AUTOMATIC TEST PATTERN GENERATION FOR DIGITAL IC'S AND IMPLEMENTATION OF A PROGRAMMABLE STIMULUS GENERATOR

by

Mustafa L. Civelek

B.S. in Electronics Dept., Turkish Naval Academy, 1989

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Electrical and Electronics Engineering

Boğaziçi University

1997

ACKNOWLEDGMENTS

I am owing a tremendous debt of gratitude to many individuals without whom this work would never have come to fruition.

Special thanks to Assoc. Prof. Dr. Günhan Dündar, for his contributions to this work from the beginning till the end.

I am grateful to Prof. Dr. Ömer Cerid who is one of my thesis advisors, for his invaluable support to the hardware implementation part of this work.

Thanks to Assoc. Prof. Dr. Sina Balkır who is one of my thesis advisors, for his valuable support.

Sincere thanks to Assoc. Prof. Dr. Levent Akın from Computer Engineering Department, for his careful corrections and close interest to the subject.

I am grateful to Prof. Dr. Yorgo Istefanopulos who is head of our department, for his kind support during my education.

My most sincere thanks to Alper Altınordu, Hakan Binici, İsmet Bayraktaroğlu, Selçuk Öğrenci and Bilgin Kerim, who spend a lot of their time for my problems.

Thanks to my friends, Esra Barlas, Gürcan Elbek, Şule Özev, Emel Yücetürk and to the people of BETA Laboratory, for their close friendship.

Thanks to the telecom. company NETAŞ and their valuable administrators of R&D department Mr. Önder Bicioğlu and Mr. Yılmaz Bingöl, for their support to this work.

Thanks to many people in all around the world who are studying on ATPG business namely, Dr. Flemming Stassen, Prof. Dr. Dong S. Ha, Prof. Dr. Melvin A. Breuer, Prof. Dr. M. Abramovici, Dr. Elizabeth M. Rudnick, Dr. Janak H. PATEL, Dr. Haluk Konuk, Prof. Dr. Hideo Fujiwara and Dr. Andrej Zemva, for their valuable contributions to this work, by giving invaluable answers to my questions.

Special thanks to the Turkish Navy Headquarters, for their invaluable support during my MS. education.

Thanks to my dear friend Yasemin Uçar, for her endurance and patience during my long working hours.

Last but not least, I would like to express my deepest gratings to my family who always encouraged and supported me throughout my life.

ABSTRACT

In this thesis work, two different topics are studied in order to perform a complete digital integrated circuit (IC) testing system, except an output analyzer. These topics can be separated as development of an automatic test pattern generation (ATPG) software for digital ICs and implementation of a programmable stimulus generator.

A fault oriented and deterministic ATPG tool is developed running on SUN™ Sparc workstations. This tool is capable of generating test patterns, using a functional description of the IC under test, and a specific fault model which is called single stuck faults, for combinational circuits. This tool and similar tools which were developed in Virginia Polytechnic & State University are plugged to each other via a user interface to perform a ready to use complete test generator and fault simulator system for both combinational and sequential circuits. During the implementation of our tool, a recursive version of a path sensitization algorithm, named "fan-out-oriented (FAN) test generation" is used with some modifications.

On the other hand, a 32 bit stimulus generator is implemented which is controlled by a 68000 microprocessor based system. This generator is capable of running up to 125 MHz. clock rates and has an interface to any PC via RS-232 serial communication protocol standards. Test vectors generated from our ATPG tool, can be downloaded to the stimulus generator in order to apply these vectors physically to the circuit under test at real operational clock rates.

The implementation details, necessary theoretical background information for both topics and the analysis of the performance achieved are presented in this thesis.

ÖZET

Bu tez çalışmasında bir sayısal tümleşik devre test sistemi oluşturmak üzere iki farklı konu üzerinde çalışılmıştır. Bunlar, sayısal tümleşik devreler için Otomatik Test Vektörü Üretimi (OTVÜ) yazılımının geliştirilmesi ve bir programlanabilir işaret üretecinin gerçeklenmesidir.

SUN Sparc iş istasyonlarında çalışan, hata tabanlı ve rastlantısal olmayan yöntemler kullanan bir OTVÜ programı geliştirilmiştir. Anılan program, ardışıl olmayan devreler için "tek değere yapışan hata" olarak isimlendirilen özel bir hata modelini ve test edilecek tümleşik devrenin işlevsel tanımını kullanarak, test vektörleri yaratabilecek kabiliyettedir. Bu program ve Virginia Politeknik ve Devlet Üniversitesinde geliştirilen benzer programlar, bir kullanıcı arayüzü programı ile birleştirilerek, hem ardışıl hem de ardışıl olmayan devreler için bir test vektörü üreteci ve hata benzetim sistemi oluşturulmuştur. Programın gerçeklenmesinde, "çıkış-tabanlı test üretimi (FAN)" isimli bir hat uyarımı algoritmasının içsel (kendini çağıran) bir biçimi, bazı değişiklikler yapılarak kullanılmıştır.

Diğer yandan, 68000 mikroişlemci tabanlı bir sistem tarafından denetlenen bir 32 bit işaret üreteci tasarlanarak gerçeklenmiştir. Bu üreteç, 125 MHz.'lik bir saat hızında çalışabilmektedir ve herhangi bir kişisel bilgisayara RS-232 seri iletişim protokolü ile arayüz oluşturabilmektedir. Bu arayüz vasıtası ile OTVÜ programının yarattığı test vektörleri, test edilecek devreye gerçek çalışma hızlarında uygulanabilmeleri için, işaret üretecine yüklenebilmektedir.

Her iki konu için gerekli kuramsal bilgiler, uygulamanın ayrıntıları ve ulaşılan başarımın analizi tezde sunulmuştur.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Page

LIST OF ABBREVIATIONS

ATPG	Automatic Test Pattern Generation
CAD	Computer Aided Design
CPU	Central Processing Unit
DUT	Device Under Test
EDIF	Electronic Design Interchange Format
FAN	Fan Out Oriented
IC	Integrated Circuit
ISCAS	International Symphosium on Circuits And Systems
LSI	Large Scale Integration
LSSD	Level Sensitive Scan Design
NP	None Polynomial
PC	Personel Computer
PODEM	Path Oriented Decision Making
RTL	Register Transfer Level
RS-232	Recomended Standard - 232
SOCRATES	Structure Oriented Cost-Reducing Automatic TG System
SSF	Single Stuck Fault
TG	Test Generation
TMR	Triple Modular Redundancy
TOPS	Topological Search
VHDL	Very High Speed Hardware Description Language
VLSI	Very Large Scale Integration

1. INTRODUCTION

With the progress of LSI/VLSI technology, the problem of fault detection for logic circuits is becoming more and more difficult. As the logic circuits under test get larger, the test generation cost increases and test becomes harder. Recent studies have shown that the problem of test generation is NP-complete [1]. Hence it appears that the computation is, for the worst case, exponential with the size of circuit. One approach to overcome this problem is to utilize several techniques known as design for testability. The techniques using shift registers such as Level Sensitive Scan Design (LSSD) [2], Scan Path [3], etc., allow the test generation problem to be completely reduced to one of the generating tests for combinational circuits. That is why it is sufficient to develop a fast and efficient test generation (TG) algorithm only for combinational circuits.

TG is the process of determining the input stimuli necessary to test a digital system. It is a complex problem with many interacting aspects [4]. The most important ones are;

- the cost of TG,
- the quality of generated test, and
- the cost of applying generated test.

The cost of TG depends on the complexity of the chosen method. Random TG is a simple process that involves only generation of random vectors. However, to achieve a high-quality test we need a large set of random vectors. On the other hand, random TG does not take into account the structure or the functionality of the circuit under test. In contrast, deterministic TG produces test by processing a description of the circuit and a specific fault model. The circuit description can be in different formats namely, Very high speed Hardware Description Language (VHDL), Verilog Hardware Description Language, Electronic Design Interchange Format (EDIF), netlists generated by CAD tools, ISCAS-89 netlist format, etc. Compared to random TG, deterministic TG is more expensive in terms of memory requirements and CPU time needed but it produces shorter [5], and higher-quality tests.

Quality is measured by the fault coverage ability of the applied test. This metric can be calculated as the ratio of detected number of faults with generated test vectors, to the total number of faults possible in the circuit. Even the TG itself can be simple but the determination of the quality of generated test vectors may be an expensive process, for example with fault simulation.

The cost of applying test is a function of the total number of test patterns and the quality of test generated. A high quality and a small number of patterns are the primary goals, in order to decrease the cost of applying the test. Notice that every set of vectors

generated has to be applied per IC produced and it is necessary to generate a different set of vectors, for each type of product portfolio.

Figure 1.1. shows a combinational circuit with n-inputs. To test this circuit exhaustively, a sequence of 2^n test vectors are required. When we convert the same circuit to a sequential circuit with the addition of m-storage registers, a minimum of $2^{(n+m)}$ of input vectors are required to exhaustively test the new circuit [5]. With LSI, this may be a network with n=25 and m=50. Hence 2^{75} or approximately 3.8×10^{22} patterns are necessary, for exhaustive testing and 100% fault coverage. Assuming one had the patterns and applied them at an application rate of 1000 ns/pattern, the test time required would be over a billion years [5]. Consider another example with a 16X16 multiplier circuit which has 32 input nodes for data only and ignore the necessary control inputs. 2^{32} patterns are necessary for an exhaustive testing and 100% fault coverage. With the same testing rate, the necessary time required would be approximately 72 minutes per IC.

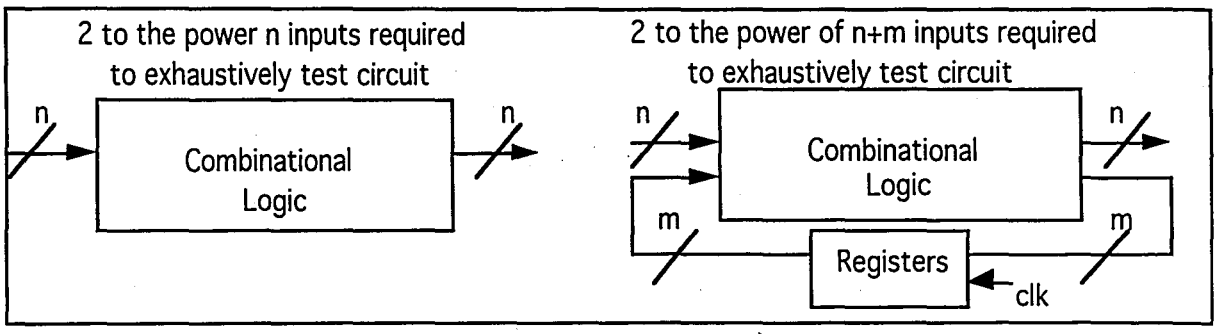


FIGURE 1.1. Combinational explosion of test vectors.

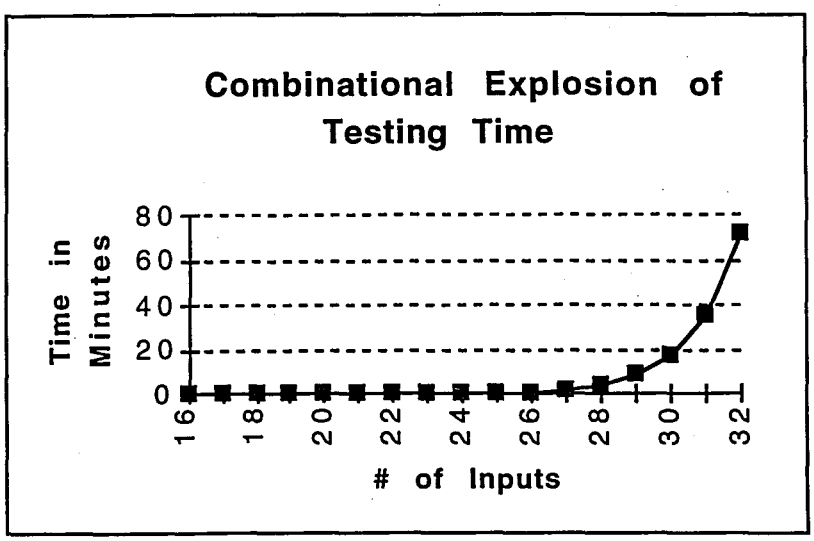


FIGURE 1.2(a). Combinational Explosion of Testing Time.

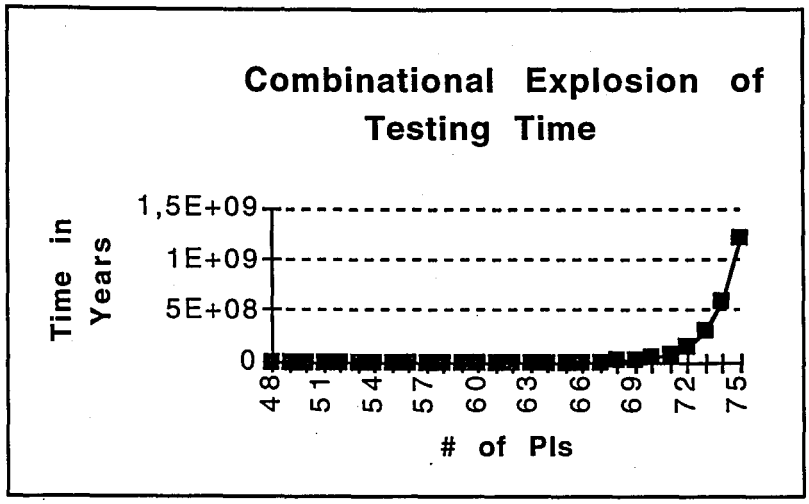


FIGURE 1.2(b). Combinational Explosion of Testing Time.

Figure 1.2 (a) and (b) are presenting the necessary time for exhaustive testing of an IC versus the number of its inputs at a 1MHz. testing rate. It is obvious that deterministic and automated test pattern generators are crucial for the ICs produced with today's technology. Million transistor ICs are common and demands are pushing the technology for further integration of transistors on smaller silicon areas.

On the other hand, in order to perform a meaningful testing system, it is necessary to generate the test vectors physically and apply them to the IC under test with real operational clock rates.

To meet above mentioned requirements we have developed a fault-oriented deterministic ATPG tool named "Poyraz", based on one of the path sensitization algorithms FAN, which was proposed by Fujiwara in 1983 [6]. Our tool runs on SUN™ Sparc workstations and also some other tools which are developed in Virginia Polytechnic & State University are plugged to each other via an user interface software. They all perform a powerful test generation and fault simulation system for both combinational and sequential circuits.

In most of the test systems, there are two common inputs namely, functional description (netlist) of Device Under Test (DUT) and a fault model. Target faults can be generated by a preprocess for all possible cases by processing the input netlist or can be injected by user. Regardless of the source type of fault list, an optional process of fault collapsing can be realized in order to prevent unnecessary repetition of TG process for the same class of faults. The final target fault list is processed by the TG software one by one, using the fault model given. Whenever a test generated by this software, (optionally) it is possible to continue with the remaining faults or determining the quality of test generated, possibly using a fault simulator tool. Generated tests are more likely to be the tests for other

faults, most probably which are on the sensitized path for the target fault currently being processed. Detecting and erasing such faults from fault list will both speed up the TG process and increase the fault coverage of final set of test vectors. Once the fault list is exhausted, it is necessary to compact the set of test vectors in order to minimize the test set. This concludes the software aspects of any TG system. The rest of the blocks shown in Figure 1.2. are the hardware blocks. They are necessary to generate the test vectors physically and observe the responses of DUT.

The organization of the thesis is as follows. In chapter two, the necessary theoretical background for ATPG is given. The evaluation of the ATPG algorithms and their comparisons are made based on this background. In chapter three, the implementation of our ATPG tool based on FAN algorithm is given. Its performance analysis with respect to other implementations, deficiencies and superiority, algorithmic descriptions and implementation specific details are also covered in this chapter. In chapter four, the implementation of stimulus generator and the necessary hardware details are presented. Finally in chapter five achieved results are summarized. User and/or programmer details for the implemented tools are explained in various appendices and they are referred to when necessary throughout the text.

2. THEORETICAL BACKGROUND FOR ATPG

2.1. Fault Modeling

Representation of physical faults by logical faults and the major concepts in logical fault modeling will be briefly presented in this section. A relatively detailed discussion of single stuck at fault model will be given, since this is the fault model used in our ATPG tool implementation.

2.1.1. Logical Fault Models

Logical faults represent the effect of physical faults on the behavior of the modeled system. In modeling any system we differentiate between the logic function modeling and timing; that is why we also distinguish between faults that affect the logic function and delay faults that affect the operation speed of the system. In this section, we will mainly concentrate on the former case. [4]

The advantages of modeling physical faults as logical faults are, the problem of fault analysis becomes logical rather than a physical problem, a logical fault can simulate many of the physical faults which reduces the complexity of the problem also, most of the logical fault models are technology independent which makes the fault analysis problem also independent of technology.

A logical fault can be explicit or implicit. An explicit fault model defines a fault universe in which every fault is individually identified and all faults are explicitly enumerated. These kind of models are practical unless the number of faults generated does not prohibit the application such as simulation or test generation. An implicit fault model defines a fault universe by collectively identifying the faults of interest, typically by defining their characterizing properties.

Given a logical fault and a fault model of a system, we should be able to determine the logic function of the system (I/O relation) in the presence of the fault in principle. Thus,

fault modeling is very closely related to the type of modeling used for the system. Faults defined in conjunction with a structural model are referred to as structural faults; their effect is to modify the interconnections among components. Functional faults are defined in conjunction with a functional model; for example, the effect of a functional fault may be to change the truth table of a component or to inhibit an Register Transfer Level (RTL) operation.

On the other hand, faults can be classified as; intermittent, transient, and permanent. Modeling of intermittent and transient faults require statistical data on their probability of occurrence. Since our attention is on deterministic test generation, we will concentrate only on permanent fault models.

At this point we will make a simplifying single-fault assumption, which is justified by the frequent testing strategy, which states that we should test a system often enough so that the probability of more than one fault developing between two consecutive testing experiments is sufficiently small. In some cases, frequent testing justification may not be valid and our single-fault assumption fails, for example a physical defect in the system can cause multiple faults, in especially newly manufactured systems prior to their first testing experiment. But even under those cases tests, generated under single-fault assumption can detect (most of the times) multiple faults.

In general, structural fault models assume that the components are fault-free and only the interconnections are affected. Typically, these effects are shorts and opens. In many technologies, a short between power or ground rails and a signal line can make that signal line remain at a fixed voltage level. The corresponding logical fault for this physical defect is signal line stuck at a fixed voltage v (v {0,1}), and is denoted by **s-a-v**. A short between two signal lines are known as *bridging faults*, they transform the logic function to a new one. In our study we will concentrate on single-stuck at faults but not the bridging ones.

In many technologies, the effect of an open on a unidirectional signal line with only one fanout is to make the input that has become unconnected due to the open assume a constant logic value and hence appear as a stuck fault (see Figure 2.1(a)). This effect may also result from a physical fault internal to the component driving the line, and without probing the two end points of the line we can not distinguish between the two cases. This distinction, however, is not necessary in the edge-pin testing, where we can assume that the entire signal line is stuck. Note how a single logical fault, namely the line i stuck at value v{0,1}, can represent many totally different physical faults: i open, i shorted to one of the rails, and any internal fault that keeps the line i at value v. An open in a signal line with fanout may result in a multiple stuck fault involving a subset of its fanout branches as illustrated in Figure 2.1(b). In order to obey our single stuck at value assumption we need to consider each fanout branch separately and one at a time.

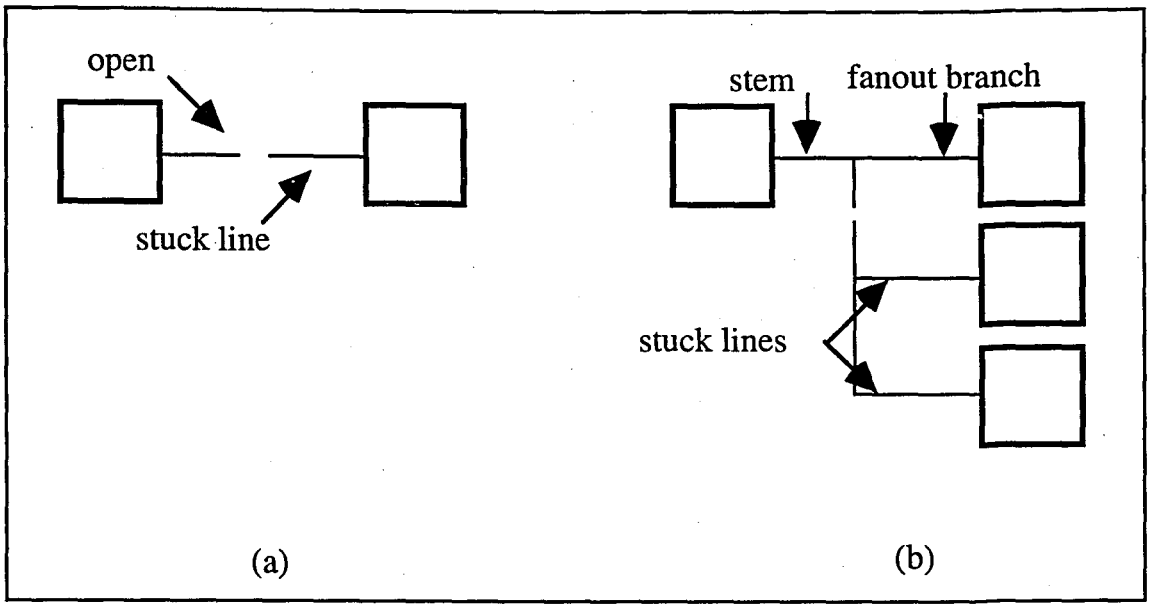


FIGURE 2.1. Stuck Faults Caused by Opens (a) Single (b) Multiple

2.1.2. Fault Detection and Redundancy in Combinational Circuits

Detecting a given fault problem can be viewed as a search of the n-dimensional 0-1 state space of primary input patterns of an n-input combinational logic circuit. [7]. In Figure 2.2. *g* is an internal node and the objective is to generate a test for g s-a-0.

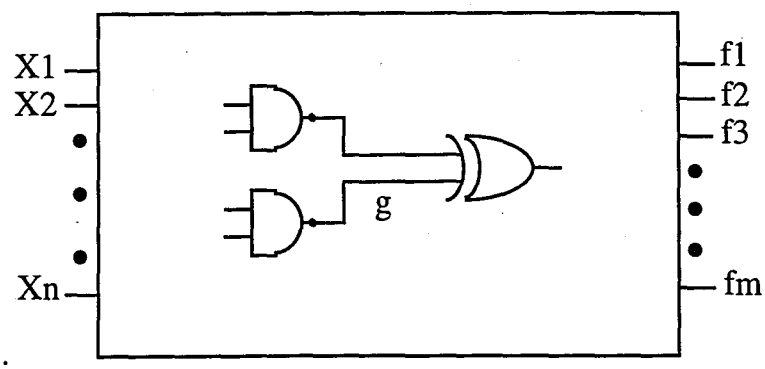


FIGURE 2.2. n-Dimensional 0-1 state space search problem

The state of g can be expressed as a Boolean function of primary inputs, X_1, X_2,...,X_n. Similarly, each primary output (f_j,j =1,2,...,m) can be expressed as a Boolean function of the state on net g as well as primary inputs X_1,X_2,...X_n. Let,

$$g = G\ (X_1,X_2,...X_n) \tag{2.1}$$

and

$$f_j = F_j(g,X_1,X_2,...X_n) \tag{2.2}$$

where $(1 \leq j \leq m)$ and $X_i = 0$ or 1 for $(1 \leq i \leq n)$

The problem of TG for *g* s-a-0 can be stated as one of solving the following set of Boolean equations [8].

$$G(X_1,X_2,...,X_n) = 1 \tag{2.3}$$

$$F_j(1,X_1,X_2,...,X_n) \oplus F_j(1,X_1,X_2,...,X_n) = 1 \tag{2.4}$$

for at least one j, $(1 \leq j \leq m)$ and $X_i = 0$ or 1 for $(1 \leq i \leq n)$

The set of equations for g s-a-1 are the same as above equations except that G is set to zero. Hence, the test generation problem can be viewed as a search of n-dimensional 0-1 space defined by the variables $X_i(1 \leq i \leq n)$ for points that satisfy the above set of equations. More generally the search will result in finding a k-dimensional subspace $k \leq n$ such that all points in the subspace will satisfy the above set of equations (2.3) and (2.4).

