Reza Sedaghat

Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping

December 1999

Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping

Dem Fachbereich Elektrotechnik und Informationstechnik
der Universität Hannover

zur Erlangung des akademischen Grades
Doktor-Ingenieur

genehmigte Dissertation

von

Dipl.-Ing. Reza Sedaghat Maman
geboren am 9. Februar 1964 in Teheran

1999

1. Referent:	Prof. Dr.-Ing. Erich Barke
2. Referent:	Prof. Dr.-Ing. Joachim Mucha
Tag der Promotion:	08.12.1999

Acknowledgments

This work originated during my activity as a Research Assistant in the Institute of Microelectronic Systems at the University of Hanover, Germany.

My thanks to my principle advisor Prof. Dr.-Ing. E. Barke for the opportunity to complete this work as well as to Prof. Dr.-Ing. J. Mucha for serving as co-advisor and Prof. Dr.-Ing. P. Pirsch for his chairmanship of the examination committee. In addition I would like to thank Prof. Dr. rer. nat. D. Müller, Institute of Computer Science, University of Hanover, who looked at parts of this work and provided helpful comments and suggestions. My appreciation to my former colleague Dipl.-Ing. Jörn Stohmann and to Dr.-Ing. Jürgen Alt and Dr.-Ing. Jan Otterstedt whose ideas during our many productive discussions contributed to the success of this work.

My gratitude to my father-in-law, Prof. Dr. Ulrich Petersen, Harvard University, Cambridge, USA. I have profited greatly from his academic and personal experiences and am grateful to him for his invaluable help. I am indebted to my parents for accompanying me along my journey. They have been a great source of strength for me. Finally, a special thanks to my son Sam who always managed to amuse me at high-stress times and to my wife Valerie for her endless patience and support. Without her I would not have been able to complete this work.

Hanover, December 1999

Reza Sedaghat

Abstract

Reza Sedaghat Maman

Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping

Various approaches to test vector evaluation exist for ascertaining the effectiveness of a test vector set for a specific fault model by computing the ratio between the number of faults detected by this set and the number of modeled faults**.** The traditional approach to test vector evaluation is software-based, utilizing programs to simulate the effects of the faults on circuit behavior. The simplest method, serial fault simulation, simulates faulty circuits one at a time. In the recent past, more advanced approaches to fault simulation have been proposed and can be categorized, in general, as either parallel or concurrent. These differ from serial fault simulation in their effort to minimize the number of simulation passes by processing faults or test vectors simultaneously. However, the circuit elements must still be processed sequentially in order to simulate the complete circuit. The fault simulation approach is becoming increasingly impractical nowadays, not only because the runtime for simulating one test vector increases linearly to quadratically with the number of circuit elements, but also because circuit complexity increases faster than computing speed.

A new approach to fault simulation involves the use of a hardware logic emulator. Logic emulation represents a new method of design validation utilizing a reprogrammable prototype of a digital circuit. In contrast to fault simulation, all circuit elements can be emulated in parallel by the emulation hardware. Therefore, emulation runtime is based solely on the number of faults, which of course also depends on circuit size, and the number of test vectors. Emulation runtime increases only linearly with circuit size making it possible to attain a speedup over software fault simulation. With the goal of satisfying the requirements of rapid fault injection including fault activation, emulator technology independence, optimal fault emulation runtime, minimal hardware overhead, and optimized mapping into reconfigurable hardware, two approaches to fault emulation, FES/1 and FES/2, were developed and implemented. Both approaches use identical methods of fault injection and fault activation in the FPGAs. However, FES/1 uses the so-called in-circuit mode of the emulator, in which test generation and emulation analysis are made feasible through the expansion of the logic emulator by additional hardware modules. FES/2, in contrast, operates in emulator acceleration mode and does not require additional hardware for test vector evaluation.

An objective of hardware-based fault injection is the reduction of the FPGA overhead, which results from the fault emulation mapping procedure. This method of fault injection includes mapping the faulty circuit for an optimized partitioning, technology mapping, and placement and routing. The Delta-Path algorithm was developed and utilized in the course of this research for the node assignment optimization problem. The problem is described here as a quadratic assignment problem and its solution using the Delta-path algorithm results in a reduction in FPGA overhead through an improved usage of FPGA resources. In contrast to previously published fault emulation approaches, FES/1 and FES/2 use additional logic functions for fault injection and decoders for fault activation. Faster fault injection is feasible without reconfiguration of the emulator hardware and without dependency on a specific logic emulator technology.

In addition, the dependability of a system can be evaluated using a logic emulator for hardware-based fault injection. Real time fault injection into a target system hardware is an important application of fault emulation for the evaluation of system behavior and involves fault injection into the system for the identification of dependability deficiencies of the system, the observation of system behavior with the given faults, as well as the determination of the degree of fault coverage.

