SYNTHESIS OF CLASSICAL AND NON-CLASSICAL CMOS TRANSISTOR FAULT MODELS MAPPED TO GATE-LEVEL FOR RECONFIGURABLE HARDWARE-BASED FAULT INJECTION

By

Raha Abedi

Bachelor of Electrical Engineering

Amir Kabir University of Technology

Tehran, Iran, 2002

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2005

ABSTRACT

Title of Thesis: **SYNTHESIS OF CLASSICAL AND NON-CLASSICAL CMOS TRANSISTOR FAULT MODELS MAPPED TO GATE-LEVEL FOR RECONFIGURABLE HARDWARE-BASED FAULT INJECTION**

Raha Abedi, Master of Applied Science, 2005

Thesis Directed By: Dr. Reza Sedaghat,
Electrical and Computer Engineering Department

One of the main goals of fault injection techniques is to evaluate the fault tolerance of a design. To have greater confidence in the fault tolerance of a system, an accurate fault model is essential. While more accurate than gate level, transistor level fault models cannot be synthesized into FPGA chips. Thus, transistor level faults must be mapped to the gate level to obtain both accuracy and synthesizability.

Re-synthesizing a large system for fault injection is not cost effective when the number of faults and system complexity are high. Therefore, the system must be divided into partitions to reduce the re-synthesis time as faults are injected only into a portion of the system. However, the module-based partial reconfiguration complexity rises with an increase in the total number of partitions in the system. An unbalanced partitioning methodology is introduced to reduce the total number of partitions in a system while the size of the partitions where faults are to be injected remains small enough to achieve an acceptable re-synthesis time.

Acknowledgment

I would like to thank Dr. Sedaghat and OPR-AL lab for their contribution and support.

I would like to express my gratitude to my parents who have always immensely encouraged and inspired me. They are my great teachers in life.

I am grateful to my husband who guided and helped me all the way. Without his support this work would have not been accomplished.

Table of Contents

ABSTRACT iii

Table of Contents vi

List of Tables viii

List of Figures x

Nomenclature xii

Chapter 1 Introduction 1

1.1 Accurate Fault Model 2
1.2 FPGA-based Fault Injection 3
1.3 Summary of Contributions 4
1.4 Thesis Outline 4

Chapter 2 Fault Modeling 6

2.1 Logical Fault Model 6
2.2 Logical Fault Types 7
2.2.1 Classical Faults 7
2.2.2 Non-classical Faults 8
2.3 Transistor level fault model 9

Chapter 3 Transistor Level to Gate Level Comprehensive Fault Model Mapping 11

3.1 CMOS Physical Failures and defects 11

3.2 Transistor to Gate Level Fault Mapping 12
3.2.1 Transistor Level Representation of Primitive Gate Circuits 13
3.3 NORn Complete Fault List Pattern 14
3.3.1 Input / Output Stuck-at Faults 15
3.3.2 Short Faults 16
3.3.3 Open Faults: 22
3.4 NANDn Complete Fault List Pattern 25
3.5 Inverter Complete Fault List 28
3.6 D-Latch Complete Fault List 29
3.6.1 Input / Output Stuck-at Faults 30
3.6.2 Short Faults 32
3.6.3 Open Faults 34
3.7 D Flip-Flop Complete Fault List 35
3.7.1 Input / Output Stuck-at Faults 36
3.7.2 Short Faults 37

3.7.3 Open Faults 40

Chapter 4 User-Defined VHDL Library of Transistor Level Faults 41

4.1 VHDL Presentation of Faulty Gates 41
4.2 Creating Faulty VHDL Components Automatically 44
4.3 Designing a User-defined Library 47
4.3.1 Existing Libraries 47
4.3.2 Library management 47
4.3.3 Aliasing the faulty components to WORK library 48

Chapter 5 Fault Injection into Circuit VHDL Description 50

5.1 Behavioural Feature of Hardware Languages 51
5.2 Structural Feature of Hardware Languages 51
5.3 Fault Injection in VHDL Description 53

Chapter 6 Binary Tree-based Partitioning Methodology 56

6.1 Balanced Versus Unbalanced Partitioning 56
6.2 Binary Tree Approach 60
6.2.1 Analyzing Merge and Sort Algorithm 61
6.3 Generating VHDL Full Binary Tree 64

Chapter 7 Automating the Synthesis Procedure of Module-Based Dynamic Partial Reconfiguration 67

7.1 Partial Reconfiguration 68
7.2 Modular Design 69
7.2.1 Modular Design Entry and Synthesis Phase 70
7.2.2 Modular Design Implementation Phase 71
7.3 Automating the Synthesis Procedure of Module-based Fault Injection Method 73
7.4 Experimental Results 80

Chapter 8 Conclusion 87

8.1 Research Contribution 87
8.2 Future Work 88

Appendices 89

Appendix A: ORn Complete Fault List Pattern 89
Appendix B: ANDn Fault Pattern 92
Appendix C: Buffer Fault list 95

List of Tables

Table 1: A Short between PMOS Drain and Power

Table 2: Short between Two Inputs

Table 3: Short between One Input and One PMOS Drain

Table 4: When D_i is connected to D_k ($1 \leq k \leq n-1$ and $K \neq i$)

Table 5: When Di ($1 \leq i \leq n-1$) is shorted to Dn

Table 6: I_{iP} Open

Table 7: I_{iN} Open

Table 8: NANDn Input/Output Stuck-at-faults

Table 9: NANDn Short Faults

Table 10: NANDn Open Faults Categories

Table 11: NANDn Open Faults

Table 12: Inverter Input / Output Stuck-at Faults

Table 13: Inverter Short Faults

Table 14: Inverter Open Faults

Table 15: D-Latch Input / Output Stuck-at Faults

Table 16: Short between Each Node and VCC or the Ground in D-latch

Table 17: D-Latch Short Faults between Two Nodes

Table 18: D-Latch Open Faults

Table 19: Fault Free D Flip-Flop

Table 20: D Flip-Flop Input / Output Stuck-at Faults

Table 21: D Flip-Flop Short Faults

Table 22: D Flip-Flop Open Faults

Table 23: Total Number of Faults for Each Gate

Table 24: Total Number of Subroutines

Table 25: S38417 Partition Size

Table 26: ORn Input/Output Stuck-at-faults

Table 27: ORn Short Faults

Table 28: ORn Open Faults Categories
Table 29: ORn Open Faults
Table 30: ANDn Input/Output Stuck-at-faults
Table 31: ANDn Short Faults
Table 32: ANDn Open Faults Categories
Table 33: ANDn Open Faults
Table 34: Buffer Input/Output Stuck-at-faults
Table 35: Buffer Shore Faults
Table 36: Buffer Open Faults

List of Figures

Figure 1: DRAM Cell

Figure 2: 2-input NOR (gate-level)

Figure 3: 2-input NOR (transistor-level)

Figure 4: I_2 Stuck-at-0 Fault of 3-input NOR

Figure 5: A Short between D_1 and Ground

Figure 6: NORn Transistor Level

Figure 7: Faulty NOR3 Transistor Level

Figure 8: NANDn Transistor Level

Figure 9: CMOS Inverter

Figure 10: T-gate

Figure 11: CMOS D-Latch

Figure 12: D Flip-Flop

Figure 13: NOR2_Short_Fault Synthesizable VHDL code

Figure 14: NOR2_open_fault synthesizable VHDL code

Figure 15: Perl Program Section for NORn Input stuck-at-1 Fault

Figure 16: C17.vhd

Figure 17: C17.prj

Figure 18: Half-adder Behavioural Model

Figure 19: Half-adder Structural Model

Figure 20: Faulty AND2 Instantiation

Figure 21: Balanced Partitioning

Figure 22: Balanced Partitioning Graph

Figure 23: Unbalanced Partitioning

Figure 24: Unbalanced Partitioning Graph

Figure 25: Full Binary Tree (Depth level = 4)

Figure 26: The Construction of a Recursion Tree for the Recurrence $T(n) = 2T(n/2) + cn$

Figure 27: The Flow of Binary Tree Based Partitioning and Modification of the VHDL Code for Fault Injection

Figure 28: Modular Design Entry and Synthesis Flow
Figure 29: Circuit Name Assignment
Figure 30: S1238 Step 2
Figure 31: Verifying the Entered Branch Length
Figure 32: Verifying the Entered Branch Characters
Figure 33: Making a Partition Directory inside the Circuit Directory
Figure 34: Copy a Partition VHDL Code to Its Directory
Figure 35: An Example of Fault List for S1238
Figure 36: S1238 Circuit Directory
Figure 37: S1238_Partition_LLLL XST Script File
Figure 38: S1238_Partition_LLLL Directory Files
Figure 39: Executing the Script File (xst.scr)
Figure 40: Size of Circuits
Figure 41: Full Binary Tree Depth Level of Circuits
Figure 42: Re-synthesis Time of the Faulty Partition
Figure 43: Benchmarks Synthesis Time
Figure 44: S38417 Partition synthesis time
Figure 45: Partition Selection of a Full Binary Tree (Depth level = 9)
Figure 46: Unbalanced Versus Balanced Partitioning
Figure 47: ORn Transistor Level
Figure 48: ANDn Transistor Level
Figure 49: Buffer Transistor Level

Nomenclature

c	Constant
C	Clock
D_i	Drain of the i^{th} CMOS transistor
G	Ground
G_T	Total number of gates
i, j, k, n	Integers
I_i	i^{th} input of a gate
I_{iP}	Input of the i^{th} PMOS transistor
I_{iN}	Input of the i^{th} NMOS transistor
nC	Not Clock
N	Total number of partitions
P_i	i^{th} partition
P_f	Partition with injected faults
Q_i	i^{th} CMOS transistor
Q^-	State of Q at time t-1
S_i	Source of the i^{th} CMOS transistor
S-on	Stuck-on
$S(P_i)$	Size of the i^{th} partition
T_i	T-gate number i
ν	Logic value
X	unknown
Z	high impedance
$\oplus$	XOR
$\forall$ **i**	For all values of i
<0 | 1>	Zero or One
γ	Cost function

Chapter 1

Introduction

As systems become more complex it becomes increasingly difficult to provide comprehensive fault testing to determine the validity of a system. Hence, faults can remain in a system and manifest themselves as errors. Furthermore, faults may be introduced into a hardware system from external sources such as electromagnetic interference. Not only can components within a system fail, no transistor will function forever. These faults can ultimately cause a system to fail. The ability of a system to function in the presence of faults, i.e. to become fault tolerant, is a rapidly growing area of research. A fault tolerant system has the ability to respond gracefully to an unexpected hardware or software failure. The need for fault tolerant systems is driven by various factors such as extremely high reliability and availability needs, reduced life-cycle costs, and long-life requirements.

Most real-time systems must function with very high availability even under hardware fault conditions. To design a fault tolerant system many fault injection techniques to evaluate the dependability of the system have been proposed. In all of these techniques single or multiple faults are intentionally inserted into the system to study the behaviour of the system in the presence of faults.

Physical failures or defects in a circuit may cause faulty circuit behaviour and thus reduce the fault tolerance rate of the system. To evaluate the fault tolerance of a system, an accurate fault model is required. Next, an approach to inject each fault into the system is essential and finally, an adequate test pattern to determine if the fault has changed the behaviour of the

circuit. We will focus mainly on defining an accurate fault model and a method of injecting faults into a system.

1.1 Accurate Fault Model

More accurate fault models are usually defined below the gate level. The major disadvantage of using a transistor level (low level) fault model is the performance degradation of fault simulation, fault emulation and test pattern generation [3]. Therefore, there is a critical need to map the transistor level fault model onto the gate level without any performance degradation, while at the same time obtaining an accurate fault model.

For many years the only practical approach has been based on the stuck-at fault model. However, problems arise in CMOS LSI and VLSI circuits. The stuck at fault model is not sufficient for systems that require high reliability or high availability such as pacemakers, ABS (anti-lock braking systems) in automobiles, or air traffic control. There are other types of physical failures and defects, i.e. short or open faults that disturb system performance.

There are many studies which model physical failure and defect in CMOS circuits. In [32, 33, and 34], the main focus is on modeling short faults between a gate and a source of one CMOS transistor in a circuit. A methodology to obtain a minimal set of faults is proposed in [35]. This methodology is based upon theoretical basis allowing the determination of the equivalence and dominance of non-classical CMOS faults. In [5, 36] short, open and stuck-at faults are presented for CMOS circuits based on simulation results and thoroughly cover open and stuck-at faults. However, the fault model they present only considers short faults for one transistor in the circuit at a time, such as a short between a gate and a drain of the same transistor. However, in [3] all the possible short faults are taken into account in the fault model, including connecting two nodes from different CMOS transistors.

In this research, all shorts that could possibly occur among transistors (e.g. a short between the gate of one transistor and the drain of another transistor) in the circuit in addition to all possible stuck-at and open faults are considered. By applying a complete version of accurate fault models [3, 5] to each gate to generate a complete fault list, it is revealed that the fault list for each type of primitive gate follows a specific pattern regardless of its number of inputs. The number of gate inputs can be used to calculate the total number of possible faults at the transistor level using general formulas presented in this research.

1.2 FPGA-based Fault Injection

After generating a fault model, software simulation and hardware emulation are the main techniques that researchers follow to inject faults into a system. Fault simulation provides a high degree of controllability [37, 38] and design mistakes in the fault-tolerant system can be detected and corrected at a very early stage in the design process [38]. However, the main drawback related to fault simulation is that it is time-consuming when many faults have to be injected in a complex circuit [39, 16]. For fault emulation a prototype of the system is needed. An advantage of prototyping is the possibility to perform "in-system" emulation before any manufacturing. Reconfigurable devices such as FPGAs are appropriate to implement and test the prototype. To cope with the time limitation imposed by simulations, it has been proposed to take advantage of hardware prototyping using FPGA-based hardware emulators [40]. Another advantage of emulation is to allow the designer to study the actual behaviour of the system in the application environment, taking into account real-time interactions of various hardware and software components [15].

FPGA-based fault injection has been an area of increased research. For example, [41, 42] discuss injecting faults through fault injectors for stuck-at faults and [43] applying a FPGA-based fault injection method through fault injection chain hardware. Another method is to inject faults by using scan-chain hardware for injecting bit-flip faults into flip-flop of the target system [44, 39, and 26]. In [21, 45, and 46] FPGA-based emulators have been used to only inject stuck-at faults for test pattern generation purposes without evaluating the fault-tolerance of the system. Using run-time reconfiguration (RTR) for fault injection is another approach proposed by [47] to save time by reconfiguring only a few resources of the device. However, because this approach modifies bitstreams it is not capable of accessing the drain of specific CMOS transistor in the circuit to make it an open node or to short it to the others nodes of the circuit. This is due to the fact that bitstrams represent the Look Up Table (LUT) values of the circuit and not the gate level description. During the synthesis some gates may be mapped into other gates due to optimization. Therefore, there is no possibility to access the transistor level description of those gates to inject faults. The FPGA-based fault injection into switch-level of a model which is in abstraction level between gate level and transistor level is discussed in [48].

When an emulator is used, the initial VHDL description must be synthesizable. It is apparent that a transistor level fault model cannot be synthesized into FPGA chips. Therefore, the fault model mapping from transistor level to gate level has a critical role in our study for injecting transistor level faults into the FPGA.

In this work, first the transistor fault model is mapped to gate level fault model. Next, a method is described to inject this fault model into the FPGA by considering the advantages of module-based dynamic partial reconfiguration.

1.3 Summary of Contributions

This thesis contributes to the following areas:

- Generating a complete fault list of primary gates (NOR, NAND, OR, AND, Inverter, Buffer, D-Latch, D Flip-Flop).
- Presenting a fault pattern for each type of gate regardless of its number of inputs.
- Extracting a general formula for each type of gate to calculate the total number of faults according to the gate number of inputs.
- Generating automatically a faulty component library that represents the transistor level faults at a gate level description.
- Presenting a methodology to partition a high level (e.g.VHDL) description of a circuit to reduce the synthesis time as well as module-based circuit design complexity.
- Generating faulty VHDL partitions exhaustively and providing a data base.
- Automate the synthesis process of faulty and fault free partitions based on a desired fault list and modular design regulations.

1.4 Thesis Outline

The necessity of fault modeling as well as different fault classifications are discussed in Chapter 2. Chapter 3 presents a complete transistor to gate level fault mapping of primary gates and their related fault pattern as well as general formulas to calculate the total number of faults. Generating and automating a user-defined library based on the fault patterns from Chapter 3, is described in Chapter 4. In Chapter 5 the behavioural and structural VHDL description features are discussed and an appropriate approach to inject faults in the VHDL

description of a circuit is introduced. A methodology of partitioning a system to reduce the synthesis time and system module-based complexity is discussed in Chapter 6. The automation of synthesis procedure based on modular design parameters and experimental results are presented in Chapter 7. Chapter 8 is dedicated to conclusions and intended future works.

Chapter 2

Fault Modeling

An instance of an incorrect operation of the system being tested is referred to as an observed error. Causes of observed errors may be design errors, fabrication errors, fabrication defects, or physical failures. Design errors can be detected and corrected at an early stage of the design process by simulating the design. Fabrication defects are not directly attributable to human error; rather, they result from an imperfect manufacturing process. Physical failures occur during the lifetime of a system due to component wear-out and/or environmental factors.

In general, Physical faults do not allow a direct mathematical treatment of testing and diagnosis. The solution is to deal with logical faults, which are a convenient representation of the effect of physical faults on the operation of the system. The basic assumptions regarding the nature of logical faults are referred to as a fault model [1].

2.1 Logical Fault Model

Logical faults represent the effect of physical faults on the behaviour of the modeled system. Given a logical fault and model of a system, we should be able in principle to determine the logic function of the system in the presence of the fault. Thus, fault modeling is closely related to the type of modeling used for the system.

The advantages of modeling physical faults as logical faults can be described as follows [1]:

- The problem of fault analysis becomes a logical rather than a physical problem.

- The complexity is greatly reduced since many different physical faults may be modeled by the same logical fault.
- Some logical fault models are technology-independent in the sense that the same fault model is applicable to many technologies.
- Tests derived for logical faults may be used for physical faults whose effect on circuit behaviour is not completely understood or is too complex to be analyzed.

A logical fault model can be defined in different levels of circuit descriptions. Transistor, gate, and RTL level of circuit descriptions, each have their own logical fault models. As mentioned, these fault models are related to the type of modeling used for the circuit.

2.2 Logical Fault Types

Faults can be categorized into two major groups of classical and non-classical faults.

2.2.1 Classical Faults

The classical faults can be categorized as follows [1, 2]:

- Single (line) stuck-at fault: The given line has a constant value (0/1) independent of the other signal values in the circuit. In many technologies, a short between ground or power and a signal line can make the signal remain at a fixed voltage level. The corresponding logical faults consist of the signal being stuck at a fixed logic value v (v = 0/1), and is denoted by *stuck-at-v*.
- Multiple stuck fault: Several signal stuck-at faults occur simultaneously. For a circuit with k lines, i.e., k input signals and input/output elements, there are $2k$ single stuck faults, and 3^k-1 multiple stuck faults. For a large combinational circuit with multiple outputs, almost all multiple faults can be covered by test patterns derived for single faults.
- Bridging fault: Two or more normally distinct points (lines) are shorted together. A short between two signal lines usually creates a new logic function. The logical fault representing such a short is referred to as a bridging fault. Input bridging can form a wired logic or voting model.

Input-to-output bridging can introduce feedback or cause oscillation or latching.

2.2.2 Non-classical Faults

In general, non-classical faults are categorized as follows [2, 3]:

- Pattern-sensitive fault: The presence of a faulty signal depends on signal values of nearby points (most common in DRAMs).

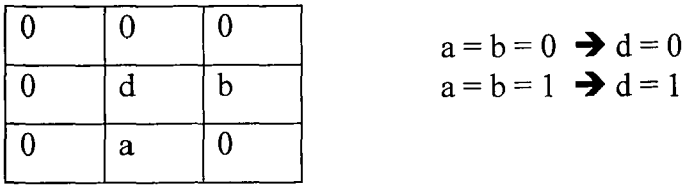


Figure 1: DRAM Cell

- Coupling fault: Pattern sensitivity between a pair of cells.
- Crosspoint fault: A PLA (Programmable Logic Array[1]) inherently has a device (diode or transistor) at every crosspoint in the (AND and OR) arrays, even if not used. The connection of each diode is programmed to realize the desired logic. A crosspoint fault can be caused by an extra or missing device.
- Transistor stuck-open fault: Transistor (switch) is always off, not controllable by gate input. This fault can turn the circuit into a sequential one. This type of fault is more difficult to test and needs a sequence of at least 2 tests to detect a single fault. Transistor stuck-open fault is unique to CMOS circuits.
- Transistor stuck-on fault: Transistor (switch) is always on, not controllable by gate input. This kind of fault can be caused by a permanently conducting transistor and also occurs in CMOS circuits.
- (Line) break (stuck-open) fault: An open wire.

[1] PLA is an array of gates having interconnections that can be programmed to perform a specific logical function.

- (Line stuck) short fault: A short is formed by connecting points not intended to be connected.
- Delay fault: Propagation (transition) delays along a path (gate) that fall outside the desired limits are referred to as either path delay faults or gate delay faults.
- Function conversion fault: Defects inside a CMOS gate may result in an incorrect function of the gate. If there is a short between input and output of an inverter the gate is no longer inverting. Furthermore, conversion of an AND into a NAND, OR into NOR etc. is possible.
- Conditional fault: A conditional fault is defined as a fault which can only be detected if one or more conditions are satisfied. Each condition consists of a fixed logical value (0 or 1) at a specified mode. This kind of fault is defined for CMOS and it is similar to pattern-sensitive fault of DRAMs.

Stuck-open and stuck-on faults are transistor level faults; stuck-at, bridging, delay faults are gate level faults; pattern-sensitivity, crosspoint faults are function level faults.

Most of the non-classical faults can be mapped onto classical stuck-at faults [4], bridging faults and transition faults. The transition fault is based on the assumption that a transition at a gate never occurs in a combinational circuit. In a sequential circuit, transition does not occur within a clock cycle.

2.3 Transistor level fault model

Let's see how a transistor fault can affect the behaviour of the circuit. In the presence of transistor faults or an interconnection wiring faults circuit will not function correctly. Many things can go wrong, leading to a variety of faults. A transistor switch can break so that it is either permanently closed or open. A wire in the circuit can be shorted to VCC or to ground, or it can simply be broken. There can be an unwanted connection between two wires. A logic gate may generate a wrong output signal because of a fault in the circuitry that implements the gate. CMOS logic circuits present some special problems in terms of faulty behaviour. The transistors may fail in a permanently open or shorted (closed) state. Many such failures manifest themselves as stuck-at faults. However, some produce entirely different behaviour.

For example, transistors that fail in the shorted state may cause a continuous flow of current from VCC to ground, which can create an intermediate output voltage that may not be determined as either logic 0 or 1. Transistors failing in the open state may lead to conditions where the output capacitor retains its charge level because the switch that is supposed to discharge it is broken. The effect is that a combinational CMOS circuit starts behaving as a sequential circuit [7].

As CMOS has emerged as an important technology for VLSI, testing of large CMOS networks has become a crucial issue. The classical stuck-at fault model assumptions are not sufficient for modeling certain faults that are specific to a CMOS-based VLSI technology. This applies particularly when systems with a high reliability or high availability such as space applications are considered. Depending on the technology, typical physical defects as such as CMOS stuck-open faults may not be covered by a stuck test set. Therefore, new fault models have been introduced at different description levels to increase the accuracy of fault modeling.

Transistor level fault model is more accurate than gate level fault model. However, its fault simulation, fault emulation, and test pattern generation are degraded in comparison to the gate level.

In order to maintain the efficiency resulting from gate level modeling while the accuracy of the fault model is increased, transistor to gate level fault mapping is required. Next chapter is directed towards a mapping of classical and non-classical transistor level faults to the gate level.