A fault in a combinational circuit, is said to be detectable if and only if expression (2.4) is satisfied. This definition of detection is valid for edge-pin testing with full comparison of results. In other words, a fault is detectable if the responses of the faulty and the not faulty circuits are different for a specific input vector. TG is the search process for this specific input vector. There are many different approaches for this search problem.

The integer programming problem and various problems in the field of artificial intelligence have been approached using state space search methods and the branch and bound technique. Implicit enumeration algorithm phenomena mentioned in [7] is a subset of branch and bound algorithms which arch of an n-dimensional 0-1 state space. Detailed information about the algorithms for the TG problem will be presented in following sections of this chapter.

The function realized by the circuit given in Figure 2.3. is $Z=[(X_2+X_3).X_1]+[/X_1.X4]$. Let fault f be X_4 s-a-0. The expression $(/X_1.X_4)$ represents, in compact form, any of four tests (0001, 0011, 0101, 0111, from LSB to MSB) that detect fault *f*. In the presence of fault the function of circuit becomes $Z_f=[(X_2+X_3).X_1]$.

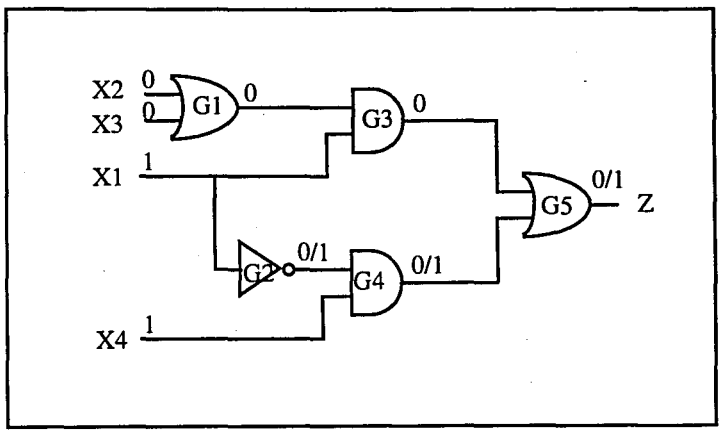


FIGURE 2.3. Fault Detection and Sensitization

Expression (2.4) says that exclusive-or of fault-free and faulty responses of any combinational circuit should yield logic one in order to say that the fault is detectable. In short;

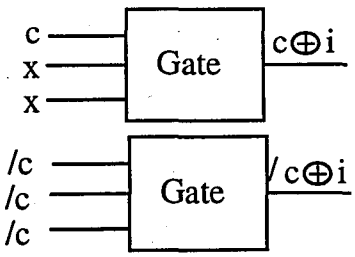
$$Z \oplus Z_f = 1 \qquad (2.5)$$

If we substitude Z and Z_f into (2.5), $\{[(X_2+X_3).X_1]+[/X_1.X4]\} \oplus [(X_2+X_3).X_1] = 1$ will be obtained which then reduces to, $(/X_1.X_4) = 1$. Thus, any test in which $(X_1= 0)$ and $(X_4 = 1)$ is a test for the fault *f*.

Let us simulate the circuit of Figure 2.3. for the test vector t = 1001, both without and with the fault G2 s-a-1 present. The results of these two simulations are shown in the same figure. The results that are different in two cases have the form **v/vf**, where *v* and *vf* are corresponding signal values in the fault-free and in the faulty circuits respectively. The fault is detected since the output values in two cases are different. Figure 2.3. illustrates two basic concepts in fault detection. First the test *t* that detects a fault *f*, activates *f*, generates an error by creating different *v* and *vf* values at the site of the fault. Second, *t* propagates the error to a primary output *w*, that is, makes the all lines along at least one path between the fault site and *w* have different *v* and *vf* values. In Figure 2.3. error propagates along the path (G_2, G_4, G_5). A line whose value under test *t* changes in the presence of the fault *f* is said to be sensitized to the fault *f* by the test *t*.. A path composed of sensitized lines is called a sensitized path.

A gate whose output is sensitized to a fault *f* has at least one of its inputs sensitized to *f* as well. Before giving some properties of such a gate let us take a look at the characteristics of some primitive gates. The set of primitive elements used in many simulation and test generation systems includes the basic gates such as AND, OR, NAND, and NOR. These gates can be characterized by two parameters, the controlling value *c* and inversion *i*. The value of an input is said to be controlling if it determines the value of the gate output regardless of the values of other inputs; then the output value is $c \oplus i$. Figure 2.4. shows the general form of the primitive cubes [4] of above mentioned primitive gates with three inputs.

	c	i
AND	0	0
OR	1	0
NAND	0	1
NOR	1	1



c x x	c ⊕ i
x c x	c ⊕ i
x x c	c ⊕ i
/c /c /c	/c ⊕ i

FIGURE 2.4. Primitive Cubes for a Gate with Controlling value c and Inversion i

The properties of a sensitized gate *G* are as follows;

a) all the inputs of *G* sensitized to *f* have the same value (say *a*),
b) all the inputs of *G* not sensitized to *f* have value */c*,
c) the output of *G* has value $a \oplus i$..

Detectability is another important aspect of test generation. A fault *f* is said to be detectable if there is a test *t* that detects *f*, otherwise *f* is an undetectable fault. For such faults $Z_f(x) = Z(x)$ and no test can distinguish between those responses or in other words when no fault is activated and a sensitized path created to any primary output, it is not possible to test the circuit. Since the undetectable faults do not change the behavior of the circuit they may be considered harmless and hence can be ignored. However, a circuit with an undetectable fault may invalidate the single fault assumption. Even a complete test set may be insufficient in the existence of an undetectable fault.

Figure 2.5. illustrates how a valid test vector becomes invalid with the existence of a second fault. In Figure 2.5.(a). shows how the test vector t = 1101 detects the fault b s-a-0.

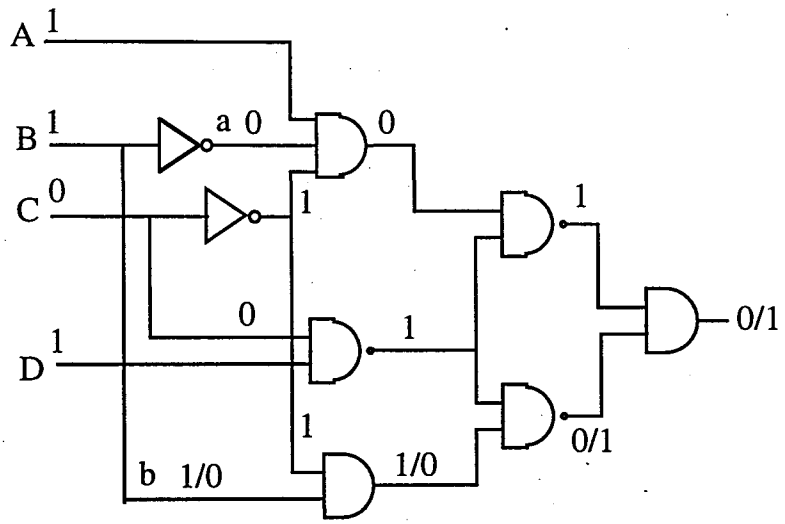


FIGURE 2.5.(a) Sensitization of Fault b s-a-0 with Test Vector t = 1101

Figure 2.5.(b). shows that b s-a-0 is no longer detected by the test vector t if the undetectable fault a s-a-1 is also present. Thus if t is the only test that detects b s-a-0 in a complete detection test set T, then T is no longer complete in the presence of a s-a-1.

The situation in which the presence of an undetectable fault prevents the detection of another fault by a certain test is not limited to faults of the same category; for example, an undetectable bridging fault can similarly invalidate a complete test set for stuck faults.

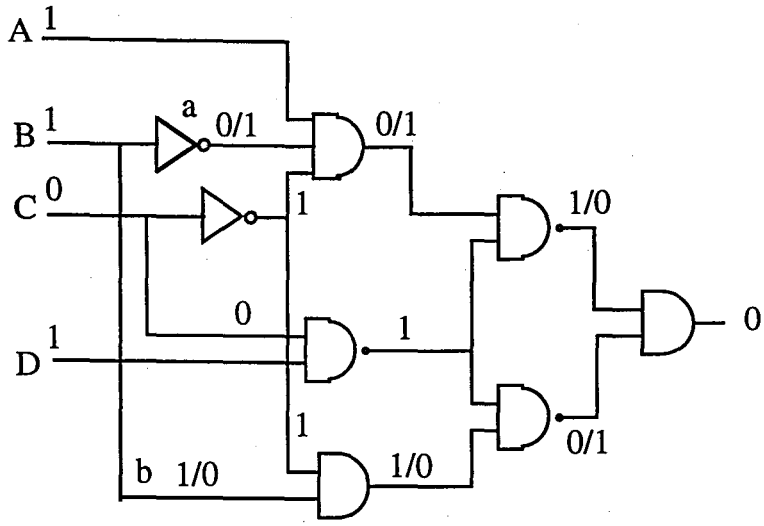


FIGURE 2.5.(b) Effect of Undetectable Fault a s-a-1 on Test Vector (t = 1101)

A combinational circuit that contains an undetectable stuck fault is said to be redundant, since such a circuit can be simplified by removing at least one gate or gate input. For instance, suppose that a s-a-1 fault on an input of an AND gate G is undetectable. Since the function of the circuit does not change in the presence of the fault, we can permanently place a 1 value on that input. But an n-input AND with a constant 1 value on one input is logically equivalent to the (n-1)-input AND obtained by removing the gate input with the constant signal. Similarly, if an AND input s-a-0 is undetectable, the AND gate can be removed and replaced by a 0 signal. Since we have a new signal with a constant value, the simplification process may continue. The simplification rules are summarized in the following Table 2.1. [4].

TABLE 2.1 Simplification Rules

Undetectable Fault	Simplification Rule
AND (NAND) input s-a-1	Remove input
AND (NAND) input s-a-0	Remove input, replace by 0 (1)
OR (NOR) input s-a-0	Remove input
OR (NOR) input s-a-1	Remove input, replace by 0 (1)

The general concept of redundancy is broader than the particular one related to the existence of undetectable faults, and it denotes a circuit that can be simplified. One possible simplification, not covered by the rules given in Table 2.1., is to replace a string of two inverters by a single line. In the following, we will restrict ourselves to the definition of redundancy related to undetectable stuck faults. The term "redundant" is also applied to undetectable stuck at faults and to the lines that can be removed. A combinational circuit in which all stuck faults are detectable is said to be irredundant.

Redundancy does not necessarily mean an inefficient design it may be desirable for some cases. For example triple modular redundancy (TMR) is a basic technique used in fault-tolerant design. For preventing the transient hazards, redundancy is injected to combinational designs intentionally also. (See chapter 4 in [4])

As mentioned in the above discussion, redundancy may invalidate a complete test set. Also, it can produce some other problems; First, if f is a detectable fault and g is an undetectable fault, then f may become undetectable in the presence of g. Such a fault f is called a second generation redundant fault. Second, two undetectable single faults f and g may become detectable if simultaneously present in the circuit. In other words, multiple fault {f,g} may be detectable even if its single-fault components are not.

Note that, in practice when we deal with large combinational circuits, even irredundant circuits may not be tested with complete detection test sets. The reason of this is the limits which are injected to the TG systems based on the cost of TG process. In practice, there is no difference between an undetectable fault f and an undetected fault g by the test applied. Clearly, fault g could be present in the circuit and hence invalidate the single-stuck fault assumption.

Identifying redundancy is closely related to the problem of test generation. To show that a line is redundant means to prove that no test exists for the corresponding fault. The TG problem belongs to a class of computationally difficult problems, referred to as NP-complete. Let n be the size of problem, which means n is the number of gates in the device under test. At present there is no polynomial time algorithm known which solves this kind of problems using a number of operations proportional to n^r, where r is a finite constant.

Although TG and identification of redundancy is a computationally difficult problem, practical TG algorithms usually run in polynomial time. They may encounter a circuit fault that can not be solved in polynomial time. Experience has shown that redundant faults are usually the ones that cause the that cause test generation algorithms to exhibit their worst-case behavior.

2.1.3. Fault Equivalence and Fault Location in Combinational Circuits

Two faults *f* and *g* are said to be functionally equivalent if $Z_f(x) = Z_g(x)$. A test *t* is said to distinguish between two faults *f* and *g* if $Z_f(t) \neq Z_g(t)$; such faults are distinguishable. There is no test that can distinguish between two faults that are functionally equivalent. The relation of functional equivalence partitions the set of all possible faults into functional equivalence class. The set of all tests that distinguish between *f* and *g* is given by the solutions of the equation,

$$Z_f(x) \oplus Z_g(x) = 1 \tag{2.6}$$

In general, functionally equivalence phenomenon is not limited to the same fault universe, but we will concentrate on functional equivalence of the same type.

With any n-input gate we can associate 2(n+1) single stuck faults. For a NAND gate all the input s-a-0 faults and the output s-a-1 are functionally equivalent. In general for a gate controlling value c and inversion i, all the input s-a-c faults and the output s-a-(c⊕i) are functionally equivalent. Thus for an n-input gate (n>1) we need to consider only n+2 single stuck faults. This type of reduction of the set of faults to be analyzed based on equivalence relations is called equivalence fault collapsing. An example is given in Figure 2.6.(a) and (b).

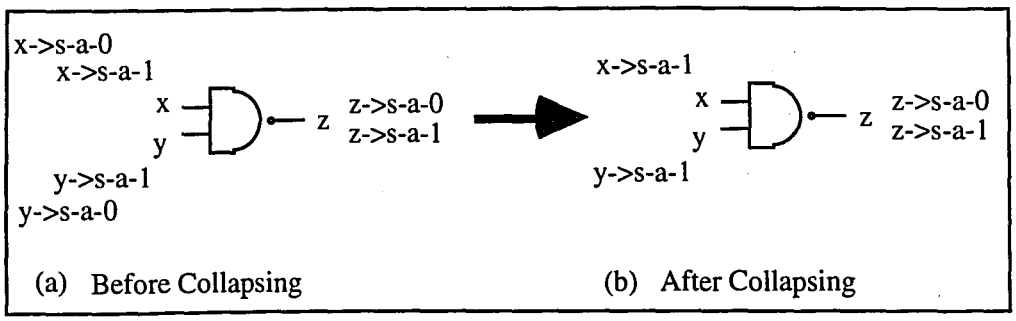


FIGURE 2.6. NAND Gate Functionally Equivalent Fault Collapsing

In addition to fault detection, the goal of the testing is fault location as well, we need to apply a test that not only detects the detectable faults but also distinguishes among them as much as possible.

It is convenient to consider that a fault-free circuit contains an empty fault denoted by ϕ. Then $Z_\phi(x) = Z(x)$. This artifact helps in understanding the relation between the fault location and fault detection. Namely, fault detection is just a particular case of fault location, since a test that detects a fault f distinguishes between f and ϕ. Undetectable faults are in the same equivalence class with ϕ. Hence, a complete fault location test set must include a complete fault detection test set.

The presence of an undetectable fault may invalidate a complete location test set. If f and g are two distinguishable faults, they may become functionally equivalent in the presence of an undetectable fault.

A complete fault location test set can diagnose a fault to within a functional equivalence class. This represents the maximal diagnostic resolution that can be achieved by edge-pin testing. In practice, large circuits are tested with test sets that are not complete. Another equivalence relation can be used to characterize the resolution achieved by an arbitrary test set. Such that, two faults f and g are functionally equivalent under a test set T iff $Z_f(t) = Z_g(t)$ for every test t, which are elements of test set T. Functional equivalence implies equivalence under any test set, but equivalence under a given test set does not imply functional equivalence.

2.1.4. Fault Dominance in Combinational Circuits

If the objective of a testing experiment is limited to fault detection only, then, in addition to fault equivalence, another fault relation can be used to reduce the number of faults that must be considered. This new relation can be defined as; Let T_g be the set of all tests that detect a fault g. We say that a fault f dominates the fault g iff f and g are functionally equivalent under T_g. In other words, if f dominates g, then any test t that detects $g(Z_g(t)\ Z(t))$, will also detect f on the same primary outputs because $Z_f(t) = Z_g(t)$. Therefore, for fault detection it is unnecessary to consider the dominating fault f, since by deriving a test to detect g we automatically obtain a test that detects f as well. This case is demonstrated in Figure 2.7.

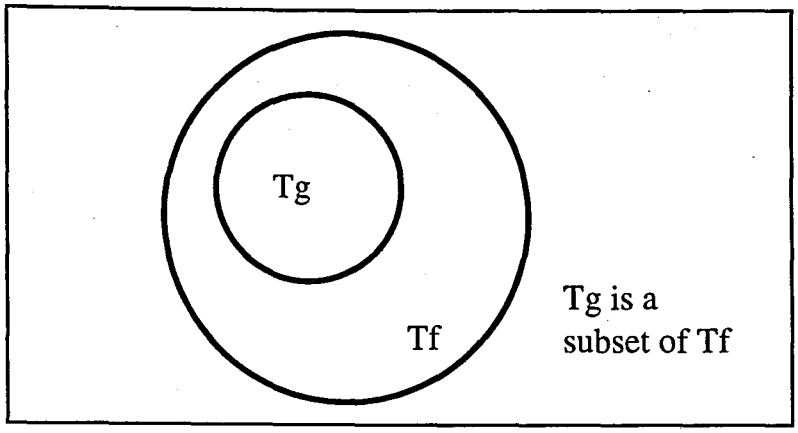


FIGURE 2.7. Fault Dominance and Test Sets for Faults g and f

For example, consider the NAND gate in Figure 2.8. and let g be the y s-a-1 and f be the z s-a-0. The set T_g consists of only one test, namely 10 which also detects *f*, and T_f consists of 10,01. As we see, T_g is a subset of T_f and z s-a-0 dominates y s-a-1. Then, we can remove the output (dominating) fault from the fault list and this sort of fault collapsing is called as "dominating fault collapsing." In general, for a gate controlling value c and inversion i; the output s-a-(/c⊕i) fault dominates any input s-a-/c and output fault can be removed from fault list for fault detection based TG computations.

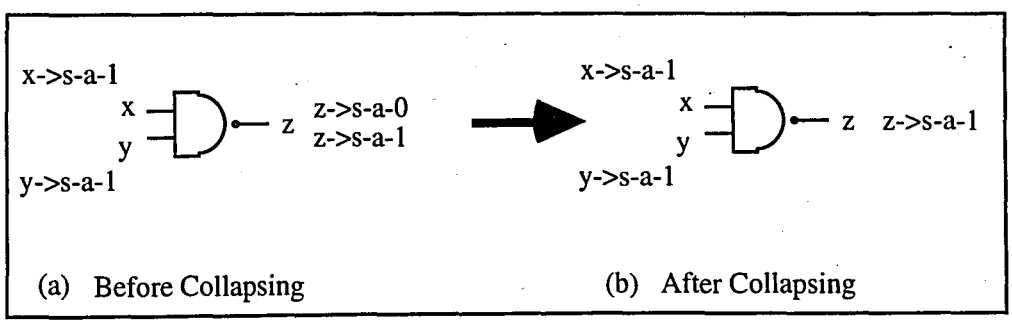


FIGURE 2.8. Dominance Fault Collapsing

It is interesting to observe that we can have two faults, f and g, such that any test that detects g also detects f without f dominating g. Consider the circuit given in Figure 2.9. Let *f* be Z_2 s-a-0 and *g* be y_1 s-a-1. The set T_g consists only of the test 10, which also detects *f* (notice that fault *f* is detected from another primary output). But according to definition of fault dominance fault *f* in this case does not dominate the fault *g*. Although for fault detection

it is not necessary to consider *f*, it would be difficult to determine this fact from an analysis of the circuit.

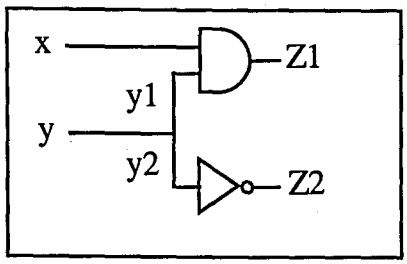


FIGURE 2.9. Collapsing is Possible even for not Dominating Case

When choosing a fault model it is important to select one whose faults are generally dominated by faults of other fault models, because a test set detecting the faults of the chosen model will also detect many other faults that are not even explicitly considered. The best fault model with such a property appears to be the single-stuck fault model.

2.1.5. The Single Stuck-Fault Model

The single stuck-fault (SSF) model is also referred to as classical or standard fault model because it has been the first and the most widely studied and used. It has following and interesting attributes that makes it famous. First, it represents many different physical faults [9]. Second, it is independent of technology, as the concept of a signal line being stuck at a logic value can be applied to any structural model. Third, experience has shown that tests that detect SSFs detect many non-classical faults as well. Fourth, compared to other fault models, the number of SSFs in a circuit is small. Moreover, the number of faults to be explicitly analyzed can be reduced by fault collapsing techniques. And fifth, SSFs can be used to model other type of faults [4].

When considering an explicit fault model it is important to know the size of the fault universe it defines. If there are n lines on which SSFs can be defined, the number of possible faults is 2n. To determine the n we have to consider every fanout branch as a separate line. We will do this computation for a gate level model. Primary inputs and the

outputs of any gate is called the signal sources of the circuit. Every signal source i with a fanout count of f_i contributes k_i lines to the number of possible fault sites, where k_i is defined as in the following expression,

$$\begin{aligned} &\text{if } f_i=1 \Rightarrow k_i = 1 \\ &\text{if } f_i>1 \Rightarrow k_i = 1+f_i \end{aligned} \tag{2.6}$$

The number of possible fault sites is,

$$n = \sum_i k_i \tag{2.7}$$

where the summation is over all the signal sources in the circuit. Let us define a variable q_i as,

$$\begin{aligned} &\text{if } f_i=1 \Rightarrow q_i = 1 \\ &\text{if } f_i>1 \Rightarrow q_i = 0 \end{aligned} \tag{2.8}$$

Then, expression (2.7) becomes,

$$n = \sum_i (1 + f_i - q_i) \tag{2.9}$$

Let us denote the total number of gates in the circuit by G and the number of primary inputs by I. The average fanout count in the circuit is given by,

$$f = \left[\sum_i f_i \div (G+I) \right] \tag{2.10}$$

The fraction of signal sources with only one fanout can be expressed as,

$$q = \left[\sum_i q_i \div (G+I) \right] \tag{2.11}$$

with these described notation we can rearrange (2.9) resulting as,

$$n = (G + I)(1 + f - q) \quad (2.12)$$

Notice that G is much larger than I in large circuits and q>0.5. So, the dominating factor in (2.12) is Gf. Thus we can conclude that total number of all possible SSFs is slightly larger than 2Gf. It is important that this number is dependent on the gate count and the average fanout count.

Let us conclude our discussion on fault modeling with the discussion of check points. Primary inputs and any fanout branches are called check points. In a combinational circuit C any test set that detects all SSFs on the check points of C detects all SSFs in C, unless the circuit C has no redundancy. If this is not the case, then additional effort is necessary for a complete fault detection test set. These methods are presented in [10].

2.2. Fault Simulation

As the name implies, fault simulation consists of simulating a circuit in the presence of faults. Comparing the fault simulation results with those of the fault-free simulation of the same circuit simulated with the same applied test T, we can determine the faults detected by the T.

2.2.1. Applications

One use of the fault simulation is to evaluate a test *T*. Usually, the grade of *T* is given by its fault coverage, which is the ratio of the number of faults detected to the total number of faults simulated. This grade is directly related to the faults simulated, as even a test with 100 percent fault coverage may still fail to detect faults outside the considered fault model. Thus the fault coverage represents only a lower bound of defect coverage, which is the probability that T detects any physical fault in the circuit. Experience has shown that a test with high coverage for SSFs also achieves a high defect coverage. Test evaluation based on fault simulation has been applied mainly to SSF model.

The quality of test greatly influences the quality of shipped product. Let Y be the manufacturing yield, that is, the probability that a manufactured circuit is defect-free. Let DL denote the defect level, which is the probability of shipping a defective product, and let d be the defect coverage of the test used to check for manufacturing defect. The relation between these variables is given by [4],

$$DL = 1 - Y^{(1-d)} \tag{2.13}$$

Assuming that the fault coverage is close to defect coverage, we can use this relation given in (2.13) and illustrated in Figure 2.10. to determine the fault coverage required for a given defect level and the yield.