Keywords: FPGA, Fault Injection, Fault Simulation

Abstract

Reza Sedaghat Maman

Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping

Mehrere Methoden der Testmusterevaluierung existieren, welche die Effektivität eines Testmustersatzes für einen spezifischen Fehlermodell feststellen können, indem das Verhältnis zwischen der Anzahl der entdeckten Fehler und der Anzahl der modellierten Fehler berechnet wird. Das traditionelle softwarebasierte Verfahren der Testmusterevaluierung setzt Programme ein, die die Wirkung von Fehlern auf das Verhalten der Schaltung simulieren. Die einfachste Methode, Serielle Fehlersimulation, simuliert Fehler einen nach dem anderen. In den letzten Jahren wurden Methoden der Fehlersimulation vorgestellt, die im allgemeinen als "Parallel" und "Concurrent" bezeichnet werden. Diese unterscheiden sich von serieller Fehlersimulation indem versucht wird, die Anzahl der Simulationsabläufe durch eine gleichzeitige Bearbeitung von Fehlern oder Testmustern zu minimieren. Um die komplette Schaltung simulieren zu können, müssen aber die Schaltungselemente immer noch sequentiell bearbeitet werden. Heutzutage werden Fehlersimulationsmethoden zunehmend unpraktisch, nicht nur weil die Simulationslaufzeit mit der Anzahl der Schaltungselemente linear bis quadratisch steigt, sondern auch weil die Schaltungskomplexität schneller als die Rechengeschwindigkeit wächst.

Logikemulation ist eine neue Methode der Designverifikation, die einen reprogrammierbaren Prototypen einer digitalen Schaltung darstellt. Fehlersimulation mittels eines hardwarebasierten Logikemulators (Fehleremulation) repräsentiert ein neues Verfahren der Schaltungsvalidierung. Im Gegensatz zur softwarebasierten Fehlersimulation können mit Fehleremulation alle Schaltungselemente in einem Emulationstakt durch die Emulationshardware berechnet werden. Die Emulationslaufzeit ist lediglich abhängig von der Anzahl der Fehler, welche natürlich von der Schaltungsgröße und der Anzahl der Testmuster abhängt. Die Emulationslaufzeit steigt nur linear mit der Schaltungsgröße und ermöglicht daher ein Speed-up über softwarebasierte Fehlersimulation. Zwei Verfahren der Fehleremulation, FES/1 und FES/2, wurden entwickelt und implementiert, welche die Bedingungen der schnellen Fehlerinjektion, einschließlich der Fehleraktivierung, Unabhängigkeit von Emulatortechnologie, optimaler Fehleremulations-laufzeit und minimalen Hardware-Overhead sowie optimierter Abbildung in rekonfigurierbarer Hardware, erfüllen. FES/1 verwendet der sogenannten In-Circuit-Mode des Emulators, in der Testgenerierung und Analyse der Emulationsergebnisse durch die Erweiterung des Logikemulators mit zusätzlichen Hardwaremodulen ermöglicht werden. FES/2 benutzt den Emulator-Acceleration Mode**,** welcher keine zusätzliche Hardware für die Generierung und Auswertung von Testmustern benötigt. Die beiden Verfahren setzen identische Methoden der Fehlerinjektion und Fehleraktivierung in den FPGAs ein.

Ein Ziel der hardwarebasierten Fehlerinjektion ist die Reduzierung des FPGA-Overheads, der aus dem FES Verfahren resultiert. Hardwarebasierte Fehlerinjektion schließt die Abbildung der fehlerhaften Schaltung für eine optimierte Partitionierung, Technologiemapping, Plazierung und Routen ein. Das Delta-Path Algorithmus wurde im Laufe dieser Forschung für die Zuordnung von Fehlerknoten der Schaltung in dem Fehleraktivator entwickelt und implementiert. Das Problem wird in dieser Arbeit als ein quadratisches Zuordnungsproblem beschrieben. Eine suboptimale Lösung des Problems ist mit dem Delta-Path Algorithmus möglich und führt durch den optimierten Verbrauch von FPGA-Ressourcen zu einer Reduzierung des FPGA-Overheads. Im Gegensatz zu früheren veröffentlichten Verfahren der Fehleremulation, setzten FES/1 und FES/2 zusätzliche Logikfunktionen für die Fehlerinjektion, sowie Zeilen- und Spaltendekoder für die Fehleraktivierung ein. Schnellere Fehlerinjektion ist möglich ohne Rekonfiguration der Emulatorhardware und ohne von einer spezifischen Logikemulatortechnologie abhängig zu sein.

Über die Testvektorevaluierung hinaus bieten die in dieser Arbeit beschriebenen Verfahren große Vorteile bei der Beurteilung der Zuverlässigkeit von Systemen. Diese kann mittels eines Logikemulators für hardwarebasierte Fehlerinjektion evaluiert werden. Echtzeitfehlerinjektion in eine Zielsystemhardware ist für die Evaluierung des Systemverhaltens einer wichtige Aspekt der Fehleremulation, welcher Fehlerinjektion in ein System für die Identifizierung von Systemzuverläßigkeitsdefiziten, sowie die Beobachtung des Systemverhaltens mit den gegebenen Fehlern, und die Feststellung des Fehlerüberdeckungsgrades einschließt.