Chapter 3

Transistor Level to Gate Level Comprehensive Fault Model Mapping

In order to have greater confidence in the fault tolerance of a system more accurate fault models are needed. An accurate fault model cannot be attained unless all faults in the transistor level (low level) are considered thoroughly. However, these transistor-level faults must be mapped onto gate level (higher level) so that the efficiency of fault simulation, fault emulation and test pattern generation on the gate level is not sacrificed. This chapter considers single physical failures for static CMOS primitive gates and shows their effects in the output behaviour in terms of gate level faults. We have found a specific fault pattern for each type of gate regardless of its number of inputs is proposed. All kinds of faults from stuck-at to short and open faults have been considered in these patterns. A general formula to calculate the total number of faults for each type of gate is extracted from these patterns.

3.1 CMOS Physical Failures and defects

Failures in CMOS circuits can be classified into shorts, opens, and circuit degradation. Shorts are due to oxide breakdown and metal bridging, and are caused by static discharge and time-dependent defects, while metallization problems caused by electro-migration or electro-mechanical corrosion can produce shorts and opens. Degradations include threshold voltage shifts caused by ionic contamination, surface-charge spreading, and the trapping of hot

electrons in the gate oxide. However, these degradations, if permanent, will consequently be translated into classical type of faults at the input(s) and/or output(s) [5].

It should be noted that 75 percent of the cases are shorts and opens, while the rest could be considered unobservable or insignificant [6]. Hence, physical faults of CMOS cells have been divided into two groups, namely shorts and opens. For example, for any two-input CMOS gate, the following faults are considered:

1) *Short Faults:*
   - Short between gate and source in both p-channels;
   - Short between gate and drain in both p-channels;
   - Short between source and drain in both p-channels;
   - Short between gate and source in both n-channels;
   - Short between gate and drain in both n-channels;
   - Short between source and drain in both n-channels.
2) *Open (Floating) Faults:*
   - Open gate in both p-channels and n-channels;
   - Open source in both p-channels and n-channels;
   - Open drain in both p-channels and n-channels.
3) *Input / Output Stuck-at Faults:*
   - Input #1 stuck-at-0 or 1;
   - Input #2 stuck-at-0 or 1;
   - Output stuck-at-0 or 1.

However, it should be noted that some faults are redundant, which consequently reduces the number of faults.

3.2 Transistor to Gate Level Fault Mapping

The complexity of integrated circuits requires that a practical approach to fault simulation, fault emulation, and test pattern generation be based on a higher level of circuit description. The adequate fault model depends on the desired accuracy for modeling the actual defects and the complexity of the circuit description. For years the only practical approach has been based on the classical stuck-at fault model and a gate level description of

the circuit. The main reason is that fault simulators and fault emulators can handle the gate level descriptions efficiently and in a timely manner. However, the conventional stuck-at fault assumptions are not sufficient for modeling certain faults that are specific to some VLSI technology, especially, when certain physical failures such as shorts between two nodes or open nodes occur in CMOS technology [3].

The general idea of transistor to gate level fault mapping is to combine the accuracy of modeling faults at the transistor level with the efficiency of fault simulation, emulation, and test pattern generation based on gate level description of the circuit. The starting point in this research is to consider all the faults that can possibly occur in a transistor level description of the standard cell library [3]. The main focus of this study will be on static CMOS library.

In this study NAND, NOR, AND, OR, Inverter, Buffer, D-latch and D flip-flop cells are considered. For each cell the effect of each transistor level fault on the gate level fault model is determined using the results given in [3, 5]. By applying an accurate fault models to each gate (cell) to generate a complete fault list in this work, we have found that the fault list for each type of primitive gate follows a specific pattern regardless of the number of inputs. The number of gate inputs can be used to calculate the total number of possible faults at the transistor level using general formulas which will be described later.

3.2.1 Transistor Level Representation of Primitive Gate Circuits

The accuracy such as internal nodes values cannot be reached by just considering a circuit at the gate level. Figure 2 shows a gate level NOR circuit. At the gate level, faults can only be injected or diagnosed on input and output pins. There is no way to access the gate internally to inject more faults, nor to observe its fault tolerance or diagnose the fault after its detection. Therefore, we must use the transistor level of primitive gates in order to cover more faults and obtain a more accurate fault model. Figure 3 shows the 2-input NOR (NOR2) transistor level circuit.

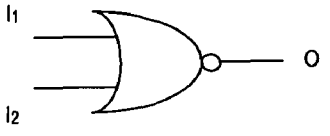


Figure 2: 2-input NOR (gate-level)

Injecting a fault in a CMOS gate may result in different output values according to different input combinations. The relationship between inputs and the faulty output shows the faulty function of the CMOS gate. For example, if I_1 is shorted to D_1 the output of NOR2 gate in Figure 3 will be:

$$O = \begin{cases} I_1 & if \quad I_2 = 0 \\ 0 & else \end{cases}$$

The relationship between inputs and the faulty output will be thoroughly discussed in following sections for all possible faults in the CMOS gates.

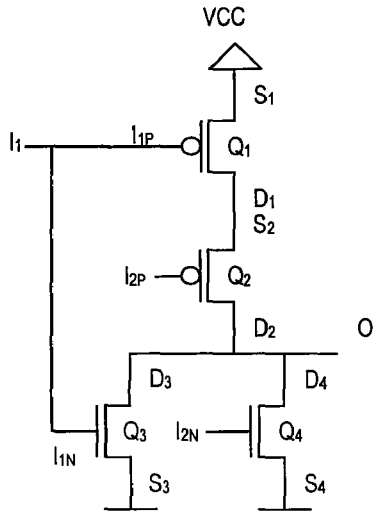


Figure 3: 2-input NOR (transistor-level)

These relations between inputs and output follow a specific pattern regardless of gate input numbers for each gate. We will discuss and explain an example of this pattern for a NOR gate with n inputs in the next section.

3.3 NORn Complete Fault List Pattern

Figure 6 shows a CMOS NOR gate with n inputs. n is an integer where ($n \geq 2$). I_i symbolizes one of the NOR ith input where ($1 \leq i \leq n$).

I_{iN} and I_{iP} are the inputs of i[th] NMOS and PMOS transistors respectively. They are both connected to I_i. For stuck-at-faults and short faults I_i , I_{iP} and I_{iN} function as the same node. Thus, they will be generally labeled as I_i. However, in the open fault case, they act as separate nodes.

The three groups of faults (stuck-at, short, and open) and their effects on the output of NORn are represented as follows with respect to the above assumptions:

3.3.1 Input / Output Stuck-at Faults

NORn is faulty if either I_i or O is stuck-at-0 or 1. When I_i is stuck-at-1 it turns its NMOS on thus, the output is connected to ground and results in low output (O = "0"). In contrast, when I_i is stuck-at-0 it turns its PMOS on and NMOS off. Therefore, the faulty NORn gate still acts as a NOR gate but without I_i input. The output is the result of NOR(I_1, I_2 ,..., I_{i-1}, I_{i+1},..., I_n). Figure 4 illustrates I_2 stuck-at-0 faults. In this case Q_2 is switched on and Q_5 is switched off. Therefore, the output is the result of NOR with only two inputs of I_1 and I_3 (NOR(I_1,I_3)).

If (n = 2) and one of the two inputs is stuck-at-0 then the output will be the NOR of the remaining non-faulty input which acts as an inverter. When the output is stuck-at-0 or 1, it will no longer be dependent on inputs.

The total number of input/output stuck-at faults for NORn gate is *(2n+2)*.

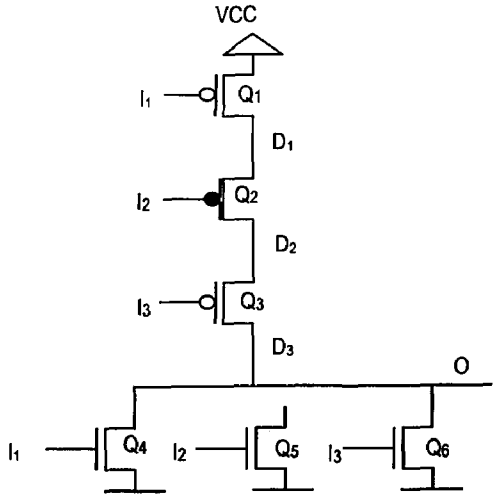


Figure 4: I_2 Stuck-at-0 Fault of 3-input NOR

3.3.2 Short Faults

The many types of short faults that may occur in a NORn circuit are categorized as follows:

1) Short between a PMOS drain and ground or power:

As far as output stuck-at-0 and stuck-at-1 are concerned, there is no need to analyze the D_n connection to ground and power due to the similarities in results. For any i ($1 \leq i \leq n-1$), D_i can be shorted to ground or power thus, the total number of faults for this category will be $2(n-1)$.

If D_i is shorted to ground, noted as ($D_i \leftrightarrow G$), the output O will always be low. When the circuit is fault free by applying low inputs the output is connected to VCC through conducting PMOS transistors. Given the presence of a short fault between D_i and ground, the output will be connected to ground through the PMOS transistors located below the node D_i and the short between D_i and ground. This will result in a low value for the output of this faulty circuit. For the rest of the input combinations, where at least one input with a high value exists, i.e. at least one NMOS transistor is conducting; the output is forced to low again. Figure 5 illustrates $D_1 \leftrightarrow G$ fault in NOR3 gate. When I_2 and I_3 are low Q_2 and Q_3 are switched on and the output is connected to ground through Q_2, Q_3, and D_1.

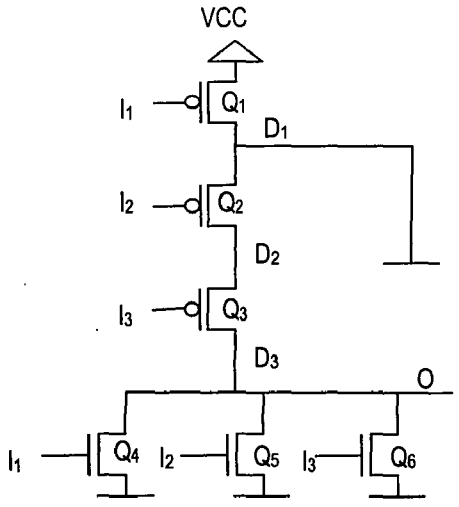


Figure 5: A Short between D_1 and Ground

If D_i is connected to *VCC*, ($D_i \leftrightarrow VCC$), the circuit will behave differently with regard to its inputs. When all inputs are low ($I_j=0,\ 1\leq j\leq n$) the output will be high. As soon as one of the inputs, connected to the PMOS transistor located above D_i, goes high it switches one NMOS transistor on and the related PMOS transistor off. In this case, a path is established between ground and *VCC*, through D_i, output and NMOS transistor. This condition takes the output to an uncertain value between 0 volt and *VCC*. The output is called stuck-on in this situation [5]. For the rest of the input combinations the output will be low while D_i is shorted to *VCC*. All the conditions are summarized in Table 1.

In fact Table 1 shows the output pattern with regard to input conditions when any drain of the n-input NOR gate is shorted to VCC. In Table 1, $I_j = 0 \quad \forall \ 1\leq j\leq n$ indicates all inputs from I_1 to I_n are low. In this case the output is high. $I_j = 0$ ($\forall$ i+1$\leq$ j$\leq$ n) denotes all inputs of PMOS transistors below the shorted drain (D_i) are low and ($I_1 \parallel I_2 \parallel \ldots \parallel I_i$) =1 indicates at least one of the PMOS transistor inputs above the shorted drain is high. In this case the output has an unknown value between 0 and VCC volt (stuck-on).

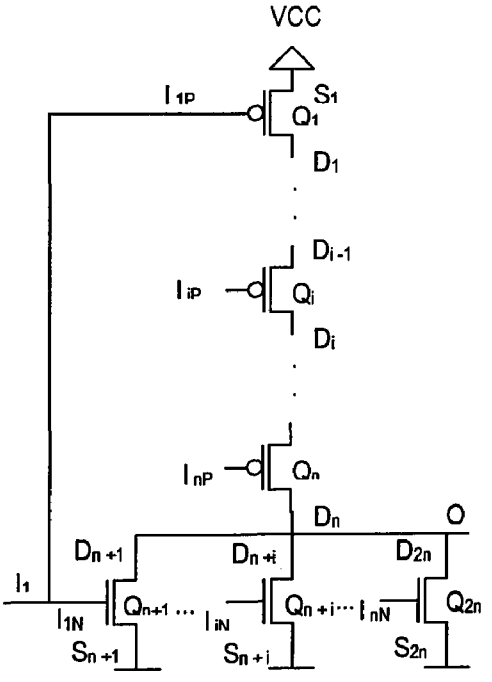


Figure 6: NORn Transistor Level

Short faults	Input conditions	Faulty output (O)
$D_i \leftrightarrow VCC$	$I_i = 0 \ \forall \ 1 \le j \le n$ (means all inputs from I_1 to I_n are low)	1
	$I_j = 0 \ (\forall \ i+1 \le j \le n)$ & $(I_1 \| I_2 \| \ldots \| I_i) = 1$	S-on (Stuck-on)
	For all other input combinations	0

Table 1: A Short between PMOS Drain and Power

2) Short between two inputs: ($I_i \leftrightarrow I_k$ where $1 \le k \le n$ and $k \ne i$)

Assume I_i is shorted to I_k. The output is unknown when these two inputs have opposite values while the rest of the inputs are low. The circuit output can have a high or low value depending on the input combinations. The faulty output pattern for this kind of fault is shown in Table 2.

In Table 2, $I_j = 0 \ \forall \ (1 \le j \le n$ and $j \ne i, j \ne k)$ denotes all inputs except I_i and I_k are low and $I_i \oplus I_k = 1$ indicates I_i and I_k have different values. In this case the output is unknown.

The total number of short faults that connect two inputs undesirably can be computed from equation (3.3.2.1).

$$(n-1)+(n-2)+\ldots+(n-(n-1)) = \sum_{j=1}^{n-1}(n-j) \qquad (3.3.2.1)$$

Short faults	Input conditions	Faulty output (O)
$I_i \leftrightarrow I_k$ $(1 \le k \le n$ and $k \ne i)$	$I_i = 0 \ \forall \ 1 \le j \le n$	1
	$I_j = 0 \ \forall \ (1 \le j \le n$ and $j \ne i, j \ne k)$ & $I_i \oplus I_k = 1$	X
	For all other input combinations	0

Table 2: Short between Two Inputs

3) Short between one input and one PMOS drain: ($I_i \leftrightarrow D_k$ where $1 \le k \le n$)

The total number of faults in this category is n^2. There are two cases for determining the output value given this type of fault. The first case occurs when I_i is

shorted to a drain of any PMOS transistor located below it ($I_i \leftrightarrow D_k$ where $i \leq k \leq n$). The second case considered is when I_i is shorted to the drain of one of the PMOS transistors above it ($I_i \leftrightarrow D_k$ where $1 \leq k \leq i-1$).

In the first case the output is equal to I_i if all the PMOS inputs located below the D_k are low. Thus, the output is connected to I_i through conducting PMOS transistors and the short between I_i and D_k. But even if one of the inputs, located below D_k, has a high value it turns its related PMOS transistor off and NMOS transistor on. Therefore, the output will be low [3]. If I_i is shorted to D_n the output equals I_i for any inputs combination. These are shown in Table 3.

In Table 3, $I_j = \langle 0|\ 1\rangle \ \forall\ 1 \leq j \leq k$ points out that PMOS transistor inputs above D_k can be either high or low and $Ij = 0 \ \forall\ k+1 \leq j \leq n$ indicates all PMOS transistor inputs below D_k are low. In this case the output has the exact value of I_i.

In the second case, the output remains low for all input combinations [1]. If all inputs located below D_k including I_i are low the output, which is connected to I_i through conducting PMOS transistors and the short between D_k and I_i, will be low. If one of these inputs has a high value the path between the output and I_i is disconnected; the output will be connected to the ground through conducting NMOS transistors.

Short faults	Input conditions	Faulty output (O)
$I_i \leftrightarrow D_k$ $(i \leq k \leq n-1)$ $(1 \leq i \leq n-1)$ D_k is located below the I_i.	$I_j = \langle 0\|\ 1\rangle \ \forall\ 1 \leq j \leq k$ & $I_j = 0 \ \forall\ k+1 \leq j \leq n$	I_i
	For all other input combinations	0
$I_i \leftrightarrow D_n$ $(1 \leq i \leq n)$	For all input combinations	I_i
$I_i \leftrightarrow D_k$ $(1 \leq k \leq i-1)$ D_k is located above the I_i.	For all input combinations	0

Table 3: Short between One Input and One PMOS Drain

4) Short between two PMOS drains: ($D_i \leftrightarrow D_k$ where ($1 \leq k \leq n$ and $k \neq i$))

Another type of short faults occurs when D_i is shorted to another drain. There will be a difference in the output results if D_i is shorted to D_n, considered the same as the output, or if it is shorted to one of the other PMOS drains.

- If D_i is connected to D_k ($1 \leq k \leq n-1$ and $K \neq i$):

The output is stuck-on if all inputs, except the inputs between two shorted drains, are low. If at least one input of any transistor between the two shorted drains is high, the related NMOS transistor is turned on and a conducting path between *VCC* and ground appears. Figure 7 illustrates the short fault between D_1 and D_2 for NOR3. By applying (010) as an input vector, Q_1, Q_3 and Q_5 turn on and *VCC* is connected to the ground through Q_1, D_1D_2, Q_3 and Q_5.

The output will be high if all inputs are low and will be low for all other input combinations. The output faulty pattern for short between two drains is shown in Table 4.

In Table 4, $I_j = 0$ ($\forall$ $1 \leq j \leq i$ and $m < j \leq n$ $(i+1 \leq m \leq n-1)$) denotes all PMOS transistor inputs except those between D_i and D_k are low and $(I_{i+1} \parallel I_{i+2} \parallel \ldots \parallel I_m) = 1$ shows at least one of the inputs of PMOS transistors between D_i and D_k is high. In this case the output has an unknown value between 0 and VCC volt (stuck-on).

Short faults	Input conditions	Faulty output (O)
$D_i \leftrightarrow D_k$ ($1 \leq k \leq n-1$, $1 \leq i \leq n-1$ and $k \neq i$)	$I_j = 0$ $\forall$ $1 \leq j \leq n$	1
	$I_j = 0$ ($\forall$ $1 \leq j \leq i$ and $m < j \leq n$ $(i+1 \leq m \leq n-1)$) & $(I_{i+1} \parallel I_{i+2} \parallel \ldots \parallel I_m) = 1$	S-on (Stuck-on)
	For all other input combinations	0

Table 4: When D_i is connected to D_k ($1 \leq k \leq n-1$ and $K \neq i$)

- If D_i ($1 \leq i \leq n-1$) is shorted to D_n the output will correspond to Table 5.

In this case, the reason behind a stuck-on faulty output is similar to the explanation of stuck-on for Table 4.

Figure 7 illustrates this situation for NOR3 when D_1 is connected to D_3 ($n=3$). If a (010) input vector is applied, a conducting path from *VCC* to ground will go through Q_1, D_1D_3 and Q_5 whereas when a (001) input vector is applied, the conducting path will go through Q_1, D_1D_3 and Q_6. Thus, the output is stuck-on if the input of at least one transistor between two shorted drains is high.

The total number of faults that short two drains can be computed from equation (3.3.2.2).

$$(n-1)+(n-2)+\ldots+(n-(n-1))=\sum_{j=1}^{n-1}(n-j) \quad (3.3.2.2)$$

Short faults	Input conditions	Faulty output (O)
$D_i \leftrightarrow D_n$ $(1 \le i \le n-1)$	$I_i=0 \ \forall \ 1 \le j \le n$	1
	$I_j = 0 \ (\forall \ 1 \le j \le i)$ & $(I_{i+1} \| I_{i+2} \| \ldots \| I_n)=1$	S-on (Stuck-on)
	For all other input combinations	0

Table 5: When D_i ($1 \le i \le n-1$) is shorted to D_n

5) Short between one input and the output ($I_i \leftrightarrow O$ where($1 \le i \le n$))

This fault was discussed in 3), where inputs are shorted to D_n.

All five types of short faults have been covered in detail. The total number of short faults from the first to the last category is:

$$2\sum_{j=1}^{n-1}(n-j)+n^2+2n-2 \qquad (3.3.2.3)$$

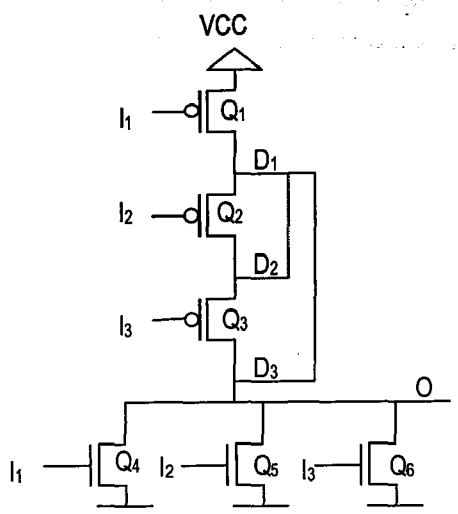


Figure 7: Faulty NOR3 Transistor Level

3.3.3 Open Faults:

Open faults occur after breaking at least one connection (channel) in the circuit. The source of the open state is a logic fault, which prevents the channel from conducting [5]. Under these circumstances every node in the circuit must be considered separately, since each node contributes to a different output result in the fault analysis. For instance, in the case of NORn, I_i, I_{iN} and I_{iP} as well as D_n, $D_{n+1}....D_{2n}$ and O are all separate nodes.

Different types of open faults that can occur in the circuit are classified as follows:

1) I_{iP} $(1 \leq i \leq n)$*:* Open

When I_{iP} is open the logical fault type, which represents the I_{iP} open physical failure, is equivalent to *VCC* stuck open. This results in I_{iP} stuck- at –1and $I_{iN} = I_i$ [5]. When I_{iP} is stuck-at-1 the output will be in a tri-state if all inputs are low. In this case all PMOS transistors except the one with I_{iP} input, which is stuck-at-1, are turned on and all NMOS transistors are off due to the low input values. The output has neither a path to *VCC* nor to ground. This is shown in Table 6.

The total number of faults for I_{iP} open fault is equal to n.

Open faults	Input conditions	Faulty output (O)
I_{iP} :Open ($1 \leq i \leq n$)	$I_j = 0 \ \forall \ 1 \leq j \leq n$	Z
	For all other input combinations	0

Table 6: I_{iP} Open

2) I_{iN} ($1 \leq i \leq n$): Open

The logical fault type of I_{iN} open fault is identical to I_i open fault which is similar to the situation where I_{iN} is stuck-at-0 and $I_{iP} = I_i$ [5]. In the I_{iN} stuck-at-0 condition, the output is high impedance if all NORn inputs except I_i are low. In this case, all PMOS transistors conduct except that with input I_i. Although I_i is high, its NMOS transistor (Q_{n+i}) is off since I_{iN} is stuck-at-0. Therefore, the output is neither connected to VCC nor ground. However, if I_i and the other inputs have low values the NORn output will be connected to the *VCC* and will go high. This is shown in Table 7.

The total number of faults for I_{iN} open fault is equal to n.

Open faults	Input conditions	Faulty output (O)
I_{iN} :Open ($1 \leq i \leq n$)	$I_j = 0 \ \forall \ 1 \leq j \leq n$	1
	$I_j = 0$ ($\forall \ 1 \leq j \leq n$ and $j \neq i$) & $I_i = 1$	Z
	For all other input combinations	0

Table 7: I_{iN} Open

3) I_i ($1 \leq i \leq n$): Open

I_i open fault acts similar to when I_{iN} is stuck-at-0 and $I_{iP} = I_i$ [5]. The NORn output results for I_i open fault is the same as I_{iN} open fault. The total number of faults in this case is also n.

