level synthesis tools have been proposed with some that include testability considerations. Furthermore, Several stand-alone test insertion and analysis tools were proposed . In this chapter, we will review some High Level Synthesis and Testability tools.

The five systems that will be presented in this chapter attempt to provide solutions to the testing problem.

2.1 ADEPT

ADEPT [CLPa92] is a DFT tool that has been proposed at the center for reliable and high performance computing at the university of Illinois at Urbana-Champaign. It is based on the partial scan method in order to improve the test - ability of a given circuit.

The input to this tool is a high level description of the circuits specified in VHDL. ADEPT converts the VHDL specification into an Execution Flow Graph (EFG) which is a compact representation of the data and control flow suitable for extracting testability information. A node in the EFG is either a register node, a combinational node or a fan node. The directed arcs represent the control orders which transfer data from the source node to the destination node.

A testability measure defined as Testability Sequence Range (TSR) is then derived for the nodes of the EFG. The TSR is the summation of two basic controllability and observability measures namely the Controllability Sequence Range (CSR) and the Observability Sequence Range (OSR). The EFG for each module of the circuit is extracted independently and the TSRs for this Primitive EFG (PEFG) are calculated.

Each PEFG is combined next with the PEFGs that interact with it and the TSRs are updated accordingly to give finally the Global EFG (GEFG). The TSRs are then scaled by certain weight factors to give values appropriate for calculating profit functions for each scan register. The profit functions are a measure of the improvement in testability achieved by converting a register to a scan register.

Given a cost function for converting a register to a scan register and a profit function, the scan selection problem reduces to an optimization problem. The o - jective of this optimization problem is to choose the most profitable combination of profit and cost that has a determined maximum cost.

2.2 SYNTEST

The SYNTEST system [HPCN92] is a system that integrates both synthesis and test at the system level. The input to this system is a behavioral description that describes the function to be performed and the output is a testable datapath and a controller. There are two key aspects on which SYNTEST bases its testability approach: the structural testability model and the functional testability model (Fig 2.1).

SYNTEST uses the BIST technique in order to make its datapath testable. It uses the concept of Testable Functional Blocks (TFB) and focuses on the problem of testing data paths using TFBs that do not contain self loops using the conventional BIST techniques. Later, the authors proposed an improved structural testability model based on Extended TFBs or XTFBs that have more than one output register, one of which can act as an MISR.

SYNTEST improves the functional testability by minimizing the number of test registers. The idea is that the need for a TPGR is to produce random patterns. If a register can still provide enough random patterns, it does not need to be a TPGR. On the other hand, if the output of a module can be observed at an observable point through intermediate modules, the register at the output of that module need not be an MISR.

Thus, SYNTEST uses two testability metrics, namely randomness and tran - parency to minimize test points. The randomness metric gives the probability of a register to produce random patterns (a randomness of one means a perfect pattern generator). The transparency metric gives the probability that an error at a point can be observed at an observable point. This way the test area overhead can be further reduced by being able of removing unnecessary test points. However, this may reduce fault coverage. Therefore, SYNTEST allows a tradeoff between test area overhead and fault coverage.

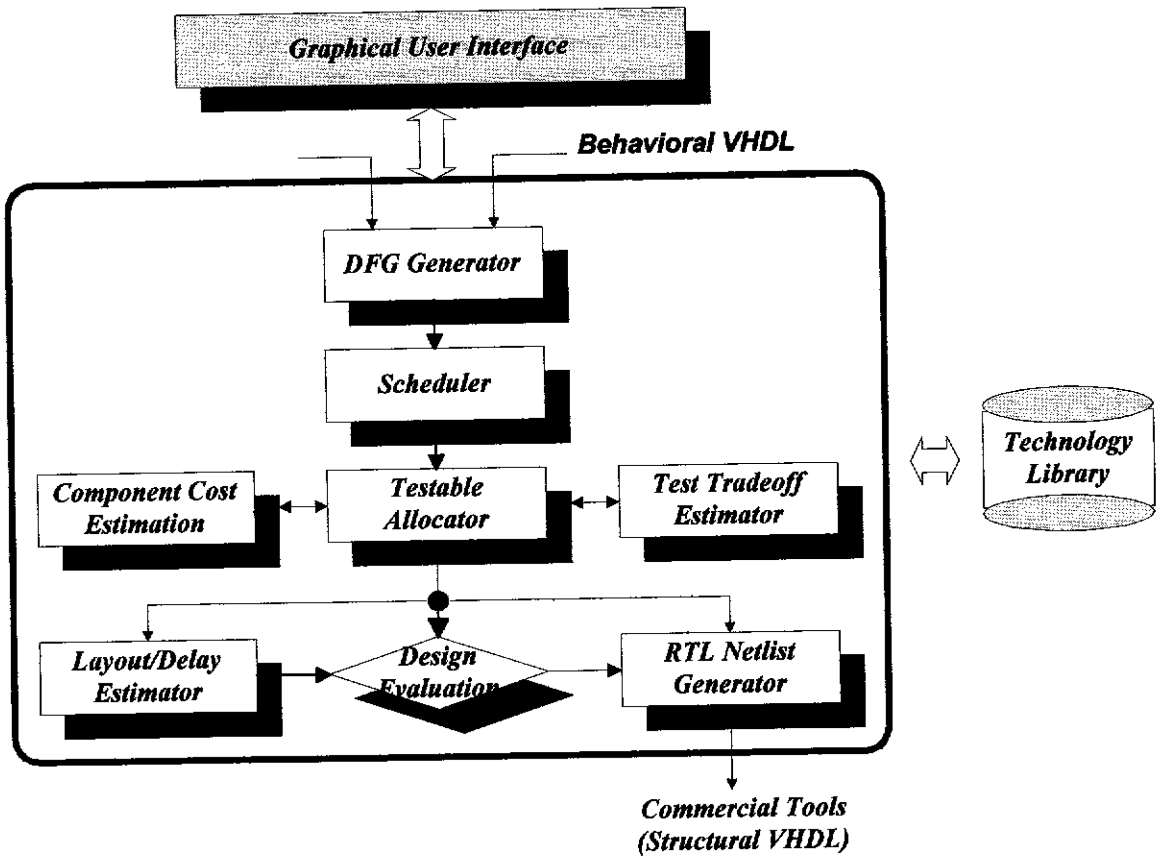


Figure 2.1: General Organization of the Syntest Environment.

2.3 Breuer's BILBO Methodology

A group lead by Professor Melvin Breuer at USC proposed a design system based on the BILBO methodology [LNBr93]. It aims at providing the designer with the option of making a tradeoff between test time and area overhead. Figure 2.2 shows an overview of this BILBO design system.

The input to the system is represented in the Cbase (an object oriented data-base developed by the USC test group). The first step is to determine I-paths that are the datapaths between input and output ports through which data can be tran - ferred unchanged.

Second, from the I-paths embeddings for each kernel can be constructed. In the system, an embedding is the structure formed of a kernel and its associated Pseudorandom Pattern Generators PRPGs and Signature Analyzer SA. Thus, in complex circuits, there may be more than one embedding for a single kernel.

During the third step, the compatibility between each two embeddings is ob-tained. Two embeddings are compatible if they do not have resource conflicts and can be executed concurrently.

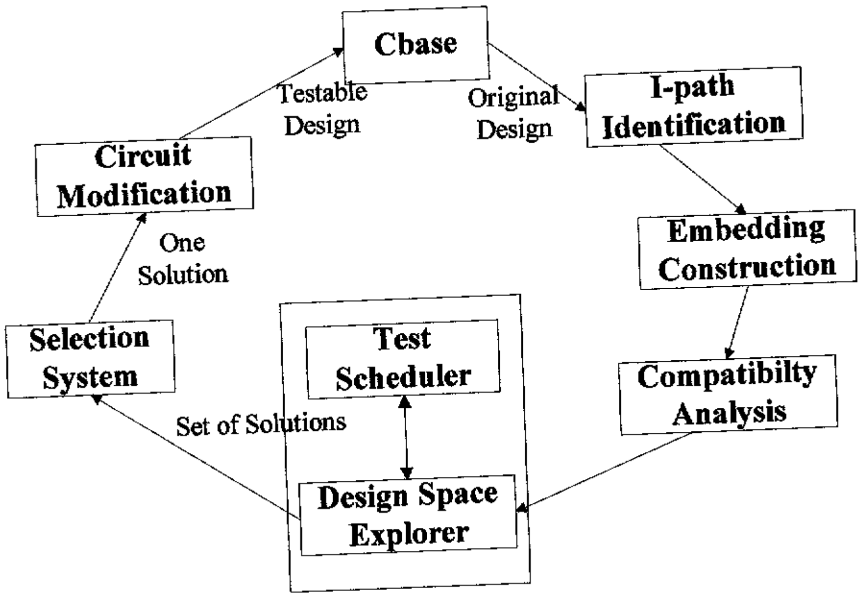


Figure 2.2 : An Overview of the BILBO Design System

Afterwards, a design space explorer is used to generate representative designs for the testable design space using a branch and bound procedure. A representative design is one of the designs that the designer may choose. During this step, a test scheduler is used to determine the minimal time each design will need to execute a complete test. Each representative design has its own test time and area overhead and the designer uses these data to choose between representative designs.

A Self Adaptive Expert Selection System (SAESS) has been implemented to help a designer in making a good selection. Once the designer makes his selection, a modification process is carried out to add the test hardware to the circuit under test.

Finally, the testable version of the original design is stored back in the data-base for future processing or layout.

2.4 A Behavioral Level Synthesis for Testability System

Chen and Saab presented a system that considers testability at a higher level of abstraction [ChSa92]. By determining Hard To Detect areas (HTD) and by inserting statements in the behavior of a circuit, this system tries to resolve the testability problem of a given circuit. This system can be integrated into a high level synthesis system as a pre-scheduling step (see figure 2.3).