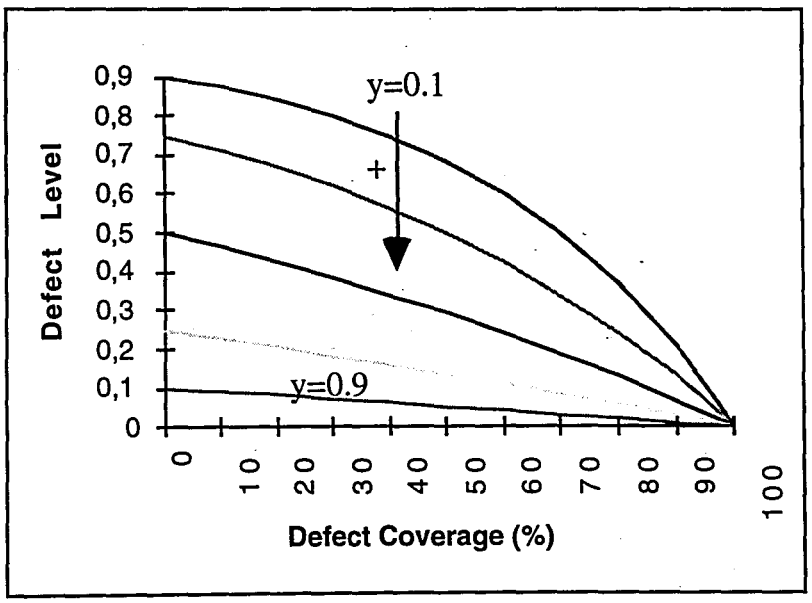


FIGURE 2.10. Defect Level as a Function of Defect Coverage and the Yield

Fault simulation plays an important role in test generation; many TG systems use fault simulation as an evaluation tool for the generated test set T. This set is iteratively modified by a program or by a test designer until the desired fault coverage is achieved.

Another use of fault simulation is during the TG phase is illustrated by Figure 2.11.

Many TG algorithms are fault-oriented; that is, they generate a test for one specified fault, referred to as target fault. Often, the same test is also detects many other faults that can be determined by fault simulation. Then, all the detected faults are discarded from the fault list and a new one is selected from the remaining ones if any.

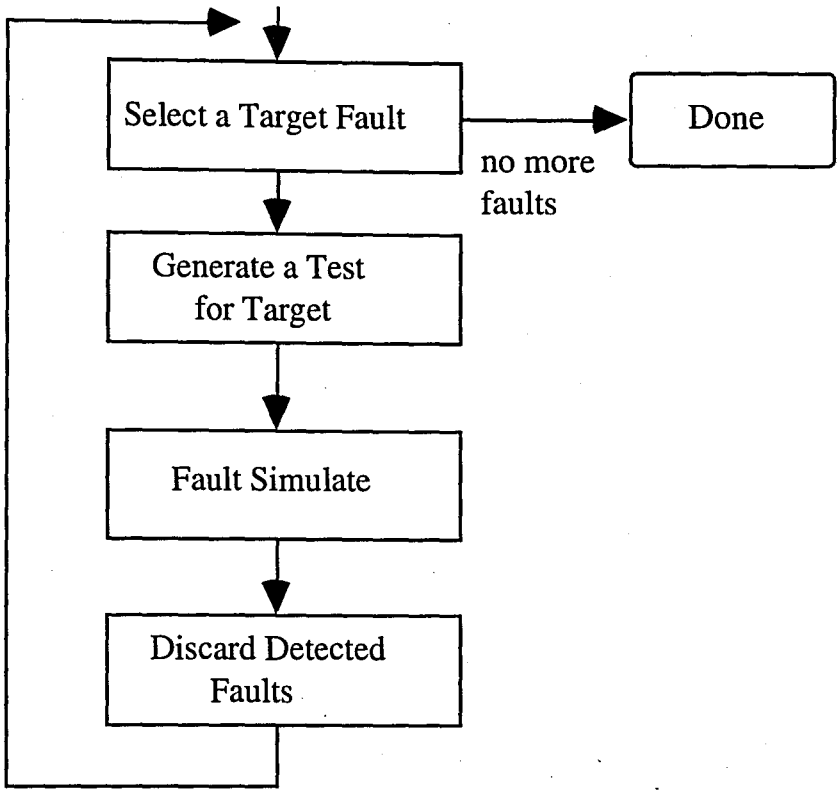


FIGURE 2.11. Fault Simulation Usage in TG

Fault simulation is also used for creating fault dictionaries, for analysis of high-reliability systems operation in the presence of faults, and so on. See the reference [4] for more details.

Fault simulation plays an important role in ensuring a high quality for digital circuits. Its high computational cost motivates new research in this area. The main research directions are hardware support, new algorithms, and the use of hierarchical models.[4]

2.3. Testing for Single Stuck Faults

The TG process depends primarily on the type of experiment for which the stimuli are generated. This thesis work is devoted to off-line, edge-pin, stored pattern testing with full comparison of the output results. (See reference [4] for the definitions of the terms)

2.3.1. Basic Issues

Figure 2.12. shows a general view of a deterministic TG system. Tests are generated based on a model of a circuit, and a given fault model. The generated tests include the stimuli to be applied and the expected response of a fault-free circuit. Some of them also produce diagnostic data to be used in fault location.

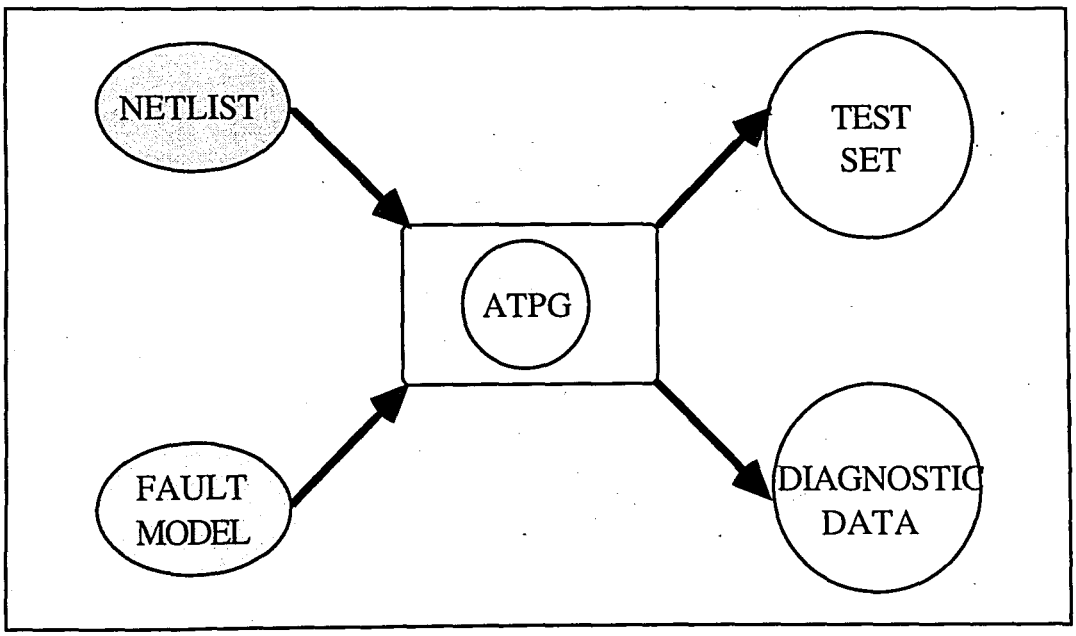


FIGURE 2.12. Deterministic TG System

2.3.2. Fault Oriented ATPG

The fundamental steps in generating a test for a fault l s-a-v are, to excite the fault and propagate it to any primary output (PO). Activating a fault means applying a stimulus to the primary inputs (PI) that cause the line l to have value /v. This is an instance of line justification problem which deals with finding an assignment of PI values that results in a desired value setting with the specified line in the circuit. To keep track of error propagation, we must consider both the values for fault-free circuit N and faulty circuit N_f defined by the target fault f. For this, we define composite logic values of the form v/v_f, where v and v_f are the values of the same signal in N and N_f. The composite logic values that represent errors are denoted by D and /D [11]. The other two composite values (0/0 and 1/1) are denoted by 0 and 1. Any logic operation between two composite values can be done by separately processing the fault-free and faulty values. (/D + 0 = 0/1 + 0/0 = 0 + 0/1+0 = 0/1 = /D) [4]. As in this three valued logic, we add an undefined logic value which is X to this set of four composite values. Described new logic levels and their representations are summarized in Table 2.2.

TABLE 2.2. Five Valued Composite Logic Levels

v/vf	0/0	1/1	1/0	0/1	Unspecified
Denoted	0	1	D	/D	X

Basic goals of any TG algorithm are, error activation, justification of line values needed for error activation, error propagation, and iterating the last two items until an error is propagated to any PO, or detecting that the target error in process is redundant. In fanout-free circuits the problem is straightforward, because the line justification problems are independent from each other and you can solve them in any order without contradicting to any line value found. In contrast, circuits with fanout have big problems at this point because the line justification problems are dependent on each other and you have to solve them one by one without contradicting any of your particular solutions. This brings the need of choosing the correct choice, at every step of TG process generally. If you are lucky enough, in a finite search space which is bounded by an exponential function of the number

of the gates, you can always take the correct choice and reach the result without returning to a previous step and changing your incorrect choice and its implications. This optimistic case can occur rarely and most of the time, at any step of any algorithm returning back and changing the last choice made becomes inevitable, which we call backtracking. A backtracking strategy is necessary for the systematic exploration of the complete space of possible solutions and recovery from the incorrect decisions. Recovery involves restoring the state of the computation to the state existing before the incorrect decision.

Usually, assignments which are the results of decisions made uniquely, determine other values. Computing these values and checking their consistency with the previously determined ones is referred to as implication process. An example is given in Figure 2.13. to describe the concepts given up to now as, backtracking, error propagation, implication, and justification.

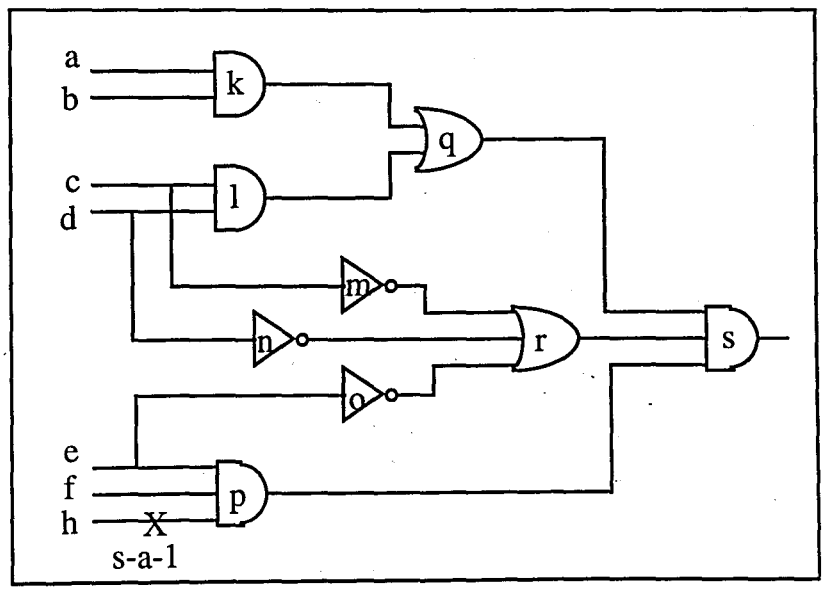


FIGURE 2.13. An Example for the General TG Concepts [4]

Consider the line h to be stuck at one in Figure 2.13. To activate this fault, we must set h=0. To propagate this fault there is a unique path passing from p and s gates. For error propagation we need to set e=f=1, and q=r=1. The value q=1 can be justified by l=1 or by k=1. First let's try to set l=1. This leads to c=d=1. However, these two assignments together with the previously specified e=1, would imply r=0 which would also imply s=0 preventing the detection of fault h s-a-1 from the output, which is an inconsistency. Hence, we must choose the alternative choice k=1, which implies a=b=1. Now the only remaining line-justification problem is r=1. Either m=1 or n=1 leads to consistent solution. Changing

the choice l=1 to k=1 is called as backtracking and resultant assignments of l=1 has to be removed also in order to turn to previous state and to make a new choice.

2.3.3. Common Concepts

There are several fault oriented TG algorithms and they all have some common points which are going to be presented in the following paragraphs. After brief descriptions of those concepts, we will present FAN TG algorithm in detail, since it is the predecessor to the ones which are being used today and also it is the one that we have implemented in this thesis work.

Decision tree; A backtracking based TG algorithm can be visualized with the aid of a decision tree. Figure 2.14 shows an example decision tree where the nodes of the tree shows the problems to be solved and the branches between any of the two nodes are the decisions taken. Dashed branches represents the untried alternatives and the boxes are the ends where the backtrack occurs or a successful result is found, and they are discriminated by F and S letters respectively. F boxes are met when an inconsistent choice is made or an error propagation is precluded from that state on.

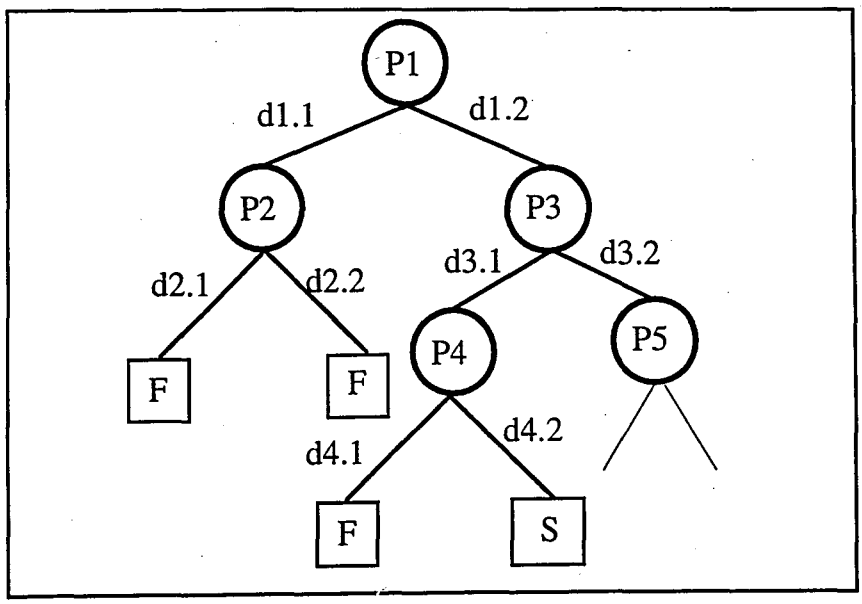


FIGURE 2.14. Decision Tree

Implicit and explicit enumeration; The algorithms for TG can be classified with respect to their methods of searching in their solution space, as implicit enumeration or explicit enumeration algorithms. Enumeration here means finding a test if one exists, or describing a complete algorithm which exhaustively and systematically searches the solution space. Their methods of search is hidden under the words implicit or explicit. Explicit means repeatedly generating tests and checking whether it detects the target fault or not. In contrast, in implicit enumeration algorithms, search process is directed toward the vectors that can satisfy the set of constraints imposed by the set of lines whose values must be simultaneously justified. As the set of constraints grows during the execution of the algorithm, the set of vectors that can satisfy them becomes smaller and smaller. The advantage of implicit enumeration is that it bounds the search space and begins to do so early in the search process.

In Figure 2.15., an example combinational circuit is given for better visualization of different enumeration approaches. Using implicit enumeration, we start by limiting the search space by rejecting the b=0 and c=0. These imposed limits are the constraints for not meeting an inconsistent assignment which will fail the algorithm in the future by not activating the target fault and precluding the error propagation via gate h. Since there are 2^4 possible choices, an explicit enumeration algorithm will run over 16 different input vectors for the worst case but by rejecting b=0 and c=0 an implicit enumeration algorithm would require searching in total 4 different input vectors for the worst case.

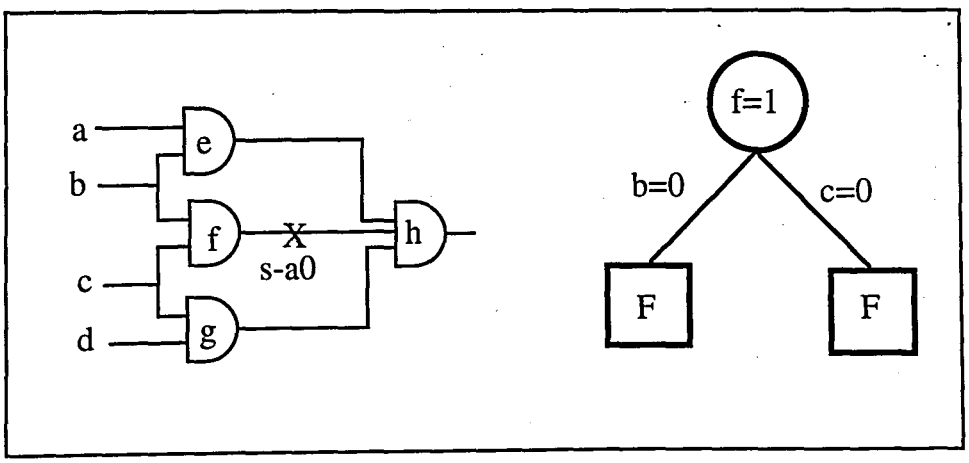


FIGURE 2.15. Implicit versus Explicit Enumeration

Complexity Issues; For any exhaustive algorithm, the number of operations needed for finding a solution for the worst case is can be stated as;

$$\#\text{ of Operations Needed for TG} = f(\exp(\#\text{ of gates})) \tag{2.14}$$

to minimize the total search or TG time any TG algorithm is allowed to do only limited amount of search. This limitation can be in terms of number of incorrect decisions (backtracks) and/or elapsed CPU time. This will result in not generating tests for some detectable faults. The trade-off has to be done in and between the target fault coverage ratio and the cost of TG that can be tolerated as illustrated in Figure 2.16.

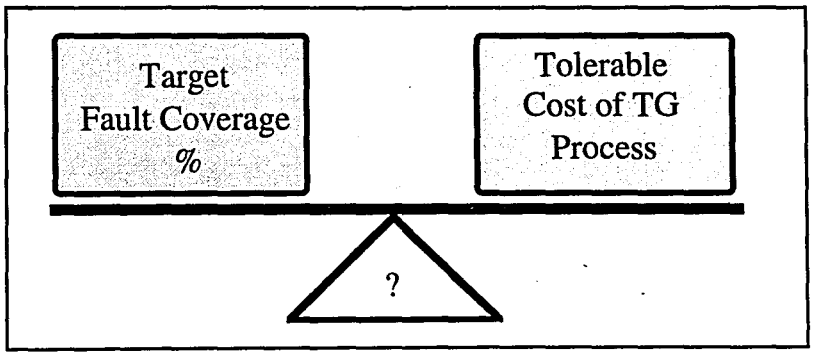


FIGURE 2.16. The Balance Necessary for Reciprocal Parameters of TG

The worst case behavior is mainly observed for undetectable faults, the best case behavior occurs when the result is obtained without backtracking which may will be possible when finding a result only with implications or by chance taking only the correct decisions. Under such optimistic cases,

$$\#\text{ of Operations Needed for TG} = f(\#\text{ of gates}) \tag{2.15}$$

Notice that in expression (2.15), function f is a linear one in contrast to the function in expression (2.14) which is an exponential one.

The key factor in controlling the complexity of any TG algorithm is to minimize the number of incorrect decisions or in other words the need for backtracking. Several heuristics can be introduced to any TG algorithm in order to minimize the number of backtracking. A simple one is the maximum implication principle which will decrease the number of problems to be solved and point any possible inconsistency sooner. This method will be detailed later in the description of FAN algorithm. Other methods like distance based cost functions related to controllability and observability measures can be found in chapter 6 of reference [4].

D-Frontier; is the set of gates whose outputs are in X state and having error (D or /D) on one or more of their inputs. Error propagation consists of selecting one gate from D-frontier and assigning values to the unspecified gate inputs so that the gate output becomes D or /D. This process is also called as D-drive operation. If the D-frontier is empty and there is no error at any primary output than a backtrack should occur from the last decision made and the circuit under test should be recovered to the state prior to that useless choice made.

J-Frontier; to keep the track of currently not solved line justification problems, we use a set called J-frontier, which consists of all gates whose outputs are determined but not implied yet by the gate inputs.

The implication process; is the process whose tasks are, Computation of all values that can be uniquely determined by implication, Checking the consistency of all new assignments with the assignments made before and maintaining the D-frontier and J-frontier. This process is one of the most important parts of any fault-oriented TG algorithms and should be well understood prior to any attempt for implementation of any TG algorithm. In this thesis work, considerable time has been spent on finding and understanding the process due to the reason that there is not much detail about this process in literature. This point make the implementation specific approaches possible for this process which highly effects the overall performance of TG algorithms. Very brief and maybe the most comprehensive definition of this process within the literature can be found in the implementation details chapter of this thesis work.

2.3.4. Algorithms

Many of the concepts presented above are common to a class of TG algorithms. In

general, they are referred to as path-sensitization algorithms. In this section, we will present briefly a couple of algorithms but mainly will concentrate on FAN algorithm.

The D-Algorithm; is presented by Roth in 1966 [12], a characteristic feature of this algorithm is its ability to propagate errors on several reconvergent paths. This feature is referred as multiple-path sensitization, is required to detect certain faults that otherwise would not be detected. A recursive version of this algorithm can be found in [4].

Single Path Sensitization; Experience has shown that faults whose detection is possible only with multiple-path sensitization are rare in practical circuits. To reduce computation time, TG algorithms are often restricted to single path sensitization. To restrict the D-algorithm to single-path sensitization, after we select a gate from D-frontier and D-drive operation, we consider only that D or /D for further propagation and ignore the other gates from D-frontier.

The PODEM (Path-Oriented Decision Making) is presented by Goel in 1981 [7]. Direct search process of PODEM is its characteristic in which decisions consist of only PI assignments. Each problem is backtraced to the PIs in order to map them onto the PI value/s. Those mappings are simulated in the forward direction in order to make assignments to the lines. This procedure is iterated until an error signal is propagated to any PO or a redundancy is detected. The superiority of this approach is that there is no need for line justification and keeping the track of J-frontiers since all assignments are made onto the PIs and line values are assigned only with forward simulations.

Let us concentrate on the backtrace mechanism of PODEM; the algorithm treats a value v_k to be justified for line k as an objective(k,v_k) to be achieved via PI assignments. A backtracing procedure maps a desired objective into a PI assignment that is likely to contribute to achieving the objective.

A further improvement is proposed by Fujiwara in 1983 [6], with the new algorithm which is called FAN (Fanout-Oriented TG). FAN introduces two major extensions to the backtracing concept of PODEM. First, rather than stopping at PIs, backtracing in FAN may stop at special internal lines. Second, rather than trying to satisfy one objective FAN uses a multiple-backtrace procedure that attempts to simultaneously satisfy a set of objectives.

FAN classifies internal lines and uses this classification to decide where to stop its multiple-backtrace subroutine. There are three classes of lines namely bound, free and head lines. A line that is reachable from at least one stem is called bound line. A line that is not bound is called a free line. A free line that directly feeds a bound line is called a head line.

Figure 2.17. presents an example for the line classification definitions of the FAN algorithm. PI lines a,b and c are not reachable from any stem so they are called as free lines. Line d is not reachable from any stem, so it is a free line also, but it is feeding a bound line directly, so at the same time it is a head line. The rest of the lines are classified as bound lines since they are reachable at least from one stem.

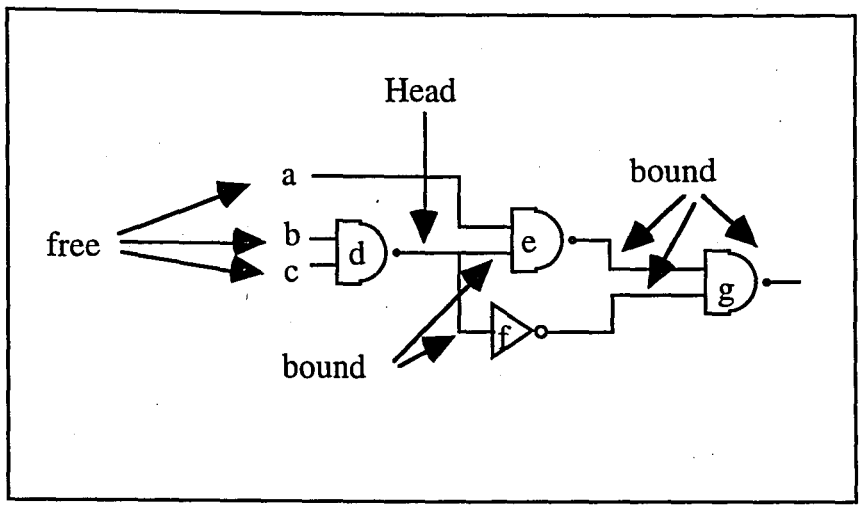


FIGURE 2.17. Line Classification

Since a subcircuit feeding a head line l is fanout free circuit, a value of l can be justified without contradicting any other value previously assigned. Which means it is not necessary to backtrace the objectives until the PIs, we can stop at head lines and postpone the justification of head lines to the end of the TG process.