Schlagwörter: FPGA, Fehlerinjektion, Fehlersimulation

Contents

Notation Index .... **III**

List of Figures and Tables .... **VI**

1. Introduction **1**

2. Logic Emulation **4**

2.1 Logic Validation Techniques 4

2.1.1 Logic Simulation 4

2.1.2 Formal Verification 5

2.1.3 Logic Emulation 6

2.1.4 Design Prototyping 8

2.1.5 Comparison 9

2.2 FPGA-based Logic Emulation Design Flow 11

3. Fault Simulation **13**

3.1 Fault Models 13

3.2 Fault Redundancy, Equivalence, and Dominance 16

3.3 General Approach to Fault Simulation 19

3.4 Fault Simulation Techniques 20

3.5 Test Generation 28

4. Fault Emulation **33**

4.1 State of the Art 33

4.1.1 Timoc Approach 33

4.1.2 Fault Grading Method 35

4.1.3 Serial Fault Emulation (SFE) 37

4.1.4 KRONE 40

4.1.5 Objectives of this Work 42

4.2 Fault Emulation System (FES) 43

4.2.1 FES/1 44

4.2.2 FES/2 47

4.2.3 Fault Injection 48

4.2.4 Fault Activation 50

4.2.5 Combinational Circuits 52
4.2.6 Sequential Circuits 53

5. Node Assignment 61
5.1 Introduction 61
5.2 Placement and Quadratic Assignment Problems 63
5.3 Fault Activator and Node Assignment 67
5.3.1 Partitioning of an Expanded Circuit 69
5.3.2 Technology Mapping of an Expanded Circuit 73
5.3.3 Placement and Routing of an Expanded Circuit 74
5.4 Optimized Node Assignment 77
5.4.1 Algorithms 79
5.4.1.1 Simulated Annealing 80
5.4.1.2 Min-Cut 81
5.4.1.3 Delta-Path 83

6. Experimental Results 90
6.1 CLB-Overhead 90
6.2 Fault Emulation Runtime 95

7. Conclusion and Future Work 104

8. References 107

Notation Index

CLB	Configurable Logic Blocks
LUT	Look-Up Table
PI	primary input of a circuit
f_d	delay fault
δ	additional delay (delay fault)
Del	delay
ΔDel	slack
T_N	nominal time
f_t	transition fault
V	test vector set
$v_1,v_2,...,v_n$	test vectors
$Bf(v)$	logical function of faulty circuit
$B(v)$	logical function of fault-free circuit
$F=\{f_1, f_i, ..., f_n\}$	set of faults
f	fault
s-a-0	stuck-at- 0 fault
s-a-1	stuck-at-1 fault
γ	fault coverage
val	value of a signal
$mask1$, $mask2$	parallel fault simulation masks
$\hat{E}$	number of gate calculations per simulation run
g	gates are evaluatuation per second
sp	seril fault emulation speed
GT	gate
$m_1, m_2, ..., m_r,$	PRTG feedback connections
$P(a)$	polynomial function of a PRTG
Rf	fault emulation runtime (Cheng&Dai)
Rt	total runtime of fault emulation (Cheng&Dai)
T_{reconf}	time required for reconfiguration of a BLP (SFE)
AN	average number of test vectors necessary for fault detection (SFE)
SP	speed of SFE approach
$FLO=\{flo_1, flo_i, ..., flo_n\}$	set of fault locations
$FI=\{fi_1, fi_i, ..., fi_n\}$	set of Fault Injectors
L	control signal of the line in the Fault Activator
C	control signal of the column in the Fault Activator

N_{in}	data input of the Fault Injector
N_{out}	data output of the Fault Injector
EN	control signal of the Fault Injector
SF	control input of the Fault Injector to select the stuck-at fault model
$A=\{A_1, ..., A_n\}$	input data of the Fault Activator
Nf	number of faults
i	number of line decoders
j	number of column decoders
t	clock period
ξ	state variables of a sequential circuit
I^t	inputs of a sequential circuit
r	next clock period of a sequential circuit
η	function of dependency of the state variables ξ^{t+r} on the inputs I^t
VS	test vector sequence
$v_{Initial}$	initial test vector
IS	initial state of a sequential circuit
x	feasible point of a cost function
$\tilde{n}$	neighborhood of x
$\tilde{N}(x,\tilde{n})$	set of neighborhoods of x
x_{local}	local minimum of a cost function
x_{global}	global minimum of a cost function
$d_{lk(i)lk(j)}$	distance of the locations of modules i and j
$lk(i) = [lk(i)^x, lk(i)^y]$	locations of module i
$lk(j) = [lk(j)^x, lk(j)^y]$	locations of module j
π	permutation
Π	the set of permutations
π_{opt}	optimized permutation
$MD=\{md_1, md_2, ..., md_n\}$	set of modules
$SG= \{sg_1, sg_2, ..., sg_k\}$	set of signals
$LK=\{lk_1, lk_2, ..., lk_n\}$	set of locations
$G(U_{I/O}, U_N, U_I, E)$	extra node graph
$U_{I/O}$	set of *I/O* nodes
U_N	set of extra nodes
U_I	set of instance nodes
E	set of edges