The input to the system is a behavioral description of the circuit that has a special intermediate format, consisting of a symbol table and a statement list. The symbol table describes the variables defined in the circuit. On the other hand, the statement list is a list of control, assignment, logical, and user defined statements. This intermediate format is fed to a behavioral testability analyzer called BETA. It produces a control flow graph where a node corresponds to a statement in the statement list and an edge determines the flow of execution of the statements. Then, a procedure is followed to determine HTD areas and the reasons which make each point HTD. HTD areas are the areas that are not identified as Completely Controllable (CC) or Completely Observable (CO). Since the goal of the selection process is to find the minimum number of test points needed to remove all HTD points, then the problem becomes NP-hard. Thus, a heuristic is devised to over - come this problem.

Finally, after test points have been determined, Test Statement Insertion (TSI) is done. During TSI, a test statement is inserted such that it assumes the original statement at the normal mode but is able of making a completely controll - ble design at test mode.

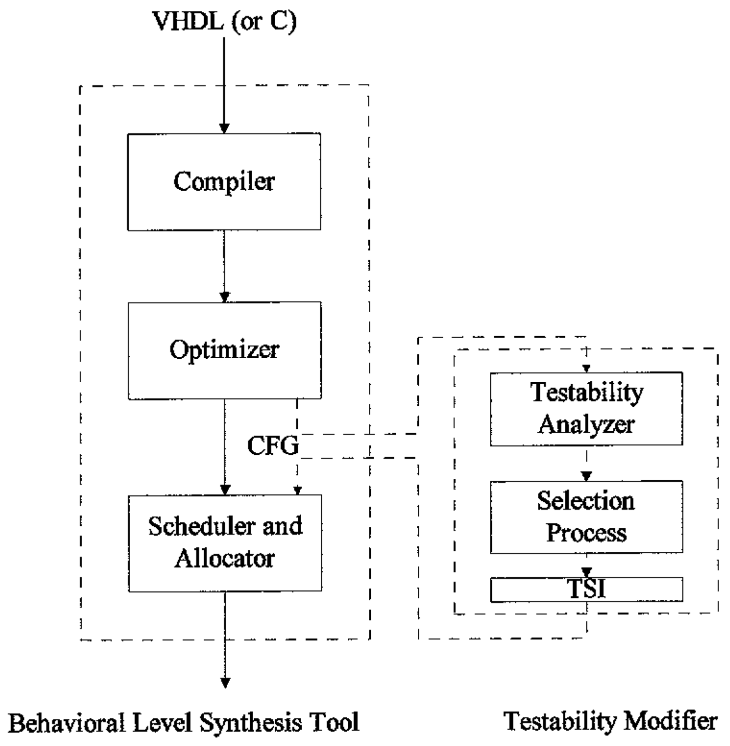


Figure 2.3: An Integrated System for Behavioral Synthesis for Testability

2.5 Automatic Insertion of BIST Hardware (AIBH)

The department of electrical engineering at Virginia Polytechnic Institute and State University proposed a system that automatically inserts BIST hardware to a circuit [KTHa88]. The input to the system is a structural description of the non-testable circuit written in VHDL. AIBH is implemented under the BILBO design methodology, and it uses an algorithmic and rule based approach to insert BILBO registers.

The tasks of AIBH are summarized in five steps:

1. Translation of VHDL description into Prolog description.
2. Allocation of PRPG and MISR for each Combinational Logic

Block (CLB) in a design.

3. Scan-path Organization.
4. Test Scheduling.
5. Distribution of register control signals.

The first step parses the VHDL code and transforms it into Prolog represe - tation which can be processed by a rule based system. The representation is in form of facts describing the individual modules in the design. The predicates repre - sent the name of the module, and the arguments represent the type of the module, its inputs, and its outputs. The inputs and outputs contain a lot of details such as the name of the input port, the name of the block connected to the input port and the name of its output port, the type of signal at the port, and the size of data path width of the port.

The second step consists of the allocation of the PRPGs and the MISRs. This is done by first searching all paths from registers to CLBs and from CLBs to registers. Then, each CLB is allocated PRPGs and MISR from the existing registers. A covering table shows, for every CLB, the registers that can be configured as PRPGs and as MISRs. The selection process is done using a set covering algorithm in order to minimize the number of PRPGs and MISRs. However, the set covering algorithm is not sufficient to solve the allocation problem. Two problems arose : some CLBs are not covered at all due to lack of registers, while others face the problem of having the same register as their PRPG and MISR at the same time. Therefore, a heuristic has been developed to solve these two problems. This heuristic was implemented using a rule based system.

The third step is the scan-path organization. During this step the registers are to be configured as a single shift register chain. Therefore, an optimal order of the registers should be found in order to minimize the wire length of a scan-path. The problem is formulated as a graph and the all-pairs shortest path algorithm is used to solve it.

The fourth step is the test scheduling. AIBH uses the scheduling scheme rep resented by Kime and Saluja. The CLBs are represented by nodes in a Test Com patibility Graph (TCG). The edges between the nodes represent the compatibility of these two CLBs (i.e. if they can be tested during the same test session). This depends on the sharing of resources such as MUXes and Data Buses.

The last step deals with control signal distribution. It tries to minimize the number of control signals needed to control the test registers. This reduction is performed using the procedure proposed by Kalinowki et al .

The AIBH system solves the testing problem using the BILBO test insertion scheme. However, it's not a pure register insertion scheme (i.e. not only BIST registers are inserted); in some cases, in order to reduce the area overhead MUXs may be inserted instead in order to use existing test registers.

Chapter 3 Redesign for Testability

Many high level synthesis systems with test considerations have emerged in the last decade. Some integrate testing at an early stage of the synthesis process, trying to reduce the test overhead by trading off design and test cost.

However, there are a lot of existing High Level Synthesis systems without test considerations. In order to test such systems, test insertion must be used. Test insertion may involve inserting new registers or just modifying existing registers to support additional test mode. Test points are important since they provide additional control to the data path internals by providing patterns and compressing signatures. Many test insertion schemes have been developed; however, they did not fully take advantage of the ALUs' functional features, namely randomness and transparency.

In this chapter, a test insertion scheme based on the BIST methodology is presented. It guarantees data path testability, by inserting appropriate test registers. Initially, inserted registers are considered as normal registers, the randomness and transparency metrics are used later to minimize the number of test points. The inserted normal registers that are not selected as test points will be removed from the data path. Figure 3.1 describes our redesign for testability approach (ReTest).

A data path, expressed in structural VHDL is fed to our system. Each module is described with a behavioral logic level model. The VHDL code is parsed and components information are collected.

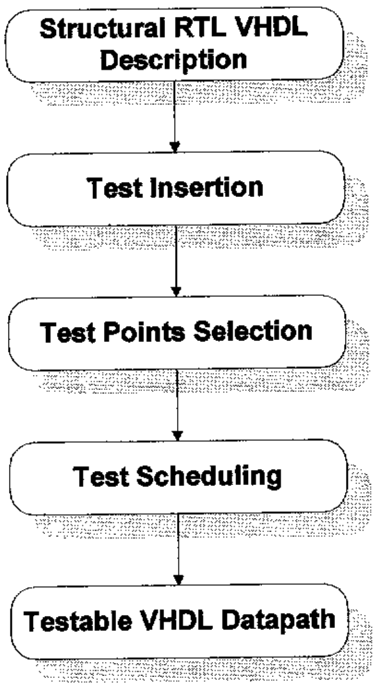


Figure 3.1 : The ReTest System

Such information include, among others, components type, bit-width, inputs and outputs. The design is next analyzed, in order to insert and select test points, taking into consideration randomness and transparency metrics. Test scheduling is accomplished next in in order to determine the necessary test ses sions. Finally, The resulting data path corresponds to the testable redesigned data path and a VHDL description of it is generated.

The next section introduces the different problems that make a test point insertion necessary, and presents the test insertion scheme used. Section two elaborates on the randomness and transparency measures and their usefulness in minimizing test point selection. Section 3 presents the selection of test registers and the heuristics used to implement the selection process and to apply randomness and transparency considerations. Section 4 presents the scheduling process.

3.1 The different testing considerations

The testing scheme presented in this Thesis is based essentially on the notion of Data path Structural Testability using Testable Functional Blocks (TFB). A functional block is considered a TFB if its combinational unit is controlled at its inputs by TPGRs and is observed at its output through an MISR (see Fig 3.2). A circuit that is based on TFBs is structurally testable. Based on the notion of structural testability, There are three general cases in which a test insertion becomes necessary.