4) *Di*: Open

D_i represents a drain belonging to NMOS or PMOS transistors. When ($1 \leq i \leq n$), D_i is a PMOS drain. The logical fault type D_i open fault is *VCC* stuck open, which can be replaced by I_{iP} stuck-at-1, $I_{iN} = I_i$ fault model. Therefore, the output will be the

same as I_{iP} open fault. D_i is a NMOS drain if $(n+1 \leq i \leq 2n)$ and the output result will be the same as an I_{iN} open fault.

The total number of faults in this category is $2n$.

5) *Si:* Open

If $(i = 1)$ the result of S_i open will be the same as I_{iP} open fault and if $(n+1 \leq i \leq 2n)$ then S_i open fault behaves like an I_{iN} open fault [5]. The PMOS sources, other than S_1, have not been considered since they are the same node as other PMOS drains which have been taken into account in the previous category.

The total number of faults is $2n+1$ for source open faults.

6) *O*: Open

A stuck open fault causes the output to be connected neither to power nor to ground. Therefore, the output is in a tri-state $(O = Z)$. The number of faults in this category is 1.

All open faults of the NORn gate have been analyzed and the total number of open faults from all six categories is $6n+2$.

The general formula for the total number of faults for NORn is concluded by adding the total number of faults from all groups of faults (stuck-at, short, and open):

$$2\sum_{j=1}^{n-1}(n-j)+n^2+10n+2 \qquad (3.3.3.1)$$

The patterns and formulas for the total number of faults for other primary gates can be calculated in the same manner. The final results for NAND, Inverter, D-latch, and D flip-flop gates are presented in the following sections. Fault list patterns for AND and OR gates can be concluded from NOR, NAND, and Inverter. The D flip-flop fault list can be concluded from the D-latch fault list.

3.4 NANDn Complete Fault List Pattern

Figure 7 presents a NANDn gate. The output pattern in the presence of faults is presented in three groups of faults in Table 8 for I/O stuck-at –faults, Table 9 for short faults and Table 10 and Table 11 for open faults.

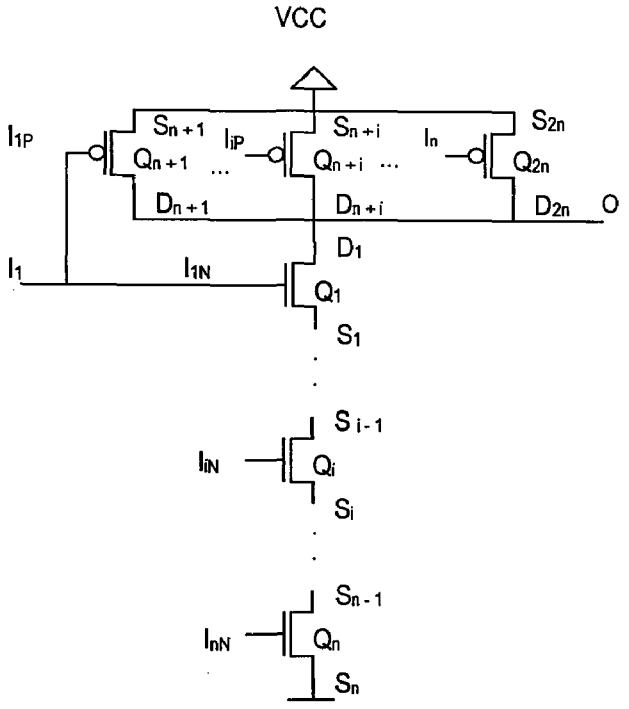


Figure 8: NANDn Transistor Level

Input/output stuck-at-faults	Faulty output (O)
I_i: stuck-at-0	1
I_i: stuck-at-1	NAND (I_1, I_2 ..., I_{i-1},I_{i+1},..., I_n)
O: stuck-at-0 (for any input combinations)	0
O: stuck-at-1 (for any input combinations)	1

Table 8: NANDn Input/Output Stuck-at-faults

The many types of short faults that may occur in a NANDn circuit are categorized as follow:

1) Short between a NMOS source and ground or power

2) Short between two inputs

3) Short between one input and one NMOS source

4) Short between two NMOS sources

5) Short between one NMOS source and the output

6) Short between one input and the output

Short faults	Input conditions	Faulty output (O)
$S_i \leftrightarrow$ G (ground) $(1 \le i \le n-1)$	$I_j=1 \ \forall \ 1 \le j \le n$	0
	$I_j=1 \ (\forall \ 1 \le j \le i)$ & $(I_{i+1} \& I_{i+2} \& \ldots \& I_n)= 0$	S-On
	For all other input combinations	1
$S_i \leftrightarrow$ VCC $(1 \le i \le n-1)$	For any input combination	1
$I_i \leftrightarrow I_k$ $(1 \le k \le n$ and $k \neq i)$	$I_j=1 \ \forall \ 1 \le j \le n$	0
	$I_j=1 \ (\forall \ 1 \le j \le n$ and $j \neq i, j \neq k)$ & $I_i \oplus I_k=1$	X
	For all other input combinations	1
$I_i \leftrightarrow S_k$ $(i \le k \le n-1)$	For any input combination	1
$I_i \leftrightarrow S_k$ $(1 \le k \le i-1)$ $(1 \le i \le n)$	$I_j=1 \ \forall \ 1 \le j \le k$ & $I_j = <0\|1> \ \forall \ k+1 \le j \le n$	I_i
	For all other input combinations	1
$S_i \leftrightarrow S_k$ $(1 \le k \le n-1$ $1 \le i \le n-1$ and $k \neq i)$:	$I_j=1 \ \forall \ 1 \le j \le n$	0
	$I_j= 1 \ (\forall \ 1 \le j \le i$ and $m < j \le n \ (i+1 \le m \le n-1))$ & $(I_{i+1} \& I_{i+2} \& \ldots \& I_m) = 0$	S-on
	For all other input combinations	1
$S_i \leftrightarrow$ O $(1 \le i \le n-1)$	$I_j=1 \ \forall \ 1 \le j \le n$	0
	$I_j= 1 \ \forall \ (i+1 \le j \le n)$ & $(I_1 \& I_2 \& \ldots \& I_i)= 0$	S-on
	For all other input combinations	1
$I_i \leftrightarrow$ O $(1 \le i \le n)$	For any input combination	I_i

Table 9: NANDn Short Faults

Many open faults turn out to have a similar logical fault type. Table 10 puts these faults into logical fault categories and Table 11 shows the output result for each category.

Open faults	Logical fault type
I_{iP} : open ($1 \le i \le n$)	I_{iP}: stuck-at-1
S_i: open ($n+1 \le i \le 2n$)	
D_i: open ($n+1 \le i \le 2n$)	
I_i : open ($1 \le i \le n$)	
I_{iN} : open ($1 \le i \le n$)	I_{iN}: stuck-at-0
S_i: open ($1 \le i \le n$)	
D_i: open ($i = 1$)	
O: open	Z

Table 10: NANDn Open Faults Categories

Logical fault type of open faults	Input conditions	Faulty output (O)
I_{iP}: stuck-at-1	$I_j = 1 \ \forall \ 1 \le j \le n$	0
	$I_j = 1$ ($\forall \ 1 \le j \le n$ and $j \ne i$) & $I_i = 0$	Z
	For all other input combinations	1
I_{iN}: stuck-at-0	$I_j = 1 \ \forall \ 1 \le j \le n$	Z
	For all other input combinations	1
Z (O: open)	For any input combination	Z

Table 11: NANDn Open Faults

The general formula for the total number of faults for NANDn is:

$$\sum_{j=1}^{n-1}(n-j)+\sum_{k=2}^{n-1}(n-k)+n^2+11n+1 \qquad (3.4.1)$$

The fault list patterns of OR and AND gates with n inputs can easily be concluded from NORn and NANDn. The complete tables of patterns for these gates are presented in the appendices.

Inverter, D-Latch, and D Flip-Flop have a specific number of inputs. Therefore, it is not appropriate to say they are following a specific fault list pattern due to their number of inputs. Each has only one complete fault list which will be presented in Sections 3.6 and 3.7.

3.5 Inverter Complete Fault List

The most important CMOS gate is the CMOS inverter. It consists of only two transistors, a pair of one N-type and one P-type transistor.

Figure 9 shows an inverter gate. The output pattern in the presence of faults is presented in three groups of faults in Table 12 for input / output stuck-at faults, Table 13 for short faults and Table 14 for open faults. The total number of faults for the inverter is: *13*

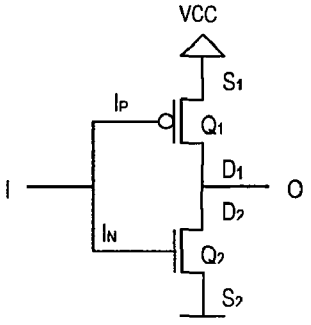


Figure 9: CMOS Inverter

Input/output stuck-at fault	Faulty output (O)
I: stuck-at-0	1
I: stuck-at-1	0
O: stuck-at-0	0
O: stuck-at-1	1

Table 12: Inverter Input / Output Stuck-at Faults

Short faults	Faulty output (O)
O ←→ I	I

Table 13: Inverter Short Faults

Open faults	Input conditions	Faulty output (O)
O: open	For any I	Z
I_P: open	I = 1	0
	I = 0	Z
I_N: open	I = 0	1
	I = 1	Z
I: open	N/A	Z
S_1: open	I = 1	0
	I = 0	Z
S_2: open	I = 0	1
	I = 1	Z
D_1: open	I = 1	0
	I = 0	Z
D_2: open	I = 0	1
	I = 1	Z

Table 14: Inverter Open Faults

3.6 D-Latch Complete Fault List

In CMOS technology, T-gates allow efficient realizations of several important logical functions [9]. Such a circuit consisting of one N-type and one P-type transistor connected in parallel and controlled by inverted gate voltages is shown in Figure 10. This circuit is defined as a transmission gate (T-gate) circuit. If the gate voltage of the N-type transistor is 'GND' and the P-type transistor has a gate voltage of 'VCC', both transistors are non-conducting. On the other hand, if the gate voltage of the N-type transistor is 'VCC' and the gate voltage of the P-type transistor is 'GND', both transistors are conducting. If the source voltage is near VCC, there is a voltage drop across the N-type transistor but (almost) no voltage drop across the P-type transistor. If the source voltage is near GND, the N-type transistor has (almost) no voltage drop. Due to the symmetry of standard MOS transistors, there is no reason to differentiate between source and drain in a T-gate. The contacts are therefore usually referred to as 'L' (left) and 'R' (right).

Figure 11 illustrates a D-latch by using only 8 transistors (2 inverters and 2 T-gates) for cases where both the clock (C) and the inverted clock (nC) signals are available. If the inverted clock is not available, an additional inverter is required to provide the control signal for the two T-gates [9].

A standard D-latch can be built from four 2-input NAND gates [8]. Thus, 16 transistors are needed for one D-latch. However, in this study we consider the D-latch circuit in Figure 11 in order to reduce the number of transistors leading to a lower number of possible faults. The case of a faulty inverter has been discussed in previous sections. Here, we assume that:

1. Both the clock and the inverted clock signal are available.
2. All signals are propagated up to the edge of the D-latch cell correctly.

The similar approach to NOR gate is applied to the D-latch to obtain its faulty output in the presence of a single fault. Later, the results from a D-latch fault list can be used to determine the fault list of a D flip-flop.

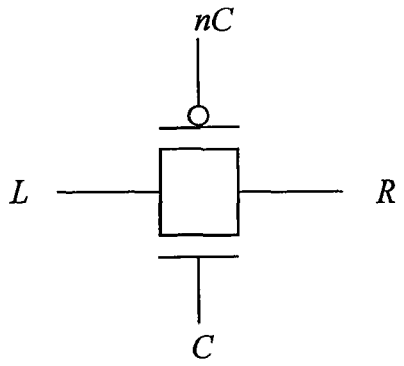


Figure 10: T-gate

3.6.1 Input / Output Stuck-at Faults

D, C (Clock), *nC* (inverted clock), and *Q* represent the input/output nodes of the D-latch in Figure 11. These nodes can be stuck-at-1 or stuck-at-0 leading to a faulty output. The results are shown in Table 15.

Q^- represents the previous state of output *Q*. When *D* is a stuck-at-fault, the output remains in its previous state as long as the clock is low. The output shows a faulty value for *D* when the clock goes high. When *C* and *Q* are stuck-at-faults the result of the faulty output can be easily computed. For example, when *C* is stuck-at-0 the feedback T-gate (T_2) is always conducting so the output remains in its previous state if *nC* is high. When *nC* is low and *C* is stuck-at-0 T_1 gate also conducts,

therefore the output has the value of D. When C is stuck-at-1 T_1 is always conducting and the value of D goes directly to the output.

T_2 conducts while nC is stuck-at-1. The output remains in its previous state if C is low. Once C becomes high, the T_1 gate conducts and the value of D goes to the Q. If nC is stuck-at-0 gate T_1 conducts and the output has the value of D.

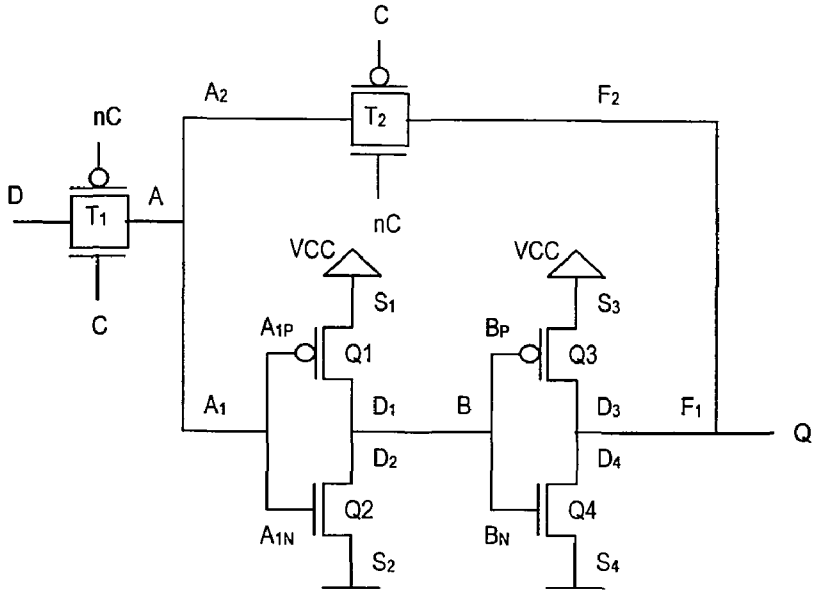


Figure 11: CMOS D-Latch

Stuck-at-fault	Input conditions	Faulty output (Q)
D: stuck-at-1	C = 0	Q^- (previous state)
	C = 1	1
D: stuck-at-0	C = 0	Q^-
	C = 1	0
C: stuck-at-1	N/A	D
C: stuck-at-0	nC = 1	Q^-
	nC = 0	D
Q: stuck-at-1	For any D and C	1
Q: stuck-at-0	For any D and C	0
nC: stuck-at-1	C = 1	D
	C = 0	Q^-
nC: stuck-at-0	C = 1	D
	C = 0	

Table 15: D-Latch Input / Output Stuck-at Faults

3.6.2 Short Faults

A short fault can occur at the connection of two nodes in a D-latch circuit. In this category of faults A, A_1, A_2, A_{1P} and A_{1N} are considered one node and are referred to as A because their behaviour is identical in the presence of short faults. Also, B, B_P, B_N, D_1 and D_2 are collectively referred to B and finally F_1, F_2 and Q are named Q. By these assumptions there are 6 nodes in the D-latch transistor-level circuit (D, C, nC, Q, A, B). Short faults can be categorized in two groups for the D-latch.

1) Short between each node and VCC or ground

The shorts between D, C, nC or Q and VCC or shorts between D, C, nC or Q and ground have not been considered here because their stuck-at-1 and stuck-at-0 have the same effect and they have already been discussed in Section 3.6.1. The results from connecting A and B to VCC or ground are presented in Table 16.

A and B have direct control over Q without depending on C or D. Therefore, when they are shorted to VCC or ground they change their output value independently.

Short faults	Input conditions	Faulty output (Q)
A ←→VCC	For any C and D	1
A ←→ G	For any C and D	0
B ←→ VCC	For any C and D	0
B ←→ G	For any C and D	1

Table 16: Short between Each Node and VCC or the Ground in D-latch

2) Short between two nodes

There are six nodes in the D-latch circuit D, C, nC, Q, A, B. If these nodes are connected to each other incorrectly, a faulty output will result. Table 17 shows the results that can be attained from a D-latch truth table.

A short between D and nC leads to different output states depending on their value. If D and nC are both low D is passed to the Q through T_1 gate. If these two signals are both high T_2 gate conducts and Q remains in its previous state. When D and nC have opposite values, the value of the short outcome is unknown. Therefore. the output is in an unknown state.

A short between D and A, B, or Q passes the value of D to these nodes asynchronously.

When C is connected to nC, one of the transistors of the T_1 and T_2 gates conducts and the output has the value of the D constantly. A short between C and Q forces the output to C value. A connection between C and A results in $Q = C$ for any value of D. When C is shorted to B, the output is $NOT(C)$ due to an inverter between nodes B and Q.

Any short between nC and Q, A, and B forces the value of nC to one of these nodes. When node A and B are shorted together one of the inverters is bypassed and the output Q is always inverted compared to a fault free D-latch. The same situation occurs when B is shorted to Q. When A is shorted to Q the output behaviour is the same as a fault free circuit.

Short faults	Input conditions	Faulty output (Q)
D ←→ C	D = C = 0 , (D and C are low)	Q^-
	D = C = 1 , (D and C are high)	D
	D = NOT(C), (D and C have opposite values)	X
D ←→ nC	D = nC = 0 , (D and nC are both low)	D
	D = nC =1 , (D and nC are both high)	Q^-
	D = NOT (nC) , (D and nC have opposite values)	X
D ←→ Q	For any C and Q^-	D
D ←→ A	For any C and Q^-	D
D ←→ B	For any C and Q^-	NOT (D)
C ←→ nC	For any C , D and Q^-	D
C ←→ Q	For any D and Q^-	C
C ←→ A	For any D	C
C ←→ B	For any D and Q^-	NOT (C) = nC
nC ←→ Q	For any C , D and Q^-	nC
nC ←→ A	For any C , D and Q^-	nC
nC ←→ B	For any C , D and Q^-	NOT(nC) = C
A ←→ B	C = 1	NOT (D)
	C = 0	NOT (Q^-)
A ←→ Q	C = 1	D
	C = 0	Q^-
B ←→ Q	C = 1	NOT (D)
	C = 0	NOT (Q^-)

Table 17: D-Latch Short Faults between Two Nodes

3.6.3 Open Faults

The open fault *D* causes a tri-state for the output when *C* is high and the gate T_1 is closed. When *C* is low the gate T_2 is switched on and causes the output to keep its previous state. Open faults *C* and *nC* do not affect the output behaviour since one T-gate is conducting at a time.

Open faults	Input conditions	Faulty output (Q)
D: open	C = 1	Z
	C = 0	Q^-
C: open	nC = 0	D
	nC = 1	Q^-
nC: open	C = 0	Q^-
	C = 1	D
A: open	C = 1	Z
	C = 0	Q^-
A_1: open	For any *D*, *C* and Q^-	Z
A_2: open	C = 1	D
	C = 0	Z
A_{1P}:open	(C = D =1) || (C = 0 & Q^- = 1) (*C* and *D* both high or *C* low and Q^- high)	1
	(C = 1 & D = 0) || (C = Q^- = 0)	Z
A_{1N}:open	(C = D = 1) || (C = 0 & Q^- = 1)	Z
	(C = 1 & D = 0) || (C = Q^- = 0)	0
B : open	For any *D*, *C* and Q^-	Z
B_P: open	(C = D = 1) || (C =0 & Q^- = 1)	Z
	(C = 1 & D = 0) || (C = Q^- = 0)	0
B_N: open	(C = D = 1) || (C = 0 & Q^- = 1)	1
	(C =1 & D = 0) || (C = Q^- = 0)	Z
F_1: open	For any *D*, *C* and Q^-	Z
F_2: open	C = 1	D
	C = 0	Z
Q: open	For any *D*, *C* and Q^-	Z
S_1: open	Like A_{1P}: open	
D_1: open	Like A_{1P}: open	
S_2: open	Like A_{1N}: open	
D_2: open	Like A_{1N}: open	
S_3: open	Like B_P: open	
D_3: open	Like B_P: open	
S_4: open	Like B_N: open	
D_4: open	Like B_N: open	

Table 18: D-Latch Open Faults

Open fault A has the same effect as open fault D. Open faults A_1, B and F_1 produce a tri-state output. When A_2 or F_2 is opened the output has the value of D when gate T_1 is switched on. The output goes to a tri-state when gate T_1 is off because the feedback circuit is disconnected. Once A_{1P} is opened the output is high if the input of the first inverter (transistors Q_1 and Q_2) is high. While the input of the first inverter is low the output goes to a tri-state due to disconnection of Q_1 PMOS transistor. The same approach is used when A_{1N} is opened. In this case Q_2 is disconnected and the low input value of the first inverter results in a low output. A high input value forced the output to a tri-state. The same conclusions can be drawn for open faults B_P and B_N. The total number of faults for the D-latch is *49*.

3.7 D Flip-Flop Complete Fault List

An edge sensitive D flip-flop can be made from two D-latches in master and slave mode [7]. Although some VLSI libraries indicate alternative configurations, the case study discussed here considers only this configuration. The same fault analysis can be extended to a custom D flip-flop library. The fault list for a D flip-flop can be easily concluded from a D-latch fault list. Figure 12 shows a D flip-flop circuit. In this figure the inverters are shown with gate formats. The faults related to the inverters have been discussed in the D-latch and inverter sections. Therefore, in this section we assume that all of the inverter gates are fault free. All the faults related to a circuit's nodes are taken into account in this section.

To consider the edge sensitivity of a D flip-flop, we divide the circuit into two level sensitive D-latches. In the presence of one fault, the output B of first D-latch *(DL1)* is first computed and then the second D-latch *(DL2)* output Q is studied. Table 19 illustrates the result of the fault free D flip-flop for Figure 12. This approach shows that the D flip-flop fault list can be determined by following the D-latch fault list pattern.

Input conditions	Fault Free Q	Fault Free B
C = 1	Q^- (previous state)	NOT (D)
C = 0	NOT (B)	B^- (previous state)

Table 19: Fault Free D Flip-Flop

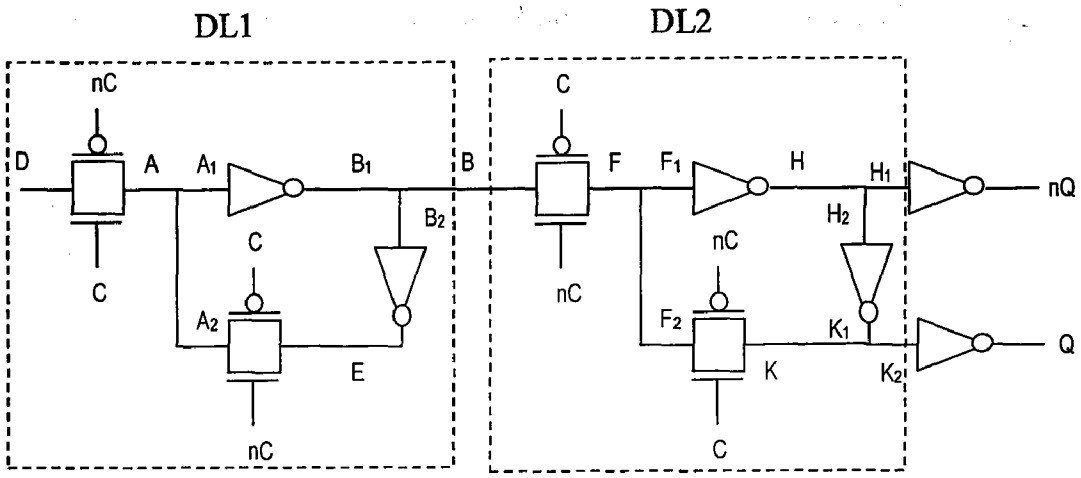


Figure 12: D Flip-Flop

3.7.1 Input / Output Stuck-at Faults

D, C, nC, Q and *nQ* are the inputs and outputs of D flip-flop. Table 20 shows the output results in the presence of a single stuck at fault.