$w(e)$	weight of an edge
Φ	cutsize
ρ	placement
ε_i	switch block side i
$\psi\rho(\varepsilon)$	capacity of the switch block side
$\eta\rho(\varepsilon)$	number of nets on each side of a switch block
$\hat{F}(W, D)$	cost function of the node assignment problem.
NA	node assignment
QAP	quadratic assignment problem
D	distance matrix
En	energy
$Temp$	Temperature
P	probability
Θ	partition of a graph
u^s	source nodes
$\Gamma^-_{(u)}$	set of predecessor node u
$\Gamma^+_{(u)}$	set of successor node u
$\Lambda_1, \Lambda_2, \ldots, \Lambda_k$	subsets of U
$\deg(u^{(v)})^+$	number of successor nodes of v
Δ_z	delta path
Δ	set of delta paths
$u \mapsto fi$	node $u \in \Delta$ is assigned to a Fault Injector $fi \in lk$
$Freq$	the emulation frequency
$RT_{FE}^{FES/1}$	total fault emulation runtime for FES/1
$RT_{FE}^{FES/2}$	total fault emulation runtime for FES/2
P_{avg}	average number of test vectors necessary to detect a fault
$\hat{O}$	number of test vectors
RT_F	fault emulation runtime (FES)
RT_G	good emulation runtime (FES)
RT_{total}	total runtime of fault emulation (FES)

List of Figures and Tables

Fig. 1.1	Y-diagram of digital circuit
Fig. 1.2	Overview of fault simulation approaches
Fig. 2.1	A typical logic emulator
Fig. 2.2	Time frames
Fig. 2.3	A typical design flow of FPGA-based emulation system
Fig. 3.1	CMOS INVERTER at electrical and logic level
Fig. 3.2	Stuck-at fault model
Fig. 3.3	Modeling of bridging fault as wired-AND
Fig. 3.4	Fault free (a) and faulty (b) signal delay
Fig. 3.5	Undetectable fault
Fig. 3.6	Redundancy
Fig. 3.7	Equivalence fault collapsing
Fig. 3.8	Fault coverage curve
Fig. 3.9	Parallel fault simulation
Fig. 3.10	Fault injection on signal *h*
Fig. 3.11	Fault list propagation
Fig. 3.12	Concurrent fault simulation
Fig. 3.13	Fault propagation of f_1 to output *p*
Fig. 3.14	Principle of two-valued Parallel Pattern Single Fault Propagation
Fig. 3.15	Classification of test generation
Fig. 3.16	Backward trace and forward trace along a sensitive path
Fig. 3.17	Conceptual Linear Feedback Shift Register (LFSR)
Fig. 3.18	Hardware realization of a LFSR as a pseudo-random test generator
Fig. 3.19	Test generation of Fig. 3.18
Fig. 4.1	Hardware fault simulator for an AND gate [Timo79]
Fig. 4.2	Primitive serial fault emulation process [ChHu95]
Fig. 4.3	Fault injection and activation of fault grading method [ChHu95]
Fig. 4.4	SFE flowchart [BuRe96]
Fig. 4.5	An example of FPGA reconfiguration [BuRe96]
Fig. 4.6	Delay lines [Kron96]
Fig. 4.7	FPE-cell [Kron96]
Fig. 4.8	FES/1 flow diagram
Fig. 4.9	Fault emulation hardware
Fig. 4.10	Comparison hardware module
Fig. 4.11	FES/2 flow diagram

Fig. 4.12	Stuck-at-0 Fault Injector
Fig. 4.13	Stuck-at-1 Fault Injector
Fig. 4.14	Stuck-at-0/1 Fault Injector
Fig. 4.15	Fault Activator
Fig. 4.16	Combinational circuit in a CLB
Fig. 4.17	Fault Injector s-a-0 in a CLB
Fig. 4.18	Generic schematic of sequential circuits
Fig. 4.19	State transition graph
Fig. 4.20	Initialization of storage element with 3- and 4-valued logic
Fig. 4.21	Fault emulation for a sequential circuit
Fig. 4.22	Mapping of the sequential circuit into an FPGA
Fig. 4.23	Hardware initialization of individual storage elements
Fig. 5.1	Local versus global optima
Fig. 5.2	Wirelength estimation methods
Fig. 5.3	An example of the placement problem
Fig. 5.4	Model of a symmetrical FPGA
Fig. 5.5	Node assignment model
Fig. 5.6	Circuit without Fault Injectors (left) modeled as extra node graph
Fig. 5.7	Circuit partitioning of expanded circuit
Fig. 5.8	Circuit with Fault Injectors
Fig. 5.9	Bipartition with optimized node assignment and cutsize Φ=7
Fig. 5.10	Bipartition with non-optimized node assignment and cutsize Φ=9
Fig. 5.11	Technology mapping of an expanded circuit
Fig. 5.12	Placement of circuit with non-optimized node assignment in an FPGA
Fig. 5.13	Placement of circuit with optimized node assignment in an FPGA
Fig 5.14	Optimized node assignment compared to random node assignment
Fig. 5.15	Vertical and horizontal cut lines
Fig. 5.16	Cut tree
Fig. 5.17	Slice/Bisection procedure
Fig. 5.18	Dividing of U into k subsets $\Lambda_1, \Lambda_2, \ldots, \Lambda_k$
Fig. 5.19	One-dimensional symbolic placement
Fig. 5.20	Assignment of u to fi
Fig. 5.21	Node assignment with Delta-Path algorithm
Fig. 5.22	One-dimensional symbolic placement
Fig. 5.23	Super node splitting
Fig. 6.1	Comparison of node assignment runtime for different algorithms
Fig. 6.2	Reduction of CLB number for s-a-0
Fig. 6.3	Reduction of CLB number for s-a-1