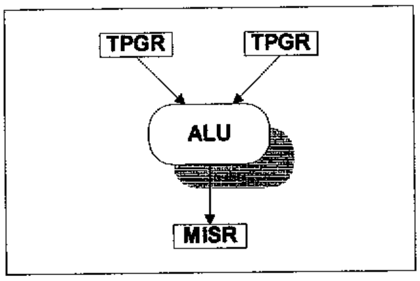


Figure 3.2: A Testable Functional Block

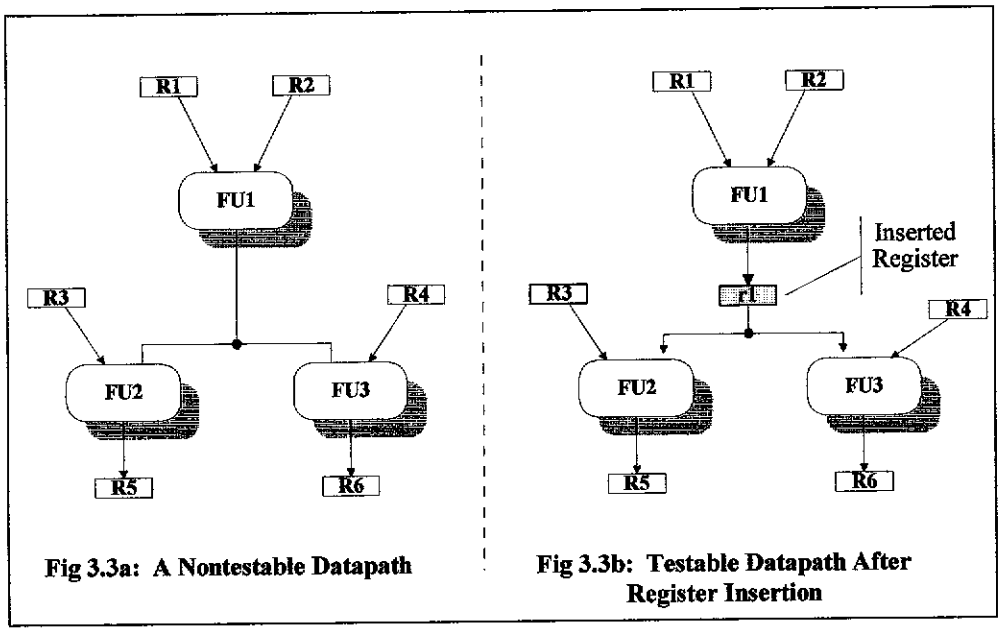


Fig 3.3a: A Nontestable Datapath

Fig 3.3b: Testable Datapath After Register Insertion

Figure 3.3 : Test Insertion Between two ALUs

In the first case, a register insertion is necessary if the ALU output is immediately used as an input to one or more ALUs (Fig. 3.3). The inserted register may work as an MISR for FU1 and as a TPGR for FU2 and FU3, before consi - ering the randomness and transparency measures. However, if the output of the ALU is also fed to a register, the ALU is then considered testable and there is no need to insert any more registers.

The second case results due to register self-adjacency problem. A register is self-adjacent if an output of this register feeds back into itself through combinational logic. Figure 3.4a depicts such a situation. This creates a testing problem, and can be solved if the self-adjacent register is configured as a CBILBO which is very costly in terms of area and delay [HaPa93].

This problem in this case is solved by inserting a register between the combinational logic and the self-adjacent register as shown in Figure 3.4b. However, not all self-adjacent registers create a testing problem as shown in Figure 3.4c. If an ALU has 2 or more output registers, one of which is a self-adjacent register, then the self-adjacent register may be configured as a TPGR to the ALU and one of the other output registers would be configured as an MISR.

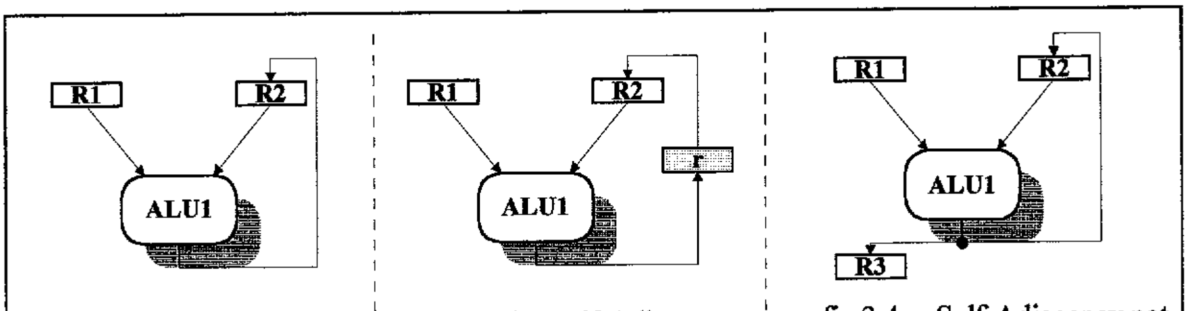


Fig 3.4a: An example of self-adjacency

Fig 3.4b: Self-Adjacency solved

fig 3.4c: Self-Adjacency not creating a testing problem

Figure 3.4: Test Insertion with Self-Adjacency

In the third case, if the input to the ALU is a MUX, then in order to test the ALU, the MUX inputs need to be checked. If the MUX inputs include a register (i.e. its inputs are not only outputs of other MUXes or ALUs), then the ALU is considered testable. For example, in Figure 3.5 we need to check the testability of

ALU1 whose inputs are the output of MUX1. In this case, the inputs to MUX1 need to be checked.

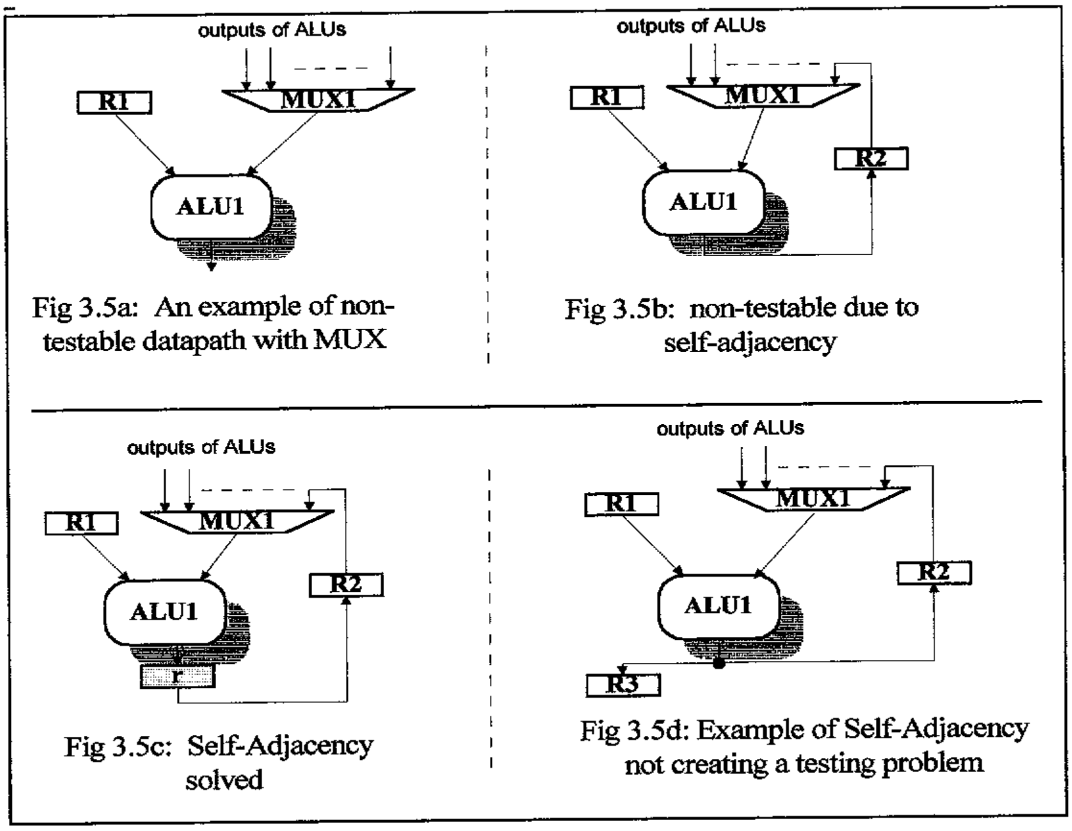


Fig 3.5a: An example of non-testable datapath with MUX

Fig 3.5b: non-testable due to self-adjacency

Fig 3.5c: Self-Adjacency solved

Fig 3.5d: Example of Self-Adjacency not creating a testing problem

Figure 3.5: Test insertion with Multiplexer at Input Port.

Figure 3.5a shows a case where ALU1 is not testable. In this case, none of the inputs to the MUX1 are registers outputs (i.e. they are all output ports of other ALUs). The problem here is solved automatically by solving the problem of the ALUs that are input to MUX1. The output ports of these ALUs are observed by register insertion. This register insertion transformed the input to MUX1 to be formed of registers, and therefore the ALU becomes testable. Figure 3.5b shows a non-testable combinational block, even though one of the inputs to MUX1 is a register. This is the self-adjacency case presented in case two; it is also resolved by inserting a register between the multiplexer and the ALU1 (see Figure 3.5c). Furthermore, if the output of the combinational logic feeds into more than one output register, one of which is a self-adjacent register, then the self-adjacency does not create a test ing problem (see Figure3.5d).

On the other hand, if the output of the ALU feeds directly into a MUX, then in order for the ALU to be testable, then the output of the MUX needs to feed into a register, otherwise a register has to be inserted. Moreover, if the output of this MUX feeds into a register that is self-adjacent, then there is also a need to insert a register. Figure 3.6 depicts several examples illustrating this case.