Stuck-at-fault	Input conditions	Faulty B	Faulty output (Q)
D: stuck-at-1	C = 1	0	Q^-
	C = 0	B^-	NOT(B)
D: stuck-at-0	C = 1	1	Q^-
	C = 0	B^-	NOT(B)
C: stuck-at-1	nC = 0	NOT(D)	Q^-
	nC = 1		NOT(B)
C: stuck-at-0	nC = 0	NOT(D)	NOT(B)
	nC = 1	B^-	
Q: stuck-at-1	C = 1	NOT(D)	1
	C = 0	B^-	1
Q: stuck-at-0	C = 1	NOT(D)	0
	C = 0	B^-	0
nQ: stuck-at-1	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT(B)
nQ: stuck-at-0	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT(B)
nC: stuck-at-1	C = 1	D	NOT(B)
	C = 0	B^-	
nC: stuck-at-0	C = 1	NOT(D)	Q^-
	C = 0		NOT(B)

Table 20: D Flip-Flop Input / Output Stuck-at Faults

3.7.2 Short Faults

To examine short faults for a D flip-flop, some nodes are considered as one node due to their similar behaviour in the presence of a short fault. Therefore, A, A_1, and A_2 are addressed as A. B, B_1, and B_2 are addressed as B. F represents F, F_1 and F_2. H represents H, H_1, and H_2 and finally K, K_1, and K_2 are addressed as K in short faults study. Table 21 shows all the possible short faults in a D Flip-Flop.

Short faults	Input conditions	Faulty B	Faulty output (Q)
A ←→ VCC	C = 1	0	Q^-
	C = 0		1
A ←→ G	C = 1	1	Q^-
	C = 0		0
E ←→ VCC	C = 1	NOT(D)	Q^-
	C = 0	0	1
E ←→ G	C = 1	NOT(D)	Q^-
	C = 0	1	0
B ←→ VCC	C = 1	1	Q^-
	C = 0		0
B ←→ G	C = 1	0	Q^-
	C = 0		1
F ←→ VCC	C = 1	NOT(D)	0
	C = 0	B^-	
F ←→ G	C = 1	NOT(D)	1
	C = 0	B^-	
H ←→ VCC	C = 1	NOT(D)	1
	C = 0	B^-	
H ←→ G	C = 1	NOT(D)	0
	C = 0	B^-	
K ←→ VCC	C = 1	NOT(D)	0
	C = 0	B^-	
K ←→ G	C = 1	NOT(D)	1
	C = 0	B^-	
D ←→ C	D = C = 0	B^-	NOT(B)
	D = C = 1	D	Q^-
	D = NOT(C)	X	X
D ←→ nC	D = nC = 0	D	Q^-
	D = nC = 1	B^-	NOT(B)
	D = NOT(nC)	X	X
D ←→ Q	C = 1	NOT(D)	D
	C = 0	B^-	
D ←→ nQ	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
D ←→ A	C = 1	NOT(D)	Q^-
	C = 0		D
D ←→ E	C = 1	NOT(D)	Q^-
	C = 0		D
D ←→ B	C = 1	D	Q^-

	C = 0		NOT(D)
D ←→ F	C = 1	NOT(D)	NOT(D)
	C = 0	B^-	
D ←→ H	C = 1	NOT(D)	D
	C = 0	B^-	
D ←→ K	C = 1	NOT(D)	NOT(D)
	C = 0	B^-	
C ←→ nC	For any *C, nC*	NOT(D)	NOT(B)
C ←→ Q	C = 1	NOT(D)	C
	C = 0	B^-	
C ←→ nQ	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
C ←→ A	C = 1	NOT(C)	Q^-
	C = 0		C
C ←→ E	C = 1	NOT(D)	Q^-
	C = 0	1	0
C ←→ B	C = 1	C	Q^-
	C = 0		1
C ←→ F	C = 1	NOT(D)	nC
	C = 0	B^-	
C ←→ H	C = 1	NOT(D)	C
	C = 0	B^-	
C ←→ K	C = 1	NOT(D)	nC
	C = 0	B^-	
nC ←→ Q	C = 1	NOT(D)	nC
	C = 0	B^-	
nC ←→ nQ	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nC ←→ A	C = 1	NOT(nC) = C	Q^-
	C = 0		NOT (B)
nC ←→ E	C = 1	NOT(D)	Q^-
	C = 0	0	1
nC ←→ B	C = 1	nC	Q^-
	C = 0		0
nC ←→ F	C = 1	NOT(D)	C
	C = 0	B^-	
nC ←→ H	C = 1	NOT(D)	nC
	C = 0	B^-	
nC ←→ K	C = 1	NOT(D)	C
	C = 0	B^-	
Q ←→ nQ	C = 1	NOT(D)	X
	C = 0	B^-	
Q ←→ A	C = 1	NOT(D)	D
	C = 0	B^-	NOT (B)
Q ←→ E	C = 1	NOT(D)	D
	C = 0	B^-	NOT (B)
Q ←→ B	C = 1	NOT(D)	B
	C = 0	B^-	
Q ←→ F	C = 1	NOT(D)	B
	C = 0	B^-	
Q ←→ H	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)

Q ←→ K	C = 1	NOT(D)	NOT(Q^-)
	C = 0	B^-	B
nQ ←→ A	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nQ ←→ E	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nQ ←→ B	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nQ ←→ F	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nQ ←→ H	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
nQ ←→ K	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
A ←→ E	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
A ←→ B	C = 1	D	Q^-
	C = 0	NOT(B^-)	NOT (B)
A ←→ F	C = 1	NOT(D)	NOT(D)
	C = 0	B^-	X
A ←→ H	C = 1	NOT(D)	D
	C = 0	B^-	NOT (B)
A ←→ K	C = 1	NOT(D)	NOT(D)
	C = 0	B^-	X
E ←→ B	C = 1	NOT(D)	Q^-
	C = 0	NOT(B^-)	B
E ←→ F	C = 1	NOT(D)	D
	C = 0	B^-	X
E ←→ H	C = 1	NOT(D)	NOT (B)
	C = 0	B^-	
E ←→ K	C = 1	NOT(D)	D
	C = 0	B^-	X
B ←→ F	C = 1	NOT(D)	NOT (B)
	C = 0	B^-	
B ←→ H	C = 1	NOT(D)	B
	C = 0	B^-	
B ←→ K	C = 1	NOT(D)	NOT (B)
	C = 0	B^-	
F ←→ H	C = 1	NOT(D)	NOT(Q^-)
	C = 0	B^-	B
F ←→ K	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT (B)
H ←→ K	C = 1	NOT(D)	NOT(Q^-)
	C = 0	B^-	B

Table 21: D Flip-Flop Short Faults

3.7.3 Open Faults

Open faults	Input conditions	Faulty B	Faulty output (Q)
D : open	For any *C*	Z	Z
C : open	nC = 1	B^-	NOT(B)
	nC = 0	NOT(D)	Q^-
nC : open	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT(B)
Q : open	For any *C*	Z	Z
nQ : open	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT(B)
A: open	C = 1	Z	Q^-
	C = 0	B^-	NOT(B)
A1: open	For any *C*	Z	Z
A2: open	C = 1	NOT(D)	Q^-
	C = 0	Z	NOT(B)
E: open	C = 1	NOT(D)	Q^-
	C = 0	Z	NOT(B)
B: open	For any *C*	Z	Z
B1: open	For any *C*	Z	Z
B2: open	C = 1	NOT(D)	Q^-
	C = 0	Z	NOT(B)
F: open	C = 1	NOT(D)	Z
	C = 0	B^-	
F1: open	C = 1	NOT(D)	Z
	C = 0	B^-	
F2; open	C = 1	NOT(D)	Z
	C = 0	B^-	NOT(B)
H: open	C = 1	NOT(D)	Z
	C = 0	B^-	
H1: open	C = 1	NOT(D)	Q^-
	C = 0	B^-	NOT(B)
H2: open	C = 1	NOT(D)	Z
	C = 0	B^-	
K: open	C = 1	NOT(D)	Z
	C = 0	B^-	NOT(B)
K1:open	C = 1	NOT(D)	Z
	C = 0	B^-	
K2: open	C = 1	NOT(D)	Z
	C = 0	B^-	

Table 22: D Flip-Flop Open Faults

The total number of faults for the D Flip-Flop is *98.*

Chapter 4

User-Defined VHDL Library of Transistor Level Faults

In a structural level of abstraction a system closely corresponds to the actual hardware and thus can easily be understood and used by a hardware designer. To describe a system at the structural level, the components of that system are listed and the interconnections between them are specified. A term often used to describe this form of description is netlist.

VHDL provides language constructs for concurrent instantiation of components, the primary constructs for structural specification of hardware [10].

A complete transistor mapping to a gate level fault list of each primary gate was presented in the previous chapter. In the presence of a fault from a gate fault list, the changes in gate function lead to a different output. All possible gate functions in the presence of faults are discussed in Chapter 3. These functions can be presented in VHDL format describing the behaviour of the faulty gates in a synthesizable hardware language.

4.1 VHDL Presentation of Faulty Gates

Each gate has a specific number of faults which can be calculated from the general formula obtained for each type of gate in Chapter 3. Table 23 shows the results when $2 \leq n \leq 5$. The upper bound for the number of inputs is five due to the fact that the selected benchmarks used in our case study were restricted to a maximum of five inputs.

Gate Name	Total number of faults
NOR2	28
NOR3	47
NOR4	70
NOR5	97
NAND2	28
NAND3	47
NAND4	70
NAND5	97
AND2	41
AND3	62
AND4	87
AND5	116
OR2	39
OR3	60
OR4	85
OR5	114
Inverter	13
Buffer	24
D-latch	49
D Flip-Flop	98
Total =	1272

Table 23: Total Number of Faults for Each Gate

The total number of necessary VHDL codes to represent each fault in a gate can be extracted from Table 23. For example, to represent all possible faults in a NOR2 gate, 28 VHDL codes are needed. For instance, a short fault between D_1 and I_1 makes the NOR2 gate (Figure 3) behaves differently. The difference of behaviour between a faulty NOR and a fault free NOR gate can be observed at the output. Thus, the output will be:

$$O = \begin{cases} I_1 & if \ I_2 = 0 \\ 0 & if \ I_2 = 1 \end{cases} \qquad (4.1)$$

This output can be presented by a synthesizable VHDL code shown in Figure 13:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY NOR2_short_fault is
  port ( I1,  I2  : IN  STD_LOGIC;
         O        : OUT  STD_LOGIC);
END NOR2_short_fault;

ARCHITECTURE faulty_module of NOR2_short_fault IS
 SIGNAL A: STD_LOGIC;
BEGIN
   -- I1 is shorted to D1
   A <= I2;
   O <= I1 WHEN A = '0' ELSE '0';
END faulty_module;
```

Figure 13: NOR2_Short_Fault Synthesizable VHDL code

The open node, I_{1P}, can be an example of an open fault. In this case the behaviour of the faulty output will be:

$$O = \begin{cases} Z & if \ I_1 = I_2 = 0 \\ 0 & else \end{cases} \qquad (4.2)$$

This faulty output is presented by the synthesizable VHDL code in Figure 14.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

 ENTITY NOR2_open_fault is
 port(  I1,  I2  : IN  STD_LOGIC;
        O        : OUT  STD_LOGIC);
 END NOR2_open_fault;
```

```
ARCHITECTURE faulty_module of NOR2_open_fault IS
 SIGNAL A: STD_LOGIC;
BEGIN
 -- I1P is open
 A <= I1 or I2;
 O <= '0' WHEN A = '1' ELSE 'Z';
 END faulty_module;
```

Figure 14: NOR2_open_fault synthesizable VHDL code

The above codes can be considered as components in a circuit specified by structural (gate level) VHDL code. The next section discusses how to automatically create faulty VHDL components.

4.2 Creating Faulty VHDL Components Automatically

As shown in Table 23, 1272 VHDL codes will be needed to create all faulty components of primitive gates. It is neither cost-efficient nor error-free to write a large number of VHDL modules manually especially when modifications may be required. Therefore, the process of creating faulty VHDL modules must be automated. For this purpose we will use Perl as our scripting language. In the Perl automated program, the patterns of each gate obtained from the tables in Chapter 3 are used and the proper VHDL components are generated. In general, the program runs as follows:

Step 1: Write a FOR loop for changing the number of input, *n*, from 2 to 5 for a specific type of gate.

Step 2: Define an array, called @fault_number[1]. Each member of this array shows the total number of faults for each group of faults which belongs to one of the rows of fault pattern tables defined in Chapter 3. For example, the total number of input stuck-at-1 in NORn is *n*. Therefore, its related member of the array, $fault_number[i][2], has the value of *n*.

[1] An array is defined by @ character in Perl language.
[2] A scalar is defined by $ character in Perl language.

Step 3: Write a general subroutine, called subroutine *entityvhdl*, to create a VHDL file in a desired directory. In this subroutine the entity part of the VHDL code is written with regard to the value of *n*.

Step 4: Write specific subroutines which will represent each row of fault patterns in the tables. For example, the subroutine of input stuck-at-1 in NORn assigns a low value to the output. These subroutines create the architectural body of VHDL module based on their related fault patterns.

Table 24 shows the total number of Step 4 subroutines for each type of gate.

Step 5: Write a FOR loop for each row of tables. Number of iteration of the loop depends on the value of $fault_number[i] from Step 2. In this loop, subroutine *entityvhdl* and one of the subroutines of Step 4 are called and the related VHDL component is created.

Figure 15 shows a section of the program for input stuck-at-1 in NORn.

```
for($n=2;$n<6;$n++)                    #[1] FOR loop of step 1
{
 $fault_number[0]= $n;                 # step 2

 for($m=1;$m<=$fault_number[0];$m++)

                                       # input stuck-at-1
 {
     &entityvhdl($n,$m);               # calling step 3 subroutine

     &inputstuck1fault;                # calling step 4 subroutine
                                       # for input stuck-at-1
  }
}
##########################################
# Subroutines
##########################################

sub entityvhdl
 {
  $underline="_F";
  $component_name="NOR$n$underline$m";
  $file_name= "NOR$n$underline$m.vhd";

  $first_part_VHDL_body=" LIBRARY ieee;\n
```

[1] Comments are specified with # in Perl

```
                                USE ieee.std_logic_1164.all;\n\n
                                ENTITY $component_name is\n port(";

  open (FILE, ">$file_name");
  print FILE "$first_part_VHDL_body\n";
  @inputs= &inputgenerator($n);

  print FILE "@inputs : IN  STD_LOGIC;\n
              O : OUT  STD_LOGIC);\n
              END $component_name;\n
              ARCHITECTURE faulty of $component_name IS\n ";

  }
##############################################

   sub inputgenerator                    # defining inputs for
    {                                    # entityvhdl subroutine
     my @INPUT;
     my $x;
     for ($x=1; $x<=($n-1); $x++)
     {
      $INPUT[$x]="I$x, ";
     }
     $INPUT[$n]= "I$n ";
     return @INPUT;

    }
################################################

   sub inputstuck1fault                    # step 4 subroutine
    {
     print FILE "BEGIN \n
                  --one input is stuck-at-1\n
                  O <= '0';\n
                END faulty; ";
    }
```

Figure 15: Perl Program Section for NORn Input stuck-at-1 Fault

Gate Type	Total number of Step 4 subroutines
NOR	12
NAND	14
OR	18
AND	21
D-Latch	41
D Flip-Flop	76
Inverter	8
Buffer	14

Table 24: Total Number of Subroutines

4.3 Designing a User-defined Library

Logic families or group of components can be categorized according to their physical characteristic, price, complexity, usage, or other properties. The VHDL language supports the use of design libraries for categorizing component or utilities. In general, libraries are used for design organization. Specific applications of libraries include sharing components between designers, grouping components of standard logic families, and categorizing special purpose utilities such as subprograms or types.

4.3.1 Existing Libraries

Predefined libraries in VHDL are the STD and the WORK libraries. The STD library contains all the standard types and utilities and is visible to all designs. The WORK library is simply a name that refers to the current working library. When a VHDL environment is created for a user, the keyword WORK refers to the root library of the user. As new libraries are created, the user can designate a new default library by equating one of the libraries to the WORK library [10].

4.3.2 Library management

Library management tasks, such as the creation or deletion of a library or aliasing it to WORK, are not part of the VHDL language. These tasks are done outside of

VHDL and depend on the specific tool which, in this research, is the Xilinx ISE CAD tool. The use of a library, however, is supported by VHDL. The LIBRARY keyword followed by the name of a library makes it visible to a design. The following statement is assumed by all designs:

```
LIBRARY WORK;
```

4.3.3 Aliasing the faulty components to WORK library

The faulty VHDL components, created automatically with respect to each gate type pattern, can be added to the WORK library so that it is visible to all circuit VHDL codes. The components can be added to the WORK library manually by using GUI format of Xilinx ISE tool or by using ISE command line. The number of VHDL components is 1272. Adding all components to WORK library is inefficient. Only necessary components must be considered. Therefore, based on the circuit specification, only some components are added to WORK library. This job can be done automatically by knowing the component list of each circuit and ISE command line in Perl language.

For example, we considered the benchmark C17, a combinatorial circuit. Then we injected a fault to the code by replacing the NAND2 module by a faulty module called NAND2_F1 shown in Figure 16. There are only two gates used in this code therefore, only NAND2.vhd and NAND2_F1.vhd are added to the WORK library. To automate this process, Perl program gets two lists: the component list of a specific circuit and the user fault list. By using these two lists, it opens a new file, called project files (*.prj*), and associates all fault free components of the component list and faulty components of the user fault list to WORK library. When the ISE synthesizer tool (XST) is synthesizing the circuit VHDL code it will look at all components listed in the project file and then synthesize them. Figure 17 shows the project file related to the fault injected benchmark C17.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity c17 is
   port( PI1, PI2, PI3, PI4, PI5 : in std_logic;
         PO1, PO2 : out std_logic);
end c17;
architecture STRUCTURE of c17 is
   signal G10, G11, G16, G19 : std_logic;
begin
 G10_NAND2:NAND2 port map(E1=>PI1,E2=>PI3,A=>G10);
 // fault is injected in G11_NAND2 gate
 G11_NAND2:NAND2_F1 port map(E1=>PI3,E2=>PI4,A=> G11);
 G16_NAND2:NAND2 port map(E1=>PI2,E2=>G11,A=> G16);
 G19_NAND2:NAND2 port map(E1=>G11,E2=>PI5,A=> G19);
 G22_NAND2:NAND2 port map(E1=>G10,E2=>G16,A=> PO1);
 G23_NAND2:NAND2 port map(E1=>G16,E2=>G19,A=> PO2);
end STRUCTURE;
```

Figure 16: C17.vhd

```
vhdl work ../faultycomponent/NAND2.vhd
vhdl work ../faultycomponent/NAND2_F1.vhd
vhdl work c17.vhd
```

Figure 17: C17.prj

Once the related VHDL components are added to WORK library, the user-defined faulty component library will be ready for use. In the next chapter we will discuss how to use the faulty components in the VHDL codes of circuits as well as the method of injecting faults.

Chapter 5

Fault Injection into Circuit VHDL Description

A circuit can be modeled in several manners according to the desired abstraction level (e.g. architectural, logic, geometric), to the wanted view (e.g. behavioural, structural, physical) and to the modeling method being used (e.g. language, diagram, mathematical model).

In recent years, there has been a trend towards using hardware description languages (HDLs) for circuit specification. The conciseness of HDL models has made them preferable to the corresponding flow, state and logic diagram, even though some diagram models are more powerful in visualizing the circuits' functions. Circuit specifications shall not be described in terms of diagrams, because the information that most of them convey can be expressed in equivalent form by HDL models [11].

VHDL is the VHSIC, i.e. Very High Speed Integrated Circuit, hardware description language. It can describe the behaviour and structure of electronic systems, but is particularly suited as a language to describe the structure and behaviour of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits [49].

This chapter will discuss behavioural and structural features of HDL models. It will also present different approaches of fault injection into both behavioural and structural HDL models based on their features. In this research we are using a structural VHDL description of a circuit for injecting faults. One reason is that the structural features of VHDL descriptio-

are compatible with our user-defined library introduced in Chapter 4. Section 5.3 will discuss our fault injection approach in more detail.

5.1 Behavioural Feature of Hardware Languages

The behavioural description deals with the system as if it were a kind of "black box" with its inputs and outputs, with no regard to its structure. The goal is to ignore the redundant details and to concentrate on the necessary function specifications.

Behavioural modeling for circuits is considered in increasing levels of complexity. Combinational logic circuits can be described by a set of ports (input / output) and a set of equations that relate variables to logic expressions. The declarative paradigm applies best to combinational circuits, which are by definition memory-less. Indeed, they can be seen as an interconnection (i.e. a structure) of operators, each operator evaluating a logic function. These models differ from structural models in that there is not a one-to-one correspondence between expressions and logic gates, as a single gate may not exist for some expressions for implementation [11]. Figure 18 shows the half-adder circuit in the VHDL language, using its behavioural modeling capability.

```
Architecture BEHAVIOUR of HALF_ADDER is
Process
        Begin
                Carry <= (a and b) ;
                Sum   <= (a xor b) ;
        End process;
End BEHAVIOUR;
```

Figure 18: Half-adder Behavioural Model

5.2 Structural Feature of Hardware Languages

The structural description defines the way that the system is to be built up. The focus is on the blocks and how they interact with each other to form a system's structure. The subsystems, which are to provide its functional execution, are defined somewhere else.

Structural languages models describe an interconnection of components. Hence, their expressive power is similar to that of circuit schematics, even though specific language constructs can provide more powerful abstractions. Hierarchy is often used to make the description modular and compact. The basic features of structural languages place them close to the declarative class, even though some structural languages also have procedural features. Variables in the language correspond to ports of components [11]. Figure 19 shows the half-adder circuit in the VHDL language, using its structural modeling capability.