Fig. 6.4	Runtimes of fault emulation FES/1 and FES/2 and Comsim
Fig. 6.5	Speedup of fault emulation FES/1 and FES/2 over fault simulation with Comsim
Fig. 6.6	Simulated runtime of VED and calculated runtime of fault emulation FES/1
Fig. 6.7	Speedup of fault emulation FES/2 over fault simulation with VED
Fig. 7.1	Evaluation of system dependability using logic emulation

Table 4.1	Comparison of available hardware-based fault simulation approaches
Table 4.2	State transition table
Table 4.3	Emulation results for initial states (A) and (D)
Table 6.1	Comparison of node assignment results for stuck-at-0 faults
Table 6.2	Comparison of node assignment results for stuck-at-1 faults
Table 6.3	Logic emulation of circuit without Fault Injectors
Table 6.4	Fault emulation with stuck-at-0 Fault Injectors
Table 6.5	Fault emulation with stuck-at-1 Fault Injectors
Table 6.6	Comparison of fault emulation and Comsim
Table 6.7	Comparison of fault emulation FES/1 and VED
Table 6.8	FES/1 and FES/2 approaches based on the comparison of Table 4.1
Table 6.9	Overview of existing approaches

1. Introduction

An essential part of modern electronic systems are Very Large Scale Integrated (VLSI) circuits. These circuits contain between thousands and millions of transistors, diodes, and other components such as resistors, capacitors, and interconnections within a very small area. The design of such circuits is a complicated and time-consuming process. During the design process an integrated circuit is modeled on different abstraction levels. These abstraction levels represent the information necessary for the actual step in the design process. The "Top-Down" design begins at a high level and precedes downwards to the next level where more detailed information of the circuit is examined. The level of abstraction can be characterized by the type of information processed, as shown in Fig. 1.1. A circuit design can generally be subdivided into three main categories according to the view from which a circuit is considered. These include the design with which the behavior of a circuit is observed, the design in which the structure of a circuit is established, and the physical design (geometry).

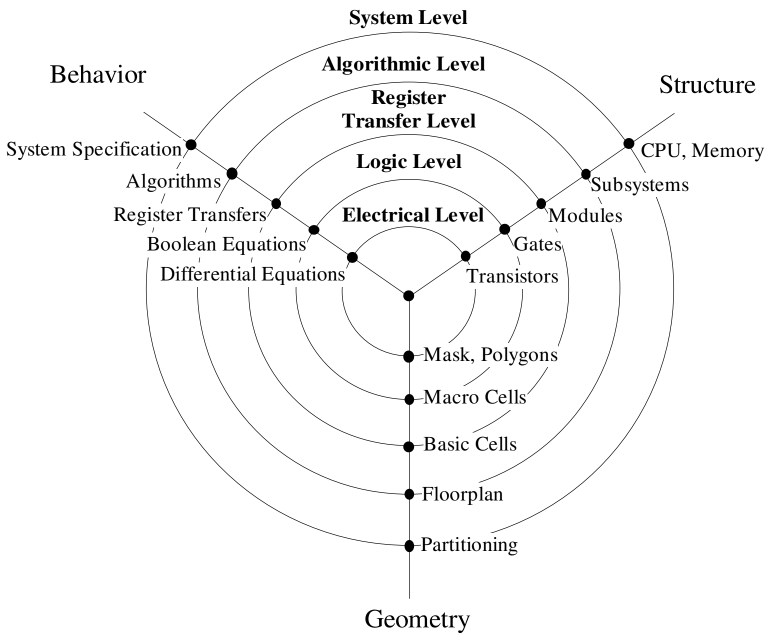


Fig. 1.1: Y-diagram of digital circuit

Starting with an idea about the design of the circuit, a behavioral description of the circuit is written in a high-level language like VHDL [Waxm89] [Lips89] [Coel89]. The behavioral specification is then converted into a register transfer level (RTL) description of the circuit using a synthesis tool. After a register transfer level description has been obtained it is mapped into logic equations. Typically, a structural register transfer level description is an interconnection of predefined modules such as adders, multipliers, memory, etc. At logical level the design is represented by a combination of primitives, for example AND-, OR- , XOR-gates, Flip-Flops etc. which are present in a library. The characteristics of these basic elements are defined in the library and reproduce in a simplified form the characteristics of the target technology. The behavioral aspects at the logical level of the circuit can be represented using Boolean equations. The next step in the process is the production of a mask level description or a layout of the circuit in a given technology. Module generators [OnLL89] can be used to produce a layout for each module in the design. The modules or gates are placed and routed using placement and routing tools [Leng90]. The mask level description is used to manufacture the integrated circuit (IC). Since no manufacturing process can not guarantee 100% yield, manufacturing defects are usually introduced during the manufacturing process. The actual type of defect is technology dependent. The larger the circuit in terms of area, the higher the probability of defects. Clearly, testing is crucial to the VLSI manufacturing process.

As described in Chapter 3, one application of fault simulation includes the simulation of a circuit in the presence of faults. Faults are detected in a circuit by comparing the results of a fault simulation to the results of a fault-free or good simulation using a test vector set. Differing results indicate the detection of a fault. Hence, fault simulation is implemented for the evaluation of a test vector set. Furthermore, fault simulators are also used to increase the efficiency of programs used for test vector calculation [ScTr87]. Fault simulation is generally divided into two approaches: software-based and hardware-based fault simulation.