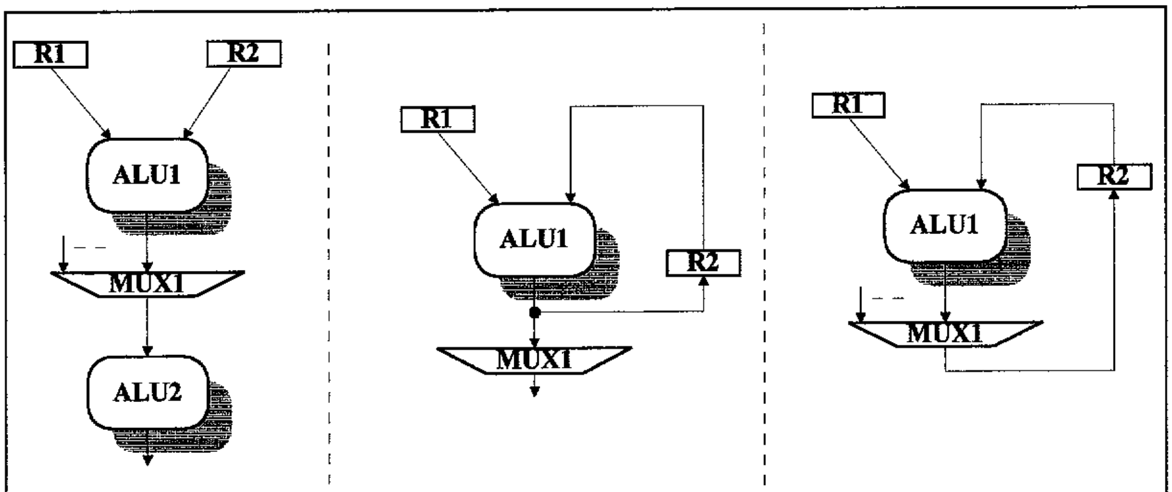


Fig 3.6a: non-testable datapath with MUX

Fig 3.6b: non-testable due to self-adjacency

Fig 3.6c: non-testable due to self-adjacency

Figure 3.6: Examples of non-testable ALUs with MUXs at the output

The test insertion scheme implemented in this thesis tries to resolve all the problems listed in the three cases described above. It does so using a heuristic that checks all these cases and inserts registers accordingly. Figure 3.7 shows the algorithm for the test insertion problem. The order of this algorithm in the worst case is $O(n^3)$ where n is the number of components in the input circuit.

3.2 Test point minimization

A certain module is considered testable when it can yield a high fault coverage when tested. Under the BIST methodology, a given circuit is tested by supplying random patterns at the modules inputs, and analyzing faults at their outputs.

```
Read Input file.
Create a list of all the ALUs and determine their arguments.
For every ALU check its arguments;
        For every argument check its type
                (whether it's a register, a multiplexer or another ALU).
        If the argument is a register:
                If this register is an output register and there is no other output register
                        If this register is also an input register to the ALU
                      (self-adjacency case)
                                Insert a register;

        If the argument is an ALU:
               If the ALU is at the input port
                        insert a register between the two ALUs.
                If the ALU is at the output port
                        If the main ALU has other outputs which are registers
                                If the registers create a self adjacency problem
                                        Insert a register
                        else
                                Insert a register

        If the argument is a MUX:
        If the MUX is at the input port of the ALU
                If the inputs of the MUX do not include registers
                        insert a register
                If the inputs to the MUX include only one register
                        If this register is an output of the ALU
                        (self-adjacency case)
                                insert a register
        If the MUX is at the output port of the ALU
                If the ALU under consideration has a register at its output port
                        If the register is also an input register
                        (self-adjacency case)
                                insert a register
                If the ALU has no registers at its output port
                        If at the output of the MUX there is a register
                                If this register is an input register of the ALU
                                        insert a register
                        If there is no registers at the output of the MUX
                                insert a register
During the insertion three lists are created, they contain respectively pairs of ALUs and
the TPGRs at the first input port, pairs of ALUs and TPGRs at the second input port, and
pairs of ALUs and MISRs.
```

Figure 3.7: The Test Insertion Algorithm.

Thus, the input registers need to be configured as TPGRs that generate random patterns, while output registers need to be configured as MISRs to analyze the outputs of the modules.

In Figure 3.8, if the output of module A is random, then the output of re - ister r can be fed into module B with random enough patterns and it needs not be a TPGR. Such a case depends on the "randomness" of module A. On the other hand, if the fault generated at the output of A can propagate through B and be ob- served at R2, then r need not be an MISR. This case depends on the "transparency" of module B.

The probability of module A providing random output patterns is calculated and defined as randomness of module A. The probability of module B propagating a fault from its input to its output register is also calculated as the transparency of module B. This work will not address the generation of the randomness and trans- parency measures, and will assume that they were derived directly from the system library To have more details about their calculation, refer to [ChPa91].

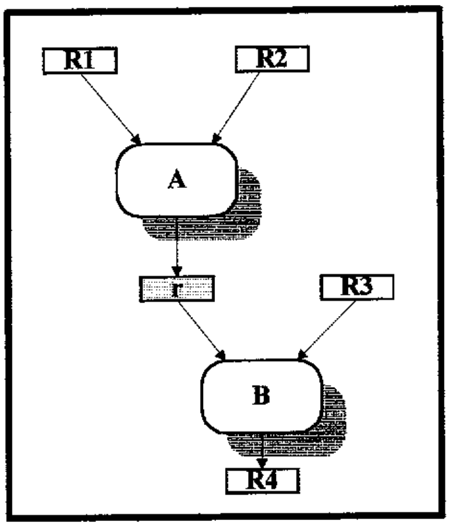


Figure 3.8 : Example Data Path

The randomness and transparency metrics are applied in the selection proc- ess. Their use will result in fewer BIST registers and thus, a decrease in the area overhead and in the hardware cost. This feature will help in the aim of this thesis namely to make a tradeoff between area overhead and testability improvement.

3.3 The selection process

The selection of test points for the data path ALUs is the next step, after the test insertion process. Several conditions have to be taken into consideration during the selection process. Each ALU must have TPGRs feeding each of its input ports and an MISR fed by the output port of the ALU. The two conditions that have to be checked always during the selection process are the following:

1 The TPGRs at the input ports of a single ALU cannot be the same due to correlation problems.

2. A TPGR cannot be used as an MISR for the same ALU in order to avoid the self-adjacency problem.

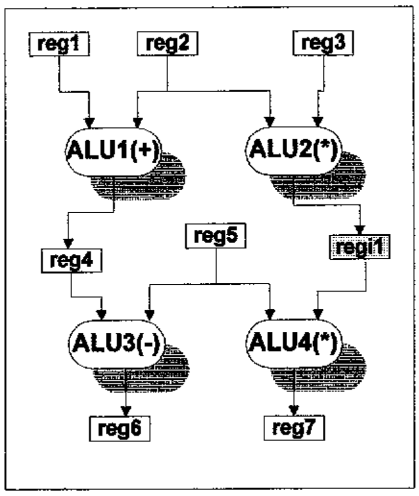


Figure 3.9: Example Design

Different ALUs may share the same TPGRs or MISR. By applying the randomness and transparency metrics, the MISR of a transparent ALU may be s - lected as an MISR of the ALUs that are connected to it at its input ports. For example in Figure 3.9, ALU3 is transparent, therefore its MISR, namely REG6 may be selected as an MISR of ALU1 connected to it through its first input port. On the other hand, if an ALU is random, then its TPGRs May be selected as TPGRs of the ALUs connected to it through its output port. In the example of Figre 3.8, ALU1 is random, therefore its TPGRs, namely REG1 and REG2 may be selected as TPGRs of ALU3 connected to ALU1 at its output port. This feature helps minimizing the number of test points selected to ensure the testability of the circuit.

The selection process is divided into two main phases. In the first phase we generate mappings of the data path ALUs with the different possible TPGRs and MISRs . In the second phase, we select one of the initial different mappings for every ALU.

The selection process begins by determining which registers cover the ports of each ALU. A list of test mappings is generated in which every ALU is assigned the direct registers at its ports as test points. For the example of figure 3.9, the initial list of test mappings is the following.

ALU name	TPGR1	TPGR2	MISR
ALU1	REG1	REG2	REG4
ALU2	REG2	REG3	REGI1
ALU3	REG4	REG5	REG6
ALU4	REG5	REGI1	REG7

Table 3.1 : initial list of test mappings for example of Figure 3.9.

Next, randomness and transparency metrics are applied. Every ALU, ac - cording to its function, may be random (randomness = 1) or not (randomness = 0). Furthermore, it may be transparent (transparency = 1) or not (transparency = 0). If an ALU is random, it can provide random enough patterns; thus, this ALU can act as a random pattern generator for all the ALUs connected to it at its output port. For example, in Figure 3.9, ALU1 can act as a random pattern generator for ALU3 and ALU2 can act as a random pattern generator for ALU4 . Under a different perspective, this means that the possible TPGRs of the ALU may be TPGRs of any combinational logic connected to this ALU through an intermediate port or register. For the example of Figure 3.9, ALU1 is random and its TPGRs are REG1 and REG2. Since ALU1 is random, it can generate enough random patterns that can be fed into ALU3, and thus, the input register REG4 of ALU3 does not necessarily need to be a TPGR. In other words the TPGR of ALU3 at this port may be the registers REG1 or REG2 the TPGRs of ALU1. Therefore, mappings of the two TPGRs with ALU3 is done. The same logic goes for ALU2 of Figure 3.9. The list of mappings becomes after the randomness check as shown in Table 3.2.

ALU name	TPGR1	TPGR2	MISR
ALU1	REG1	REG2	REG4
ALU2	REG2	REG3	REGI1
ALU3	REG4	REG5	REG6
ALU4	REG5	REGI1	REG7
ALU3	REG1	REG5	REG6
ALU3	REG2	REG5	REG6
ALU4	REG5	REG2	REG7
ALU4	REG5	REG3	REG7

Table 3.2 : Test Mappings of Example of Figure 3.9 after Randomness check

On the other hand, if an ALU is transparent, then it can pass the results generated from the ALUs at its input port to its MISR and these results can be observed at that register. In other words, the MISR of the transparent ALU may be an MISR of any combinational logic connected to the ALU through any of its two input ports. For the example of Figure 3.9, ALU3 is transparent and its MISR is REG6. Thus ALU1 connected to this ALU through one of its input ports may have REG6 as an MISR. Therefore additional mappings of ALU1 are added to the list of test mappings. These mappings are a duplicate of all the previous mappings of ALU1 with the MISR changed. In this case only one mapping is added. Table 3.3 shows the list of test mappings after the transparency check.