```
Architecture STRUCTURE of HALF_ADDER is

        component AND2
                Port (x, y: in bit; o: out bit);
        End component;

        component XOR2
                Port (x, y: in bit; o: out bit);
        End component;

        Begin
            G1: AND2 port map (a, b, carry);
            G2: XOR2 port map (a, b, sum) ;
End STRUCTURE;
```

Figure 19: Half-adder Structural Model

The model contains two declarations of other models, AND2 and XOR2, as well as two instantiations of the models, called G1 and G2. Additional information on the components AND2 and XOR2 is provided elsewhere.

In this research the structural VHDL model of some combinational and structural benchmark circuits are used. A fault can easily be injected into the VHDL description of the circuit by just replacing one fault free component with a faulty one. All the fault free and faulty components are declared in a user-defined library which is added to WORK library of

the VHDL codes as explained in Chapter 4. As an example, the G1 instantiation in Figure 19 can be replaced by the following:

```
G1: AND2_F1 port map (a, b, carry);
```

Figure 20: Faulty AND2 Instantiation

The next section describes the method of injecting faults into VHDL description of a circuit and a solution will be proposed to limit the synthesis time and complexity along the way.

5.3 Fault Injection in VHDL Description

Due to the evolution of technology, the probability of faults occurring in the field of integrated circuits is noticeably increasing. Single Event Effects (SEEs) such as Single Event Upsets (SEUs) or Single Event Latchup, which are noticeable sources of failures in space applications, will become obvious sources of collapses even at the sea level, for the next generation of technology. The interest in integrated on-line fault detection mechanisms and/or fault tolerance is therefore rapidly increasing for circuits designed in deep sub-micron technologies. Therefore, there is an urgent need to evaluate circuit fault tolerance. A key issue in designing fault tolerant digital systems is the validation of the design with respect to the dependability requirements.

The final assessment of the circuit including system dependability and reliability is classically executed after manufacturing of the circuits using fault injection on a system prototype. If the preliminary analysis carried out during the circuit design was not complete, unacceptable behaviour can be identified only at that stage. The re-work of the circuit implies then very high costs and a lot of wasted time. It becomes therefore crucial, especially with the advent of systems-on-chip, to perform a thorough analysis of the failure modes of the circuit as early in the design process as possible and at least before any manufacturing. Classical injection techniques (e.g. pin-level fault injection, memory corruption, heavy-ion injection, power supply disturbances, laser fault injection or software fault injection) all apply on a fabricated circuit and cannot be used in this context [12]. By contrast, some

studies have proven that performing fault injection in high-level models of the circuit can be a practical approach to an early analysis of faulty behaviours [13, 14].

More recently, several researchers proposed the application of fault injection early in the design process. The main approach consists in injecting the faults in high level descriptions (most often, VHDL models) of the circuit or system. Delong et al. [18] described, for example, the injection of faults in behavioural VHDL descriptions of microprocessor-based systems. The approach uses instruction-level models of the processors. During simulation stuck-at faults are injected in the memory or in the processor registers, through a fault injection controller that is a module added to the initial description of the system. The injection process relies on specific data types and on the bus resolution functions; such an approach is therefore difficult to generalize for architectures that do not use tri-state buses.

Another approach presented in [19], and then [13] or [20], is more general and considers the injection of different types of faults in the VHDL model of a circuit at several abstraction levels and using various techniques based on the modification of the initial VHDL description or on the use of simulation primitives. As mentioned in [16], the main drawback related to the use of simulation is the huge amount of time required to run the experiments when many faults have to be injected in a complex circuit.

In-system emulation using hardware prototyping on FPGA-based logic emulation systems has also been proposed to consider the effects of the circuit environment in the application [15]. Another advantage expected from such emulation is to noticeably reduce the time needed for the fault injection experiments when compared to simulations [16]. Therefore, based on the discussion above and in Section 5.2, we have considered in this work fault injection into structural models written in VHDL, and FPGA-based hardware prototyping for early failure mode analysis of a complex digital circuit.

When an FPGA-based emulator is used, the initial VHDL description must be synthesizable. In some cases, the approaches developed for fault grading using emulators (e.g. [21, 22]) may be used to inject faults. However, such approaches are classically limited to permanent stuck-at fault injection. Several specific approaches have therefore been recently considered [23-28]. In most cases, modifications are introduced in the circuit description.

In the structural fault injection technique, a fault model was provided by replacing one module with another. The replacement module is a predefined or user-defined module. Thus, this fault model is general enough to cover many other fault models which affect the functionality of modules. If the replaced module is also synthesizable model synthesizability is not affected [17].

In this work the modules for replacement are generated by mapping transistor level faults into gate level format. In other words, the replacement modules are defined at the gate level user-defined library with a synthesizable VHDL format as shown in Chapter 4. The fault injection technique is based on replacing the fault free module in the initial system VHDL description with a faulty module from the user-defined library. Examples of this replacement are shown in Figure 16 and Figure 20.

Although the transistor level faults are injected in the initial structural VHDL description of the system by using faulty modules, the synthesis time is high for modifying large circuits. To alleviate this problem, a large circuit must be divided into small partitions. Then one partition is considered at a time where faults are injected. Therefore, synthesis and implementation time is reduced because all partitions are synthesized and implemented only once, whereas the partition with injected faults is re-synthesized and re-implemented as many times as there are faults. It should also be noted that the bitstreams of the fault free partitions are generated and downloaded into FPGA only once. The bitstream of the faulty partition must be generated and downloaded into FPGA several times depending on the number of faults to be injected into the FPGA.

In the next chapter we will introduce our method of partitioning which leads to a reduction in design synthesis time and complexity.

Chapter 6

Binary Tree-based Partitioning Methodology

Partitioning is a common method for reducing the design complexity of a system. As system complexity increases, the timing problem plays an important role in the whole design process. The goal of partitioning is to divide a complex system into small subsystems subject to balance constraints while minimizing interconnections among subsystems. Partitioning optimizations are critical to the synthesis of large-scale VLSI systems.

In this chapter the advantage of using unbalanced partitioning over balanced partitioning will be discussed. Then, the methodology to apply a full binary tree to perform an unbalanced partitioning of a circuit VHDL description will be described.

6.1 Balanced Versus Unbalanced Partitioning

A system has a balanced partitioning if all partitions are of nearly equal size, whereas unbalanced partitioning can be defined as a system with different sized partitions.

The primary focus of this research is to reduce the synthesis time while injecting different types of faults into FPGAs. To reach this goal the system is divided into small partitions where faults are injected in the desired locations. Thus, fault free partitions are synthesized once while the faulty partition is re-synthesized after modifying its structural VHDL description to inject different faults. Thus, if the faulty partition size is reduced the total synthesis time for injecting faults into that partition is also reduced.

If, in a balanced partition circuit, N is the total number of partitions and G_T is the total number of gates in the circuit, the size of each partition $S(P_i)$ is:

$$S(P_i) = G_T / N \quad \text{Where } N \geq 1 \text{ and } 1 \leq i \leq N \quad (6.1)$$

G is constant for a specific circuit. Therefore, to reduce the size of partitions the total number of partitions N must be increased. Figure 21 shows a balanced partitioned circuit with 5 partitions where faults are injected into P_4. Figure 22 shows the graph related to this type of partitioning. In this case:

$$S(P_1) = S(P_2) = S(P_3) = S(P_4) = S(P_5) = G_T/5 \quad (6.2)$$

In this research we also focus on module-based partial reconfiguration of FPGAs. Partial reconfiguration involves defining a distinct portion of an FPGA design to be reconfigured while the rest of the device remains in active operation. These portions are referred as reconfigurable modules. The partial reconfiguration flow utilizes a modified form of the modular design process [29]. We will expand more on this subject in the next chapter. In this work, the reconfigurable module is the partition where faults are to be injected. The lower the number of modules, the better the module-based partial reconfiguration performance will be. The reason is that the complexity of the design process will increase due to following factors: complexity of defining bus macros[1], having more constraints (local and global clock routing), difficult routing caused by the fixed location of bus macros, and wasting device resources when sizes of modules are small. These factors are further explained in Section 7.1.

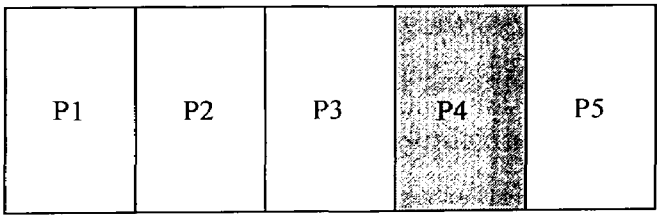


Figure 21: Balanced Partitioning

Clearly, when using the balanced partitioning method there is a conflict between reducing synthesis time and reducing the number of partitions. The suggested method to solve this

[1] Bus macros are used as fixed data paths for signal exchange between a reconfigurable module and other modules.

problem is to divide the system into an unbalanced subsystem by applying full binary tree methodology which uses the iterative genetic algorithm with minimum ratio cut. This algorithm will be discussed in Section 6.3.

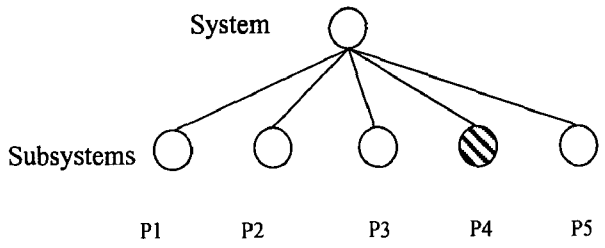


Figure 22: Balanced Partitioning Graph

Figure 23 shows an unbalanced partitioning for the total number of 5 partitions. First, the circuit is divided into two equal partitions. One of these two partitions is called P_1. The other part is again divided into two equal partitions. This recursive process continues until 5 partitions are obtained. Figure 24 illustrates the graph format of this type of partitioning.

Assume faults are injected into P_4. By applying this method the size of P_4 is considerably smaller than P_1, P_2, and P_3. Therefore, the synthesis time is lower than when all partitions are of equal size. P_1, P_2, P_3, and P_5 are synthesized once and P_4 is re-synthesized each time after modifying the P_4 structural VHDL code.

Figure 24 is a part of a full binary tree graph where each node is half the size of its parent node. The complete form of a binary tree graph with depth level of 4 is shown in Figure 25. Faults can be injected into any of the 4^{th} level partitions. Once one partition in the 4^{th} level is selected for fault injection, partitions from the higher levels are selected respectively in order to constitute an unbalanced partitioning in the system. To improve the module-based partial reconfiguration performance, fewer partitions are needed. There is no need for the partitioned system to contain all the 16 small partitions from the 4^{th} level when faults are to be injected into only one partition. If we assume partition "LLLR" in Figure 25 is a selected partition in the 4^{th} level for injecting the faults, the other partitions would be "LLLL", "LLR", "LR", and "R" (shown as gray circles in Figure 25) to create an unbalanced partitioning. In this approach nodes in the binary tree are selected vertically instead of horizontally.

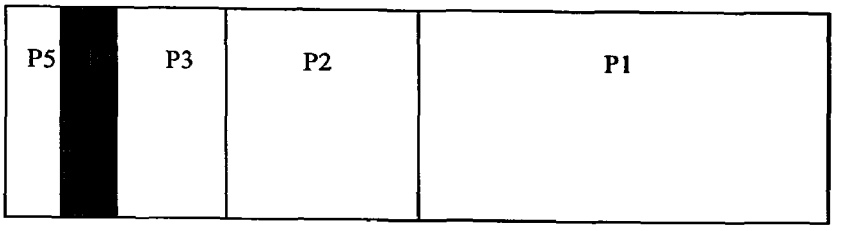


Figure 23: Unbalanced Partitioning

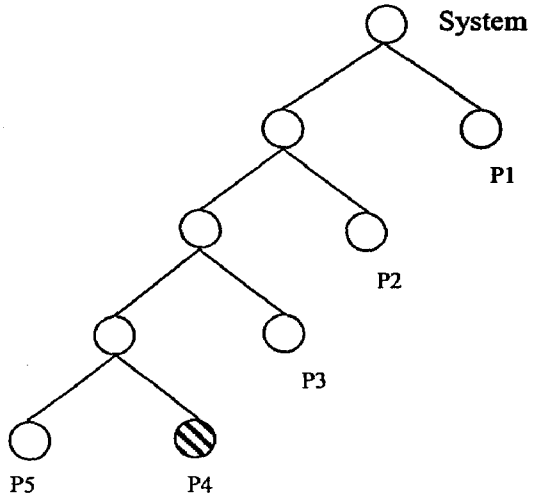


Figure 24: Unbalanced Partitioning Graph

The relation between the sizes of partitions in Figure 25 when faults are injected into "LLLR" is:

$$S(R) = 2S(LR) = 4S(LLR) = 8S(LLLL)$$
$$S(LLLL) = S(LLLR) \qquad (6.3)$$

If N is the total number of partitions in an unbalanced partitioning method and G_T is the total number of gates in the circuit the size of the smallest partition $S(P)$ is:

$$S(P) = G_T / 2^{(N-1)} \qquad (N \geq 1) \qquad (6.4)$$

By comparing (6.1) and (6.4), it can be proven that the size of the smallest partition in the unbalanced partition is smaller than the size of the balanced partition. Consequently the synthesis time is less in unbalanced partitioned systems.

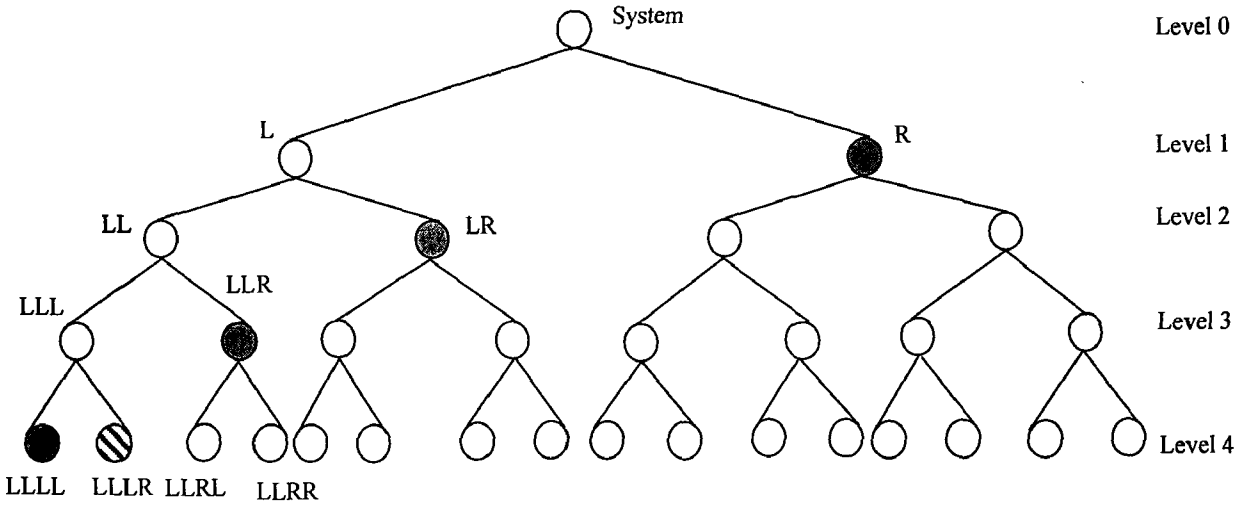


Figure 25: Full Binary Tree (Depth level = 4)

6.2 Binary Tree Approach

Binary trees are defined recursively. A binary tree is a structure defined on a finite set of nodes that either

- contains no nodes, or
- is composed of three disjoint sets of nodes; a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

The binary tree that contains no node is called the empty tree or null tree, sometimes denoted NIL. If the left subtree is nonempty, its root is called the left child of the root of the entire tree. Likewise, the root of a non-null right subtree is the right child of the root of the entire tree. If a subtree is the null tree NIL, we say that the child is absent or missing.

If a node in a binary tree has just one child the position of the child- whether it is the left child or the right child- matters.

In a tree that results is a full binary tree, each node is either a leaf or has exactly a degree of 2. There are no degree-1 nodes. Consequently, the order of the children of a node preserves the position information [31].

In this research the full binary tree is generated based on the merge sort algorithm features. In other words every child in the tree has the half size of its parent and each child has the same size as other children in the same level of depth. The features of the merge sort algorithm and its worst-case running time are discussed in subsection 6.2.1.

6.2.1 Analyzing Merge and Sort Algorithm

Many useful algorithms are recursive in structure. To solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach. They break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- **Divide** the problem into a number of subproblems.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblem in a straightforward manner.
- **Combine** the solution to the subproblems into the solution for the original problem.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation or recurrence, which describes the overall running time of a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide limits on the performance of the algorithm.

We reason out as follow to set up the recurrence for the worse-case running time of merge sort on n numbers, $T(n)$. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **Divide:** The divide step computes the middle of the subarray, which takes constant time c.
- **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** The merge procedure on an n-element subarray takes time cn.

Therefore the recurrence for the worst-case running time $T(n)$ of merge sort is:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases} \qquad (6.5)$$

Figure 26 shows how we can solve the recurrence (6.7). For convenience to simplify matters, we assume that n is an exact power of 2. Part (a) of Figure 26 shows $T(n)$, which in part (b) has been expanded into an equivalent tree representing the recurrence. The cn term is the root (the cost at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost for each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem size is reduced to 1, each with a cost of c. Part (d) shows the resulting tree.

Next, we add the costs across each level of the tree. The top level has total cost cn. In general the level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that the ith level below the top has total cost $2^i c(n/2^i) = cn$.

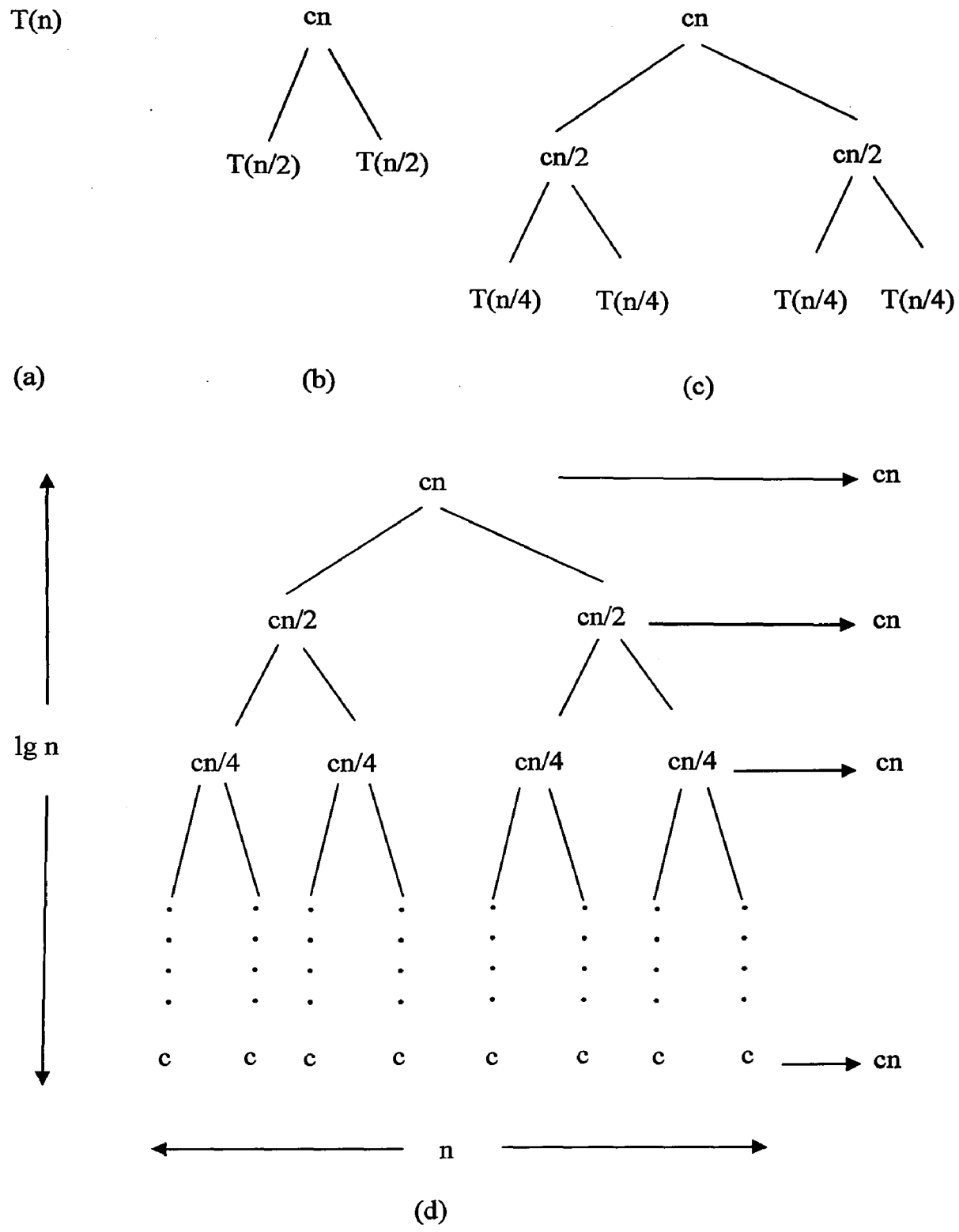


Figure 26: The Construction of a Recursion Tree for the Recurrence *T(n) = 2T(n/2) + cn*

The total number of levels of the "recursion tree" in Figure 26 is $lgn + 1$ where *lgn* stands for $log_2 n$. To compute the total cost represented by the recurrence (6.5), we simply add up the costs of all the levels. There are $lgn + 1$ levels, each costing cn, for a total cost of $cn\ (lgn + 1) = cnlgn + cn$. Ignoring the low-order term and the constant c gives the desired result of $O(n * lgn)$ [31]. Therefore, the complexity of generating a full binary tree based on the merge sort algorithm is $O(n*lg\ n)$.

6.3 Generating VHDL Full Binary Tree

We note that unbalanced partitions are extracted from a full binary tree. Each parent in the tree has two equally sized children. In other words, we generate a full binary tree based on the balanced partitioning method. In this research, we are using the genetic algorithm to divide each VHDL code of the parent node into two equally sized VHDL codes.

The initial VHDL code is divided into two balanced partitions, "L[1].vhd" and "R[2].vhd" to create a binary tree. Each of these two VHDL codes is again divided into two balanced partitions which result in "LL.vhd", "LR.vhd", "RL.vhd", and "RR.vhd". This process continues until the proper depth level is reached.

The depth level of the binary tree depends on the size of the partition and the limit for the cutset size between this partition and the others. The proper partition size, which depends on the application, results in an acceptable synthesis time. The cutset size limit depends on the type of FPGAs since the reconfigurable module (the faulty partition) in the module-based partial reconfiguration communicates with other fixed modules by using a special bus macro. Bus macros are used as fixed data paths for signals going between a reconfigurable module and another module. Each bus macro provides 4 bits of inter-module communication signals. As many bus macros as needed must be instantiated to match the number of bits traversing the boundaries of the reconfigurable module [29]. However, there is an upper boundary limit for the number of bus macros in each type of FPGA. This boundary limits the cutset size of the partition circuit. If the cutset size of the reconfigurable module is more than the maximum number of bus macros of the specific device, the tree must go through deeper levels. For example, the maximum number of bus macros for XC2V1000 is 160 signals.

---

[1] L stands for left child
[2] R stands for right child

As previously mentioned, the Genetic algorithm is used in this work in an attempt to divide a VHDL code into two balanced partitions. The evolutionary computational approach (sometimes called a genetic algorithm) represents the circuit as a chromosome. In a chromosome, each node is given a fixed index in an integer array; and the value at each index determines which partition a node belongs to. For example, if we wished to place each circuit node in either Partition 0 or Partition 1, then the entire circuit configuration could be represented as a string of ones and zeros called chromosomes.

A unique chromosome exists for every possible circuit configuration in the solution space. Each chromosome has a weight that is determined by the size of the cutset and how balanced the circuit is. The best solution will have the lowest weight. In a circuit with 10 million nodes, the size of the solution space would be $2^{10,000,000}$ chromosomes. Unfortunately, finding the best solution to such a problem would take years even by using the fastest computers of today. For this reason we do not concern ourselves with trying to find the best solutions, but rather a good solution that is considered acceptable.

In order to solve the problem, an iterative approach is used. A series of techniques called genetic mutation and genetic crossover are applied to the chromosomes for each iteration, resulting in new offspring chromosomes. The best chromosomes are repeatedly mated and evolved until a solution with acceptable parameters is declared the winner. One of the parameters in this work is to approach to the minimum ratio cut. Minimum ratio cut seeks the partition such that (cost (A,B) / |A|.|B|) is minimized where A and B are two disjoint partitions [30]. The (cost (A,B) / |A|.|B|) is minimized when A and B have an equal size. The winner will determine what the final circuit configuration will look like, including in which partition each node will be contained.

The partitioning program is implemented by getting a VHDL file which describes the structural configuration of the circuit. This configuration is converted into a directed hypergraph, and from there into a chromosome. After the winning chromosome is determined, it is used to construct a VHDL representation of the resulting partitioned circuit.

The structural VHDL codes based on full binary tree are generated and saved once for each circuit and the result can be used several times based on the unbalanced partition selections. Figure 27 shows the flow of the partitioning structural VHDL code.

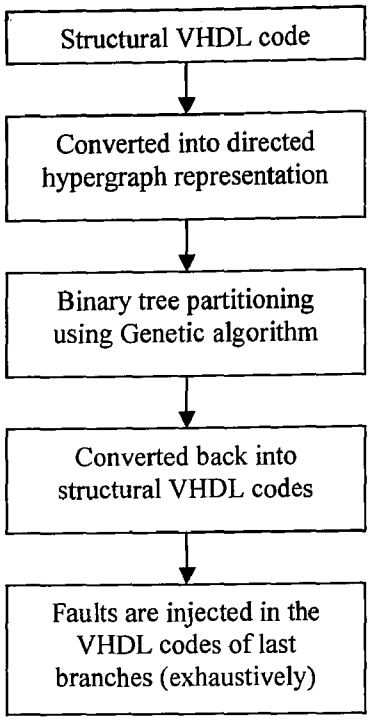


Figure 27: The Flow of Binary Tree Based Partitioning and Modification of the VHDL Code for Fault Injection

After generating the full binary tree of partitioned structural VHDL codes, the structural VHDL codes of the last branches must be modified for faults injection based on the transistor to gate level library modules. The process of modifying the VHDL code to inject faults is done exhaustively, i.e. all possible faults in the library that can occur on each partition are injected into their respective VHDL codes one at a time (single fault) and then faulty VHDL codes are saved in a database. Based on the partition selection and the user fault list, relative faulty and fault free structural VHDL codes from the database can be synthesized and injected into the FPGA. The fault list might include either all possible faults that can occur in a selected partition from the last branches or specific faults.

The next chapter will describe how to use the VHDL code database to automate the synthesis procedure according to the partition selection and the user fault list.

Chapter 7

Automating the Synthesis Procedure of Module-Based Dynamic Partial Reconfiguration

FPGAs are extensively employed today for rapid system prototyping and in countless finished products. The main feature of FPGAs is their ability to change the system's hardware structure in the field, a process known as reconfiguration. Device reconfiguration enables the incremental development of systems, the correction of design errors after implementation, real time fault injections, and the addition of new hardware functions.

Systems built with FPGAs can be reconfigured either statically or dynamically. A static reconfigurable system must be completely reconfigured each time any change of the hardware is required. In this case, system execution stops during reconfiguration. A dynamic reconfigurable system (DRS) allows parts of the system, referred to as reconfigurable modules, be modified, while the rest of the system continues to operate [50]. In this research the intent is to apply dynamic partial reconfiguration for fault injection into a portion of a circuit. Therefore, certain portions of the device can be reconfigured while the rest of the device remains operational. These portions are referred to as reconfigurable modules.

Styles and features of partial reconfiguration will be discussed as well as the method of applying module-based partial reconfiguration for fault injection into FPGA. In addition, a presentation of the automated process of synthesizing system VHDL codes for the partial reconfiguration of FPGAs will follow.

7.1 Partial Reconfiguration

By definition partial reconfiguration or local run-time reconfiguration is a form of reconfiguration which allows that only a portion of a system (e.g. DR FPGA device) to be reconfigured. A partial reconfiguration can be non-disruptive, i.e. the portions of the system, which are not being reconfigured remain fully operational during the entire reconfiguration cycle or disruptive, i.e. partial reconfiguration affects other portions of the system - typically need for a clock hold. Non-disruptive partial reconfiguration is often shorted to partial reconfiguration, as this form is more practical and common in today's DR FPGAs [51].

One of the features of partially reconfigured FPGAs is that the size of the bitstream is proportional to the size of the reconfigured resources. Thus, as the size of the bitstream decreases, the speed of the reconfiguration increases. This feature further clarifies why, in Chapter 6, we have concentrated on the unbalanced partitioning methodology to reduce the size of the fault injected partition as much as possible. This partition has to be reconfigured each time after injecting a new fault. Therefore, its size has an important impact on the speed of fault injection.

There are two main styles of partial reconfiguration: module-based and difference-based. The difference-based method of partial reconfiguration is accomplished by making a small change to a design, and then generating a bitstream based only on the differences between the two designs. This method is undesirable for injecting faults into a large circuit because after a small change in the initial structural VHDL code the whole circuit has to be re-synthesized, which is time-consuming. The module-based method is based on dividing a design into modules and reconfiguring only a few modules while the rest of the design remains in active operation.

The height of an FPGA is predefined. In this method, the reconfigurable module height is always the full height of the device and its width ranges from a minimum of four slices to a maximum of the full-device width [29]. Thus, if a partition of a circuit assigned to a reconfigurable module does not fill the entire module predefined space the device resources will be wasted. Dividing the system into small partitions can also be a waste of resources.

To help minimizing problems related to design complexity, the number of modules should be minimized, so that less bus macros are defined, leading to a more optimized routing. The reason is that as the location of the bus macros are fixed, if a signal traversing the reconfigurable module to the other modules is not near the location of the bus macros the complexity of routing will increase.

This method is well-matched to our work because only the partition where faults are injected is reconfigured. It has to be mentioned that due to unbalanced partitioning the module sizes are not equal. This means there will not be many small partitions, and thus we will avoid wasting the device resources. Module-based partial reconfiguration flow is based on the modular design methodology. Consequently, in order to clarify the module-based partial reconfiguration this methodology will be studied.

7.2 Modular Design

Modular design provides a divide-and-conquer implementation approach to multi-million gate FPGA designs available. Partitioning a design into smaller functional modules reduces the complexities of design, implementation and verification. These smaller modules can then be brought through the design flow independently. Once completed, a module's implementation is preserved, guaranteeing the timing in the finished device. In other words, the basic idea underlying modular design is to organize a complex system (such as a large program or an electronic circuit) as a set of distinct components that can be developed independently and then plugged together.

Modular design requires up-front planning to ensure that the design is partitioned properly. It also requires ensuring that partitions work together during the final assembly phase.

This breakthrough technology is employed in this research to utilize partitioned VHDL codes from the database in Chapter 6 as a design entry, synthesize them and generate proper bitstreams for injection into the FPGA.

The Modular Design flow consists of the following phases:

7.2.1 Modular Design Entry and Synthesis Phase

In this phase the module designs using HDL are created and synthesized individually. In this research the VHDL code of each module can be obtained from the full binary tree database explained in Chapter 6. Then the related VHDL code from the database must be synthesized. During synthesis, behavioural information in the HDL file is translated into a structural netlist and the design is optimized for a Xilinx device.

A state of the art synthesis engine is required to produce highly optimized results with a fast compilation and quick turnaround time. To meet this requirement, the synthesis engine needs to be tightly integrated with the physical implementation tool and have the ability to proactively meet the design timing requirements by driving the placement within the physical device. In addition, cross probing between the physical design report and the HDL design code will further enhance the turnaround time [52]. Our experimental device in this project is Xilinx VirtexII (XC2V1000).

To synthesize the VHDL codes we can use either Xilinx-supported third-party tools, which produce a design file in third-party netlist formats, the Xilinx synthesis tool, or Xilinx Synthesis Technology (XST) that produces a netlist in NGC format. In this work the XST synthesis engine is chosen. The reason is that XST progresses in each release, improving clock frequencies and decreasing area, as well as reducing run time and memory utilization. XST has been tuned to Virtex architectures, inferring many of the architecture's specific primitives. Users have extensive control over inference capabilities and optimization techniques via global options and local attributes. Xilinx estimates the current language support covers at least 95% of the constructs supported by other synthesis tools. Many of the unsupported constructs are infrequently used and/or have simple work-arounds. Also, many of these constructs are not handled consistently by each synthesis tool. One tool may accept a construct in one way, another in a different way, and a third may flag a parsing error. In some situations, XST is actually more precise than other tools, requiring exact, complete descriptions when others allow incomplete or vague code. These are very common issues when moving code from one synthesis tool to another [53].

For our purpose, XST reads the entry file in VHDL format and creates an NGC file. NGC file is a netlist that contains both logical design date and constraints and can be read directly by NGDBuild. Figure 28 illustrates modular design entry and synthesis flow.

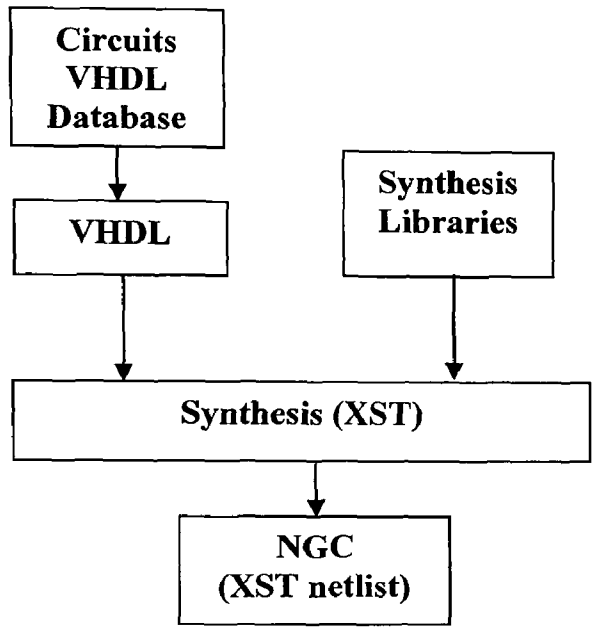


Figure 28: Modular Design Entry and Synthesis Flow

7.2.2 Modular Design Implementation Phase

Modular design implementation includes the three phases described below. After the final phase is complete, the implemented design can be used to generate a bitstream [54].

1. Initial budgeting phase: In this phase, the top level logic for the design is positioned. Properly positioning the logic in this phase is critical. Repositioning top-level logic later in the design process requires rerunning each phase of the Modular design flow, in a time consuming process. The objectives of the initial budgeting phase are to position global logic, size and position each module on the target chip, position the input and output ports for each module, and budget initial timing constraints.

The first step in this phase is to run NGDBuild in initial budgeting mode. NGDBuild generates an NGD file with all of the instantiated modules represented as unexpanded blocks. This NGD file cannot be mapped but can be used with the Constraints Editor, PACE or Floorplanner. The Constraints Editor is employed to assign timing constraints to the top-level design. PACE can position the I/O ports for the design on the targeted device. Floorplanner can be used to assign pin location constraints for the design.

2. Active module implementation phase: In this phase NGDBuild has to be run for each module. NGDBuild reads a netlist file in NGC format and creates a NGD file that contains both a logical description of the design reduced to Xilinx Native Generic Database (NGD) primitives and a description of the original hierarchy expressed in the input netlist. After generating a NGD file, it can be mapped, placed and routed. When a module is fully placed and routed and meets the desired timing constraints, it is published back for inclusion in the final design
3. Final Assembly phase: In this phase, the modules are assembled into one design by running NGDBuild in final assembly mode. NGDBuild creates a fully expanded design file that can be used for mapping, placing, and routing.

After the design is completely routed, it is necessary to configure the device so that it can execute the desired function. This is done using files generated by BitGen, the Xilinx bitstream generation program. BitGen takes a fully routed file as input and produces a configuration bitstream, a binary file with a *.bit* extension. The BIT file contains all of the configuration information that defines the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file is then downloaded into the FPGAs memory cells, or used to create a PROM file.

7.3 Automating the Synthesis Procedure of Module-based Fault Injection Method

As introduced in previous chapters, in order to inject faults into FPGAs, the initial circuit VHDL description must be divided into unbalanced partitions with respect to the full binary tree features.

All fault free and faulty VHDL partitions of a circuit are generated and saved in a database. Each VHDL partition can be considered as a module entry to our modular design.

It must be noted that in modular design, each module has its own directory. Synthesis of each module entry has to take place in its module directory. Therefore, there will be as many module directories as there are partitions. For example, if one circuit is divided into five partitions, five directories will be generated. The VHDL code of each partition will be copied from the database to its respective directory.

We have used Perl scripts, developed in Unix OS, to automate the process of generating directories, getting the appropriate VHDL entries from the database and applying XST on them. In this work the automation of the synthesis procedure of modular design is implemented. In general, the Perl automated program has the following steps:

Step 1: At the command line, get a circuit name from the user which is assigned to $circuit_name variable. Figure 29 shows this part of the Perl program.