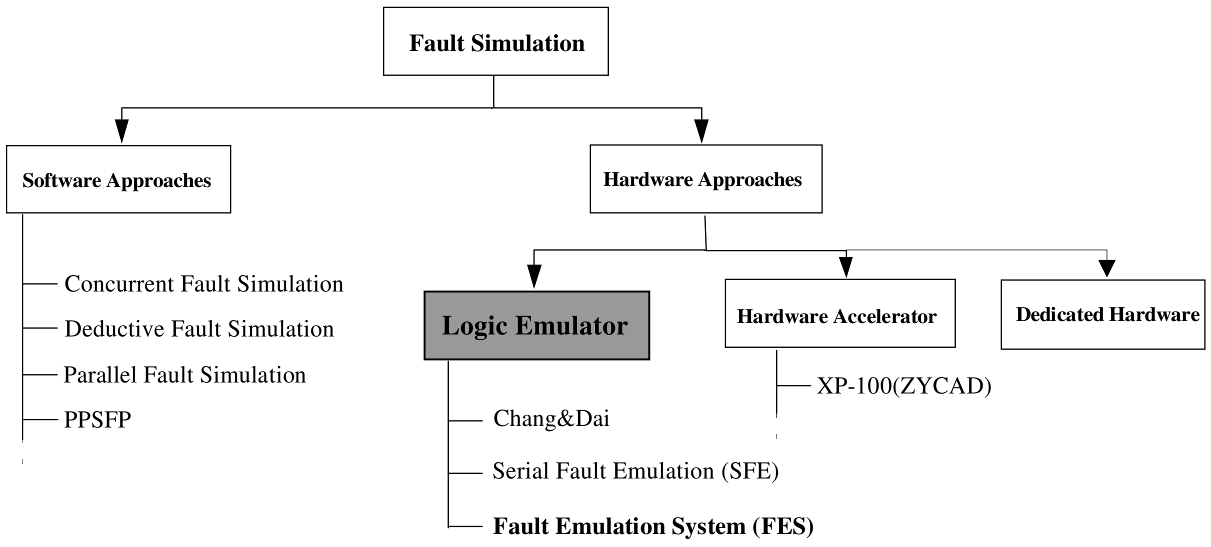

Fault Simulation
Software Approaches
Hardware Approaches
Concurrent Fault Simulation
Deductive Fault Simulation
Parallel Fault Simulation
PPSFP
Logic Emulator
Chang&Dai
Serial Fault Emulation (SFE)
Fault Emulation System (FES)
Hardware Accelerator
XP-100(ZYCAD)
Dedicated Hardware

Fig. 1.2: Overview of fault simulation approaches

Due to the increasing complexity of integrated circuits as well as the competition-based requirement for a shorter time-to-market of the product, software-based approaches can not satisfy the present demand for fault simulation. A significant amount of time, anywhere from minutes to days, is required by the simulation process of a complex circuit with millions of gates. Efforts have been made to shorten the necessary simulation time by developing specialized simulation accelerators [Zyca94a], logic emulators [BuBa90], and the first hardware-based fault simulator [Timo79]. In the past years various methods have been presented for the generation of hardware-based faulty circuits, each with its own approach to fault injection. A common feature of each of these methods is the execution of hardware-based fault simulation at gate level and the modeling of faults using a stuck-at fault model. Different techniques involving the use of a circular shift register or the reprogrammability of field programmable gate arrays (FPGAs) are utilized for the activation of faults in the circuit.

Various logic validation techniques are presented in Chapter 2, followed by a discussion of their advantages and disadvantages as well as the application of a logic emulator in the design phase. Chapter 3 introduces the most widely used fault simulation and test vector generation techniques and in addition, a hardware-based pseudo-random test vector generator utilized for hardware-based fault simulation.

In Chapter 4 the requirements for a novel approach to hardware-based fault simulation using a logic emulator (fault emulation) are described. A Fault Emulation System FES is presented, characterized by rapid fault injection and fault activation through a Fault Activator. Chapter 5 presents procedures for mapping faulty circuits while focusing on optimal partitioning, technology mapping, placement, and routing. A new algorithm for the optimized mapping of a faulty circuit is detailed in Chapter 5. Chapter 6 presents and discusses the experimental results of fault emulation FES in comparison to existing fault simulation and fault emulation approaches. A discussion follows on the optimized mapping of faulty circuits into logic emulators compared to existing algorithms such as simulated annealing. Chapter 7 concludes this thesis and discusses future work in this area.

2 Logic Emulation

2.1 Logic Validation Techniques

A comparison between the specification and the implementation of a digital circuit design is necessary in order to recognize design errors and to attain a correct implementation. Generally, such a comparison is referred to as logic validation. Rather than using the final version of a design, logic validation is usually carried out on more abstract levels of a design, often utilizing a high-level description of the circuit's functionality. Validating such high-level implementations is desirable for locating and correcting errors early, rather than identifying errors at the end of the implementation procedure when the design has developed more detail and complexity. Various logic validation methods are presented in the following sections along with a discussion of their advantages and disadvantages. As discussed in the final section of this chapter, the proper method of logic validation is not a single technique, but rather a combination of techniques that takes advantage of the different strengths of each validation technique.