ALU name	TPGR1	TPGR2	MISR
ALU1	REG1	REG2	REG4
ALU2	REG2	REG3	REGI1
ALU3	REG4	REG5	REG6
ALU4	REG5	REGI1	REG7
ALU3	REG1	REG5	REG6
ALU3	REG2	REG5	REG6
ALU4	REG5	REG2	REG7
ALU4	REG5	REG3	REG7
ALU1	REG1	REG2	REG6

Table 3.3 : Test Mappings of Example (Fig 3.9) after Transparency Check

The additional mappings increase the probability that the register which does not need to be a TPGR and the one which does not need to be an MISR will not be picked during the selection process. After applying the randomness and transparency metrics the two conditions listed above are checked and violating mappings are removed.

The selection process is implemented as a set covering problem since we are trying to minimize the number of TPGRs and MISRs. However, the set covering problem is NP-complete, therefore a near optimal approach is used in which the set covering algorithm is implemented in a greedy approach. A list of all the registers is derived, and they are sorted in a descending order according to the number of times they are used in the mappings derived during the first step of the selection. For the example of Figure 3.9, the sorted list of registers is shown in Table 3.4.

Register	Weight
REG2	5
REG5	5
REG6	4
REG1	3
REG7	3
REG3	2
REG4	2
REGI1	2

Table 3.4: List of sorted Registers

Then, the mappings are selected according to this sorting in order to min - mize the number of test registers. First, the first register is selected; the mappings are checked, and the mappings for every different ALU containing this register is picked. If there are more than one mapping for the same ALU containing this register, then the weights of the registers are added; The mapping that results in the highest weight is selected. For the example of Figure 3.9, the highest weighted register is REG2, thus the following mappings are picked.

ALU1	REG1	REG2	REG4
ALU2	REG2	REG3	REGI1
ALU3	REG2	REG5	REG6
ALU4	REG5	REG2	REG7
ALU1	REG1	REG2	REG6

ALU2, ALU3 and ALU4 have each a single mapping picked, therefore this map - ping will remain. However, ALU1 has two mappings, only one should remain. However, in the first mapping [ALU1,REG1,REG2,REG4] the sum of the weights of the registers is 3+5+2=10, while the sum of the weights in the second mapping [ALU1,REG1,REG2,REG6] is 3+5+4=12 greater than the sum of the weights of the registers of the first mapping; thus, the second mapping is picked. In this ex - ample, the selection stopped here since all the ALUs have been picked, if else a - other register would have been picked and the mappings of the remaining ALUs checked. The final list of selected mappings is shown in Table 3.5, notice that REG4 remained a normal register and was not picked as a test point.

ALU name	TPGR1	TPGR2	MISR
ALU2	REG2	REG3	REGI1
ALU3	REG2	REG5	REG6
ALU4	REG5	REG2	REG7
ALU1	REG1	REG2	REG6

Table 3.5: The Final Selection List of Example of Figure 3.9

Fig 3.10, 3.11, 3.12, and 3.13 present the algorithms of the different steps of the selection problem. The running time of the total selection algorithm is in the worst case equal to $O(a^2r^2)$ where a is the number of ALUs in the input circuit and r is the number of registers in the initial circuit.

```
read the selection lists generated from the insertion algorithm
Merge these lists to get a new list that gives all possible combinations of
[ALU,TPGR1,TPGR2,MISR]
Remove all entries that may violate the selection conditions. (i.e. If TPGR1 = TPGR2 or
if TPGR1=MISR or if TPGR2 = MISR)
add to the new list two more integer elements that will be used during the scheduling
process; Initialize these two elements to zero.
For every ALU check its randomness and transparency from the library of components.
if the ALU is random apply the randomness check (Figure 3.11 )
if the ALU is transparent apply the transparency check (Figure 3.12)
Apply the greedy set covering algorithm to do the selection (Figure 3.13)
```

Figure 3.10: The Selection Algorithm

```
For every different ALU in the list created during the selection process
Pick the MISR of the ALU
For every other ALU check their TPGRs
if TPGR1 of the second ALU is equal to the MISR picked ( these two ALUs are con -
nected through this register and the TPGRs of the first are TPGRs of the second )
        Add two new entries to the selection list where the ALU is the second, the
TPGR1 is in the first entry the TPGR1 of the first ALU, and in the second the TPGR2 of
the first ALU, the TPGR2 is the TPGR2 of the second ALU, and the MISR is the MISR of
the second ALU.
if TPGR2 of the second ALU is equal to the MISR picked ( these two ALUs are
connected through this register and the TPGRs of the first are TPGRs of the second )
        Add two new entries to the selection list where the ALU is the second, the
TPGR1 is the TPGR1 of the second ALU, the TPGR2 is in the first entry the TPGR1 of
the first ALU, and in the second the TPGR2 of the first ALU and the MISR is the MISR of
the second ALU.
```

Figure 3.11: The Randomness Check

```
For every different ALU in the list created during the selection process
Pick the TPGRs of the ALU
For every other ALU check their MISR
if MISR of the second ALU is equal to one of the TPGRs picked ( these two ALUs are
connected through this register and the MISR of the first is an MISR of the second )
        Add a new entry to the selection list where the ALU is the second, the TPGR1 is
        the TPGR1 of the second ALU, the TPGR2 is the TPGR2 of the second ALU,
        and the MISR is the MISR of the first ALU.
```

Figure 3.12: The Transparency Check

```
Put all the registers in an array and sort them descending according to their occurrence
in the selection list.
Pick the first register (the register that occurs the most)
(1): In the selection list pick the first mapping that contains the register at any one of its
input ports
    for all the mappings of the same ALU that contain this register
            add the weights of the selection points.
            Pick the highest weighted mapping.
            Put it into a new list
            Set the weight and session of the mapping to be 0 (these two items are needed
            in the scheduling process.
pick the ALU of the mapping.
In the selection list remove all the other mappings that contain this ALU.
Pick the next register and go to Label.
Continue until there are no registers left in the array.
```

Figure 3.13: The Greedy Set Cover Implementation

3.4 The Scheduling Process

In chapter one, a conflict problem has been discussed concerning the use of a register as an MISR and as a TPGR at the same time. The solution presented was either the use of CBILBOs or scheduling the conflicting ALUs to different test sessions. In this section, The scheduling problem and the scheduling scheme used in this thesis are presented. The goal of this step is to determine how many test sessions are needed to test all the ALUs and to minimize the number of test sessions. To minimize this number, we should maximize the number of ALUs to be tested during the same test session. However, there are conditions which restrict two ALUs from being tested at the same time. Such ALUs have to be tested in different test cycles. These conditions are:

- If two ALUs have the same MISR, then they cannot be tested at the same time, they have to be applied to different test sessions.
- If an ALU's MISR is another ALU's TPGR then these two ALUs cannot be tested at the same time, unless the register in concern is a CBILBO which is not permitted in this system. Therefore, these two ALUs have to be scheduled into two different test sessions.

The scheduling process has been proven to be NP complete; therefore, a heuristic has been used to derive a sub-optimal solution to it.

The scheduling process is divided into two main steps that resolve respectively the first and the second condition. In the first step, the ALUs which have the same MISR are attached to different test sessions, each ALU is weighted to the number of times its MISR is used by other ALUs. For the mappings of table 3.5, ALU1 and ALU3 have the same MISR, they are assigned to different test sessions, and they are assigned a weight of two. ALU2 and ALU4 are assigned to the first test session and their weights are one. So, after this step, there are two test sessions and the ALUs are assigned to them as following:

Session 1: ALU3,ALU2, ALU4

Session 2: ALU1

The second step deals with the second condition. In this step, in the same session, every ALU's TPGRs are compared to every other ALU's MISR; if they consist of the same register, then one of the two ALUs has to be moved to another test session. The ALU that has the minimal weight is the one to be moved. For our running example, ALU1's TPGRs does not match with ALU2's and ALU4's MISRs, and so it goes for the rest of the ALUs; therefore, these three can remain in the same test session. However, assume that ALU1 had a TPGR that is the same register used as an MISR for ALU4. Then, these two ALUs may not be te - tes during the same test session, and thus, one of them should be moved to another test session. ALU1 has a weight of 2 and ALU4 has a weight of 1. In such a case, ALU4 would have been moved; It is moved to the second test session, and no more additional sessions would be needed. Notice here the use of the weight at - tribute; according to the weight the lowest weighted is moved. If we have to move the highest weighted, then an additional test session would be required because al - ready, ALU1 conflicts with the ALU scheduled in the second test session, namely ALU3. This approach to scheduling attempts at minimizing the number of test se - sions required, and thus minimizing the overall test time. Fig 3.14 shows the algo- rithm for the scheduling process. The running time of the scheduling algorithm is in the worst case equal to O(a^2) where a is the number of ALUs in the input cir - cuit.

```
Pick the first mapping X in the selection list
/*Two ALUs that have the same MISR may not be scheduled in the same session*/
(1)  Set its session to 1
For every other mapping Y
        If the MISR of Y  is the same MISR of X
        increment the session
        Set the session of Y to the new session value
adjust the weight of all the mappings whose session has been changed to the greatest
session.
For the rest of the mappings whose session has not been changed from 0 yet go to (1).

/*Two ALUs with the TPGRs of the one is an MISR of the other may not be scheduled in
the same session*/
For every mapping X check
        for every other mapping Y that follows X in the list check
                If X and Y are scheduled in the same session
                        If one of the TPGRs of X is equal to the MISR of Y or
                        If one of the TPGRs of Y is equal to the MISR of X
                                If the weight of X is less than the weight of Y
                                (it has less conflicts with other mappings)
                                     set the session of X to be the weight of X + 1.
                                ELSE
                                     set the session of Y to be the weight of Y + 1.
```

Figure 3.14: The scheduling algorithm

Chapter 4
Results

The purpose of Redesign for Testability is to improve the testability of designs through insertion of additional registers and selection of appropriate test points. The system attempts to balance between area overhead and fault coverage while preserving a minimum test time.