```
print "Enter a circuit name for injecting faults:\n ";
#assign entered circuit name to circuit_name variable.
$circuit_name = <STDIN>;
```

Figure 29: Circuit Name Assignment

Step 2: Compare $circuit_name variable with all circuit names whose VHDL code already exist in the database. If the entered circuit name does not exist in the database the error message is printed and the user has to enter another circuit name supported by the

database. Circuits supported in this work are nine combinational and sequential benchmarks with different sizes and characteristics. These nine benchmarks are:

C17, S27, S298, S1238, C2670, S5378, C7552, S13207, and S38417.

Should the entered circuit name be among our nine benchmark circuits, the depth level of the full binary tree of the circuit will be printed on the screen and the user will be asked to choose a partition for fault injection. The selected partition is then assigned to $entered_branch variable.

For example, assume the entered circuit name is S1238. The depth level of this circuit is equal to four. Figure 30 shows Step 2 of the Perl program for S1238.

```
if($circuit_name eq $s1238)
  {
     print "This circuit depth level is four. \n";
     print "Please enter one of the tree branches
            where you want to inject faults.\n
            Example: LLLR \n ";
     $entered_branch = <STDIN>;
     $depth_level = 4;
  }
```

Figure 30: S1238 Step 2

Step 3: As mentioned before, the depth level of each circuit depends on the circuit size and bus macro limitation of the FPGA device. The proper depth level for each circuit was determined during the database generation. In this step the $entered_branch variable is verified and expected to match the circuit depth level. Note that $entered_branch represents one of the lowest level branches of the full binary tree.

For instance, an error message is printed for S1238 with depth level of four, if the entered branch for injecting faults is "LLR" or "LLLLR". The reason is that "LLR" is one of the last branches of the full binary tree with depth level of three and "LLLLR" is one of the last

branches of the full binary tree with depth level of five. Therefore, for S1238 the length of the entered branch is exactly four. Figure 31 shows this part of the program for S1238.

```
while (length[1] $entered_branch != $depth_level)
       {
         print "What you have entered doesn't
                match the circuit depth level.\n";

         print "The circuit depth level is ($depth_level).
                Please enter again: \n ";
         $entered_branch = <STDIN>;
       }
```

Figure 31: Verifying the Entered Branch Length

Step 4: In this step the $entered_branch is compared to our branch names convention (the length of $entered_branch was verified in Step 3). Each branch in the full binary tree can be addressed by some "L" and "R" strings depending on its location. However, if the specified branch has characters other than "L" or "R" an error massage will be printed. Figure 32 shows this kind of verification.

```
@branch= split[2](//,$entered_branch);
$len= length $entered_branch;
#  for loop executed till final entered branch is valid.
  for( $x=0; $x<$len; $x++ )
   {
    $path= $branch[$x];
    if (($path ne "L")&& ($path ne "R"))
     {
       print " this is not a valid branch.\n";
```

[1] Length is a function of Perl language. It simply returns the number of characters in a string variable.
[2] Splits up a string and places it into an array.

```
        print " yor branch name has to contain only
                'L' or 'R' characters. Please enter again: \n";
        $entered_branch= <STDIN>;
        @branch= split(//,$entered_branch);
        $x=(-1);    # makes the for loop start form the
                    # beginning to verify the new entered branch
    }
}
```

Figure 32: Verifying the Entered Branch Characters

Step 5: Make a directory for the specified circuit.

Step 6: Extract other partition names (fault free partitions) by using $enterd_branch variable. $entered_branch variable represents a partition where faults are injected. Other partition names have to be extracted from the entered branch so that an unbalanced partition in the system is created. These partition names are assigned to @partition_names array. For example, if the entered partition for S1238 is "LLLR" the other partition names to create an unbalanced partitioning will be: "LLLL", "LLR", "LR", and "R". Figure 25 shows how this selection is determined.

Step 7: Make a sub-directory inside the circuit directory (created in Step 5) for each member of @partition_names array.

```
system ("mkdir -p $circuit_name/$directory_name");
```

Figure 33: Making a Partition Directory inside the Circuit Directory

Step 8: Copy the related VHDL code of each partition from the database to the partition sub-directory.

```
system ("cp
/opt/research/fault_injection/perl/VHDLcodes/$circuit_name/$di
rectory_name.vhd[1]
```

[1] Database location of the partition VHDL code

```
/opt/research/fault_injection/perl/$circuit_name/$directory_na
me/$directory_name.vhd");
```

Figure 34: Copy a Partition VHDL Code to Its Directory

Step 9: Make sub-directories for a faulty partition according to the user fault list and copy the related VHDL code inside each sub-directory (Step 7 and 8 are based on the fault free partition directories). The fault list of the user is read and different sub-directories are generated for the faulty partition based on the number of faults. For example, Figure 35 illustrates a fault list for partition "LLLR" of S1238. The list contains five different faults to be injected into the "LLLR" partition. Therefore, five sub-directories with different names have to be generated. In a module-based partial reconfiguration one of these faulty sub-directories are used at a time along with all other fault free sub-directories.

```
OR2_87[1],OR2[2],F10[3],F25
AND3_22,AND3,F15,F32
I_202,INV,F4
```

Figure 35: An Example of Fault List for S1238

In this example the total number of sub-directories in S1238 circuit directory is nine. Four sub-directories are for fault free partitions ("LLLL", "LLR", "LR", and "R") and five sub-directories for five existing faults in the fault list. The S1238 directory is shown in Figure 36.

```
../S1238/
            S1238_Partition_LLLL/
            S1238_Partition_LLR/
            S1238_Partition_LR/
            S1238_Partition_R/
```

[1] Instantiation of the OR2 gate in the S1238_Partition_LLLR.vhd.
[2] Gate type.
[3] Wanted fault for injection

```
S1238_OR2_87_F10/
S1238_OR2_87_F25/
S1238_AND3_22_F15/
S1238_AND3_22_F32/
S1238_I_202_F4/
```

Figure 36: S1238 Circuit Directory

Step 10: Create a project file (*.prj*) inside each faulty and fault free sub-directories. This project file contains a list of all components used inside a particular partition. By running XST for system synthesis, the project file will be used for adding the entire listed components in the project file to WORK library. This issue is discussed in Section 4.3.3.

Step 11: To synthesize each sub-directory's VHDL code, XST can be run either in command line mode or from the Process window in Project Navigator. Due to the automated synthesis procedure, the command line mode is chosen here. To make the process completely automatic we create a script file (*.scr*) inside each sub-directory to store all required commands. The commands are extracted from the XST user guide [55] based on our requirements.

For example, the script file for partition "LLLL" in S1238 is shown in Figure 37.

```
run
-ifn[1] s1238_PARTITION_LLLL.prj
-ifmt[2] mixed
-ofn[3] s1238_PARTITION_LLLL.ngc
-ofmt[4] NGC
-p[5] xc2v1000-fg456-4
```

[1] Input/Project file name
[2] Input project format. Xilinx suggests using mixed format whether it is a real mixed language project or not.
[3] Output file name
[4] Output file format
[5] Target technology

-top[1] s1238_PARTITION_LLLL
-iobuf[2] No

Figure 37: S1238_Partition_LLLL XST Script File

All the required files for synthesizing the circuit are now available in each partition sub-directory. Figure 38 shows all available files inside S1238_Partition_LLLL sub-directory after running the Perl program. The Perl Program organizes and creates the necessary files inside each directory in the above steps.

```
../S1238/S1238_Partition_LLLL/
                                S1238_Partition_LLLL.vhd
                                S1238_Partition_LLLL.prj
                                xst.scr
```

Figure 38: S1238_Partition_LLLL Directory Files

The XST has to be executed in each sub-directory to synthesize the design. Using a batch file we obtain the *xst.scr* file as an input from each sub-directory of the specified circuit. Then a NGC file (*.ngc*) and a synthesis report in "*.log*" format is generated. The executed command is the following:

```
XST -ifn xst.scr -ofn Synthesis_report.log
```

Figure 39: Executing the Script File (xst.scr)

We note that at this stage all the faulty and fault free VHDL codes have been synthesized successfully. The NGC files can be used to generate bitstreams in the second phase of modular design. The synthesis report shows all details and considerations during the

[1] Top level block name
[2] Add I/O buffers

synthesis process. It also shows the CPU time for synthesizing each partition. The experimental results of nine benchmark synthesis procedures are presented in the next section.

7.4 Experimental Results

In order to observe the synthesis results of classical and non-classical transistor fault models mapped to gate level for FPGA fault injection, nine combinational and sequential benchmark circuits with different sizes and characteristics have been selected. As mentioned previously, these circuits are: C17, S27, S298, S1238, C2670, S5378, C7552, S13207, and S38417. Figure 40 illustrates the total number of gates for each circuit.

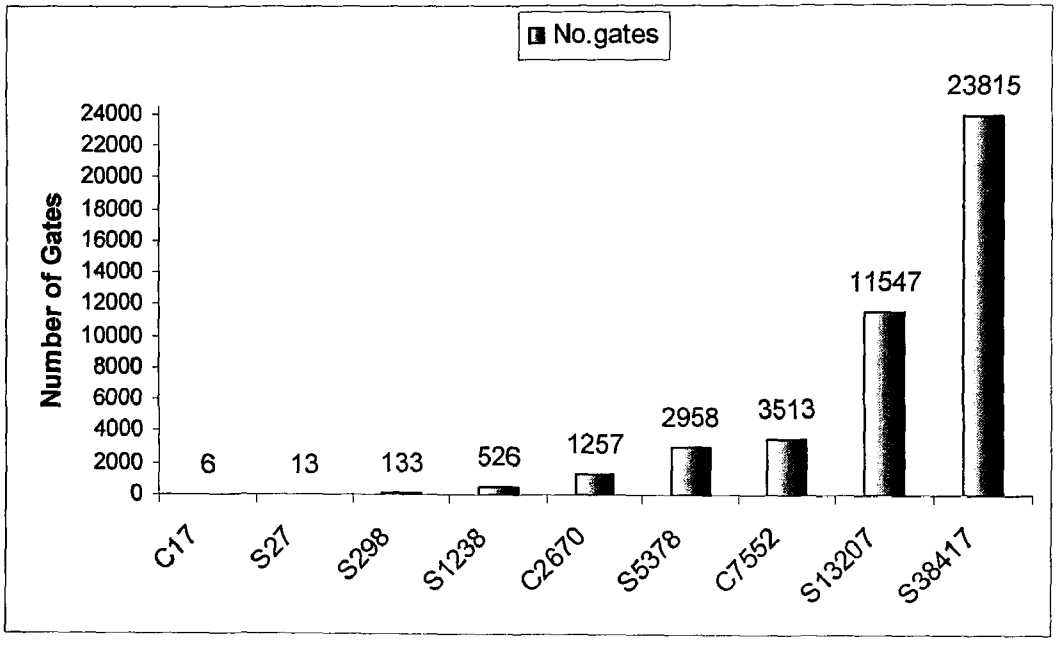


Figure 40: Size of Circuits

The unbalanced partitioning approach based on full binary tree characteristic is applied to these circuits. To determine the full binary tree depth level for each circuit, two factors are taken into account: the size of the last branches and the number of inter-module connection signals. In this experiment, our target device is the Xilinx VirtexII (XC2V1000) FPGA. With this FPGA, the maximum number of signals that can traverse from the reconfigurable

partition to the rest of the circuit is 160 signals. The depth level of each circuit full binary tree is shown in Figure 41.

Circuits C17 and S27 circuits are too small (less than 15 gates) and therefore do not require partitioning. Thus, their full binary tree depth level is zero, i.e. their respective full binary tree consists only of one node. Although S298 is not a big circuit either, the number of inter-module signals exceeds 160 signals for depth levels less than two. For other circuits we first decided to have the size of last branches partitions less than 200 gates to have a small and acceptable re-synthesis time and then verified that the inter-module signals are less than 160 as discussed in Chapter 6. In most cases the number of signals going from the reconfigurable module to the rest of the circuit exceeds the limitation. Hence we increase the depth level to satisfy all constraints.

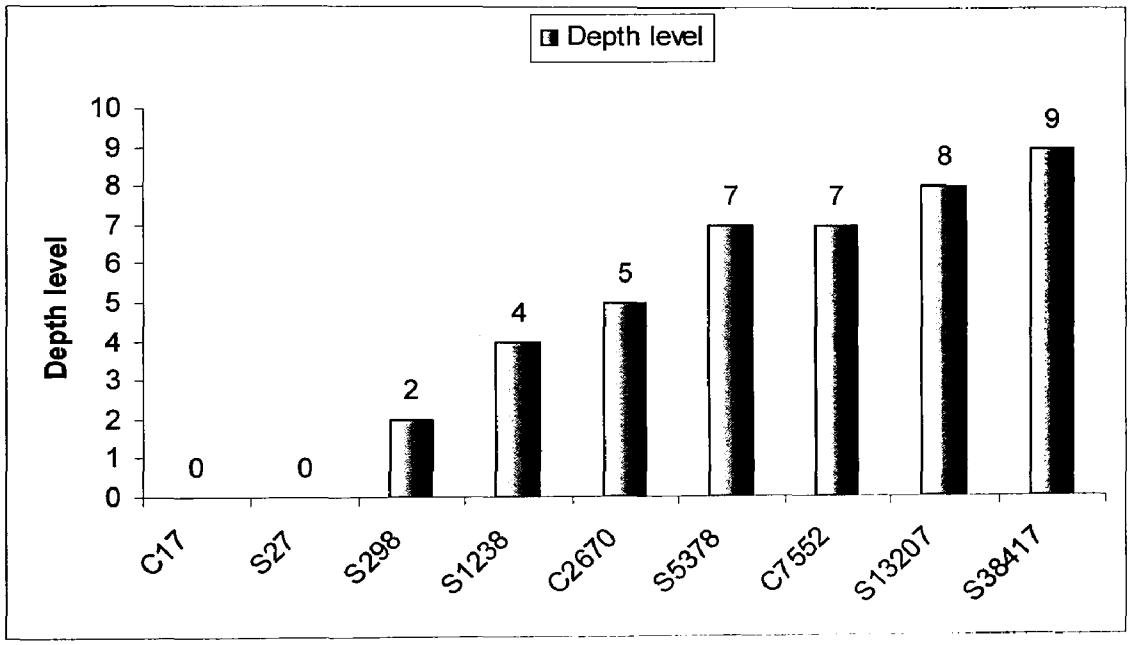
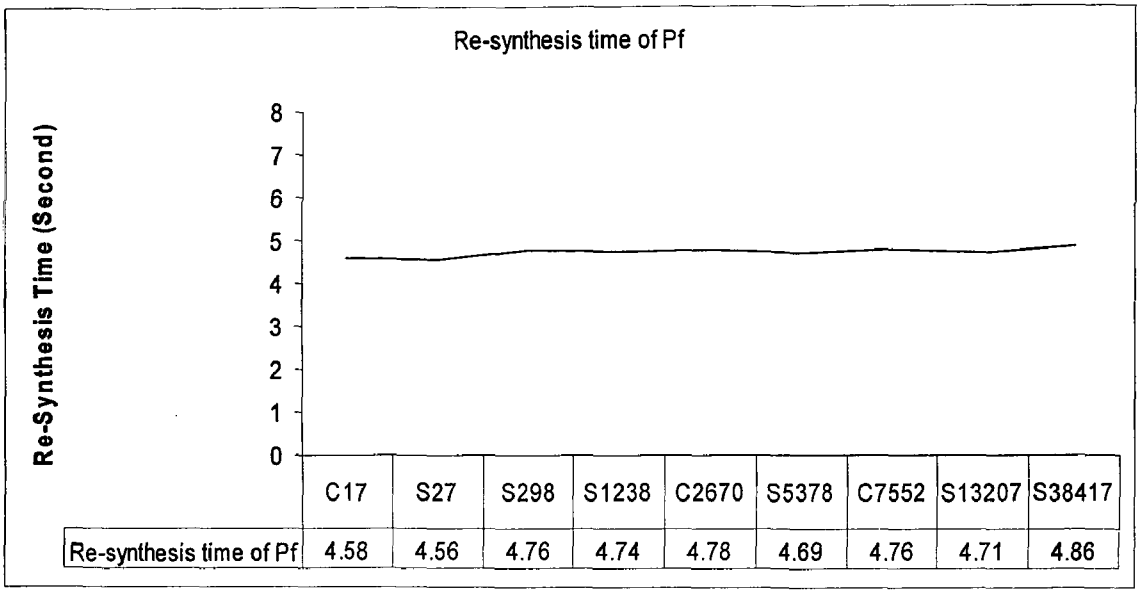


Figure 41: Full Binary Tree Depth Level of Circuits

After finalizing the depth level of each circuit full binary tree, the VHDL database for each binary tree is generated and saved. For fault injection, one partition (P_f) from the last branches of each circuit's full binary tree is selected and its faulty VHDL codes are fetched from the database according to a provided fault list. The results of re-synthesis time of the

selected partition where faults are injected into the FPGA for each circuit are monitored (see Figure 42). It has to be mentioned that all timings in this research are calculated based on the CPU time. We found that the re-synthesis time is constant for all circuits. The reason is that sizes of last branches of full binary tree partitions in all the circuits are almost the same based on their decided depth level. For example, last branches full binary tree partition size for S38417 is 46 gates with the depth level of 9. This size is equal to 45 gates for S13207 with depth level of 8. It can be concluded that the depth level of the full binary tree with respect to its limitations can be determined based on the desired re-synthesis time.



	C17	S27	S298	S1238	C2670	S5378	C7552	S13207	S38417
Re-synthesis time of Pf	4.58	4.56	4.76	4.74	4.78	4.69	4.76	4.71	4.86

Figure 42: Re-synthesis Time of the Faulty Partition

To configure the FPGA all partitions must be synthesized. Figure 43 shows the added synthesis time of all partitions in each circuit. It can be seen that the synthesis time increases linearly as a function of circuit size, whereas by applying the unbalanced partitioning approach the whole circuit is not re-synthesized when faults are injected only to one portion. For instance, by adding the synthesis times of all partitions in S13207, we have found that synthesizing the whole circuit takes 165.85 seconds. However reconfiguring the device for injecting faults requires only 4.7 seconds re-synthesis time.

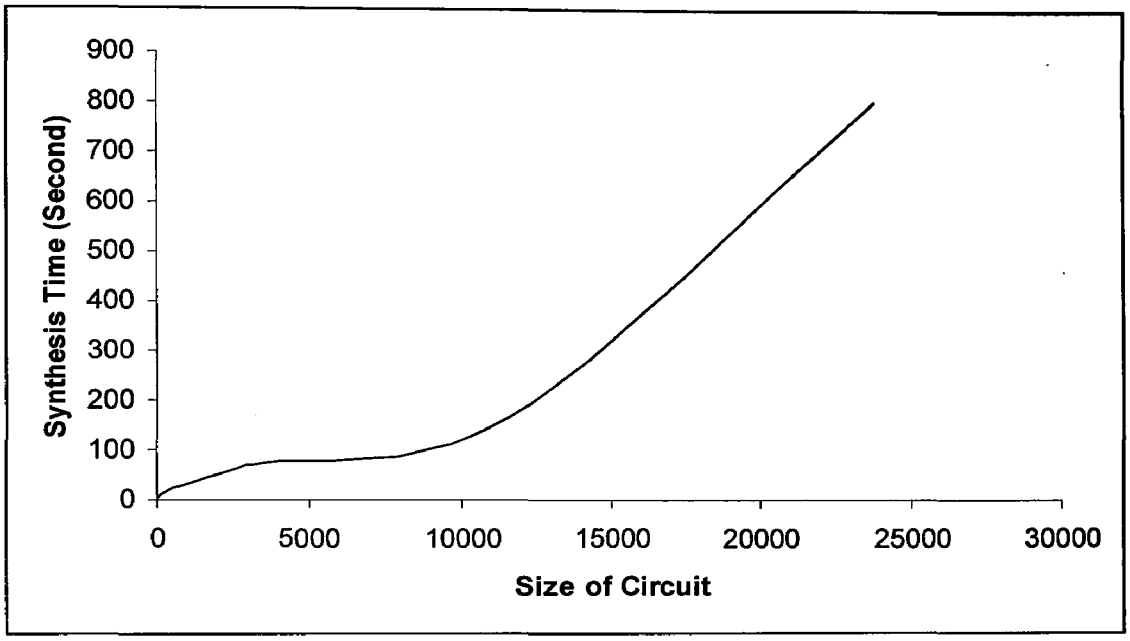


Figure 43: Benchmarks Synthesis Time

Figure 44 shows the synthesis time of each partition in circuit S38417 as a function of partition size. Assume faults are going to be injected in to the "LLLLLLLLR" partition of a full binary tree with depth level of nine. Therefore, to establish an unbalanced partitioning, specific branches of the full binary tree must be selected. This selection is shown in Figure 45. Table 25 gives the approximate size of each partition.

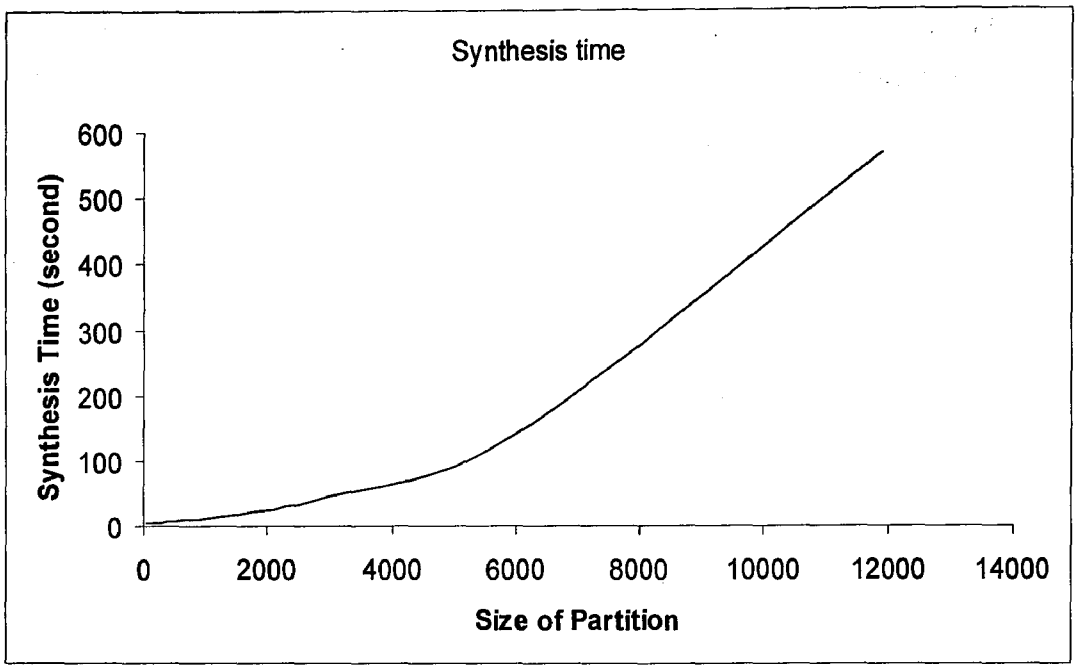


Figure 44: S38417 Partition synthesis time

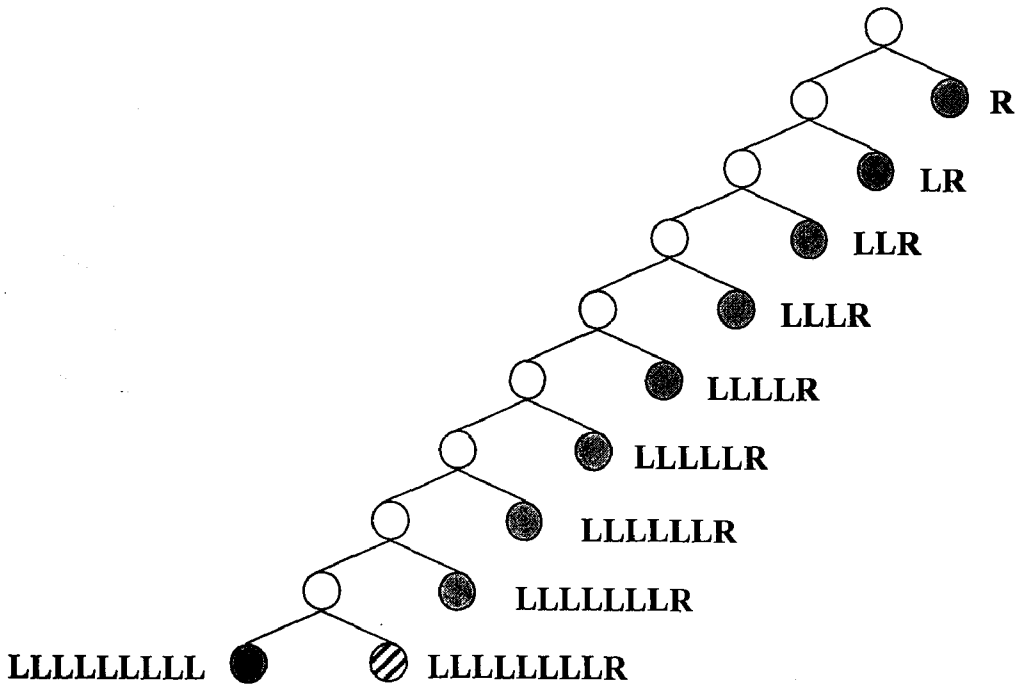


Figure 45: Partition Selection of a Full Binary Tree (Depth level = 9)

Partition Name	Number of Gates
R	11908
LR	5954
LLR	2977
LLLR	1488
LLLLR	744
LLLLLR	372
LLLLLLR	186
LLLLLLLR	93
LLLLLLLLR	46
LLLLLLLLL	46

Table 25: S38417 Partition Size

If we wanted to obtain the same re-synthesis time through balanced partitioning $2^{depth\ level}$ partitions would be required, whereas, for unbalanced partitioning only (*depth level* – *1*) partitions would be necessary. The total number of partitions for the balanced and unbalanced partitioning is shown in Figure 46.

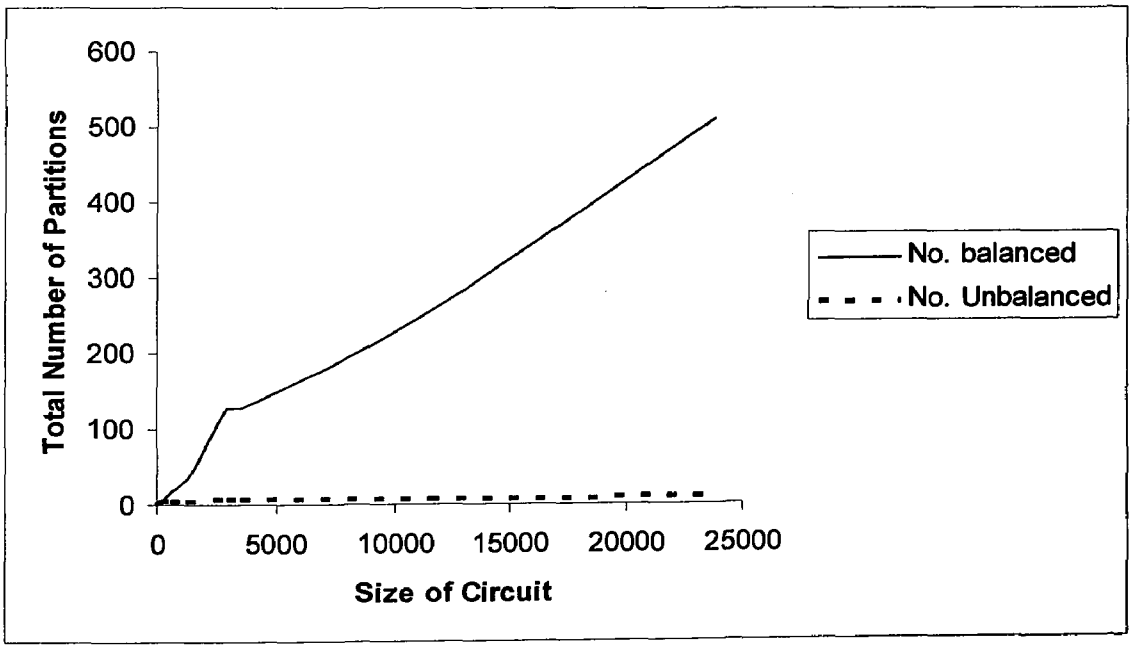


Figure 46: Unbalanced Versus Balanced Partitioning

It can be seen that the complexity of modular design increases when a balanced partitioning is applied as discussed in Chapter 6.

The minimum cost function (γ) of unbalanced partitioning approach depends on the size of the partition $S(P_f)$, and the total number of partitions (N). The minimum cost function can be summarized in (7.1) where K is a constant.

$$\min \gamma = K \frac{S(Pf)}{N} \quad (7.1)$$

The synthesis analysis of classical and non-classical transistor fault models is done in a timely manner by using the proposed partitioning approach.

Chapter 8

Conclusion

In recent years, there has been a growing interest in techniques for validating the fault tolerance properties of safety and mission critical systems and for an evaluation of their reliability. Fault injection is a validation technique of fault tolerance systems, in which the observation of the system behaviour in the presence of faults is explicitly forced by the introduction of faults.

With the advent of system-on-chip, it is crucial to perform a thorough analysis of the failure modes of the circuit before manufacturing. To inject faults into a system an accurate fault model, a proper approach for injecting faults, and adequate test patterns to observe the system behaviour is required.

8.1 Research Contribution

The transistor fault model is more accurate than the gate level fault model. Hence, to obtain an accurate fault model our intent was to use the transistor level fault model. The transistor level fault model however cannot be synthesized in FPGA chips. Therefore, we have opted for the solution where all classical and non-classical faults of static CMOS cells are mapped to gate level. A specific fault pattern for each primitive gate was determined during the mapping. To access these mapped faults a user-defined library was defined and used in structural VHDL code. By replacing a fault free component with a faulty one from the user-defined library in a structural VHDL code, faults were injected into the system VHDL code.

To reduce the re-synthesis time after injecting each fault, an unbalanced partitioning approach based on the full binary tree was introduced in this research. In this approach the re-synthesis time is reduced because the size of the partition where faults are injected is reduced by the partitioning. However, other fault free partitions have larger sizes since they do not need to be re-synthesized after each fault injection. In this case, the total number of partitions in a system is much less than balanced partitioning method. Therefore, the complexity of module-based partial reconfiguration is reduced.

Module-based partial reconfiguration was applied in the process of fault injection into FPGAs. We have also automated the generation of all required files prior to synthesis as well as the synthesis procedure itself.

Although the VHDL code database, wherein faulty and fault free VHDL partition codes reside, takes a large memory space, the experimental results have shown that unbalanced partitioning approach saves time and FPGA resources in module-based partial reconfiguration. We have shown that the re-synthesis time can be kept quasi constant in all circuits of different sizes if the sizes of the partitions where faults are injected (P_f) are the same. Whereas, the synthesis time of the whole circuit grows linearly as the size of the circuit is increased. We have compared the unbalanced and balanced partitioning approaches and shown that the former leads to a more cost-efficient fault injection method.

8.2 Future Work

More research is needed to include dynamic CMOS faults in the user-defined library. To reduce the memory space taken by the database, partitioned VHDL codes will have to be generated based on the selection of binary tree branches. Also, the implementation phase of the module-based partial reconfiguration and downloading bitstreams into FPGA will have to be automated. The actual behaviour of faulty circuits can be studied in the application environment by applying adequate test vectors.

Appendices

Appendix A: ORn Complete Fault List Pattern

- **Input/Output Stuck-at Faults**

Input/output stuck-at-faults	Faulty output (O)
I_i : stuck-at-0	OR (I_1, I_2 ..., I_{i-1}, I_{i+1},..., I_n)
I_i : stuck-at-1	1
O : stuck-at-0 (for any input combinations)	0
O : stuck-at-1 (for any input combinations)	1

Table 26: ORn Input/Output Stuck-at-faults

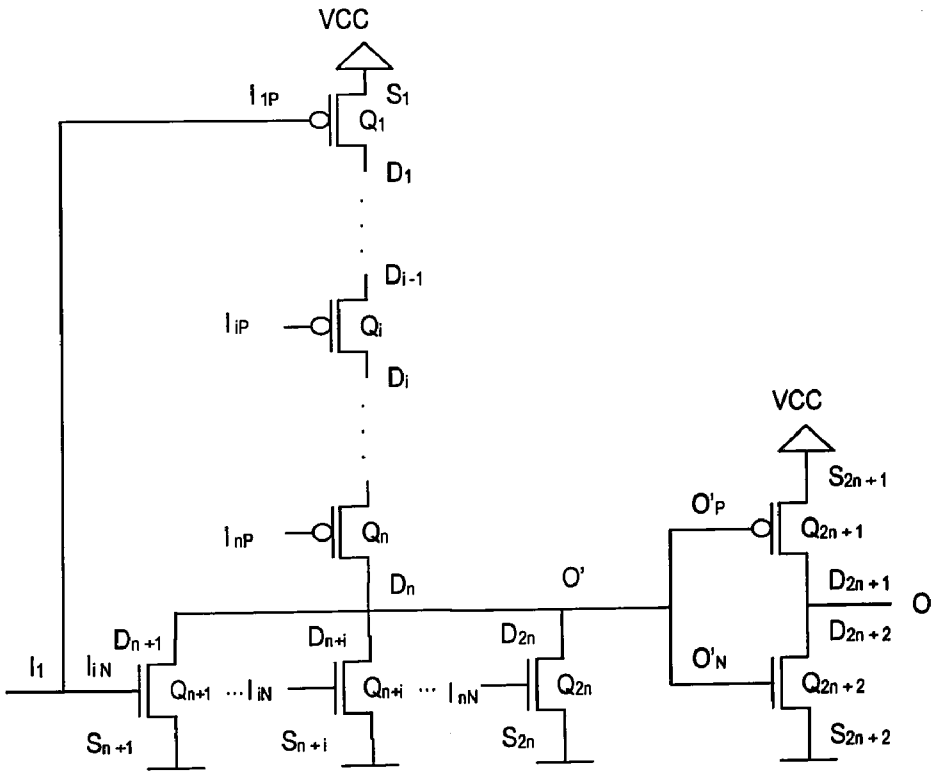


Figure 47: ORn Transistor Level

- **Short Faults**

Short faults	Inputs conditions	Faulty output (O)
$O'(D_n) \leftrightarrow G$ (ground)	For any input combinations	1
$O'(D_n) \leftrightarrow VCC$	For any input combinations	0
$D_i \leftrightarrow G$ ($1 \le i \le n-1$)	For any input combinations	1
$D_i \leftrightarrow VCC$ ($1 \le i \le n-1$)	$I_j=0 \ \forall \ 1 \le j \le n$ (means all inputs from I_1 to I_n are low)	0
	$I_j = 0 \ (\forall \ i+1 \le j \le n)$ & $I_1 \| I_2 \| \ldots \| I_i = 1$	Stuck-On
	For the rest of input combinations	1
$I_i \leftrightarrow I_k$ where $1 \le k \le n$ and $k \ne i$	$I_j=0 \ \forall \ 1 \le j \le n$	0
	$I_j=0 \ (\forall \ 1 \le k \le n$ and $j \ne i, j \ne k)$ & $I_i \oplus I_k=1$	X
	For the rest of input combinations	1
$I_i \leftrightarrow D_k$ ($i \le k \le n-1$) ($1 \le i \le n-1$)	$I_j = <0\|1> \ \forall \ 1 \le j \le k$, $I_i=0 \quad \forall \ k+1 \le j \le n$	$!I_i$
	For the rest of input combinations	1
$I_i \leftrightarrow D_n$ ($1 \le i \le n$)	For all input combinations	NOT(I_i)
$I_i \leftrightarrow D_k$ ($1 \le k \le i-1$)	For any input combinations	1
$D_i \leftrightarrow D_k$ ($1 \le k \le n-1$ and $k \ne i$):	$I_i=0 \ \forall \ 1 \le j \le n$	0
	$I_j = 0 \ (\forall \ 1 \le j \le i$ and $m \le j \le n \ (i+1 \le m \le n-1))$ & $I_1 \| I_2 \| \ldots \| I_i = 1$	Stuck-On
	For the rest of input combinations	1
$D_i \leftrightarrow D_n(O')$ ($1 \le i \le n-1$)	$I_i=0 \ \forall \ 1 \le j \le n$	0
	$I_j = 0 \ (\forall \ 1 \le j \le i$) & $I_{i+1} \| I_{i+2} \| \ldots \| I_n = 1$	Stuck-On
	For the rest of input combinations	1
$I_i \leftrightarrow O$ ($1 \le i \le n$)	For all input combinations	I_i
$D_i \leftrightarrow O$ ($1 \le i \le n-1$)	$(I_j = 0 \ (\forall \ i+1 \le k \le n) \ \& \ (I_1 \| I_2 \| \ldots \| I_i = 1))$ $\|$ $(I_i=0 \ \forall \ 1 \le j \le n)$	Stuck-On
	For the rest of input combinations	1
$O'(D_n) \leftrightarrow O$	N/A	NOR ($I_1, I_2 \ldots, I_n$)

Table 27: ORn Short Faults

- **Open Faults**

Open faults	Logical fault type:
I_{iP} : open ($1 \le i \le n$)	I_{iP}: stuck-at-1
S_i: open (i =1)	
D_i: open ($1 \le i \le n$)	
I_{iN} : open ($1 \le i \le n$)	I_{iN}: stuck-at-0
I_i : open ($1 \le i \le n$)	
S_i: open ($n+1 \le i \le 2n$)	
D_i: open ($n+1 \le i \le 2n$)	
O'_P: open	O'_P: stuck-at-1
S_i: open (i =2n+1)	
D_i: open (i =2n+1)	
O'_N: open	O'_N: stuck-at-0
S_i: open (i =2n+2)	
D_i: open (i =2n+2)	
O':open	Z
O: open	Z

Table 28: ORn Open Faults Categories

Logical fault type of open faults	Inputs conditions	Faulty output (O)
I_{iP}: stuck-at-1	I_j=0 ($\forall$ $1 \le j \le n$)	Z
	For all other input combination	1
I_{iN}: stuck-at-0	I_j=0 $\forall$ $1 \le j \le n$	0
	I_j=0 ($\forall$ $1 \le k \le n$ and $j \ne i$) & I_i=1	Z
	For the rest of input combinations	1
O'_P: stuck-at-1	I_j=0 ($\forall$ $1 \le j \le n$)	0
	For the rest of input combinations	Z
O'_N: stuck-at-0	I_j=0 ($\forall$ $1 \le j \le n$)	Z
	For the rest of input combinations	1
Z	For any input combinations	Z

Table 29: ORn Open Faults

The general formula for total number of faults for ORn is:

$$2\sum_{j=1}^{n-1}(n-j)+n^2+12n+11 \qquad \text{(A.1)}$$

Appendix B: ANDn Fault Pattern

- **Input/Output Stuck-at Faults**

Input/output stuck-at-faults	Faulty output (O)