2.1.1 Logic Simulation

Software simulation is perhaps the most widespread and effective method of logic validation and is preferred mainly due to its ease of signal observability and controllability. Software simulation is used to model and observe the functional behavior of a circuit [RuSa89] [Brew77]. A unit-delay simulator disregards all electrical characteristics of a circuit, with each gate requiring one time unit for each new input. The simulator examines all input to the gates and then calculates the correct value for each output. The behavior of the circuit can be observed by varying the values at the inputs. Other types of software simulators may also be utilized, which allow for more detailed modeling of the circuit [BARZ87][Spir85].

During the simulation process the software registers each signal value as well as changes to these values and follows the changing values over time. This information is made available to the user at all times. For various reasons the designer may want to alter the values of a net during a simulation. This is easily done because the values have been stored in the simulator. The designer may want to observe the behavior of a circuit in a specific configuration, even though in the final hardware implementation it can be difficult to force the circuit into this configuration. The user of a software simulator can simply instruct the simulator to make the desired changes to the circuit and then study the resulting behavior. Circuit changes carried out this early in the design process save effort and time that would otherwise be spent later to locate and fix errors which are more complicated in the final implementation.

The flexibility of software simulation described above also has major disadvantages. The simulation process requires a significant amount of time to evaluate a complex circuit with millions of gates, anywhere from minutes to days. Through the development of specialized simulation accelerators [Zyca94a] [Zyca94b] [Zyca94c] the runtime required by simulation has been reduced. However, software simulation with accelerators still requires more time to execute than a hardware implementation of the circuit, even when factors such as detailed timing are not considered. On the other hand, software simulation provides the ability to experiment with the logic with the goal of gaining useful information about the circuit and analyzing circuit behavior early in the design cycle.

2.1.2 Formal Verification

Formal verification is a proof for determining whether two circuit descriptions at different abstraction levels are identical, as well as a technique for deciding whether a specific behavior is implemented by a given circuit. For instance, when a designer specifies a circuit by a high-level description formal verification techniques can examine both the specification and implementation to determine whether their behavior is exactly the same [Evek91]. The majority of formal verification methods, however, only verifies the logical function and not the timing of the circuit. Circuits with up to ten thousand gates can be evaluated effectively with formal verification techniques [Evek91], Currently existing formal verification algorithms cannot handle a complex circuit with millions of gates. However, formal verification techniques can be utilized when a complex circuit is divided into smaller subcircuits, which after the verification of the individual subcircuits, form a complete circuit. In this form, the possibility of failures still exists due to the interactions of subcircuits.

2.1.3 Logic Emulation

A logic emulator is actually a reprogrammable compute engine that can be configured to implement the function of a circuit [BrFr92a]. As with a prototype, this hardware implementation is created in order to attain accurate evaluation results. In addition, design errors can be located and isolated by observing and altering an emulation, similar to software simulation. In many cases, however, a speed 10^3 to 10^6 faster than software simulation [KhHu93] can be attained with logic emulators. A complex circuit can not be mapped into a single programmable chip, thus an emulator consists of a multitude of programmable components. The programmable hardware is generally constructed hierarchically. Several of the programmable components and the corresponding routing resources are present on a board typically having an emulation capacity of 250K gates. Generally, an emulator contains several such boards interconnected through a programmable backplane in order to facilitate communication between components on different boards. Capacities of up to one or two million gates on one board can be reached. For the emulation of a complete circuit, several emulators can be cascaded in order to attain emulation systems with a capacity of several million gates. A logic emulator can contain either standard-FPGAs [Quick96a] or specially developed Full Custom Chips [BuRe96]. In general, logic emulation can be divided into two approaches involving Field Programmable Gate Array (FPGA)-based and multiprocessor-based logic emulators.

The basis of the FPGAs, such as the XILINX FPGAs, are the configurable logic blocks (CLBs). A CLB contains logic blocks for the representation of logical functions as well as flip-flops for the realization of storage elements. Logical functions are organized as look-up tables, consisting of SRAM memory. The connection between the look-up tables and storage elements of a CLB is established using programmable multiplexers. An FPGA consists of a regular array of programmable logic blocks (CLBs) as well as horizontal and vertical routing channels between the CLBs. Figure 2.1 displays a typical FPGA-based logic emulator [BuBa90] composed of emulation boards. A single emulation board is composed of multiple FPGAs (Fig.2.1a), each of which contains a multitude of CLBs (Fig.2.1b) and each CLB uses LUTs [BrFr92b] and flip-flops (Fig.2.1c).