Since the goal of ReTest is to improve designs' testability, this chapter applies the redesign for testability on some designs to show the improvement in the fault coverage. We used a fault simulator, hope, in order to determine the fault coverage of the designs before and after redesign. Figure 4.1 shows a flowchart of the steps traversed to get fault simulation.

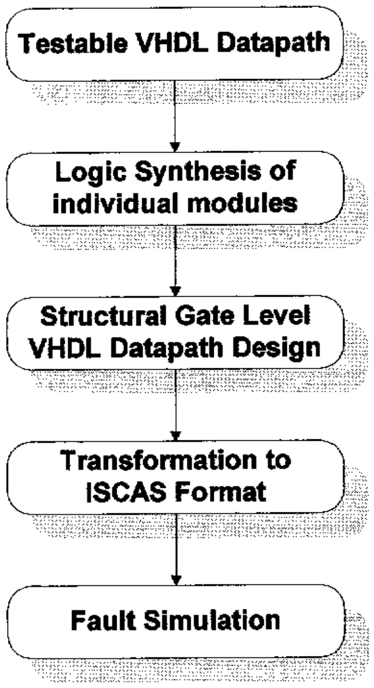

Testable VHDL Datapath
Logic Synthesis of individual modules
Structural Gate Level VHDL Datapath Design
Transformation to ISCAS Format
Fault Simulation

Figure 4.1: Experimental Procedure

The resulting testable VHDL design is then synthesized at the logic level using ALLIANCE and a gate level description of the design is generated. Note that at this stage a transistor level layout of the design may be generated by a silicon compiler. The gate level description is then transformed to ISCAS format. Finally, the ISCAS design is parsed by a fault simulation tool, hope, in order to fault grade the design. Design's fault coverage is determined before and after the redesign for testability process. The resulting fault coverages are then compared to validate our approach by showing the improvement in the design testability. The fault coverage of individual components is presented in Table 4.1.

Component Name	**Fault Coverage (%)**	**Fault Simulation Time in secs**
Adder	100	0.083
Subtractor	97.78	0.117
Multiplier	98.026	0.133
equality checker	84.906	0.083
greater	76.190	0.1
less	97.778	0.083
+ -	9.770	0.367
barrel shifter	100	0.1
2 to 1 multiplexer	100	0.05
3 to 1 multiplexer	100	0.067
4 to 1 multiplexer	88.776	0.083

Table 4.1: Fault Coverage of individual Components

This chapter presents five designs. The data path of each design is presented distinguishing between the inserted registers and other registers and showing TPGRs, MISRs and BILBOs. Then, the fault coverage of each design is determined before and after insertion; and these coverages are mapped together to show the improvement in test ability.

4.1 Example 1

The first example is the running example of figure 3.9 of chapter 3. Figure 4.2 shows its datapath along with the selected test points and their types. Notice the benefits of applying the randomness and transparency metrics. REG4 remained a normal register, while otherwise it would have been selected as a BILBO. REGI1, as well was selected only as an MISR, while otherwise, it would have been selected as a BILBO too. Therefore, randomness and transparency application decreased the area overhead required by testing. It also decreased the number of test sessions; ALU2 and ALU4 are scheduled in the same test session while othe - wise they would be scheduled in different test sessions.

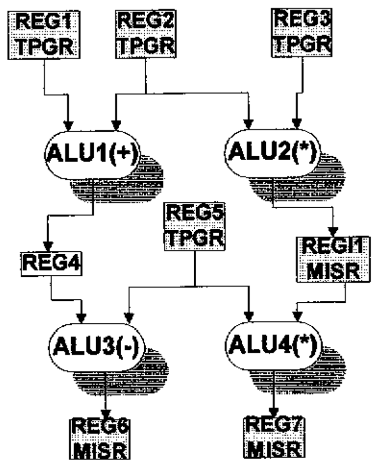


Figure 4.2: Example Design

The schedule of this example consists of two test sessions and is shown in Table 4.2:

Session	ALU name	TPGR1	TPGR2	MISR
1	ALU3	REG5	REG2	REG6
1	ALU4	REG2	REG5	REG7
1	ALU2	REG3	REG2	REGI1
2	ALU1	REG2	REG1	REG6

Table 4.2: Schedule Table of the Running Example.

The fault coverage of the redesigned circuit is 99.125%, while the fault coverage of the non-testable circuit is 98.729% with an improvement of 0.396%. The improvement of fault simulation time is from 1.5 secs to 1.383 secs. Figure 4.3 shows the fault coverage improvement chart.

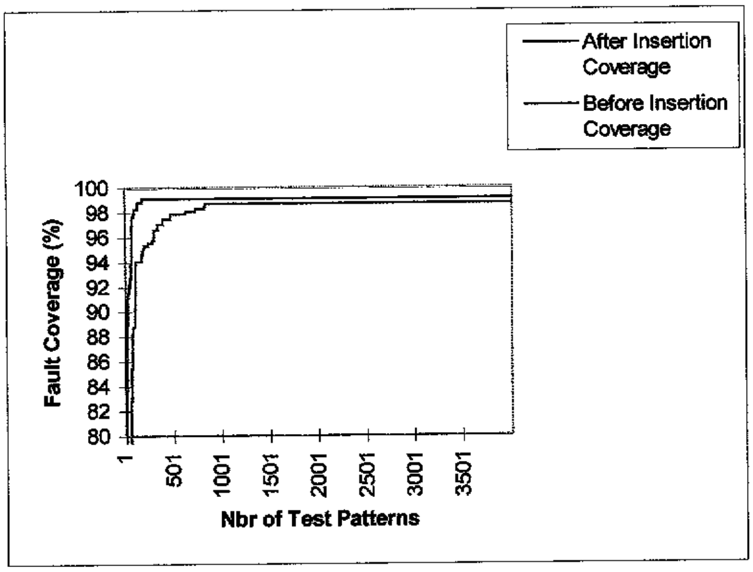


Figure 4.3: Fault Coverage of Example one

4.2 Example 2

The second example is ARYL and LYRA's[ ]. Figure 4.4 shows the data path of the example along with the selected test registers.

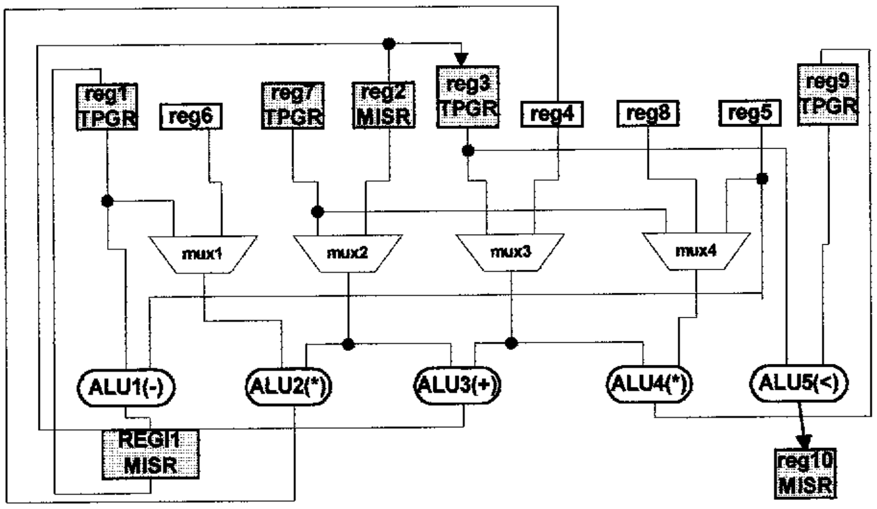


Figure 4.4 :Data path of LYRA and ARYL's example

Notice that only one register was inserted in order to improve the testability of ALU1(-). The inserted register breaks a self-adjacency and improves the testability of the overall datapath . The components are scheduled to three test sessions. Table 4.3 shows the schedule of LYRA and ARYL's datapath.

Session	ALU name	TPGR1	TPGR2	MISR
1	ALU2	REG7	REG1	REG2
1	ALU5	REG9	REG7	REG10
2	ALU1	REG7	REG1	REGI1
2	ALU3	REG3	REG7	REG2
3	ALU4	REG7	REG3	REGI1

Table 4.3: Schedule Table of ARYL and LYRA's Example.

The schedule of the LYRA and ARYL example is not an optimum one. However, it is the nearest to optimal. Remember that the scheduling problem is NP-Complete.

The fault coverage of the redesigned circuit is 98.711%, versus 87.792% fault coverage for the non-testable circuit with an improvement of 10.919%. The fault simulation time improved by 3.066 secs from 9.733 secs to 6.667 secs. Figure 4.5 shows the fault coverage improvement.

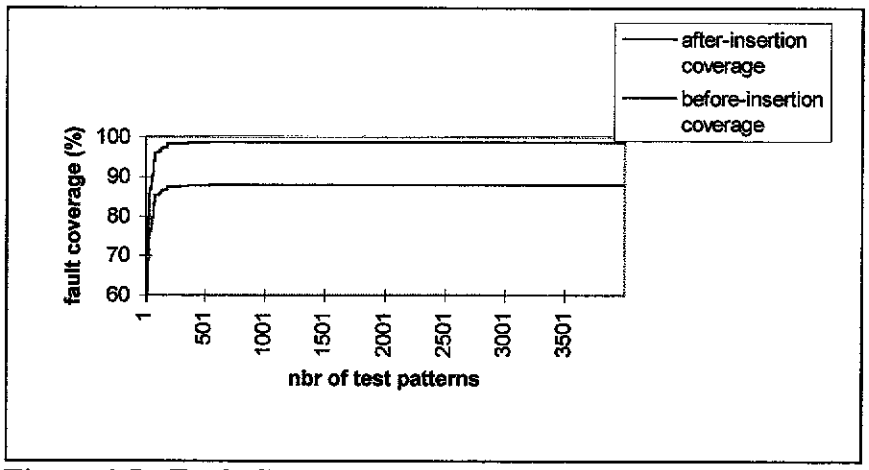


Figure 4.5: Fault Coverage of the ARYL and LYRA's Datapath

4.3 Example 3

The third example is HAL's differential equation[ ]. Figure 4.6 shows the datapath of the design along with the selected test points.