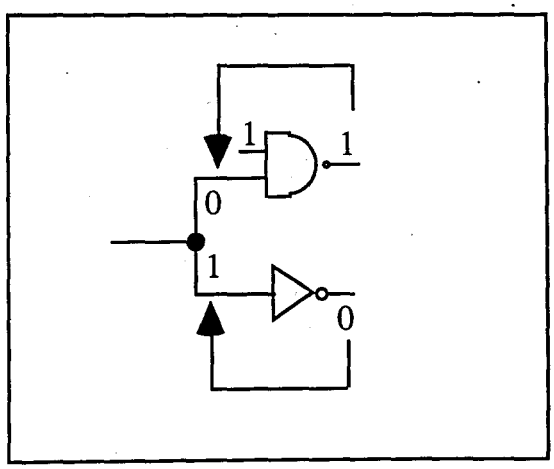


FIGURE 2.18. Conflicting Requirements on the Same Stem

On the other hand to minimize the number of backtracks FAN algorithm backtraces a set of objectives rather than a single objective using a procedure called multiple-backtrace.

This procedure determines a final assignment decision on line k with value v_k to satisfy a subset of original objectives or to detect a situation which is not possible to satisfy a subset of objectives simultaneously. The later situation may occur only when different objectives are backtraced to the same stem with conflicting values at its fanout branches as in Figure 2.18.

An outline of FAN algorithm in a brief way will be given in this paragraph and more details will be covered in the implementation details chapter. If FAN can not immediately identify a SUCCESS or FAILURE state, it marks all the unjustified values of the bound lines as current objectives, together with the values needed for the D-drive operation through one gate from the D-frontier. From these objectives multiple-backtrace procedure determines an assignment for a stem or a head line to be tried next. The decision process is similar to the one used in PODEM. Experimental results presented in [6] show that FAN is more efficient than PODEM. Its increased speed is primarily caused by a significant reduction in backtracking.

Other algorithms, namely TOPS (Topological Search) [13] and FAST (Fault-oriented Algorithm for Sensitized-path Testing) [14] are extensions of the FAN algorithm and based on different head line concepts. Also SOCRATES (Structure-Oriented Cost-Reducing Automatic TESt pattern generation system) [15] is based on FAN algorithm and added the learning ability concepts to the FAN. SOCRATES is a testing system rather than an algorithm including the efficient fault simulation and collapsing tools.

Before concluding this chapter, we will mention briefly about selection criteria and cost functions which are appropriate for embedding into ATPG problems. The search process of any TG algorithm presented above involves decisions. First type of the decision, is to select one of the several unsolved problems existing at certain stage in the execution of the algorithm. Second type is, to choose one possible way to solve the selected problem. Improvements achieved in selection criteria will directly affect the speed of any algorithm. Basic principles are, First, among different unsolved problems, first attack the most difficult one. This principle guarantees to avoid useless time spent in solving easier problems when a harder one cannot be solved. Second, among different solutions of a problem, first try the easiest one. Selection criteria differ mainly by the cost functions they use to measure "difficulty." Typically, cost functions are of two types, namely controllability measures and observability measures. Further details can be found in [4] about selection criteria and cost functions.

3. IMPLEMENTATION OF ATPG TOOL AND PERFORMANCE COMPARISON

3.1. System Description

The ATPG tool implemented with this thesis work and the similar tools developed in Virginia Polytechnic & State University are plugged to each other via a user interface software. They all form a powerful test generation and fault simulation system for both combinational and sequential circuits as stated in previous sections.

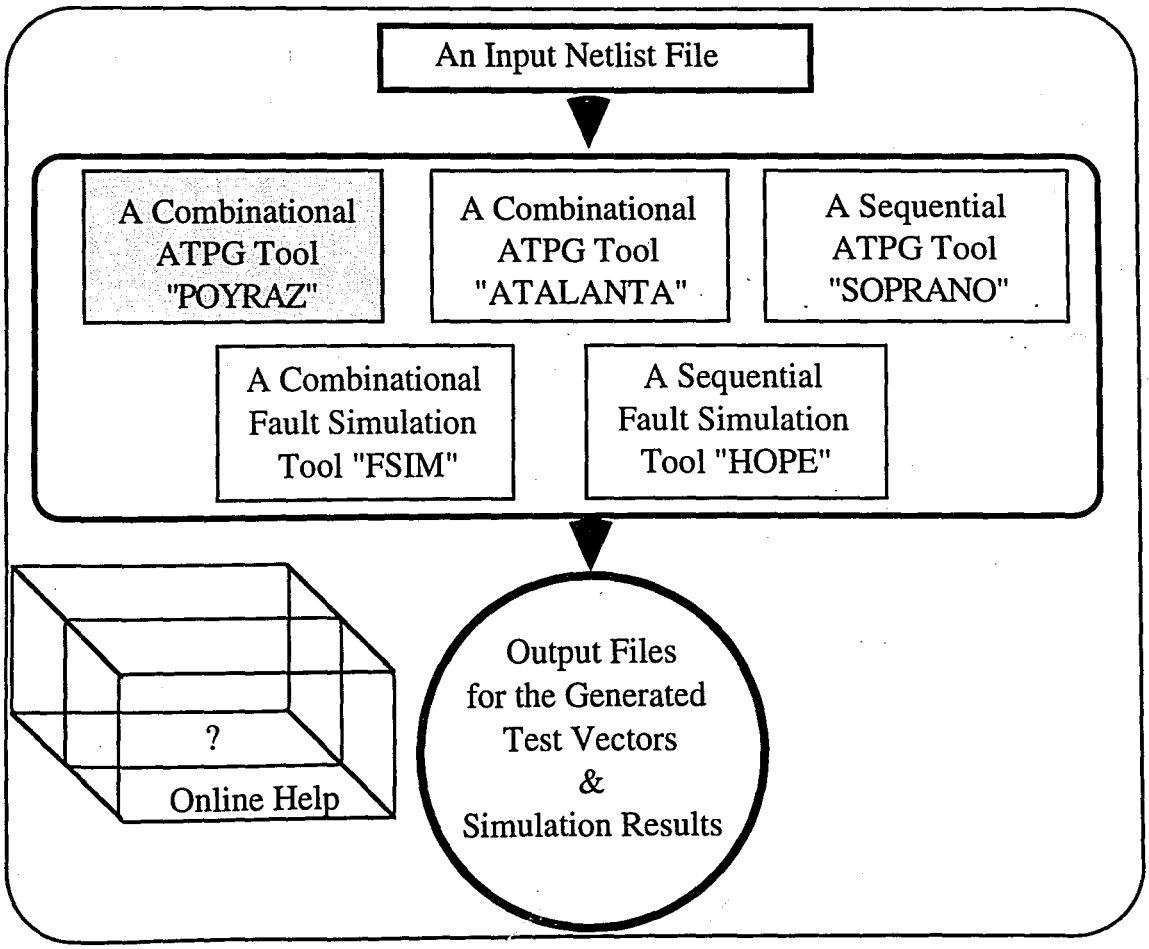


FIGURE 3.1. Block diagram of Complete Test System.

In Figure 3.1. a block diagram of complete test system is presented. The shaded sub-block describes the tool that is implemented with this thesis work. Its implementation details and performance achieved will be presented in this chapter. Shaded sub block of Figure 3.1. can be detailed as in Figure 3.2.(a).

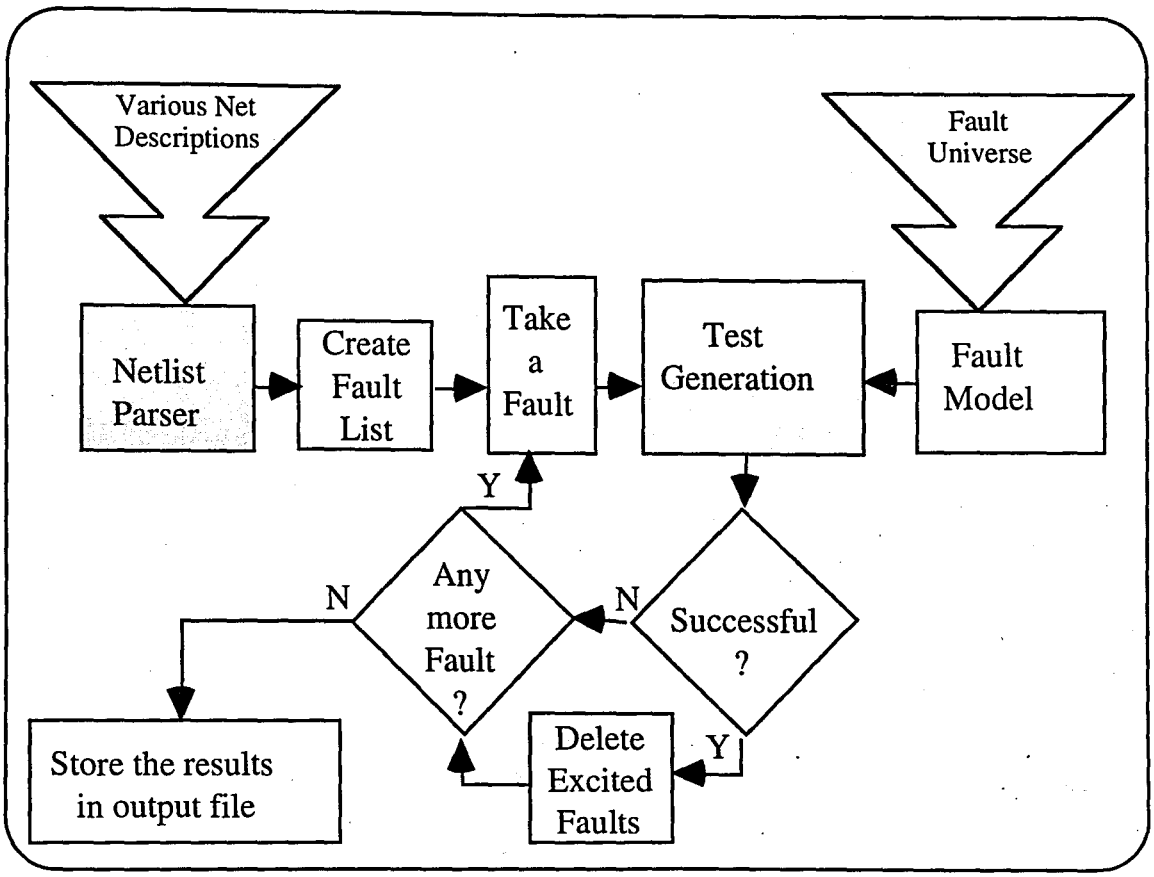


FIGURE 3.2.(a). Block diagram of our ATPG tool "POYRAZ."

In principle, netlist parser, test generation, and main sequence handler modules are the basic building blocks of this software package. In the following sections, these modules are going to be detailed. We will mainly concentrate on the test generation module but to some extent other two auxiliary modules will also be presented as much as it is necessary.

On the other hand, necessary data structures will be presented in this chapter if it is necessary for the description of the implementation, otherwise a complete description and listing for the whole data structures and variables used in this package will be detailed in Appendix-A. Also, a user guide for this package is given in Appendix-B. Figure 3.2.(b). presents a view from user interface of our tool.

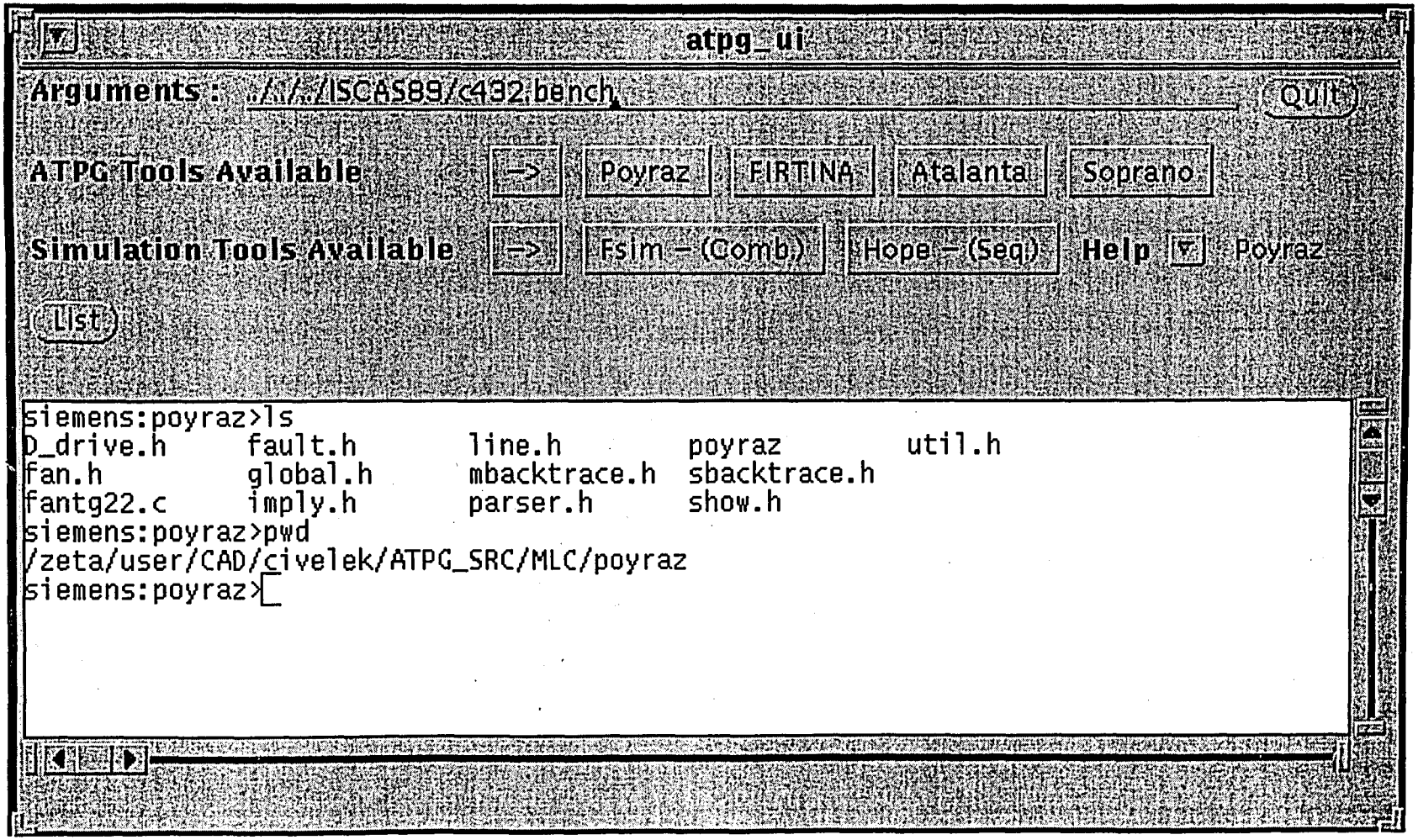


FIGURE 3.2.(b). View of our ATPG tool "POYRAZ."

3.1.1 Netlist File

As an input netlist, we use ISCAS89 netlist format which describes the circuit in the gate level. The netlist has to be flat but not hierarchical. The first line should be # followed by the name of the circuit. The lines beginning with # excluding the first line are comment lines and ignored. These comment lines may be put into any part of the netlist. It should be noted that the order of gates appearing in the netlist is not significant. The name of gates can be a string of alpha-numeric characters (0-9, A-Z or a-z). It is important not to use other characters, but especially "," and "_" characters will crash the execution of the program since they will cause the parser to fail, and create the data structures in the correct manner. An example netlist of the circuit c17 written in ISCAS89 netlist format is given in user manual. (Appendix-B).

Manageable gates are, primary input (PI), primary output (PO), AND, NAND, OR, NOR, XOR (2 input) and NOT. It is possible to add new gates or macros if necessary by adding their functional definitions into sub modules.

3.1.2 Source Files, Their Hierarchy and Compilation

We can sub-divide source codes into two pieces as the codes for the implementation of FAN algorithm, and on the other hand source codes for user interface. Second part of codes are out of our interest but for completeness they will be mentioned in hierarchical relations with the first part of the source codes. There are 12 separate code files in "c" for the implementation of our ATPG algorithm. This makes a total of 6205 lines of code, 356KBytes and approximately 81KBytes of executable binary code. Just for comparison with similar tools, namely atalanta's binary code is 106KBytes long. Additionally, 500 lines of source code is written for user interface using the XVIEW libraries which is a standard for the open windows operating systems.

A modular coding strategy is followed during the implementation and future expansions are always observed and modular interfaces are left possible with the current source code. Such as it is possible to integrate a preprocessor for being able to reading inputs from other standard netlist generators, it is also possible to embed a fault simulator, a fault collapser and/or a vector compacting tools as they are implemented. To manage this

flexibility, different jobs are assigned to different source files which communicate with each other via parameters and/or global variables (for main data structures like gate or node lists).

Figure 3.3. summarizes the all source files, their hierarchical relations to each other and also their interactions with the user interface.

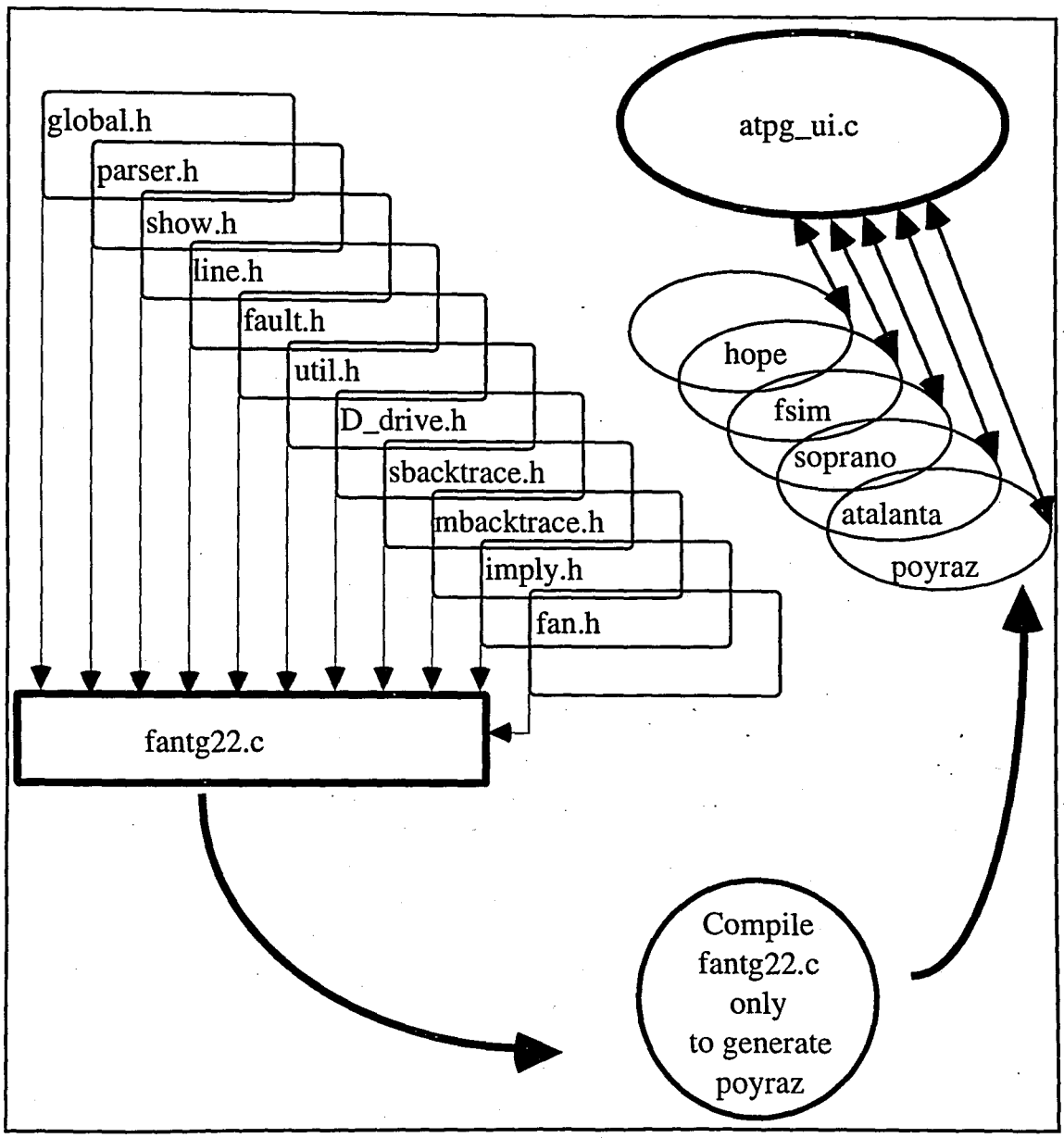


FIGURE 3.3. Block diagram of Hierarchical Relations Between Source Codes

fantg22.c is a frame that handles the sequences of the process or better to say it is the main routine of "poyraz". The rest of the files which are connected to this main file with "h" extension are responsible from different jobs that are not related to each other and they are

included at the beginning of the main file with the given order as if they were header files in fact they are not. That is why we compile only fantg22.c into "poyraz" but care must be taken about the rest of the included source codes have to be in the same directory during compilation process. Once you have compiled our tool, the resultant file with name poyraz can be called by user interface executable file named atpg_ui, with this specific name if they are in the same directory or the environment variables are set accordingly from any specific location. The simplest usage is possible if the user interface binary file, other binaries for each tool and the on-line help manual gathers in a specific location, and the each user's specific shell points this location from any working directory which contains the input files and will contain the output files after execution.

TABLE 3.1. Source Files and their Responsibilities

File Name	Responsible From
global.h	Contains the all global data structures and variables.
parser.h	Lexically parses the input file and fills the data str.
show.h	Generates the output files, prints the messages.
line.h	Makes the line classification and node search.
fault.h	Creates the fault list.
util.h	All operative utility operations, computations,...
D_drive.h	D-drive operation and its implications.
sbacktrace.h	Single backtrace calculations for every request.
mbacktrace.h	Multiple backtrace operation, uses sbacktrace.
imply.h	Imply_and_check operation.
fan.h	Implementation of FAN algo. Uses all above routines
fantg22.c	Main sequence handler of the complete tool.

Table 3.1. briefly states the responsibilities of all specific subroutines. Their independence from each other makes it easy to handle or expand the abilities of the tool developed.

3.1.3 Output Files

There are three kinds of output files. The first one has the extension of "analyze," which contains the aborted faults and the reason of failing, second one has the extension "test_mlc," which contains the test vectors generated and the simulated fault free output levels, third one has the extension of stimulus which contains the generated test vectors in a format that is compatible stimulus generator hardware but this file is generated if the number of PIs is less than 33 since the generator is 32 bits.

The output file formats are given in Appendix-B including one example from each class.

3.1.4 Data Structures

Before deciding on any approach about the data structures, primary goals are set such as; there must not be limits like the number of gates, number of fan-in or fan-out. These are the basic limits which may cause the whole system to be useless at sometime. These undefined limits and the dynamic nature of VLSI implementations of today pushed us to use dynamic data structures which are limited only with the physical data storage limits of the computing systems. On the other hand, FAN algorithm's list processing type procedures are very convenient for linear lists. For this reason linked lists were chosen as the main data structures of our implementation.

The basic idea of a linked list is that each component within the structure includes a pointer indicating where the next component can be found. Therefore, relative order of the components can easily be changed, simply by altering the pointers. In addition, individual components can easily be added or deleted, again by altering the pointers. As a result, a linked data structure is not confined to some maximum number of components. (In our application to some maximum number of gates for example.) Rather, the data structure can expand or contract in size as required.

A gate from a netlist file can have N_i inputs and M_j outputs. Also these N_i and M_j number of inputs and outputs differ from gate to gate. In addition, total number of gates is another variable from circuit to circuit. This three dimensional variability is represented by a special user defined data structure named "gatestr." First of all, "gatestr" is a linked-list data

type declaration. If we declare a gate pointer variable of this type, we can construct a linked list of gates which describes the partially device under test. Notice that we managed to describe which nodes are going/coming to/from which gate. This is an incomplete definition yet. The mentioned data type is formally written and given in Figure 3.4.(a).

```
typedef struct gatestr{
char            user_name[80];    /*Name of the gate*/
function        fnc;              /*Function of the gate*/
int             index;            /*Numeral index of the gate*/
int             nin;              /*Number of inputs*/
int             nout;             /*Number of outputs*/
boolean         marked;           /*Choose as D_drive if false*/
struct levelstr *head_in_val;     /*Pointer of head of inputs*/
struct levelstr *head_out_val;    /*Pointer of head of outputs*/
struct gatestr  *next;            /*Where next is, null if last*/
};
```

FIGURE 3.4.(a) Formal Decleration of "gatestr" Type Data Structure

Notice that some members of "gatestr" are plain, but some of them are not. Since the number of inputs and outputs is variable, this complex structure is not avoidable. Figure 3.4.(b) presents the "gatestr" more clearly. User defined data type gatestr keeps track of all necessary topological information of each gate in the netlist. In this data structure none of the members are modified during the whole process of the TG, except that the member "marked." (If marked is boolean true than in the rest of the search process that gate is not chosen as a D-drive gate, in order to prevent the algorithm from a probable failure.)