I_i : stuck-at-0	0
I_i : stuck-at-1	AND (I_1, I_2 ..., I_{i-1}, I_{i+1},..., I_n)
O : stuck-at-0 (for any input combinations)	0
O : stuck-at-1 (for any input combinations)	1

Table 30: ANDn Input/Output Stuck-at-faults

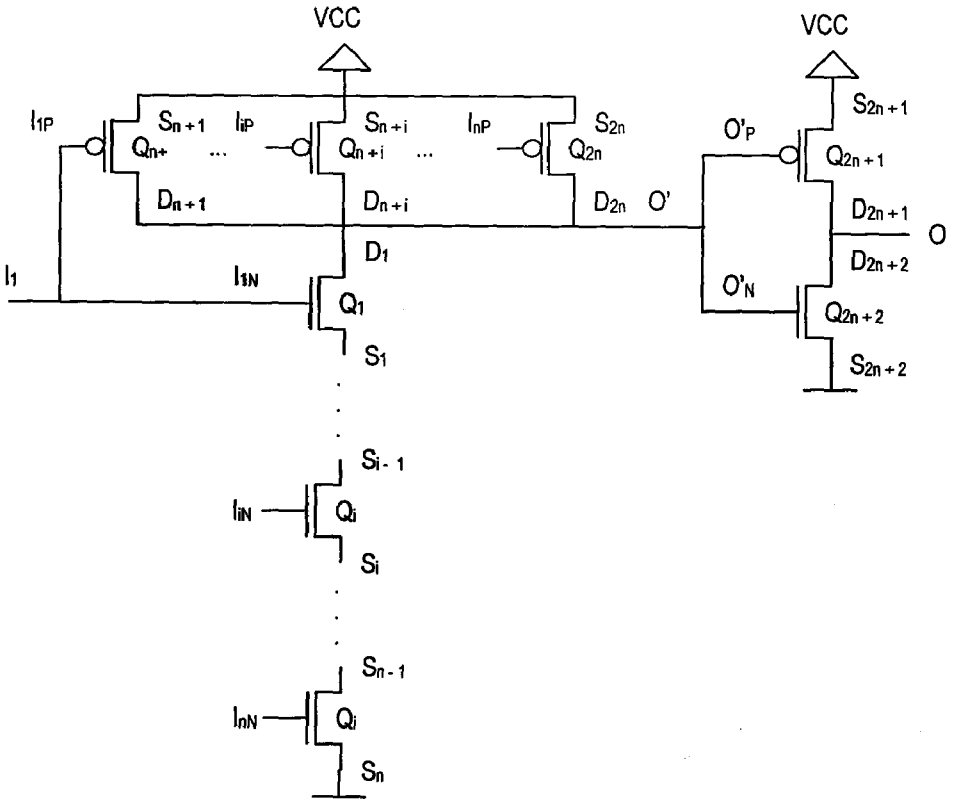


Figure 48: ANDn Transistor Level

- **Short Faults**

Short faults	Inputs conditions	Faulty output (O)
O' ←→ G	For any input combinations	1
O' ←→ VCC	For any input combinations	0
S_i ←→ G ($1 \le i \le n-1$)	$I_i=1 \ \forall \ 1 \le j \le n$	1
	$I_j = 1 \ (\forall \ 1 \le j \le i)$ & $(I_{i+1} \& I_{i+2} \& \ldots \& I_n=0)$	Stuck-On
	For the rest of input combinations	0
S_i ←→ VCC ($1 \le i \le n-1$)	For any input combinations	0
I_i ←→ I_k where $1 \le k \le n$ and $k \ne i$	$I_i=1 \ \forall \ 1 \le j \le n$	1
	$I_j=1 \ (\forall \ 1 \le k \le n$ and $j \ne i, j \ne k)$ & $I_i \oplus I_k=1$	X
	For the rest of input combinations	0
I_i ←→ S_k ($i \le k \le n-1$)	For any input combinations	0
I_i ←→ S_k ($1 \le k \le i-1$)	$I_j=1 \quad \forall \ 1 \le j \le k$, $I_j=<0\|1> \ \forall \ k+1 \le j \le n$	NOT(I_i)
	For all other input combinations	0
S_i ←→ S_k ($1 \le k \le n-1$ and $k \ne i$):	$I_i=1 \ \forall \ 1 \le j \le n$	1
	$I_j = 1 \ (\forall \ 1 \le j \le i$ and $m \le j \le n \ (i+1 \le m \le n-1))$ & $(I_{i+1} \& I_{i+2} \& \ldots \& I_m = 0)$	Stuck-On
	For the rest of input combinations	0
S_i ←→ O' ($1 \le i \le n-1$)	$I_i=1 \ \forall \ 1 \le j \le n$	1
	$I_j = 1 \ \forall \ (i+1 \le j \le n)$ & $(I_1 \& I_2 \& \ldots \& I_i = 0)$	Stuck-On
	For the rest of input combinations	0
I_i ←→ O ($1 \le i \le n$)	For all input combinations	I_i
I_i ←→ O' ($1 \le i \le n$)	For all input combinations	NOT(I_i)
S_i ←→ O ($1 \le i \le n-1$)	$(I_j = 1 \ (\forall \ 1 \le j \le i) \ \& \ (I_{i+1} \& I_{i+2} \& \ldots \& I_n=0))$ || $(I_i=1 \ \forall \ 1 \le j \le n)$	Stuck-On
	For the rest of input combinations	0
O' ←→ O	For any input combinations	NAND ($I_1, I_2 \ldots, I_n$)

Table 31: ANDn Short Faults

- **Open Faults**

Open faults	Logical fault type
I_{iP} : open ($1 \le i \le n$)	I_{iP}: stuck-at-1
I_i : open ($1 \le i \le n$)	
S_i: open ($n+1 \le i \le 2n$)	
D_i: open ($n+1 \le i \le 2n$)	
I_{iN} : open ($1 \le i \le n$)	I_{iN}: stuck-at-0
S_i: open ($1 \le i \le n$)	
D_i: open ($i = 1$)	
O'_P: open	O'_P: stuck-at-1
S_i: open ($i = 2n+1$)	
D_i: open ($i = 2n+1$)	
O'_N: open	O'_N: stuck-at-0
S_i: open ($i = 2n+2$)	
D_i: open ($i = 2n+2$)	
O':open	Z
O: open	Z

Table 32: ANDn Open Faults Categories

Logical fault type of open faults	Inputs conditions	Faulty output (O)
I_{iP}: stuck-at-1	$I_j=1 \ \forall \ 1 \le j \le n$	1
	$I_j=1$ ($\forall \ 1 \le j \le n$ and $j \ne i$) & $I_i=0$	Z
	For the rest of input combinations	0
I_{iN}: stuck-at-0	$I_j=1 \ \forall \ 1 \le j \le n$	Z
	For the rest of input combinations	0
O'_P: stuck-at-1	$I_j=1$ ($\forall \ 1 \le j \le n$)	Z
	For the rest of input combinations	0
O'_N: stuck-at-0	$I_j=1$ ($\forall \ 1 \le j \le n$)	1
	For the rest of input combinations	Z
Z	For any input combinations	Z

Table 33: ANDn Open Faults

The general formula for total number of faults for ANDn is:

$$\sum_{j=1}^{n-1}(n-j)+\sum_{k=2}^{n-1}(n-k)+n^2+13n+10 \qquad \text{(B.1)}$$

Appendix C: Buffer Fault list

- **Input/Output Stuck-at Faults**

Input/output stuck-at faults	Faulty output (O)
I: stuck-at-0	0
I : stuck-at-1	1
O: stuck-at-0	0
O: stuck-at-1	1

Table 34: Buffer Input/Output Stuck-at-faults

- **Short Faults**

Short faults	Faulty output
O'←→ I	NOT(I)
O'←→ O	NOT(I)
O'←→ VCC	0
O'←→ G	1
O ←→ I	I

Table 35: Buffer Shore Faults

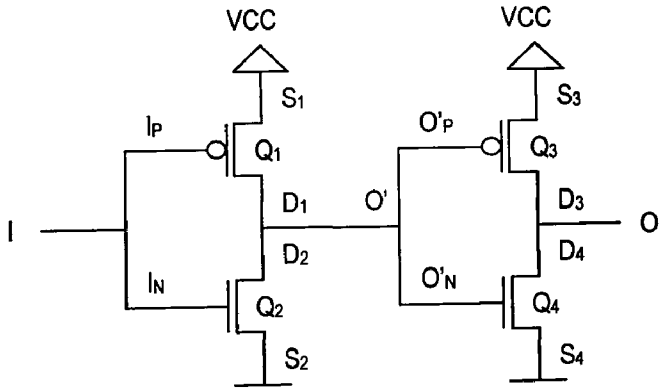


Figure 49: Buffer Transistor Level

- **Open Faults**

Open faults	Input conditions	Faulty output
I: open	N/A	Z
I_P: open	I = 0	Z
	I = 1	1
I_N: open	I = 0	0
	I = 1	Z
D_1: open	I = 0	Z
	I = 1	1
D_2: open	I = 0	0
	I = 1	Z
D_3: open	I = 0	0
	I = 1	Z
D_4: open	I = 0	Z
	I = 1	1
S_1: open	I = 0	Z
	I = 1	1
S_2: open	I = 0	0
	I = 1	Z
S_3: open	I = 0	0
	I = 1	Z
S_4: open	I = 0	Z
	I = 1	1
O': open	For any *I*	Z
O'_P: open	I = 0	0
	I = 1	Z
O'_N: open	I = 0	Z
	I = 1	1
O: open	For any *I*	Z

Table 36: Buffer Open Faults

The total number of faults for buffer is: *24*

Bibliography

[1] M.Abramovici, M.A. Breuer, A.D. Friedman, Digital System Testing And Testable Design, Revised Edition, IEEE Press, 1990.

[2] C.W. Wu, Lab for Reliable Computing (LaRC), EE, NTHU, (http://larc.ee.nthu.edu.tw/~cww/n/625/6250/02.pdf), 2002.

[3] J. Alt, U. Mahlstedt, Simulation of Non-classical Faults on the Gate Level – Fault Modeling, 11th VLSI Test Symposium, April 1993, pp. 351-354.

[4] R.D. Eldred, Test Routines Based on Symbolic Logical Statements, Journal ACM, Vol.6, No.1, 1959, pp.33-36.

[5] S.A. Arian, D.P. Agrawal, Physical Failure and Fault Model of CMOS Circuits, IEEE Transactions on Circuit and Systems, Vol. CAS-34, No. 3, March 1987, pp. 269-279.

[6] J. Galaiy, Y. Crouzet, M. Vergniault, Pphysical Versus Logical Fault Models MOS LSI circuits: Impact on their Testability, IEEE Transactions on computers, Vol. C-26, June 1980, pp. 527-531.

[7] S.D. Brown, Z.G. Vranesic, Fundamentals of Digital Logic with VHDL Design, McGraw-Hill, 2000.

[8] J.F. Wakerly, Digital Design Principles & Practices, Third Edition, Prentice Hall, 2002.

[9] http://tech-www.informatik.uni-hamburg.de/applets/cmos/cmosdemo.html

[10] Z. Nabavi, VHDL Analysis and Modeling of Digital Systems, Second Edition, McGraw-Hill, 1998.

[11] G.D. Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.

[12] R. Leveugle, Fault Injection in VHDL Descriptions and Emulation, Proceedings of the IEEE International Symposium on Defect and Fault Tolerance, Oct. 2000, pp. 414-419.

[13] S. Svensson, J. Karlsson, Dependability Evaluation of the THOR Microprocessor Using Simulation-Based Fault Injection, Technical report NO. 295, Chalmers University of Technology, Department of Computer Engineering, Sweden, 1997.

[14] E. Jenn et al, Fault Injection into VHDL models: the MEFISTO tool, FTCS, 1994, pp.66-75.

[15] E.Bohl, W. Harter, M.Trunzer, Real Time Effect Testing of Processor Faults, 5th IEEE International On-Line Testing Workshop, July 1999, pp. 39-43.

[16] R.Leveugle, Towards Modeling for Dependability of Complex Integrated Circuits, IEEE International On-Line Testing Workshop, Rhdes, Greece, July 1999, pp. 194-198.

[17] H.R. Zarandi, S.G. Miremadi, A. Ejlali, Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models, Proceedings of the 18th IEEE International Symposium Defect and Fault Tolerance in VLSI System, pp. 485-492, 2003.

[18] T.A. Delong, B.W. Johnson, J.A. Profeta III, A Fault Injection Technique for VHDL Behavioral-Level Models, IEEE Design and Test of Computers, Vol. 13, Winter 1996, pp. 24-33.

[19] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J.Karlsson, Fault Injection into VHDL Models: The MEFISTO Tool, 24th Symposium on Fault-Tolerant Computing (FTCS), 1994, pp. 66-75.

[20] J.Boue, P. Petillon, Y. Crouzet, MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance, 28th Symposium on Fault-Tolerant Computing (FTCS), 1998, pp. 168-173.

[21] K.T. Cheng, S.Y. Huang, W.J. Dai, Fault Emulation: A New Methodology for Fault Grading, IEEE Transactions on Computer-Aided Design, Vol. 18, No. 10, 1999, pp. 1487-1495.

[22] R.W. Wieler, Z. Zhang, R.D. McLeod, Emulating Static Faults Using a Xilinx Based Emulator, IEEE Symposium on FPGAs for Custom Computing Machines, February 1995, pp. 110-115.

[23] L. Antoni, R. Leveugle, B. Feher, Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Yamanashi, Japan, October 2000, IEEE Computer Society Press, 2000, pp.405-413.

[24] L. Antoni, R. Leveugle, B. Feher, Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes, the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Vancouver, Canada, 2002, IEEE Computer Society Press, 2002, pp. 242-249.

[25] P. Civera, L.Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, Exploiting FPGA for Accelerating Fault Injection Experiments, 7th IEEE International On-Line Testing Workshop, Taormina, Italy, July 2001, pp.9-13.

[26] P. Civera, L.Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, Exploiting FPGA-Based Techniques for Fault Injection Campaigns on VLSI Circuits, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, San Francisco, California, USA, October 2001, IEEE Computer Society Press, 2001, pp. 250-258.

[27] P. Civera, L.Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, FPGA-Based Fault Injection for Microprocessor Systems, Asian Test Symposium, November 2001, pp.304-309.

[28] R. Leveugle, A Low-Cost Hardware Approach to Dependability Validation of IPs, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, San Francisco, California, USA, October 2001, IEEE Computer Society Press, 2001, pp. 242-249.

[29] Two Flow for Partial Reconfiguration: Module Based or Difference based, Application Note: Virtex, Virtex-E, Virtex-II, Virtex-II Pro Families, XAPP290 (v1.1), Xilinx, 2003.

[30] S.M. Sait, H. Youssef, VLSI Physical Design Automation Theory and Practice, World Scientific, 1999.

[31] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithm, Second Edition, Mc Grow Hill, 2002.

[32] M.E. Zaghloul, D. Gobovic, Fault Modeling for Physical Failure for CMOS Circuits, IEEE International Symposium on Circuits and Systems, ISCAS'88, June 1988, pp. 677-680.

[33] P. Dahlgren, P. Liden, A Fault Model for Switch-Level Simulation of Gate-to-Drain Shorts, 14th Proceedings of VLSI Test Symposium, 1996, pp.414-421.

[34] M.E. Zaghloul, D. Gobovic, Fault Modeling of Physical Failure in CMOS VLSI Circuits, IEEE Transactions on Circuits and Systems, Vol. 37, No. 12, December 1990, pp. 1528-1543.

[35] M.L. Flottes, C. Landrault, S. Pravossoudovitch, Fault Modeling and Fault Equivalence in CMOS technology, Proceedings of the European Design Automation Conference, EDAC, March 1990, pp. 407-412.

[36] D.P. Milovanovic, V.B. Litovski, Fault Models of CMOS Circuits, Microelectronics Reliability Journal, Vol. 34, No. 5, 1994, pp. 883-896.

[37] V.Sieh, O. Tschache, F. Balbach, VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Description, Proceedings of 27th International Symposium on Fault-Tolerant Computing, 1997, pp. 32-36.

[38] P. Folkesson, S. Svensson, J. Karlsson, A Comparison of Simulation Based and Scan chain Implemented Fault Injection, Proceedings of 28th International Symposium on Fault-Tolerant Computing, 1998, pp.284-293.

[39] P.Civera, L.Macchiarulo, M. Rebaudengo, M.Sonza Reorda, M.Violante, Exploiting Circuit Emulation for Fast Hardness Evaluation, IEEE Transaction Nuclear Science 48, 2001, pp.2210-2216.

[40] R. Leveugle, Behavior Modeling of Faulty Complex VLSIs: Why and How?, Baltic Electronics Conf. Tallinn, Estonia, , October 1998, pp. 191-194.

[41] R.sedaghat-Maman, Fault Emulation with Optimized Assignment of Circuit Nodes to Fault Injectors, Proceedings of the 1998 IEEE International Symposium on Circuit and Systems, ISCAS'98, 1998, pp.135-138.

[42] R. Sedaghat-Maman, E. Barke, A New Approach to Fault Emulation, Proceedings of 8th IEEE International Workshop on Rapid System Prototyping, June 1997, pp.173-179.

[43] J.H. Hong, S.A. Hwang, C.W Wu, An FPGA-Based Hardware Emulator for Fast Fault Emulation, IEEE 39th Midwest Symposium on Circuit and Systems, Vol. 1, August 1997, pp.345-348.

[44] T.J. Chakraborty, C.H. chiang, A Novel Fault Injection Method for System Verification Based on FPGA Boundary Scan Architecture, Proceedings of International Test Conference, 2002, pp.923-929.

[45] S.A. Hwang, J.H. Hong, C.W. Wu, Sequential Circuit Fault Simulation Using Logic Emulation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17, 1998, pp. 724-736.

[46] R. Sedaghat-Maman, E. Barke, Real Time Fault Injection Using Logic Emulators, Proceedings Asia and South Pacific Design Automation Conference, 1998, pp.475-479.

[47] L.Antoni, R.Leveugle, B.Feher, Using Run-time Reconfiguration for Fault Injection Applications, IEEE Transactions on Instrumentation and Measurement, Vol. 52, No. 5, October 2003, pp. 1468-1472.

[48] A. Ejlali, S.G. Miremadi, FPGA-Based Fault Injection into Switch Level Models, Elsevier Microprocessors and Microsystems journal, 2004, pp.317-327.

[49] www.eg3.com/eCLIPS/desc/soc_vhdl_blank.html

[50] F. Moraes, N. Calazans, L. Möller, E. Brião, E. Carvalho, Chapter 1: Dynamic and Partial Reconfiguration in FPGA SoCs: Requirements Tool and a Case Study, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Brazil.

[51] http://dec.bournemouth.ac.uk/drhw_lib/terminology.html

[52] www. Xilinx.com/products/design_resources/design_tool/grouping/synthesis.htm (Xilinx: Synthesis)

[53] www.xilinx.com/xlnx (Techtips: xilinx synthesis technology)

[54] Xilinx Development System Reference Guide,
(http://toolbox.xilinx.com/docsan/xilinx6/books/data/docs/dev/dev0001_1.html)

[55] Xilinx XST User Guide,
http://toolbox.xilinx.com/docsan/xilinx6/books/data/docs/xst/xst0001_1.html

Publications:

1. R. Abedi, R. Sedaghat, Transistor-level to Gate-level Comprehensive Fault Synthesis for *n*-Input Primitive Gates, Accepted with Revision by Microelectronics Reliability Journal, ELSEVIER.
2. R. Abedi, R. Sedaghat, Synthesis of Transistor Level Fault Emulation, Submitted to Microelectronics Journal, ELSEVIER.
3. R. Abedi, R. Sedaghat, Classical and Non-classical Transistor Level Fault Injection into FPGA, Submitted to WSEAS Journal.
4. R. Abedi, R. Sedaghat, Synthesis of Exhaustive CMOS Transistor Fault Model at Gate Level for *n*-Input Primitive Gates, Submitted to ASP-DAC Conference.