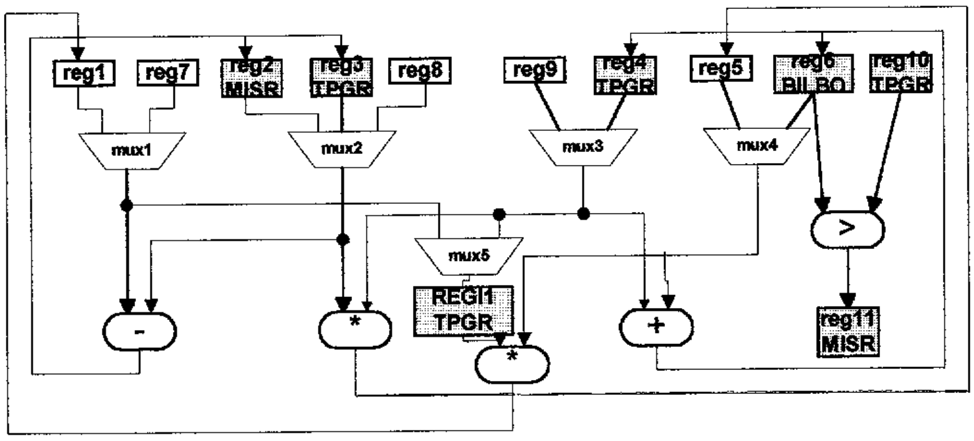


Figure 4.6: The HAL Data Path

One register was also inserted. The schedule of the HAL example is shown in Table 4.4.

Session	ALU name	TPGR1	TPGR2	MISR
1	ALU3	REG6	REGI1	REG2
1	ALU5	REG10	REG6	REG11
2	ALU1	REG3	REG6	REG2
3	ALU4	REG3	REG4	REG6
3	ALU2	REG3	REG4	REG6

Table 4.4: Schedule Table of HAL's Example.

The improvement in fault coverage was of 3.246%, from 92.164% to 96.410%. The improvement in fault simulation time is 0.6 secs, from 7.983 secs to 7.383 secs. Figure 4.7 shows the fault coverage inprovement chart.

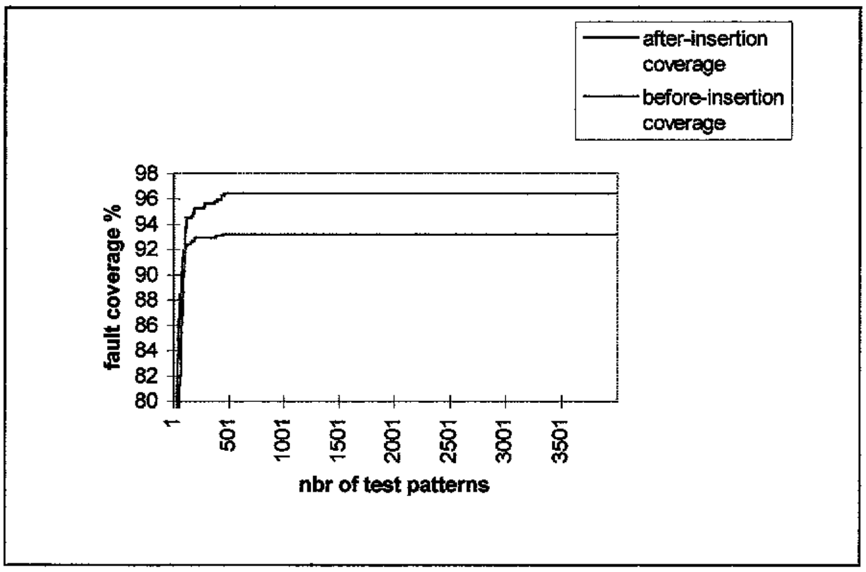


Figure 4.7: Fault Coverage of the HAL Datapath

4.4 Example 4

The fourth example is also a differential equation data path generated by another high level synthesis tool [ChPa91]. Figure 4.8 shows the datapath of the design and the test points sdected.

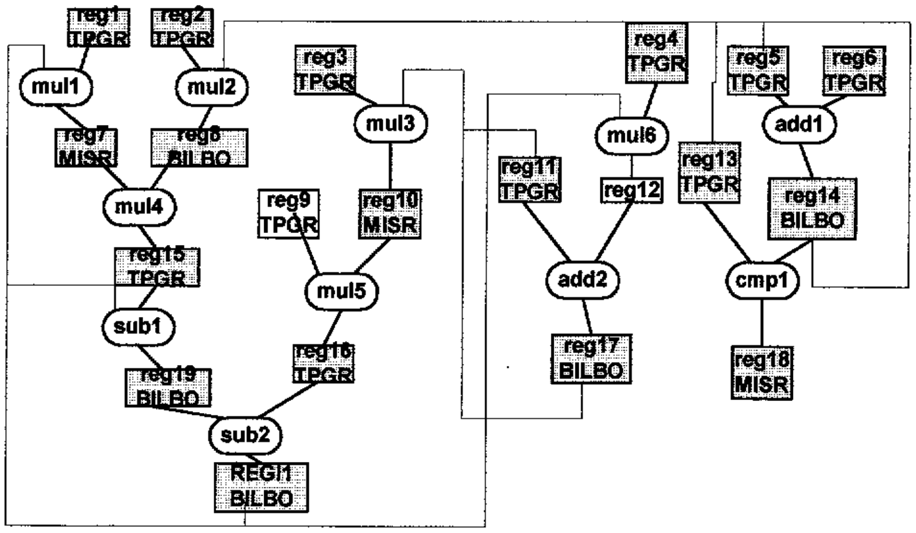


Figure 4.8: Differential Equation Data Path

One register insertion is needed. However, it substantially improves the testability of the design since it improved the testability of three components. The improvement in fault coverage is 58.539 % from 39.571% to 98.110%. The improvement in fault simulation time is 35.283 secs from 39.700 secs to 4.417 secs Table 4.5 shows the diff. Eq. Schedule and Figure 4.9 shows the fault cover age.

Session	ALU name	TPGR1	TPGR2	MISR
1	SUB2	REG16	REG19	REGI1
1	MUL2	REG14	REG2	REG8
1	CMP	REG14	REG13	REG18
1	MUL3	REG17	REG3	REG10
2	MUL1	REGI1	REG1	REG7
2	MUL4	REG8	REGI1	REG19
2	ADD2	REG11	REGI1	REG17
2	ADD1	REG6	REG5	REG14
3	MUL6	REGI1	REG4	REG17
3	SUB1	REGI1	REG15	REG19
4	MUL5	REG17	REG9	REGI1

Table 4.5: Schedule Table of Diff. Eq. Example.

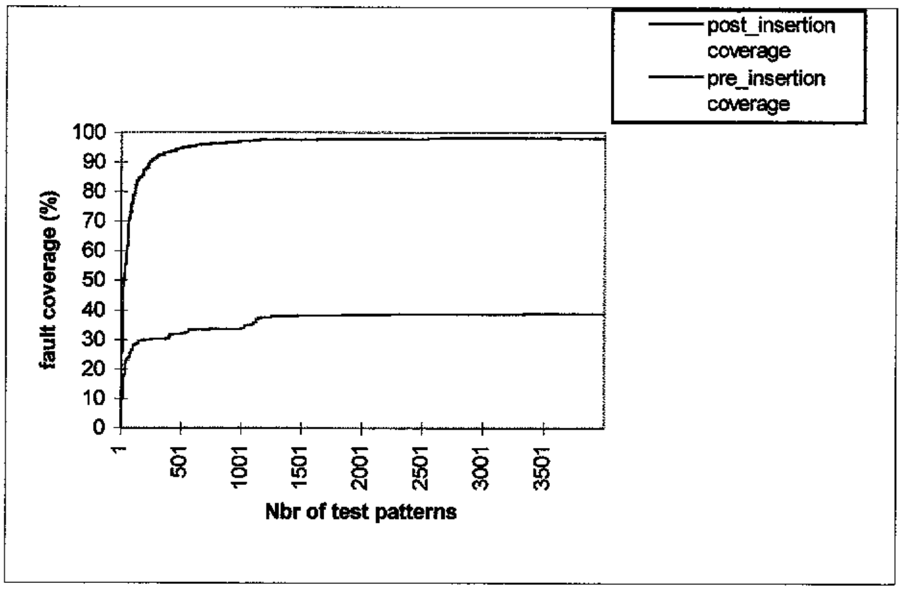


Figure 4.9: Fault Coverage of the Differential Equation Datapath

4.5 Example 5

The fifth example is the TMS32010 [KTHa88]. The datapath of this exam ple is shown in Figure 4.10. The insertion of two registers was needed. The first register (REGI1) broke the self-adjacency of REG5 and increased the testability of ALU2. The second inserted register REGI2 works as an MISR for the third ALU the Barrel shifter. The selected registers are also shown in this datapath.