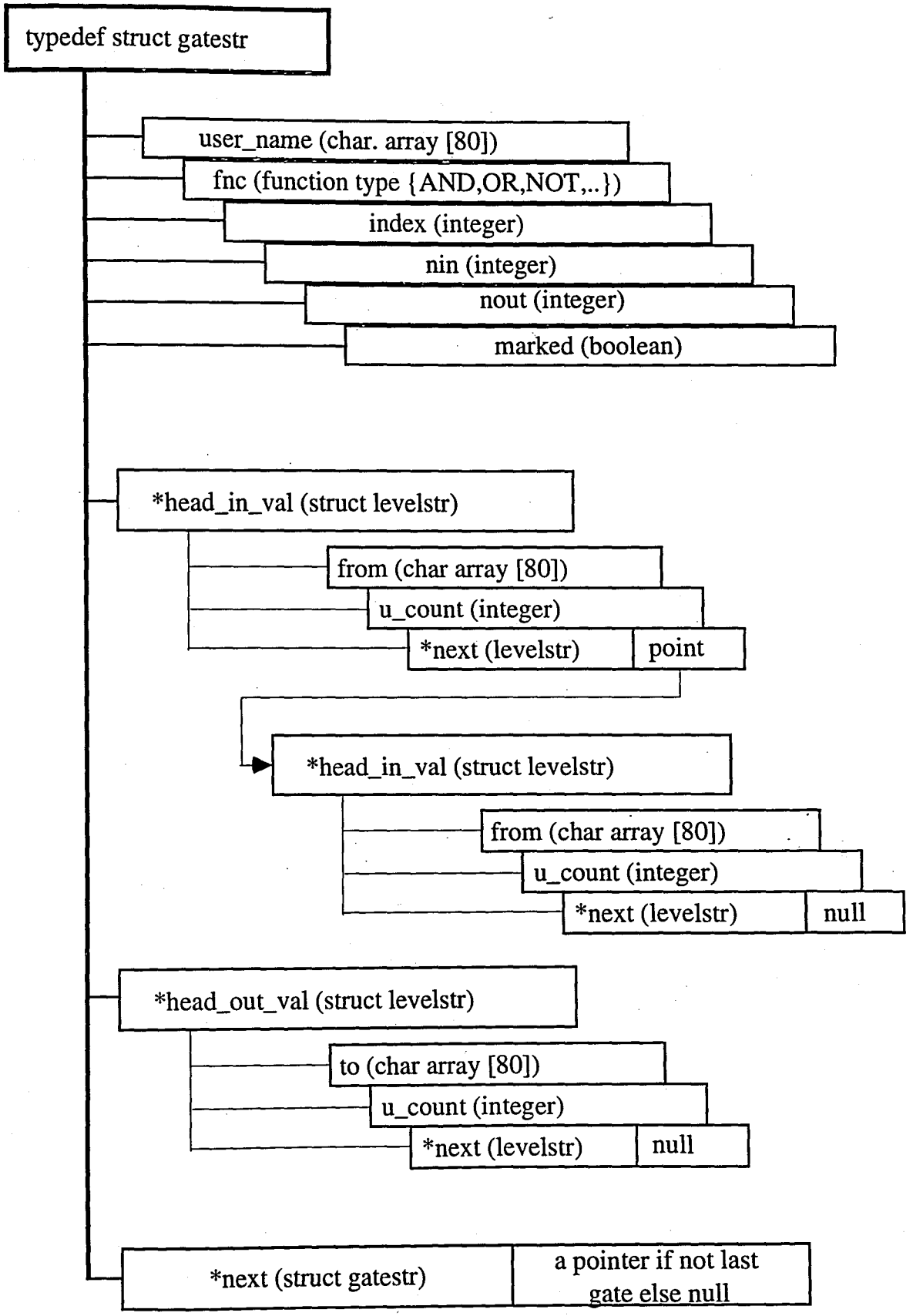


FIGURE 3.4.(b) "gatestr" Data Structure

On the other hand, all topological information stored in this structure is kept intact during all the processes, and referred whenever it is necessary, such as to determine which line is connected to which gate, which gate has how many inputs and outputs?, and so on.

Another basic user defined data type is "linestr". These type of variables contain the necessary topological information for each node of the circuit and the refreshable information which concurrently changes after each decision made, assignment taken or backtracking from our last decision made. This data type enables us to simulate our actions while searching the test vectors in the solution space that the algorithm is directed towards. Its formal definition is presented in Figure 3.5. and the visualization of its picture left to the reader which will able him/her to understand these both common data types given with "gatestr" and "linestr."

```
typedef struct linestr{
        char                user_name[80] ;
        level               value;
        int                 index;
        int                 user_count;
        char                source [80];
        char                connected_to [80];
        struct destinastr   *head_destina;
        faultype            fault ;
        line_place          lplace;
        line_type           ltype;
        boolean             justflag;
        struct linestr      *next;
};
```

FIGURE 3.5. Formal Decleration of "linestr" Type Data Structure

The other data types and variables used in our TG system is listed in Appendix-A. Generally all linked lists are created with the explained approach with Figure 3.5. and their head pointers are stored as global variables. As the end effect, this flexible and dynamic memory allocation approach enabled our implementation independent from the number of gates, number of fan-in and/or fan-out like limits. This property is a superiority with respect to other tools which are developed in Virginia Tech. & State University.

So far, we have described the system globally, as stated at the beginning of this chapter we can subdivide the complete system into three modules as, netlist parser, main sequence handler, and test generation modules. In the following section, these modules are going to be detailed.

3.2. Implementation Details of Complete ATPG System

The "fantg22.c" file is the file that performs scheduling of the events. Scheduling of the events is performed in a manner that is presented in Figure3.6. Main modules presented in Figure 3.6. will be briefly described, but the heart of the ATPG tool "Test Generation" module will be presented in detail in the following sub sections.

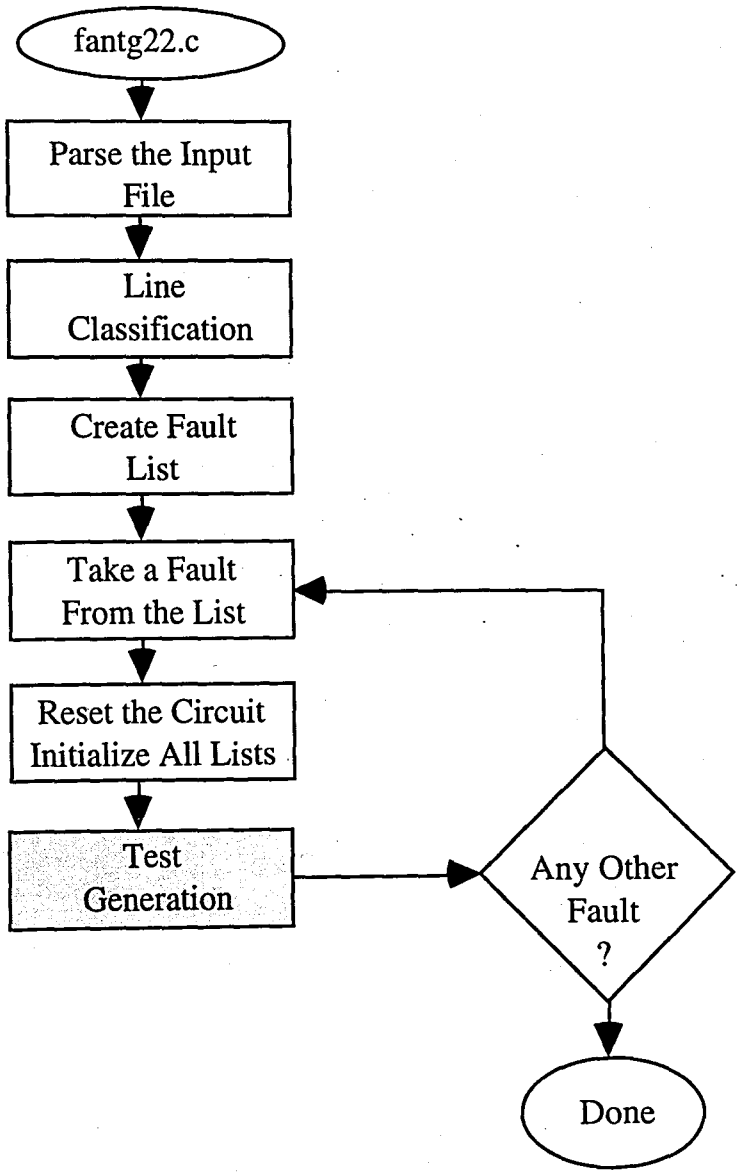


FIGURE 3.6. Sequences of the Processes in General

3.2.1 Parser

Described input netlist file in ISCAS89 format is lexically analyzed and main data structures of the system described in the previous sections are filled up. At the end of this module, from gates to nodes and from nodes to gates of connectivity information of the complete circuit under test is captured. In addition, line values are set to undefined state.

3.2.2 Line Classification

Line list which is formed of "linestr" user defined data type is exhaustively analyzed and the lines are classified as head, bound, or free lines with respect to the definitions which are given in [4] and in the second chapter of this thesis. A two pass method is followed for hundred percent correct classification of lines.

3.2.3 Fault List Creation

File "fault.h" is responsible for the creation of the fault list automatically. There are different methods for creating the fault lists each trying to maximize the fault coverage. In our implementation, we generated two faults for each node (s-a-0, s-a-1) This approach creates the maximum number of faults in the list which is possible for that netlist. There are discussions on the coverage ability of chek-points which are the PI's and the all fanout points in the netlist. These points are the arguments of fault collapsing business which is completely out of the scope of this thesis. Since we have created the maximum number of faults, it caused a big problem at run time, especially for big benchmark circuits and test generation turned into a cumbersome process which has thousands of faults to analyze and a couple of days of run-time. In addition, since we do not make vector compaction, big test sets are generated, which is not desirable. However, these problems are manageable as we progress in the field. For example, adding a fault collapsing routine into the "fault.h " file

will tremendously decrement the number of faults to be analyzed. But, at the moment, we must not forget that this solution is a method if we want fault detection only, if the demands are in the way that, the fault location like diagnostics are being requested, than we can not use this shortcut.

3.2.4 Test Generation

After the policy is decided for creating a fault list that is likely to serve our demands, the process on the queue is generating tests for these faults one by one. The TG process is independent from the generation method of the fault list since it is a routine that you pass the parameters of faults in process one by one for each fault from your list. It makes no difference from the viewpoint of TG process. Every time we call the TG subroutine which is coded in file "fan.h" it returns a SUCCESSFUL or FAILURE. Depending on what it returns, the results are written into two different output files. If the operation succeeded, current PI values and evaluated fault free PO values are recorded into the test output file; if this is not the case, whole non-X node values of the circuit and from which point on a failure is detected is dumped into the analyze output file. After this general introduction to TG module, let us continue with its detailed description.

3.2.5 FAN Algorithm Implementation

We have implemented a recursive version of the original FAN algorithm which is given in [4] with some minor modifications, which are going to be stated as we proceeded. Figure 3.7. outlines the basic flow of our implementation of FAN algorithm.

Every time we call this subroutine, initially we have two elements in the assignment queue. These are the basic or starting objectives independent from the algorithm chosen. One of them is the propagation of the fault to a PO and the other is the backtrace of the opposite state of the erroneous value in process. These queue elements are in the form of (*l,value,direction) where *l isthe pointer of line l, value is the state to be propagated and the direction is the direction of propagation from the set {forward,backward}.

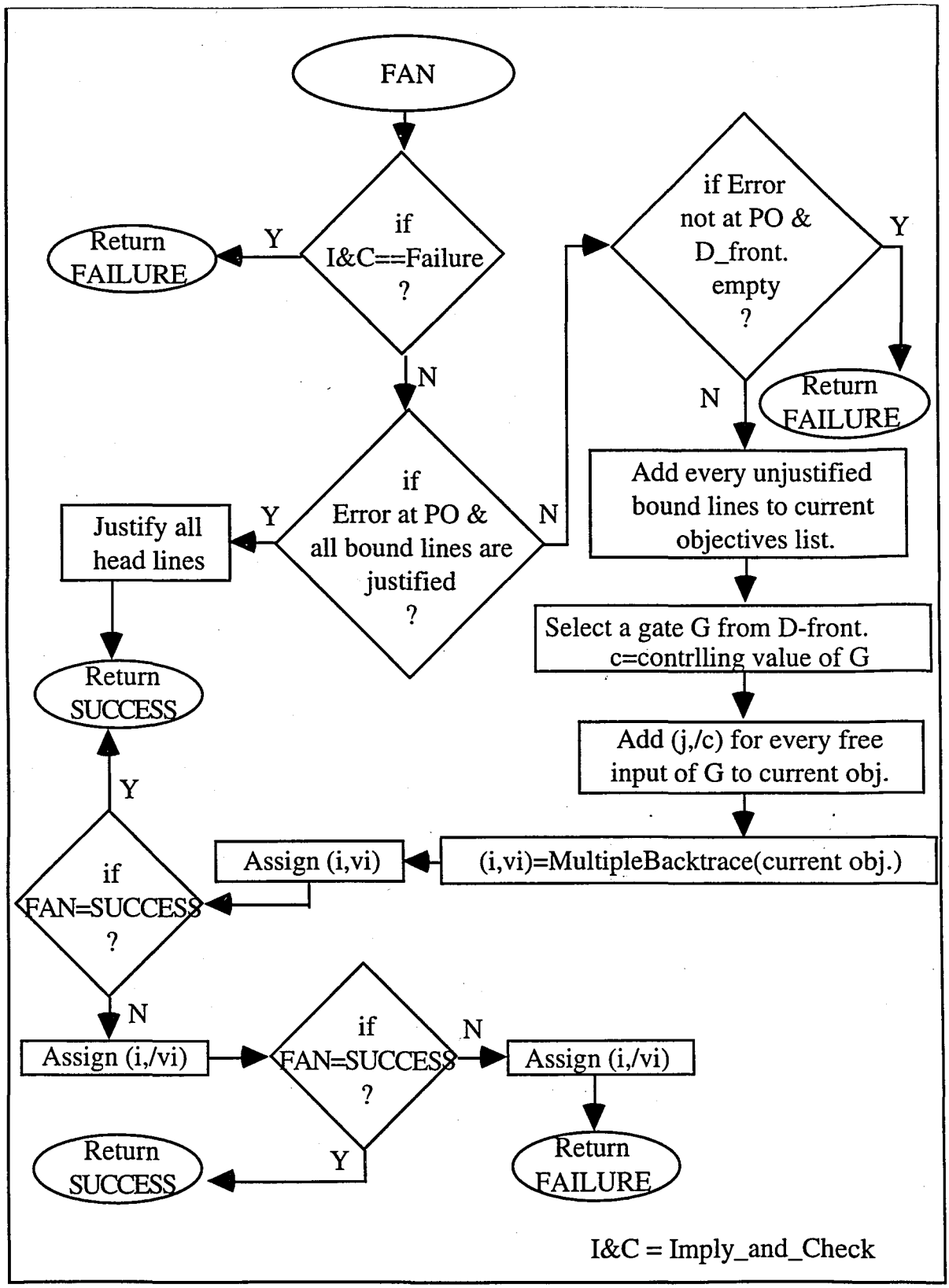


FIGURE 3.7. FAN Algorithm

If FAN can not immediately identify a SUCCESS or a FAILURE state, it marks all the unjustified values of bound lines as current objectives, together with the values needed for the D-drive operation through one gate from D-frontier. From these objectives, Multiple backtrace subroutine Mbacktrace determines an assignment for a stem or for a head line to be tried next. Here, assignment does not mean that assigning found value v_i to the decided line i directly but means that adding these parameters to the assignment queue and then calling FAN subroutine recursively until a SUCCESS or FAILURE is determined.

Whenever a SUCCESS is achieved, the justification of head lines are started as a final step of the algorithm. Recall that the lines in and between the headlines and PIs are free lines and they are fanout-free. Such lines can be justified, with no conflict, easily.

When the subroutine FAN is called, with the values on the assignment queue it first of all calls the subroutine Imply_and_Check. This is the implication process of the algorithm. The tasks of this process are, first, computing all the values that can be uniquely determined by implication [4],[6], second, checking the consistency of values which are requested and assigning them if they are consistent, third, maintaining the D-frontier and the J-frontier lists. We can view the implication process as a modified zero-delay simulation procedure. As in simulation, we begin with some values to be assigned to some lines. These values may determine (imply) new values to be assigned, and so on, until no more new values are generated. All values are retrieved from queue in turn and processed as stated. Unlike the simulation where values only propagate in forward direction or better to say towards the primary outputs, in our routine values, can also propagate to backward (to PIs).

To generate a test for the fault l s-a-1, the initial two entries in the assignment queue are (l,0,backward) and (l,/D,forward). These initial objectives in turn are retrieved by imply_and _check subroutine, the value requested to be assigned on line l, is first checked with the previous value of the line l for consistency, An inconsistent situation is reported if current value of line is not equal to undefined state (X) or requested or current values are not equal (except the faulty line). A consistent value is assigned and is further processed according to its direction.

Imply_and_Check subroutine is one of the basic modules of the TG process and its quality in implementation directly alters the quality of the complete work. On the other hand, there is no clear and complete description of this module in the literature. In the following sections, the algorithmic descriptions of Imply_and_Check, Mbacktrace will be given and a heuristic approach will be presented about choosing a gate from D-frontier for comprehending the FAN algorithm completely.

3.2.6 Algorithmic Description of Imply_and_Check

The purpose of imply_and_check is to do forward and backward implication. If the action parameter is imply, we do forward implication, if it is justify we do backward implication. Imply_and_check was called when a line value was either decided upon or implied by existing values. First thing we do is add this value to the circuit and the implied list by calling add_to_imply_list. Since this is an assignment, the action parameter to add_to_imply_list is imply. In forward implication, we compute the value of the gate with the added input (line,val). If the result is x, or it agrees with the current output value which is not x, we do nothing. If the current output is x and the new value is not x, we add the new value to the implied list, we update the line value in the line_list. We then continue with forward implication with a recursive call to imply_and_check. If the new value and the current value are both not x and disagree, we detect a conflict (this can only happen when the output line is on the j_frontier). If during forward implication, faults are either propagated or canceled, we update the d_frontier. In backward implication, we check to see if we need a controlling or non-controlling value at the input. If we need a controlling value, if more than one input is free, a decision needs to be done as to which input to set; therefore, we add the line to the j_frontier and to the implied list as a justify action. If the source gate of the implied line has erroneous inputs, it is taken off the d-frontier (since a gate is on the d-front only when its output is x and it has erroneous inputs). If we need a non-controlling value, we check to see whether one of the inputs has a fault. If not, we set all free inputs to the non-controlling value (if there's a fault on an input then setting all free inputs will propagate the fault and will clash with our required value which is either 0 or 1). For every free input we set, we recursively call imply_and_check to continue backward implication.

Algorithmically;

0. Add (line, value) with imply to imply_list (necessary for the backtrack operation)
1. If action == justify
    1.1 If source gate is a primary input, return success
    1.2 If source gate is a buffer
        1.2.1. Continue backward implication on buffer input
        1.2.2. If successful do forward implication on all other output branches, otherwise return fail
        1.2.3. If all forward implications succeed, return success, otherwise return fail.
    1.3 If the source gate has a faulty input, remove the gate from the d-frontier.
    1.4 Find required input value

1.5 If required value is the gate controlling value

1.5.1. If more than 1 input is free

1.4.1.1. Add (line,val) to the j_frontier

1.4.1.2. Add (line,val,justify) to imply_list

1.4.1.3. Return(success)

1.5.2. Set the free input to the controlling value

1.5.3. Call imply_and_check with the input line, controlling value and justify, and return result.

1.6. If one of the inputs has a fault, return fail

1.7. For each free input, call imply_and_check. If all succeed, return success, otherwise (at least 1 failed) return failure

2. action == imply

2.1. If the gate is a primary output, return success

2.2. If the gate is a buffer

2.2.1. Call imply_and_check for all output branches

2.2.2. If all return success, return success, else return fail

2.3. Compute new output value

2.4. If it is x

2.4.1. If the new input was a fault and there are no other input faults, add the gate to the d_frontier.

2.4.2. If the output is specified, and only one input is free, its value must be the controlling value of the gate. Return imply_and_check on this line with the implied value and justify.

2.4.3. Return success.

2.5. The current output is not x

2.5.1. If current output and implied output agree, return success

2.5.2. If current output is a fault, it means the output line is the fault site. therefore if implied value is equal to the excitation value, i.e. 1-fault, return success.

2.6. If current output is x

2.6.1. If the gate was on the d_frontier, take it off

2.6.2. Call imply_and_check with output line and new value and return result

2.7. Return fail (both new and current values are not x and they disagree)

3.2.7 Description of Multiple Backtrace Subroutine

Mbacktrace processes in turn every current objective until the set of current objectives is exhausted. Objectives generated for the head lines reached during this process are stored in the set Head_Objectives. Similarly, the set Stem_Objectives stores the stems reached by backtracing. After all current objectives have been traced, the highest-level stem from stem-objectives is analyzed. Selecting the highest level stem guarantees that all the objectives that could depend on this stem have been backtraced. If the stem k has been reached with conflicting values (and if k can not propagate the effect of the target fault), then Mbacktrace returns the objective (k,v_k) where v_k is the most requested value for k. Otherwise, backtracing is restarted from k. If no stems have been reached, then Mbacktrace returns an entry from the set of Head_Objectives.

A nice and complete algorithmic description of the Mbacktrace subroutine is given in [4], [6].

After giving the formal definition of Mbacktrace procedure it will be beneficial to highlight some points of the procedure again in a more qualitative manner. Each objective arriving at a fan-out point stops its backtracing while there exist other current objectives. After the set of current objectives becomes empty, a fan-out point objective closest to a primary output is taken out, if one exists. If the fan-out point objective satisfies the following condition, the objective becomes the final objective in the backtrace procedure. The condition is that the fan-out point k is not reachable from the fault line and requirements of multiple backtrace for that specific node are conflicting with each other. That is why we need to define an objective, which will be used in multiple backtrace procedure, by a triple parameter set; In order to keep the track of conflicting requirements.

$$(s, n0(s), n1(s)) \tag{3.1}$$

Where, s is an objective line, n0 (s) is the number of times the objective logic level 0 is required at s and n1(s) is the number of times the objective logic level 1 is required at s. For example (s,1,0) declares that line s has an initial objective required to set 0 once. Figure 3.8. gives an example circuit in order to clarify this new objective representation.

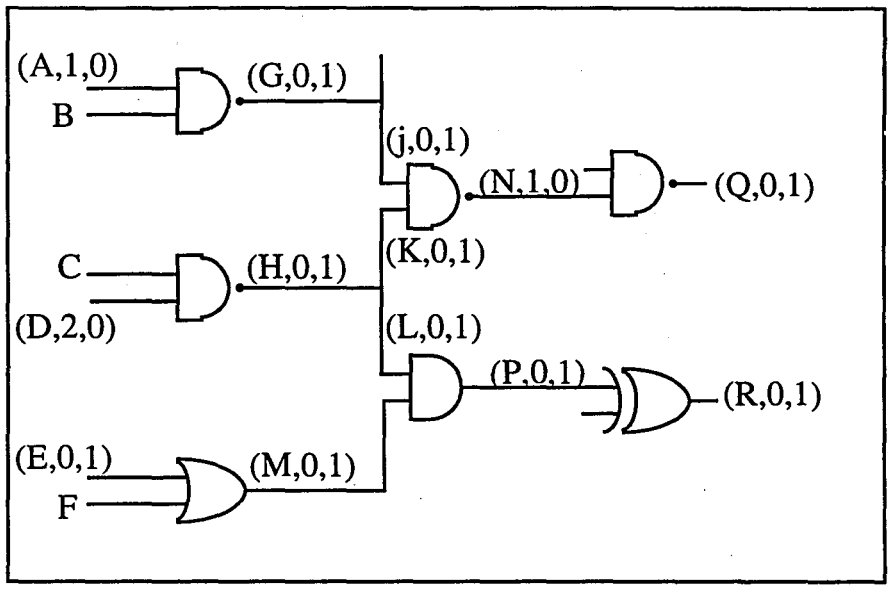


FIGURE 3.8. Computation of N0 & N1

3.2.8 A Heuristic Approach For Decision Processes

In Figure 3.7., a decision process of choosing a gate from D-frontier is considered. There are various metrics on this decision that are published in the literature [4]. They all are based on observability measures and simply to find a gate which is closer to a PO and they seek for at least one X-path from the output of the gate to any PO. In our implementation, a very simple method is used which satisfies both of the metrics (closer to PO and at least one X-path) but with no additional computation. We just observe the last generated D-frontier gate to choose if there are no limitations on that gate such as having been marked as a probable failure cause in the future steps or if one of its inputs has a controlling value which prevents error propagation and so on. Since it is just generated as the last D-frontier, it is more likely to be closer to any PO.