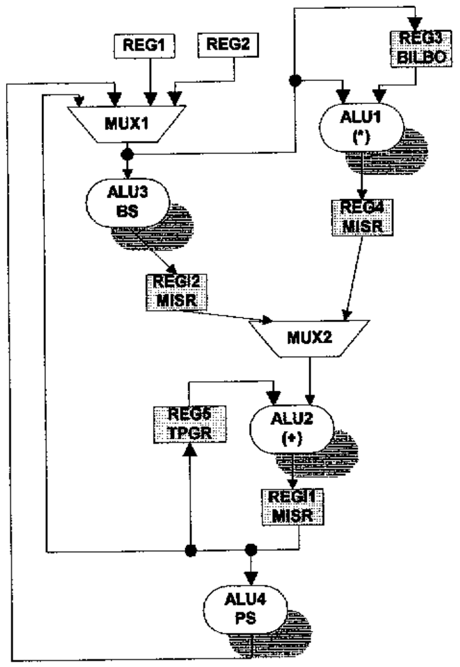


Figure 4.10: The datapath of the TMS32010 example.

The ALUs are scheduled in two test sessions. Table 4.6 shows the schedule generated.

Session	ALU name	TPGR1	TPGR2	MISR
1	ALU1	REG5	REG5	REG4
1	ALU2	REG3	REG3	REGI1
1	ALU3	REG5	-	REGI2
2	ALU4	REG5	-	REG3

Table 4.6: Schedule Table of the TMS32010 Example.

A noticeable increase in the fault coverage is deduced. The fault coverage increased by 71.240% from 5.970% to 77.210%. The fault simulation time im proved by 18.05 secs from 22.817 secs to 4.767 secs. Figure 4.11 shows the im provement in fault cover age.

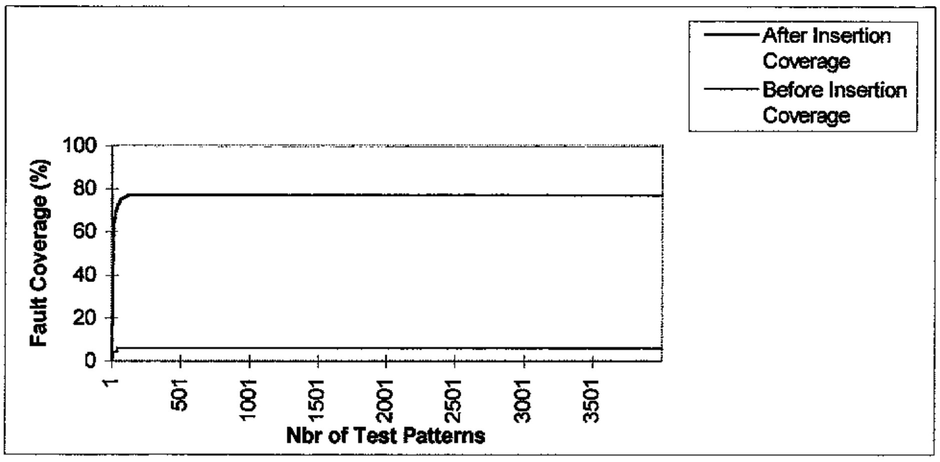


Figure 4.11: Fault Coverage of the TMS32010

4.6 Conclusion

The problem of non-testable designs is solved in this thesis by inserting additional registers. This thesis also introduces sub-optimal solutions to the selection and scheduling problems. The system improves the fault coverage of the designs. The results presented in this chapter show an improvement in the fault coverage of circuits after redesign. The area overhead is made minimal by applying randomness and transparency metrics and minimizing the number of test points. On the other hand, as shown in the graphs of fault coverages, redesigned circuits require fewer test patterns and thus demand smaller test time. The usefulness of such redesign for testability system is to improve the performance of designs. The area overhead is not very important due to the decreasing cost of silicon. The tradeoff achieved by ReTest between fault coverage and area overhead can lead to future improvements on the system in which user interaction plays a major role. In ReTest, all the steps are done by the system and the best estimations it could find on the number of additional registers needed and on the test points selected are applied to the design. As future work, the system can be further improved by allowing the designer to attempt his/her own estimations. The data path can be presented in a graphical format and the user may be allowed to work on it interactively. More specifically, after the system has inserted its additional registers, the designer may want to insert additional registers. The designer may also want to select different test points. When the user makes a change on the design, everything is re-processed. The resulting fault coverage, area overhead and test time are re-computed and presented to the designer. Then, the designer (or the system) can compare different cases and plot them to show the advantages of each design.

Bibliography

[AvMc94] L. J. Avra, and E. J. Mcluskey, "High Level Synthesis of Testable Designs: An Overview of University Systems," *International Test Conference*, 1994.

[ChPa91] S. Chiu, and C. A. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis," *28th ACM/IEEE Design Automation Conference*, 1991,pp. 271-277.

[ChSa92] C-H. Chen, and D.G. Saab, "Structural Behavioral Synthesis for Testability Techniques ," *Technical Report, University of Illinois*, June 1992.

[ChSa93] C. H. Chen, and D. G. Saab, "A Novel Behavioral Testability Measure," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol. 12, No. 12, 1993, pp.1960-1970.

[CLPa92] V. Chickermane, J. Lee, and J.H. Patel, "Design for Testability Using Architectural Descriptions," *International Test Conference*, 1992, pp.752 - 761.

[GiCa93] E. Girczyc, and S. Carlson, "Increasing Design Quality and Engineering Productivity through Design Reuse," *30'th ACM/IEEE Design automation conference*, 1993, pp. 48 - 53.

[HaPa93] H. Harmanani, and C. Papachristou, "An Improved Method for RTL Synthesis with Testability Tradeoffs," *Proceedings of the International Conference on Computer-aided Design*, November 1993.

[HPCN92] H. Harmanani, C. Papachristou, S. Chiu, and M. Nourani, "Syntest: An Environment for System Level Design for Test," *Proceedings of the European Design Automation Conference*, Sep 1992.

[JPPe89] W-B Jone, C. A. Papachristou, and M. Pereira, "A Scheme for Overlaying Concurrent Testing of VLSI Circuits," *26th ACM/IEEE Design Automation Conference*, 1989, pp. 531-536.

[KTHa88] K. Kim, J. G. Tront, and D. S. Ha, "Automatic Insertion of BIST Hardware Using VHDL," *25th ACM/IEEE Design Automation Conference*, 1988, pp. 9-15.

[LNBr93] S. Lin, C.A. Njinda, and M.A. Breuer, "Generating a Family of Testable Designs Using the BILBO Methodology," *Journal of Electronic Testing: Theory and applications*, V. 4, pp. 71 - 89, 1993.

[MaCa90] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, and J. Huisken, "Architecture-driven Synthesis Techniques for VLSI Implementation of DSP Algorithms," *Proceedings of the IEEE*, Vol. 78, No. 2, February 1990, pp. 319 - 334.

[NBDa92] V. Nagasamy, N. Berry, and C. Dangelo, "Specification, Planning and Synthesis in a VHDL Design Environment," *IEEE Design and Test of Computers*, 1992, pp. 58-68.

[PaCa95] C. Papachristou, and J. Carletta, "Test Synthesis in the Behavioral Domain," International Test Conference, 1995, pp. 693-702.

[PaKn89] P. G. Paulin, J. P. Knight, "Force-Directed Scheduling for the Behavioral synthesis of ASIC's," *Transactions on Computer Aided Design*, Vol. 8, No. 6, June 1989.

[PauK89] P. G. Paulin, J. P. Knight, "Algorithms for High Level Synthesis," *IEEE Design and Test of Computers*, 1989, pp. 18 - 31.

[Paul92] P. Paulin, "DSP Design Tool Requirements for the Nineties: An Industrial Perspective," *Sixth Intl. Workshop on High-Level Synthesis*, Laguna Niguel, CA, Nov. 1992.

[PHMo94] C. Papachristou, H. Harmanani, and M. Nourani, "An Approach for Redesigning in Data Path Synthesis," Technical report, *Department of Computer Engineering*, Case Western University, 1994.

[RaGa92] L. Ramachandran, and D. Gajski, "Architectural Tradeoffs in Synthesis of Pipelined Controls," Technical Report, *University of California/Irvine*, November 1992.

[RAKa93] S. R. Rao, J. R. Armstrong, And S. Kapoor, "On Hierarchical Test Generation for VHDL Behavioral Models," Technical Report , *Virginia Polytechnic Institute*, November 1993.

[Rude96] R. Rudell, "Tutorial: Design of a Logic Synthesis System," *33rd Design Automation Conference*, 1996.

[SeAg85] S. C. Seth, and V. D. Agrawal, "Cutting Chip-testing Costs," *IEEE Spectrum*, April 1985, pp. 38-45.

[SCMa93] J. Steensma, F. Catthoor, and H. De Man, "Partial Scna at the Register Transfer Level," *International Test Conference*, 1993, pp. 488 - 497.

[Stro88] C. E. Stroud, "Automated BIST for Sequential Logic Synthesis," *IEEE Design & Test of Computers,* 1988, pp. 22 - 32.

[ViAb92] P. Vishakantaiah, and J. Abraham, "ATKET for High Level Testability Analysis," Technical Report, *University of Texas/Austin*, July 1992.

[VisA92] P.Vishakantaiah, and J. Abraham, "High Level Testability Analysis Using VHDL Descriptions," Technical Report, *University of Texas/Austin*, December 1992.

[VTAA93] P. Vishakantaiah, T. Thomas, J. A. Abraham, and M. S. Abadir, "AMBIANT: Automatic Generation of Behavioral Modifications for Testability," Technical Report, *University of Texas/Austin*, May 1993.

[WePa92] J-P Weng, and A. C. Parker, "CSG: Control Path Synthesis in the Adam System," Technical Report, *University of Southern California*, April 1992.

[WiDe96] K. D. Wagner, and S. Dey, "High Level Synthesis for Testability: A Survey and Perspective," *33rd Design Automation Conference*, 1996.

[WiPa83] T.W. Williams, and K.P. Parker, "Design for Testability - Survey," *Proceedings of the IEEE*, Volume 71, Number 1, pp. 98 - 112, January 1983.