3.3. Results Achieved with Benchmark Circuits

ISCAS89 standard benchmark circuits are used to evaluate the performance analysis of the tool developed with this thesis work. Three circuits which are small, observable and having hand generated netlists are used for the verification of the coding.

3.3.1. Verification of Coding

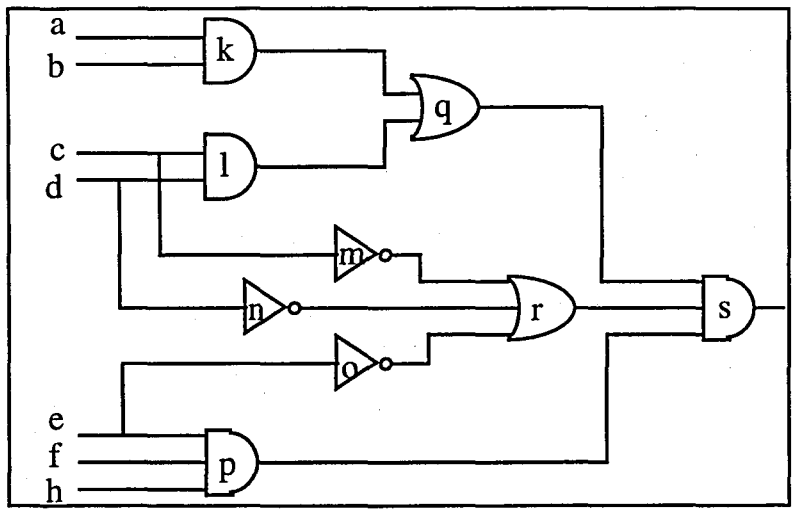


FIGURE 3.9.(a) Example Circuit for the Verification of Coding (ex63.bench)

In Figure 3.9.(a) an exercise circuit from reference [4] is given. This circuit is small enough to be observed at a glance and complex enough to present many aspects of TG problem. For example consider the question, is it possible to detect the fault "o s-a-0". The answer is obviously no, because this fault is a redundant one and redundant faults can not be detected. (to excite it we must set o=1 => e=0 => p=0 => s=0). But what happens if there are two faults even though we assume the single stuck, such as "o s-a-0" and "p s-a-1". Under the condition of single stuck, the fault on line "o" was redundant. If it happens that the fault "p s-a-1" happens at the same time, then, fault on line "o" becomes a detectable one with the vector (1111011 -> s=D). On the other hand most of the vectors detecting many of the faults requires that e=1, f=1, and h=1. This scheme is an interesting one, if you want to add any learning like skills into the algorithm, which enables you to start the search space from a reasonable point. In the following pages the output of "POYRAZ" and "ATALANTA" is partially presented in Figures 3.9.(a) and (b), for the circuit "ex63.bench" given in Figure 3.9.(a)

```
**************************************************************
*INITIALIZATION OF DATA STRUCTURES BUILDING THE LISTS       *
Checking Input File Existence      :     O.K.
Reading Input File                 :     O.K.
Creating Data Structures           :     O.K.
Classifying Lines                  :     O.K.
Creating Fault List                :     O.K.
STARTING TG PROCESS   (Resultant Statistics for file : ex63.bench)
Total # of Gates     :    17
# of Inputs         :     7
# of Outputs        :     1
# of faults generated     32
# of faults erased   :    12
# of patterns generated:  16
# of aborted faults  :     4
                TG PROCESS FINISHED
**************************************************************
Generated Test Vectors & Output Values Are in File:[ ex63.test_mlc ]
Failed Trials & Their  Failing Reasons Are in File  :[ ex63.analyze ]
Vectors  in  Stimulus Generator Format Are in File  :[ ex63.stimulus ]
PI:   (a b c d e f h)     PO                 :(s)        fault-free:   (s)
      (1 1 X 0 1 1 1)     # Output Value(s)  :(D)                     (1)
      (0 1 X 0 1 1 1)     # Output Value(s)  :(DBAR)                  (0)
      (1 1 X 0 1 1 1)     # Output Value(s)  :(D)                     (1)
      (1 0 X 0 1 1 1)     # Output Value(s)  :(DBAR)                  (0)
      (1 1 1 1 1 1 1)     # Output Value(s)  :(D)                     (1)
      (1 1 1 1 0 1 1)     # Output Value(s)  :(DBAR)                  (0)
      (1 1 1 1 1 1 1)     # Output Value(s)  :(D)                     (1)
      (1 1 1 1 1 0 1)     # Output Value(s)  :(DBAR)                  (0)
      (1 1 1 1 1 1 1)     # Output Value(s)  :(D)                     (1)
      (1 1 1 1 1 1 0)     # Output Value(s)  :(DBAR)                  (0)
      (0 X 1 0 1 1 1)     # Output Value(s)  :(D)                     (1)
      (X 0 0 0 1 1 1)     # Output Value(s)  :(DBAR)                  (0)
      (X X 0 X X 1 1)     # Output Value(s)  :(D)                     (1)
      (X X 1 X X 1 1)     # Output Value(s)  :(DBAR)                  (0)
      (X X X 0 X 1 1)     # Output Value(s)  :(D)                     (1)
      (X X X 1 X 1 1)     # Output Value(s)  :(DBAR)                  (0)
```

FIGURE 3.9.(b) Output of POYRAZ for the Circuit "ex63.bench"

Welcome to atalanta (version 1.1)
Copyright (C) 1991,
Virginia Polytechnic Institute & State University

SUMMARY OF TEST PATTERN GENERATION RESULTS

1. Circuit structure

Name of the circuit	*: ex_6.3*
Number of gates	*:16*
Number of primary inputs	*: 7*
Number of primary outputs	*: 1*
Depth of the circuit	*: 4*

2. ATPG parameters

Test pattern generation mode	*: RPT + DTPG + TC*
Limit of random patterns (packets)	*: 16*
Backtrack limit	*: 10*
Initial random number generator seed	*: 853614670*
Test pattern compaction mode	*: REVERSE + SHUFFLE*

3. Test pattern generation results

Number of test patterns before compaction	*:13*
Number of test patterns after compaction	*:8*
Fault coverage	*: 91.304 %*
Number of collapsed faults	*: 23*
Number of identified redundant faults	*: 2*
Number of aborted faults	*: 0*
Total number of backtrackings	*: 2*

4. Memory used *: 2307 Kbytes*

5.CPU time: Init.: 0.017s./Fault simulation:0.000s./FAN:0.000 s./Total:0.017 s.

** Primary inputs :*	*a b c d e f h*	*Primary outputs:*	*s*
	1: 1110110		*0*
	2: 1110111		*1*
	3: 1101101		*0*
	4: 1101111		*1*
	5: 1111111		*0*
	6: 1000111		*0*
	7: 1101011		*0*
	8: 0010111		*0*
	9: 0101111		*0*

FIGURE 3.9.(c) Output of ATALANTA for the Circuit "ex63.bench"

From the analysis of data present on preceeding pages, for the circuit in Figure 3.9.(a), ATPG tool Poyraz generated 32 faults without fault collapsing (which is the maximum possible fault count) and generated a total of 16 test vectors for a total of 28 faults. The difference of 12 faults are covered by fault dropping, which is a post process of our implementation. We may call it post fault collapsing, which is, whenever you find a test vector for a specific fault, its error propagation path is excited automatically. We are seeking such situations and dropping that kind of self excitements from our fault list, which improves both the CPU time, fault coverage and reduces the number of test vectors generated (Also similar to vector compaction.) In addition, a total of 4 errors are aborted due to redundancy or trial limit (100 trials are allowed) like reasons. With these data, fault coverage percentage achieved for this case is 87.5 percent. ATPG tool Atalanta generated a total of 23 faults for this circuit after making the fault collapsing and generated a total of 13 test vectors using first a random TG then a deterministic TG procedure and by vector compaction it reduced the number of vectors to 8. It achieved a fault coverage of 91.304 percent. Even though there is a big deal of system and ability differences between Poyraz and Atalanta, by taking these differences into account, we can make them comparable at least to some extent, even for performance analysis and the experimental verification of coding done with in the scope of this thesis. From the output data given in foregoing pages, it is possible to check the input vectors from both of the tools and their fault free outputs.

After this qualitative interpretation of the data presented as output of both tools, let us continue with some metric definitions which will help us while we are trying to compare these both unequal (by capabilities) tools.

Our metrics are, fault coverage ability of TG tool, cost efficacy factor, and coverage efficacy product. Their definitions are given in the below equations. The equation (3.2) is a common metric in this field and it directly effects the yield and a good definition of the quality of the test generation system. The other equations (3.3) and (3.4) are given in order to make a more or less equal comparison while comparing our tool and the tool which is developed in Virginia Tech. & State University. And to the best of our knowledge, they are not common or used in the literature. Cost efficacy is bounded from the left, because for the worst case it can be 1, that is a different vector per fault in the list is produced. But it is not bounded from the right as the quality increases, the number of patterns generated decreases which also increases the cost efficacy factor. Except for one case that is when the number of produced vectors is zero which means no solution for the TG problem. This case is excluded from our definition. This eradication forces the cost efficacy factor to be equal to the number of faults in the fault list for the best case, which means a unique vector detects all the faults in the fault list which is used.

$$\text{Fault Coverage} = \frac{\text{\# of covered faults}}{\text{total \# of faults on the fault list}} \text{ X } 100 \qquad (3.2)$$

$$\text{Cost Efficacy Factor (CEF)} = \frac{\text{\# of detected faults}}{\text{\# of patterns generated}} \qquad (3.3)$$

$$\text{Coverage Efficacy Product (CEP)} = \text{Fault Coverage X CEF} \qquad (3.4)$$

After the definitions of above discussed metrics, we can summarize the performance achieved by two different ATPG tools for the circuit given in Figure 3.9.(a)

In the Table 3.2, 16+12 means, 16 excited and 12 erased faults exist with the tool Poyraz and with tool Atalanta, 13->8 means, generated 13 vectors and it is decreased to 8 after vector compaction process, 2+0 means 2 faults found redundant and no error is aborted.

TABLE 3.2. Performance Summary for the Example Circuit ex63.bench

Circuit ex63.bench	Poyraz	Atalanta
# of gates	9	9
# of faults	32	23
# of covered faults	16+12	21
# of aborted faults	4	2+0
# of patterns generated	16	13->8
fault coverage	87.5%	91.3%
cost efficacy	32/16=2	23/13=1.76
coverage efficacy product	175	161.5

Another example circuit is presented in Figure 3.10. for the verification purpose of our tool, this circuit is a comparator that is finding the relations between its input vectors whether they are equal, greater or less than from each other.

We can summarize the performance achieved for the circuit in Figure 3.10 with the Table 3.3.

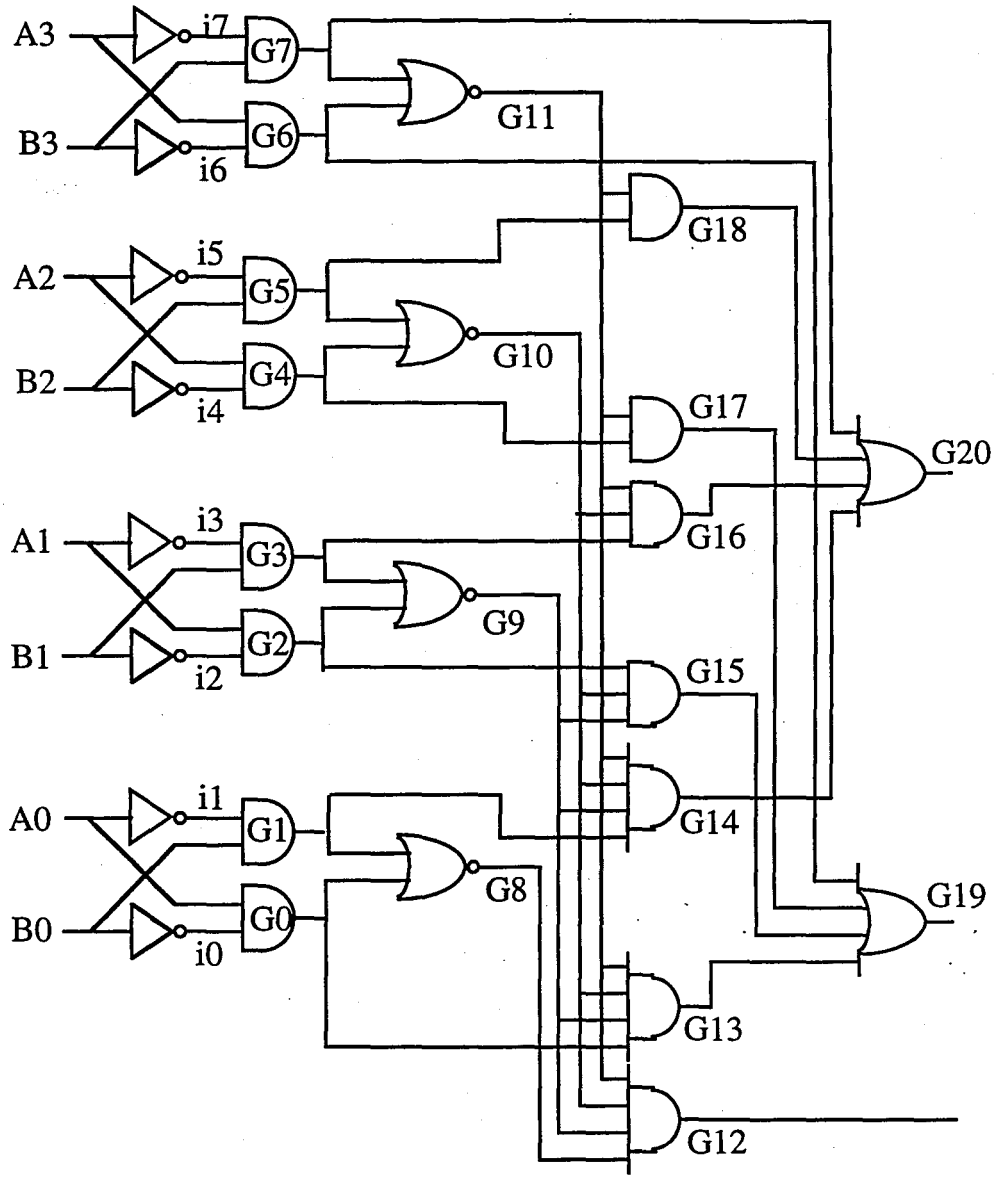


FIGURE 3.10. .Example Circuit for the Verification of Coding (magcomp.bench)

TABLE 3.3. Performance Summary for the Example Circuit magcomp.bench

Circuit magcomp.bench	Poyraz	Atalanta
# of gates	37	37
# of faults	74	98
# of covered faults	16+38	98
# of aborted faults	20	0+0
# of patterns generated	16	24->19
fault coverage	72.9%	100%
cost efficacy	74/16=4.6	98/24=4
coverage efficacy product	335	408

As a last example for the verification of coding, a 1 bit static adder is presented in Figure 3.11.

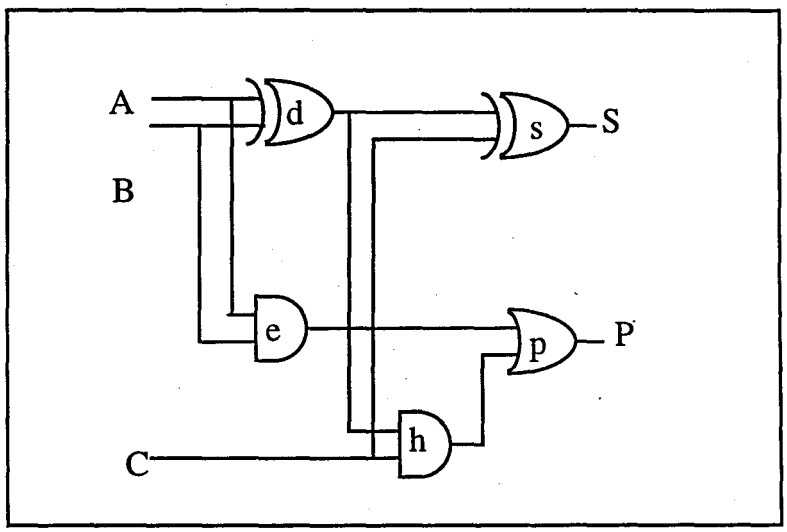


FIGURE 3.11. .Example Circuit for the Verification of Coding (1badd.bench)

And its performance results are given in Table 3.4.

TABLE 3.4. Performance Summary for the Example Circuit 1badd.bench

Circuit 1badd.bench	Poyraz	Atalanta
# of gates	8	8
# of faults	16	26
# of covered faults	6+10	26
# of aborted faults	0	0+0
# of patterns generated	6	6->5
fault coverage	100%	100%
cost efficacy	16/6=2.6	26/6=4.3
coverage efficacy product	266.6	433.3

3.3.2. Performance Comparison

We have used larger circuits to test the performance of our tool and used various ISCAS89 benchmark circuits. The performance of Atalanta was evaluated with the same circuits also. From the view point of fault coverage, our tool and Atalanta got the results which are presented in Table 3.5.

TABLE 3.5. Fault Coverage Summary

Benchmark Circuits	**Poyraz**	**Atalanta**
1badd.bench	%100	%100
ex63.bench	%87,5	%91,3
c17.bench	%72,7	%100
mcomp.bench	%72,9	%100
c27.bench	%64,7	%97,3
c432.bench	%78,3	%99,2
c880.bench	%81,4	%100
c1908.bench	%79,3	%99,5

This great difference of fault coverage percentages is mainly due to the following reasons. First Atalanta is performing a random pattern generation prior to any deterministic attempt. Second, Atalanta makes an efficient fault collapsing and fault simulation processes which greatly improve the performance of the system. Our tool is just solving TG problem and is not cocerned with any other integral process as in the Atalanta's case. Same benchmark results are presented in a more illustrative manner in Figure 3.12., as fault coverage percentage with respect to Atalanta's performance figure.

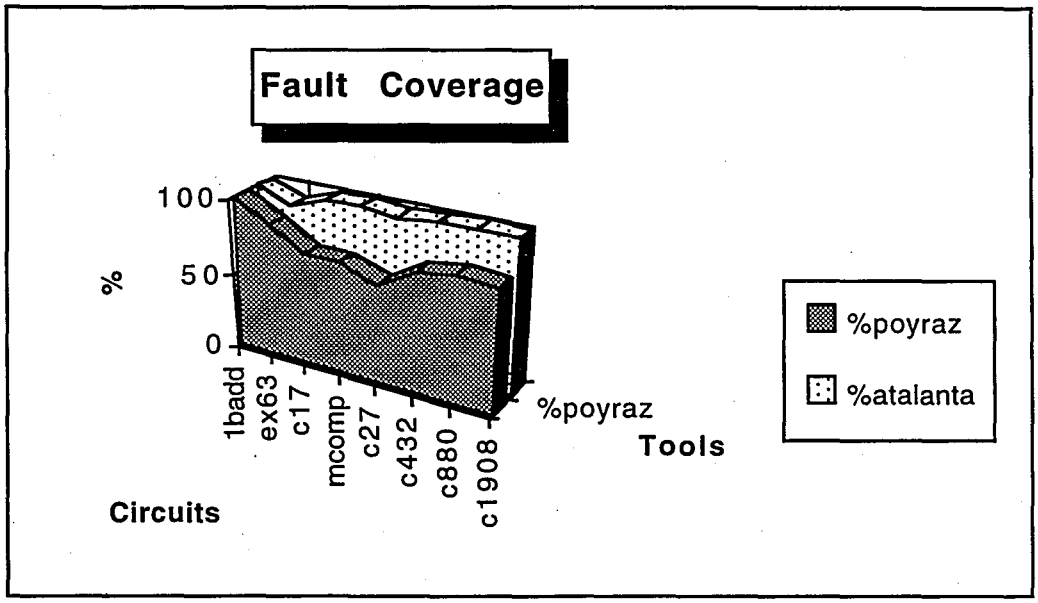


FIGURE 3.12. Fault Coverage Comparison (# of gates increasing from left to right)

On the other hand the factor CEF (Cost Efficacy Factor) that we defined with Equation (3.3) is another important metric to compare the performance of an ATPG tool because to shrink the test set as much as possible and increase the number of faults detected with a small set of test vectors are strongly desired specialties of a good ATPG system. From this point we have achieved the results given in Figure 3.13.We have seen that with small circuits we are achieving at least as good as Atalanta's, but as the circuits grows Atalanta software performs five times better than our tool but this performance figure is due to vector compaction process of Atalanta. (Consider the definition of CEF given in Equation (3.3)). CEF factor is a good but incomplete metric to evaluate an ATPG system, because it is possible to achieve high CEF's with low fault coverages. To prevent these kinds of mistakes we have defined CEP which is given by Equation (3.4). Using this metric, one can evaluate any TG system simply, including both the fault coverage and cost efficacy. Achieving a relatively higher score from this product metric, can easily be interpreted as a superior one to the other having a lower score.

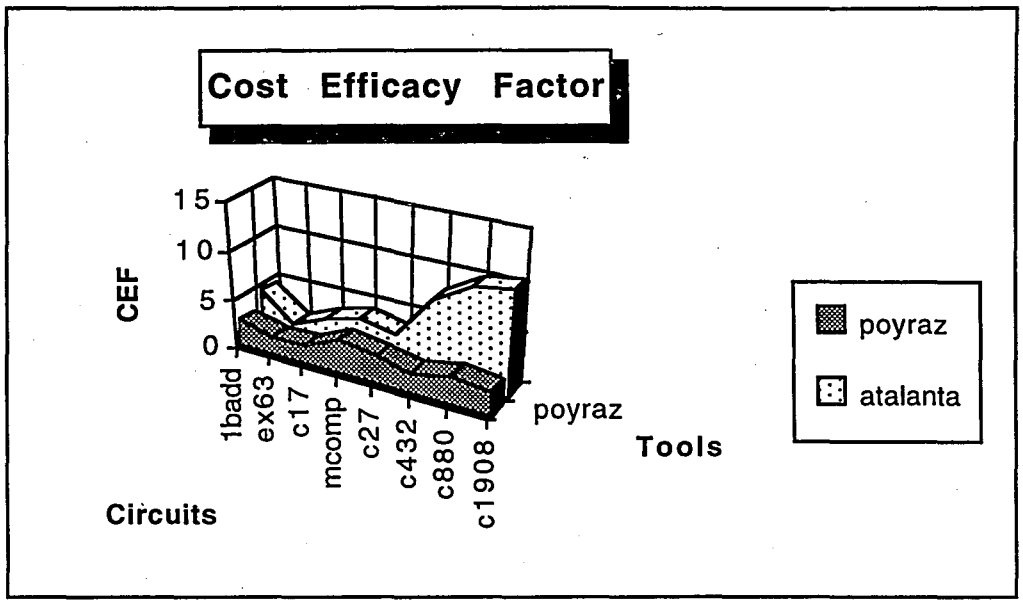


FIGURE 3.13. CEF results with respect to Atalanta (# of gates increasing from left to right)

In Figure 3.14. we have given CEP performance metric values of both systems. Obviously Atalanta software is superior to the one that we have implemented. But as we implement the integral parts of our ATPG system such as fault simulator, fault collapser and vector compaction utilities, Poyraz software will at least be comparable to Atalanta software. Notice that with small circuits where the effect of fault collapsing and/or vector compaction is not a dominating factor, our software is comparable to the other tool.

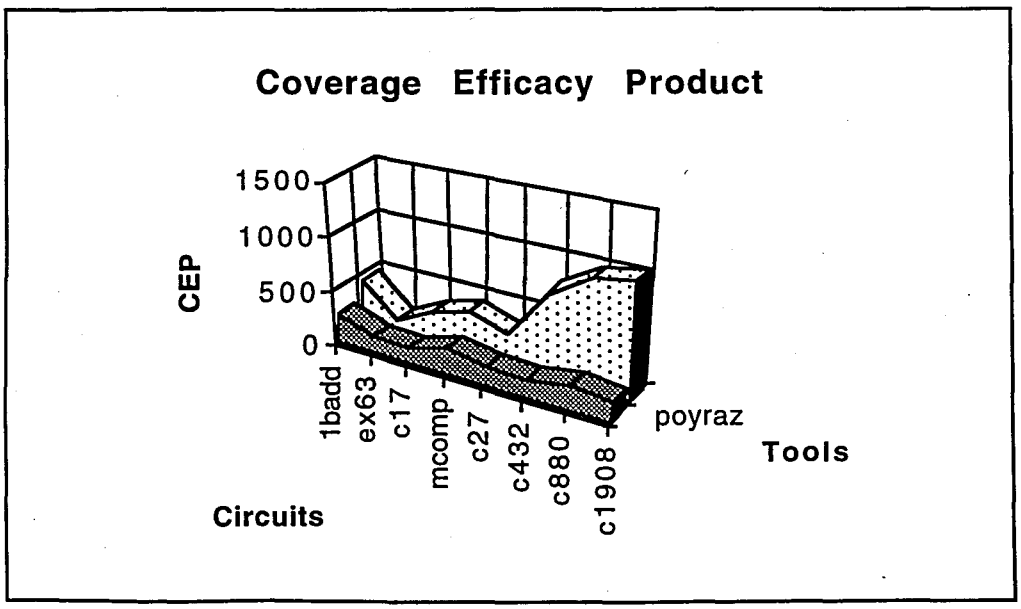


FIGURE 3.14. CEP results with respect to Atalanta (# of gates increasing from left to right)

4. IMPLEMENTATION AND HARDWARE DETAILS OF STIMULUS PATTERN GENERATOR

4.1. System Specifications and Description

The most important specification of a stimulus generator, is the very high clock rates, which is required by digital IC's of today that are going to be tested. While generating predetermined test stimuli, test vector generator (called stimulus generator) must be at least as fast as the device under test. In order to implement such a fast generator at reasonable costs some special techniques have to be used to achieve a good trade-off between the cost of implementation and the required specifications. In this thesis, at the beginning of design of such a system, we have set some specifications, which are capability of running up to 100MHz. programmable clock frequencies, 32 bit wide output driving and 32 Kbyte. depth of storage capacity, capable of interfacing a 68000 based control computer, which has an RS 232 interface to any host computer.

From this outline, it is possible to visualize a block diagram of a complete system and separate it in to sub-blocks as;

- The control computer,
- Interleaved fast RAM banks,
- Fast address generator block, and
- Programmable PLL clock frequency synthesizer.

Interleaved fast ram banks is the result of compromise between the cost and the specifications of a very fast clock rate. We could not choose, a non-interleaved ram system, which will ease the design tremendously but on the other hand will also increment the cost of the system tremendously.

After this brief introduction, in order to design and implement such a stimulus generator as an integral part of our ATPG tool, we have proposed a system which is outlined in Figure 4.1.

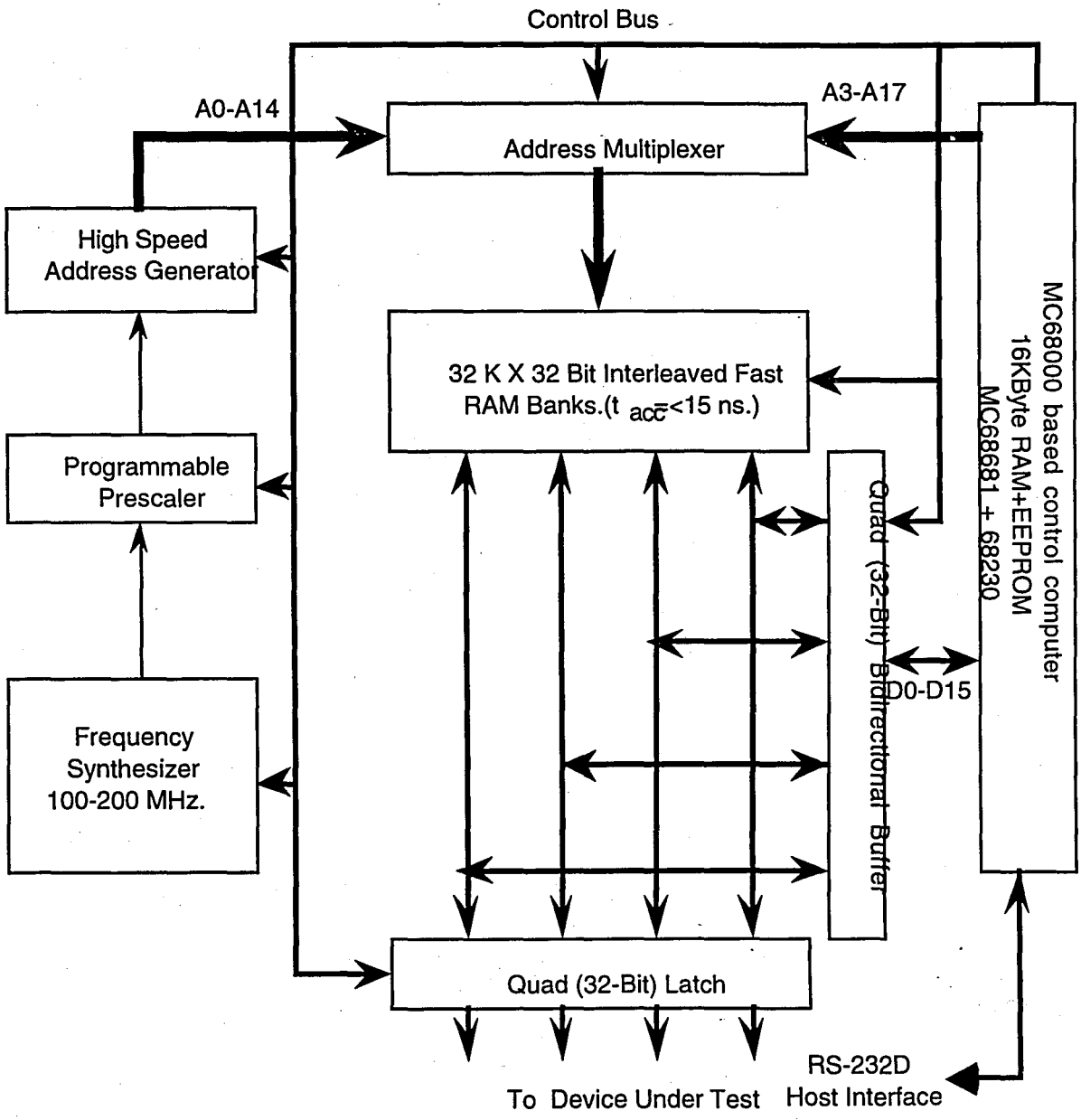


Figure 4.1. Proposed Block Diagram of Our Stimulus Generator

In the following sections of this chapter, we will present a step by step design and implementation of our hardware, by dividing the system into sub-modules listed above

4.1.1. Control Computer

The control computer is a 68000 based control computer which has a 8KWord storage capacity as RAM and the same amount of storage capacity as ROM. It has various interfaces via 68681 DUART and 68230 PIT peripheral integrated circuits. The address decoding and *dtack* signal generation logic is implemented by a programmable array logic device 16V8. A monitor software is running on the control computer, for required minimum operations like communicating with a host computer, downloading the application software and for debugging it and so on.

The control computer has two main interfaces with the rest of our the application hardware. One of them is via DUART 68681's parallel output port, which we call from now on as control bus and the other is its standard (direct) bus interface to the extension memory board which holds the fast ram banks on it. The control computer directly addresses the fast RAM bank region, whenever it is necessary to write or read operation required from that region. The extension module interfaces to the control computer from its bus, obeying the bus arbitration rules of the 68000 processor, as if it were an direct peripheral of the processor. For example it produces its own *dtack* signal, and or it with the original *dtack* signal of the processor's peripherals. On the other hand, control bus interface handles the sequences of the operations by an eight bit control byte. These signals are the mode control signal "S" which determines the operation mode of the application hardware. Three bits of serial communication bus of PLL frequency generator which are "Clk, Data, Load," start to count signal "Count", synchronous reset signal "Sr",for the address generator counters, and finally divide by control of selective multiplexer control signals of "S1 and S2"

4.1.2. Fast Ram Banks

The control computer assumes control while the signal "S" is logic low. At this mode of operation control computer can access the RAM banks, writes or reads them via its bus interface directly. When the signal "S" is logic high, the processor leaves the control completely to the application board which generates its 15 bit addresses automatically and bursts the RAMs contents of those addresses generated. At this mode of operation RAMs are accessed in the control of addresses generated since the all rams are enabled at the same time.

The burst of the data in a parallel scheme from 64 bits of memory is latched once at the rising edge of the clock rate chosen by the programmable and dividable PLL clock generator. These latched 64 bits of the data are fed to output at the falling and the rising edges of the chosen clock rate but each time from a different bank, causing an interleaved operation. We have achieved a doubled clock frequency data flow rate, using still half the speed of the data flow accessible RAMs. In other words, by using two banks of RAMs and accessing them in turn in one clock cycle, we are doubling the throughput of data flow. The detailed schematic diagram of fast RAM banks and their control logic is given in detail in Appendix-C.

4.1.3. Fast Address Generator Block

The fast address generator is performed by two eight bit fast counters which are capable of running up to 125 MHz. clock rates. When the mode is "S=high" the counters assume the generation of addresses as 15 bits starting from zero up to 32Kbytes of address space. As long as they are kept in an enabled position they count upwards ultimately and generate the necessary addresses for the RAMs in order to achieve burst mode of operation. When the "S=low " mode is entered they go into high impedance state and leave the control of address bus to processor which is achieved by bi-directional bus transceivers as illustrated in Figure 4.2.

4.1.4. Programmable PLL Clock Frequency Synthesizer

We have used a state of the art semi ASIC (Application Specific Integrated Circuit) as a high frequency programmable PLL clock generator (MC12429). It is a general purpose synthesized clock source targeting applications that require both serial and parallel interfaces. Internally it is capable of running 400 to 800 MHz. The differential PECL (Positive Emitter Coupled Logic) is configurable for a desired ratio as 2,4,8 or 16. Very few discrete elements are necessary to configure the PLL loop since all the necessary loop elements are integrated. (Even VCO and low pass loop filter.)

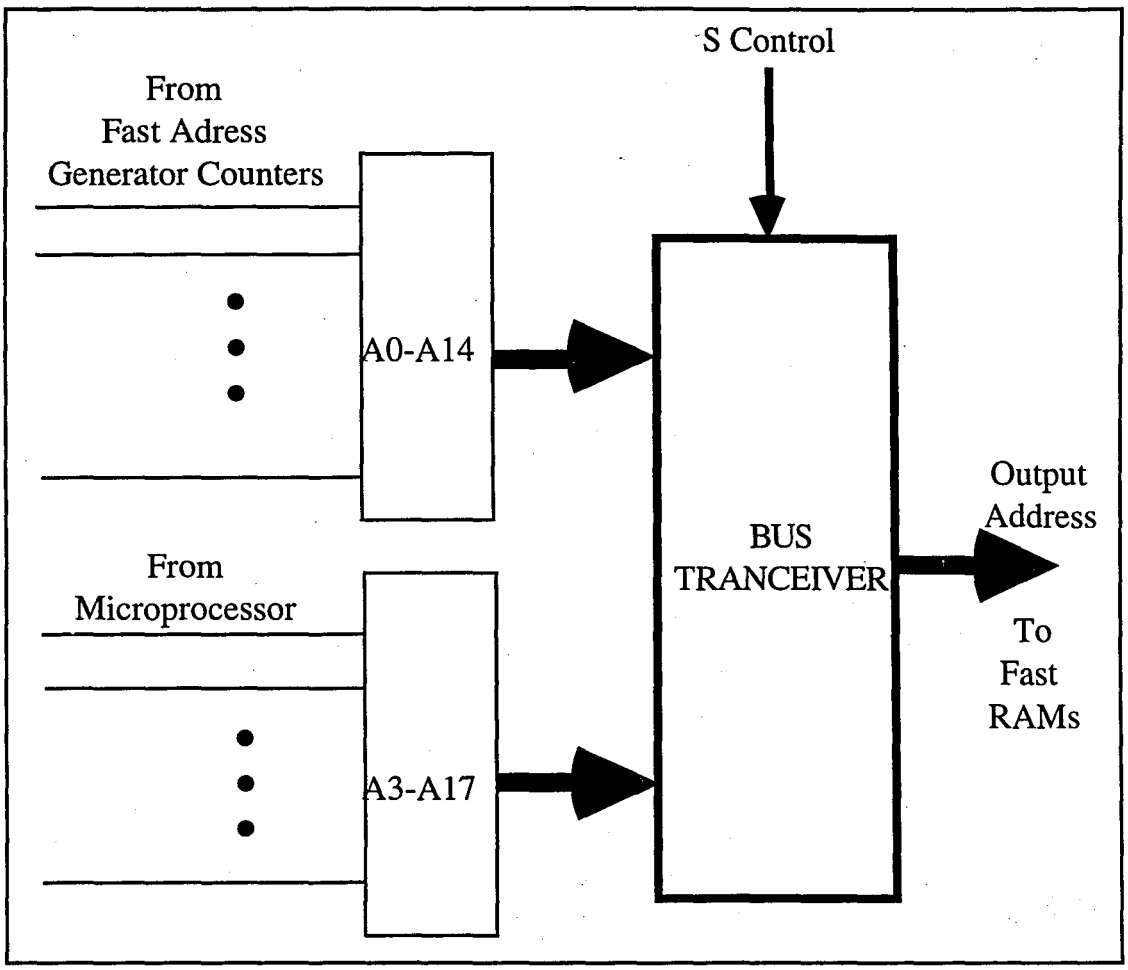


FIGURE 4.2. Address Generator Counters Operation Principle

Since the outputs of MC12429 are PECL, we have to convert this output to TTL because the rest of the circuitry is operating at TTL levels. This is achieved by a translator IC which is namely 10H10350. All the necessary schematic details are presented with the rest of the control circuitry in Appendix-C.

4.2. Software and Memory Management

4.2.1 Software

The application software is written in 68000 assembly language and have the basic functions of programming the PLL clock frequency generator and changing the control bit's state as it is necessary with respect to the operation mode of the application hardware.

At the moment it is still under construction and as we develop the code, downloading it from host computer via RS-232 interface

4.2.2 Memory Management

The Fast RAMs are located between the memory location (hexadecimal) 100000 and 13FFFF, occupying a 256Kbytes of address space.Their relative insertions can be visualized by the help of Figure 4.3.(a).

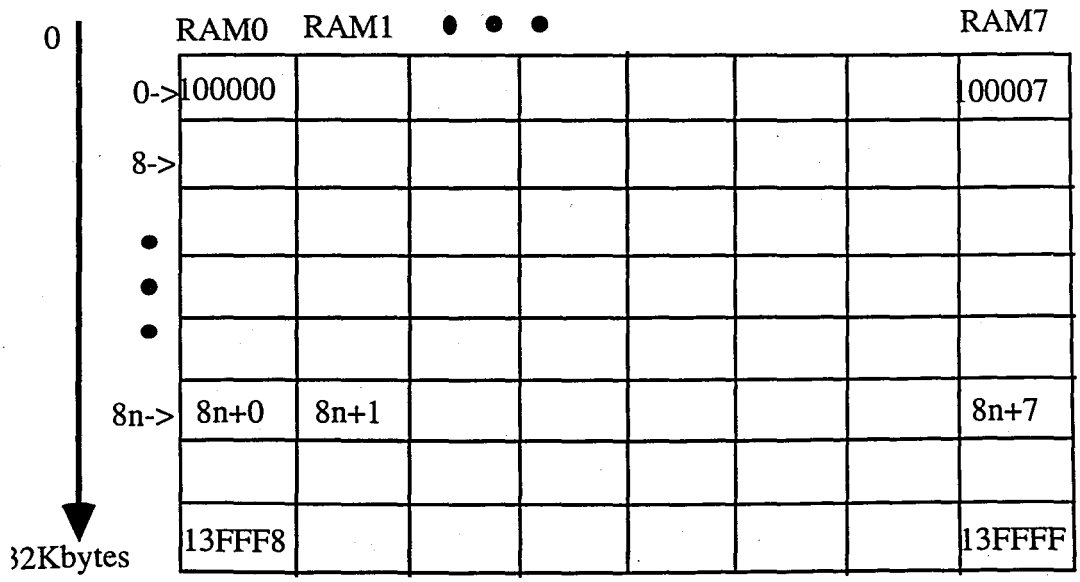


Figure 4.3.(a). Visual Presentation of Memory Organization

In Figure 4.4, we have presented a complete memory map of the whole system. In the figure, address allocations are presented not to scale but in the linear order. Also, the situation is showing the position of devices after the start up procedures are finished by the 68000 control computer. The difference before the completion of start up procedure is the RAMs and ROMs are interchanged the places shown in the Figure 4.3.(b).

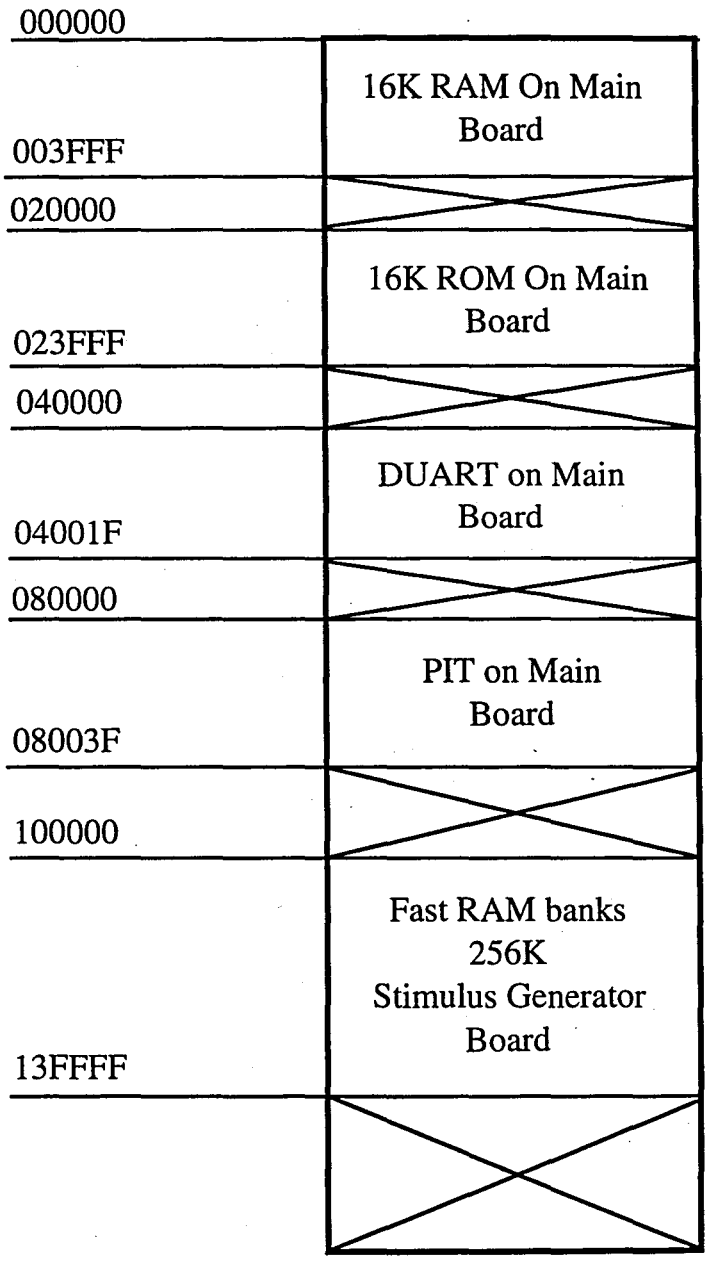


Figure 4.3.(b) Memory Map of Complete System

We have used programmable logic arrays (PAL) to implement the address decoding and direction control of the data within microprocessor system and the RAM extension board. Appendix-D details the PAL equations for both of these control functions.

5. CONCLUSION

Before concluding, we will present a summary of achieved results, mention the probable applications of this thesis and finally claim our plans on the future work.

5.1. Achieved Results

5.1.1. Results with Software Tool

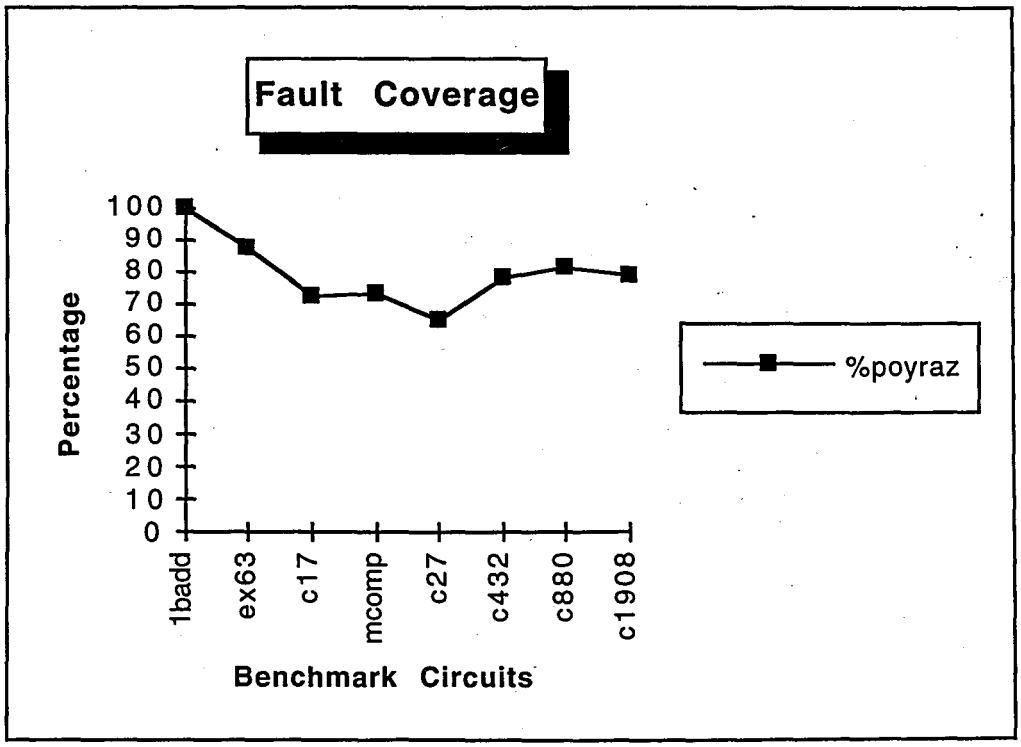


Figure 5.1. Performance as Fault Coverage (# of gates increasing from left to right)

With the software tool developed we have achieved such a performance figure presented in Figure 5.1. The fault coverage aspect of our implementation is not bad but also is not enough. This property will be developed with the development of additional tools like fault simulator, fault collapser and vector compactor rather than changing our implementation.

5.1.2. Results with Hardware Tool

Our stimulus generator performed well enough as a prototype, the aimed specifications are achieved, it is operating upto 100 MHz. clock rates. We are planning to develop it as a professional system by embedding the control computer and the extension card on a same board. Additionally, we are planning to develop a user interface software for the easy usage of it.

5.2. Future Work and Other Probable Applications

As stated through the thesis work we will develop additional tools to increase the performance figure of the ATPG tool Poyraz. These tools probably will be a fault simulator, fault collapser, vector compactor and/or a random test pattern generation tool. On the other hand as we grow in the field, we will try to use the learning schemes of artificial neural networks in ATPG field. It might be useful, to have an ATPG tool that learns from its mistakes.

Additionally, today ATPG techniques are being used in the field of "Low Power Design." The input set of vector search, which maximizes and/or minimizes the power consumption is a great center of attraction for us as a future work based on the knowledge that we achieved with this thesis.

APPENDIX - A

GLOBAL DATA STRUCTURES

```
#define EOL '\n'
typedef int boolean;
#define TRUE  1
#define FALSE 0
typedef int route;
#define FORWARD  1
#define BACKWARD 0
typedef int success;
#define SUCCESSFULL  1
#define FAILURE      0
typedef int level;
#define LOW          0
#define HIGH         1
#define X       2
#define D       3
#define DBAR         4
typedef int function;
#define NOT     0
#define AND     1
#define NAND         2
#define OR      3
#define NOR      4
#define XOR      5
#define XNOR     6
#define BUF      7
#define PI       8
#define PO       9
typedef int faultype;
#define SA0      0
#define SA1      1
#define NOTFAULTY  2
typedef int line_type;
#define FREE         0
#define HEAD         1
#define BOUND        2
typedef int line_place;
#define INPUT        1
#define CIRCUIT 0
#define OUTPUT       2
typedef int status;
#define NORMAL     1
#define FAULTY  0
typedef int objtype;
#define NONE_OBJ    0;
#define HEAD_OBJ    1;
#define FANOUT_OBJ  2;
#define STEM_OBJ    3;
#define FINAL_OBJ   4;
```

```
/*************************************************************************/
typedef struct destinastr
   {
    char to [80];
    struct destinastr *next;
   };
/*************************************************************************/
typedef struct levelstr
   {
    char from [80];
    int u_count;
    struct levelstr *next;
   };

/*************************************************************************/
typedef struct netliststr
   {
    char        readnetline [80] ;
    struct netliststr *next;
   };
/*************************************************************************/
typedef struct gatestr
   {
    char        user_name[80] ;
    function    fnc;
    int         index;
    int         nin;
    int         nout;
    struct levelstr      *head_in_val;
    struct levelstr      *head_out_val;
    boolean   marked;
    struct gatestr *next;
   };
/*************************************************************************/
typedef struct linestr
   {
    char                user_name[80] ;
    level          value;
    int         index;
    int          user_count;
    char                source [80];
    char          connected_to [80];
    struct destinastr *head_destina;
    faultype            fault ;
    line_place  lplace;
    line_type           ltype;
    boolean             justflag;
    struct linestr   *next;
   };
/*************************************************************************/
typedef struct faultstr
   {
    struct linestr *faulty_line;
    faultype  fault ;
    boolean   processed;
    struct faultstr *next;
   };
```

```
/*****************************************************************/
typedef struct assignmentquestr
  {
   struct linestr *line;
   level value;
   route direction;
   boolean   processed;
   int  fan_l;
   struct assignmentquestr *next;
  };
/*****************************************************************/
 typedef struct frontierstr
  {
   struct gatestr *gate;
   struct linestr *line;
   level value;
   boolean   marked;
   int  fan_l;
   struct frontierstr *next;
  };
/*****************************************************************/
typedef struct implystr
  {
   struct linestr *line;
   level val_bar; /* Requested new value to assign on the line */
   level val;    /* Previous value of the line, before this implication */
   int  fan_l;
   struct implystr *previous;
   struct implystr *next;
  };
/*****************************************************************/
 typedef struct objectivestr{
   struct linestr *line;
   level value;
   int n0;
   int n1;
   int  fan_l;
   objtype obj_type;
   boolean processed;
   int weight_observe;
   boolean valid;
   struct objectivestr *next;};
/*****************************************************************/
 typedef struct resultstr
  {
   struct linestr *line;
   level value;
   boolean valid;
  };
/*****************************************************************/
 typedef struct triedstr
  {
   boolean itself;
   boolean reverse;
  };
/*****************************************************************/
```

```
/* GLOBAL VARIABLE DECLERATIONS */
/* -------------------------- */
FILE *fpt;
FILE *testfpt;
FILE *analyzefpt;
FILE *stimulusfpt;
char s1[80];
struct netliststr *head_netlist, *tail_netlist;
struct linestr    *head_line_list, *tail_line_list;
struct gatestr    *head_gate_list, *tail_gate_list;
struct faultstr   *head_fault_list;
struct faultstr   *fault_in_process;
struct assignmentquestr   *head_assignmentque_list;
struct frontierstr   *head_dfrontier_list;
struct frontierstr   *head_jfrontier_list;
struct implystr      *head_implication_list;
struct objectivestr  *head_current_objective_list=NULL;
boolean from_stem = FALSE;
int total_gate_number = 0;
int total_line_number = 0;
int gate_index = 0;
int line_index = 0;
int fan_level  = 0;
int new_count = 0, old_count = 0;
int number_of_inverters  = 0;
int number_of_and_gates  = 0;
int number_of_nand_gates = 0;
int number_of_or_gates   = 0;
int number_of_nor_gates  = 0;
int number_of_xor_gates  = 0;
int number_of_xnor_gates = 0;
int number_of_buf_gates  = 0;
int number_of_pi_gates   = 0;
int number_of_po_gates   = 0;

int number_faults_in_fault_list  = 0;
int number_paterns_generated     = 0;
int number_aborted_faults        = 0;
int erased = 0;
/*****************************************************************/
```

APPENDIX - B

USER GUIDE FOR "POYRAZ" SOFTWARE ATPG TOOL

NAME: poyraz --- ATPG for stuck-at faults in combinational circuits

SYNOPSIS: poyraz [no options yet] circuit_file [> outfile]
it has an user interface and may be invoked as "atpg_ui" only.

OPTIONS: will be added in the future

OUTPUTS: Three file is created. The summary of the test pattern generation is reported to the standard output and the test patterns are stored in the circuit_name.test_mlc file.

ON-LINE HELP: Type Poyraz to see the available on-line help.

EXAMPLE:

atpg_ui

or you may directly invoke as;

poyraz c432.bench
Generates test patterns for the circuit c432.bench
The generated test patterns are stored in file
c432.test_mlc and the summary of the test pattern
is reported to the standard output (CRT terminal).

NETLIST FORMAT:
The default netlist format for poyraz is ISCAS89 netlist format except for the following two cases. The first line should be # followed by the name of the circuit. The lines beginning with # excluding the first line are comment lines and ignored. These comment lines may be put into any part of the netlist. It should be noted that the order of gates appearing in the netlist is not significant. The name of gates can be a string of alpha-numeric characters (0-9, A-Z or a-z).

EXAMPLE: ISCAS89 NETLIST FORMAT (c17.bench)

```
# c17
# 5 inputs
# 2 outputs
# 0 inverters
# 6 gates ( 6 NANDs )

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

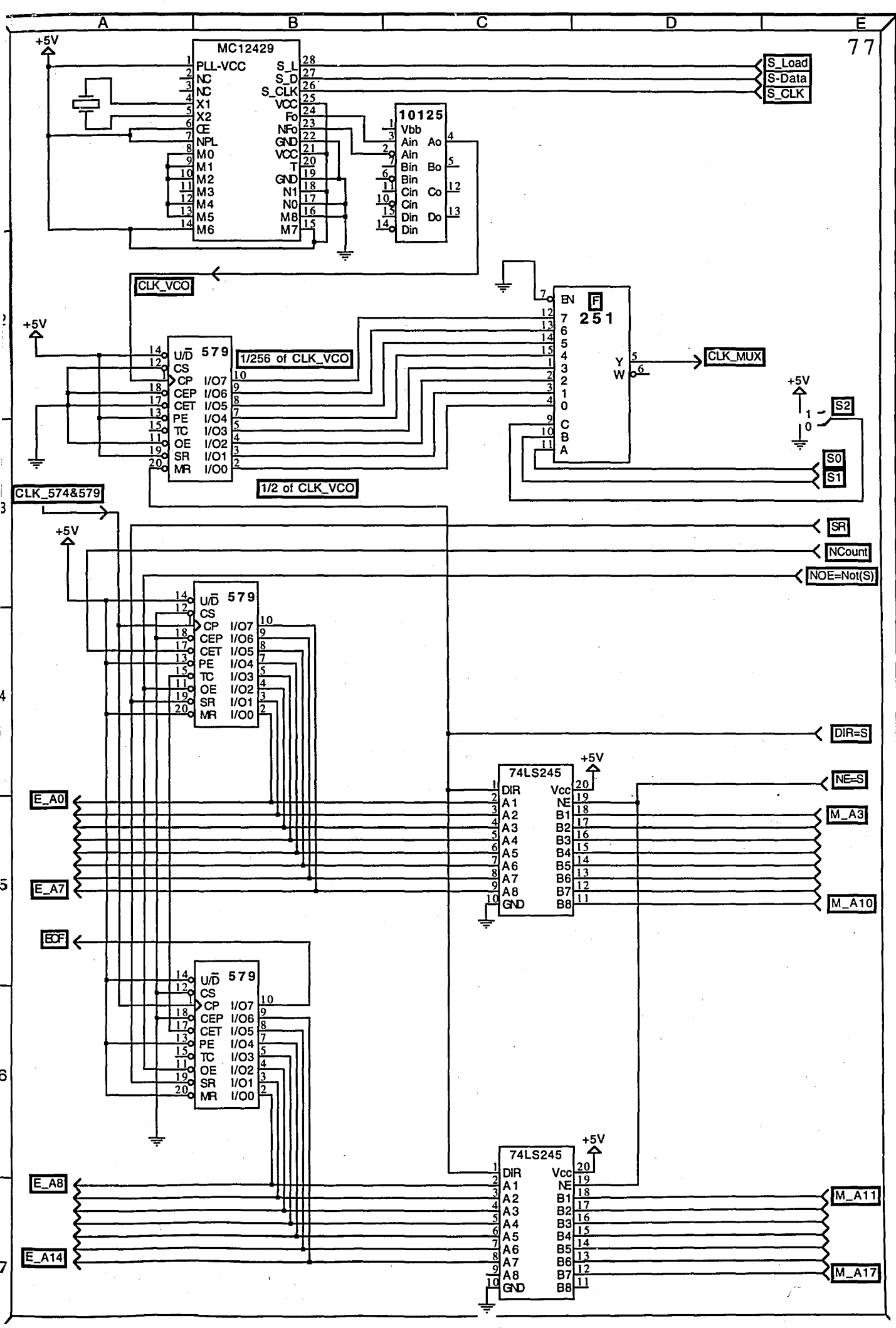
MANAGEABLE GATES:

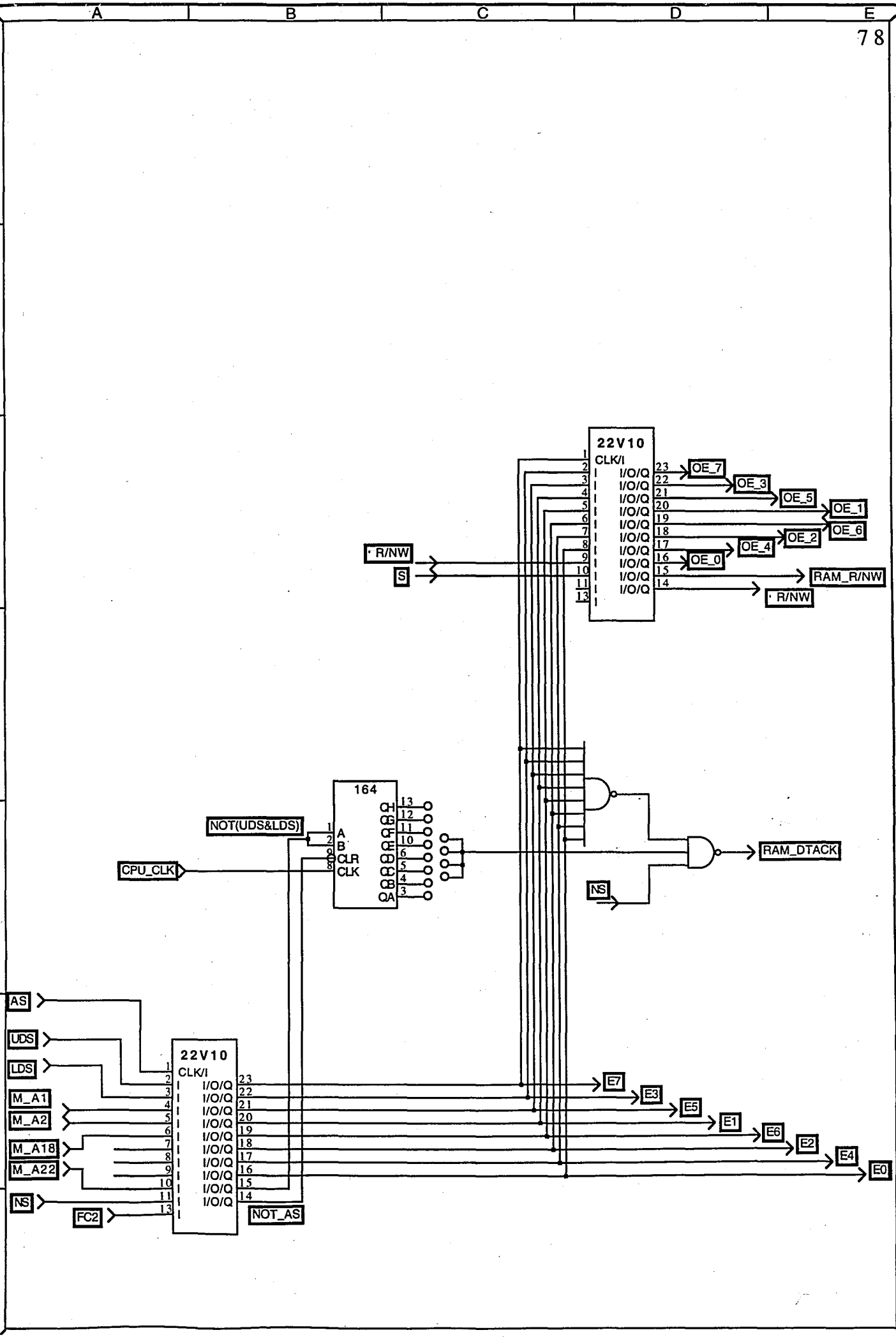
ISCAS89 NETLIST FORMAT:

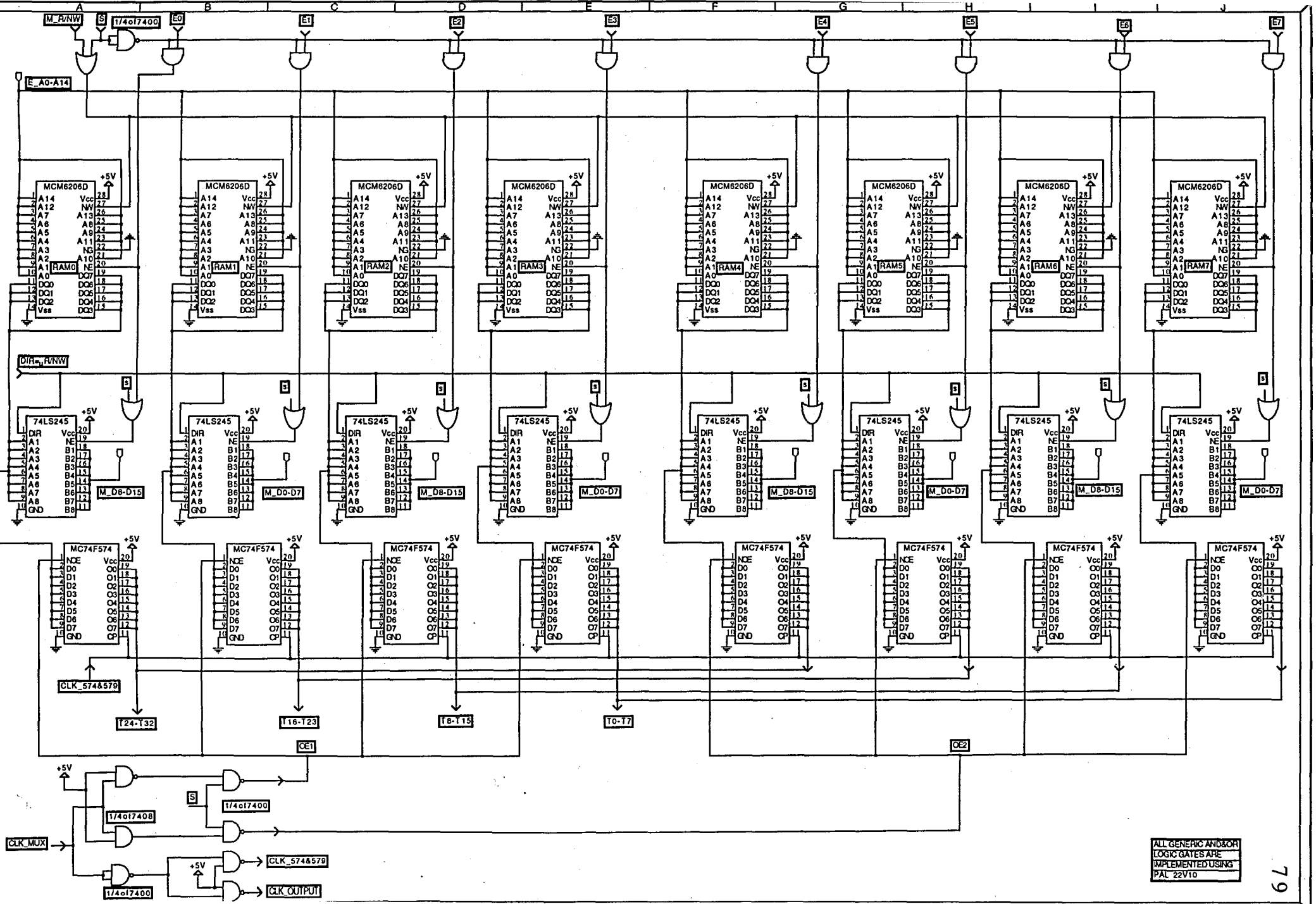
syntax	gate type
INPUT	primary input
OUTPUT	primary output
AND	and gate
NAND	nand gate
OR	or gate
NOR	nor gate
XOR	2 input exclusive-or gate
NOT	inverter

APPENDIX - C

HARDWARE SCHEMATICS
for
STIMULUS PATTERN GENERATOR


A
B
C
D
E
+5V
MC12429
PLL-VCC
NC
NC
X1
X2
CE
NPL
M0
M1
M2
M3
M4
M5
M6
S_L
S_D
S_CLK
VCC
Fo
NFo
GND
VCC
T
GND
N1
N0
M8
M7
10125
Vbb
Ain
Ain
Bin
Bin
Cin
Cin
Din
Din
Ao
Bo
Co
Do
S_Load
S-Data
S_CLK
CLK_VCO
+5V
U/D 579
CS
CP
CEP
CET
PE
TC
OE
SR
MR
I/O7
I/O6
I/O5
I/O4
I/O3
I/O2
I/O1
I/O0
1/256 of CLK_VCO
1/2 of CLK_VCO
EN F 251
Y
W
CLK_MUX
S2
S0
S1
CLK_574&579
SR
NCount
NOE=Not(S)
DIR=S
NE=S
74LS245
DIR
A1
A2
A3
A4
A5
A6
A7
A8
GND
Vcc
NE
B1
B2
B3
B4
B5
B6
B7
B8
E_A0
E_A7
E_A8
E_A14
EOF
M_A3
M_A10
M_A11
M_A17


A
B
C
D
E
22V10
CLK/I
I/O/Q
R/NW
S
OE_7
OE_3
OE_5
OE_1
OE_6
OE_2
OE_4
OE_0
RAM_R/NW
R/NW
164
NOT(UDS&LDS)
CPU_CLK
A
B
CLR
CLK
QA
RAM_DTACK
NS
AS
UDS
LDS
M_A1
M_A2
M_A18
M_A22
NS
FC2
NOT_AS
E7
E3
E5
E1
E6
E2
E4
E0


M_R/NW
S
1/4of7400
E0
E1
E2
E3
E4
E5
E6
E7
E_A0-A14
MCM6206D
RAM0
RAM1
RAM2
RAM3
RAM4
RAM5
RAM6
RAM7
+5V
DIR=M_R/NW
74LS245
M_D8-D15
M_D0-D7
MC74F574
CLK_574&579
T24-T32
T16-T23
T8-T15
T0-T7
OE1
OE2
1/4of7408
CLK_MUX
CLK_OUTPUT
ALL GENERIC AND&OR
LOGIC GATES ARE
IMPLEMENTED USING
PAL 22V10
79

APPENDIX - D

PAL - EQUATIONS FOR CONTROL LOGIC

```
module chip_sel
title 'Memory decode
MLC - Bogazici University-Bebek/Ist'

chip_sel  device  'P22V10';

AS,UDS,LDS,A1,A2,A18,A19,A20,A21,A22,NS      pin  1,2,3,4,5,6,7,8,9,10,11                  istype 'com';
FC2,NOTAS,NULDS,E0,E4,E2,E6,E1,E5,E3,E7      pin  13,14,15,16,17,18,19,20,21,22,23         istype 'com';

        H,L                 =          1,0;
  equations
NOTAS = !AS;
NULDS = !(UDS & LDS);

E0 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 & !A1 & !UDS &  LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 & !A1 & !UDS & !LDS));

E1 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 & !A1 &  UDS & !LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 & !A1 & !UDS & !LDS));

E2 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 &  A1 & !UDS &  LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 &  A1 & !UDS & !LDS));

E3 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 &  A1 &  UDS & !LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS & !A2 &  A1 & !UDS & !LDS));

E4 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 & !A1 & !UDS &  LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 & !A1 & !UDS & !LDS));

E5 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 & !A1 &  UDS & !LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 & !A1 & !UDS & !LDS));

E6 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 &  A1 & !UDS &  LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 &  A1 & !UDS & !LDS));

E7 = !((!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 &  A1 &  UDS & !LDS)#(!NS)#
          (!A22 & !A21 & A20 & !A19 & !A18 & FC2 & !AS &  A2 &  A1 & !UDS & !LDS));
END chip_sel


module output_e
title 'Bus Tranceiver
MLC - Bogazici University-Bebek/Ist'
output_e  device  'P22V10';
 E7,E3,E5,E1,E6,E2,E4,E0,MRNW,S                  pin 1,2,3,4,5,6,7,8,9,10      istype 'com';
 MRNWO,RAMRNW,OE0,OE4,OE2,OE6,OE1,OE5,OE3,OE7    pin  14,15,16,17,18,19,20,21,22,23istype  'com';
        H,L                 =          1,0;
  equations
MRNWO = MRNW;
RAMRNW = (MRNW # S);
OE0 = (E0 # S);
OE4 = (E4 # S);
OE2 = (E2 # S);
OE6 = (E6 # S);
OE1 = (E1 # S);
OE5 = (E5 # S);
OE3 = (E3 # S);
OE7 = (E7 # S);
END output_e
```

REFERENCES

1. Fujiwara, H., and S. Toida, "The complexity of fault detection: An approach to design for testability,"*Proceedings of. 12th. Int. Symp. on Fault Tolerant Comp.*, pp. 101-108, June 1982.

2. Eichelberger, E. B., and T. W. Williams, "A logic design structure for LSI testing," *Proceedings of 14th. DAC*, pp. 462-468, June 1977.

3. Funatsu, S. N, Wakatsuki, and T, Arima, "Test Generation Systems in Japan," *Proceedings of 12th DAC*, pp.114-122, June 1975.

4. Abramovici, M., M. A. Breuer, and A.D. Friedman, *Digital Systems Testing And Testable Design*, IEEE Press, 1990.

5. Weste, N. H. E., and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, Second Edition, 1993.

6. Fujiwara, H., and T.Shimono,"On the Acceleration of Test Generation Algorithms," *IEEE Trans. On Comp.*, Vol. c-32 No. 12, pp. 1137-1144, December 1983.

7. Goel, P., "An Implicit Enumaration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. On Comp,*Vol. c-30 No., pp 215-222, March 1981.

8. Sellers, F.F., M.Y. Hsiao, and L.W. Bearnson, "Analyzing Errors with the Boolean Difference," *IEEE Trans. On Comp*,Vol. c-17, pp 676-683, July 1968.

9. Timoc, C. et al., "Logical Models of Physical Failures," *Proc. Internetional Test Conference*, pp. 546-553, October, 1983.

10. Abramovici, M., P.R.Menon, and D.T. Miller, "Checkpoint Faults are not Sufficient Target Faults for Test Generation," *IEEE Trans. On Comp*,Vol. c-35, No.8, pp 769-771, August 1986.

11. Roth, J.P.,"Disagnosis of Automata Failures: A Calculus and a Method,"*IBM Journal of research and Development*, Vol. 10, No. 4, pp. 278-291, July, 1966.

12. Roth, J. P. W.G. Bouricius, and P.R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. on Electronic Computers*, Vol. EC-16, No.10, pp. 567-579, October, 1967.

13. Kirkland, T. and M.R. Mercer, "A Topological Search Algorithm for ATPG," *Proc. 24th DAC*, pp.502-508, June, 1987.

14. Abramovici, M. et al., "SMART and FAST : TG for VLSI Scan-Design Circuits," *IEEE Design & Test of Computers*, Vol.3 , No.4, pp. 43-54, August, 1986.

15. Schulz, M.H., E. Trischler, and T.M. Sarfert, "SOCRATES A Highly Efficient ATPG System," *Proc. 24th DAC*, pp.1016-1026, June, 1987.