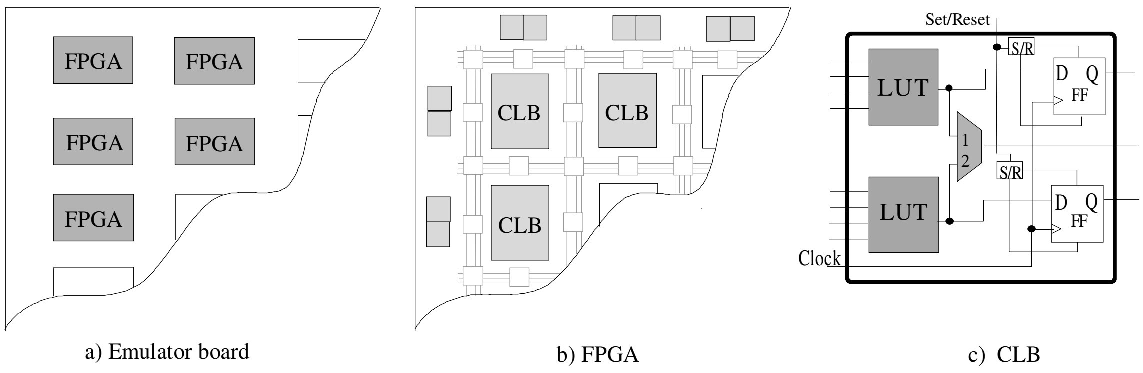


Fig. 2.1: A typical logic emulator

In an FPGA-based logic emulator the conversion of the circuit description into a form that is suitable for mapping to the logic emulator may take several or more hours. The emulator software completely automates the mapping process. After the circuit is mapped to the logic emulator, circuit functionality is completely implemented by the emulator.

A multiprocessor-based logic emulator operates with a dedicated architecture based on parallel processing and is basically a parallel logic computer. The basic components of the processors are also programmable logic blocks, which however are constructed considerably simpler than CLBs of the FPGAs. The multiprocessor-based logic emulator is composed of an array of custom multiprocessing ICs that operate collectively to emulate complex logic circuits. Each processor emulates a small portion of the circuit by performing a sequence of operations during every target system clock cycle.

During parallel processing the logical function (gates) are no longer mapped one-to-one into the physical gates of the emulator. Rather, the design is divided into several time frames and a logic block of the emulator emulates a different logical function in each time frame. Before the design is mapped into the emulator each logical function is assigned to a specific time frame. The value of the assigned logical function is calculated in each time frame. Several time frames are grouped together. These groups are processed sequentially during emulation.

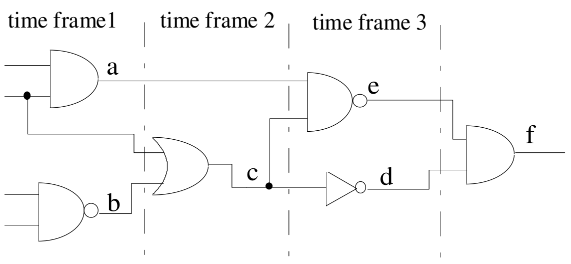


Fig. 2.2: Time frames

Each logic block and its interconnection to other logic blocks must be reconfigured for each time frame. Therefore, all program data of processor systems are loaded into the internal memory before the emulation starts. At the beginning of each time frame the logic blocks' look-up tables are configured with data of the internal memory. During a time frame, a logic block requires not only the output values of the other blocks in the same time frame, but usually also the results from the preceding time frames. The preliminary results of a time frame must be stored into memory and made available to the appropriate logic blocks' look-up tables when necessary. Thus, the logic blocks and their interconnections are processed by dynamic programming, which is similar to the processing of a program in a microprocessor.

The maximum number of time frames that can be processed in parallel is determined by the longest combinational path within the design. A large number of time frames leads to a low emulation frequency. Due to parallel processing, the emulation frequency is generally a factor of two to three times lower than with FPGA-based emulators. Today, a multiprocessor-based logic emulation system [Quick98] is capable of emulating circuits with up to 20 million gates. Because the logic emulator can implement the functionality of the complete circuit in parallel, the evaluation of millions of circuit cycles per second is possible.

Logic emulation of a circuit can be executed in two modes: logic simulation acceleration and in-circuit emulation. Applying the first mode to the mapped design, a vector set can be evaluated at an emulation speed of several MHz and is up to 10^6 times faster than software simulation. The second mode involves plugging the logic emulator into the target system. The emulated chip operates as a prototype in the target system. The logic emulator is inserted into the target environment to provide a more realistic evaluation of the system. The debugging process now includes evaluating the circuit in its real environment. Such is the case with the concurrent development of a custom ASIC and the circuit board, where the ASIC will be later inserted. By connecting the emulated ASIC to the circuit board with appropriate interfaces, the ASIC functionality can be evaluated in the board.

One restriction of the logic emulation procedure is that only the functional behavior of the circuit can be emulated. The validation of circuit timing characteristics, which is an important aspect of logic validation, is not possible. However, when combined with both software simulation and prototyping, logic emulation plays an important role in the logic validation process.

2.1.4 Design Prototyping

The process of developing a hardware implementation as a prototype of the circuit under validation is referred to as prototyping. A prototype can be completed, for example, with breadboarding and wire-wrap techniques (methods for wiring together standard components to implement the circuit) or as a first silicon, i.e. the first series of chip production. The prototype, evaluated under normal operating conditions, gives results that are most accurate without regard to modeling, abstraction or any other factors involved with software simulation [Micz87]. When the evaluation phase is completed, the prototype can be sent to users to determine whether the circuit is appropriate for their needs in a real target system. Another feature of prototypes is that the complete evaluation process runs much faster than with software simulation, due to operation at or near target system speeds.

While a high level of accuracy and high-speed evaluation are certainly advantages of prototyping, disadvantages must also be considered. The complete circuit must exist not only as a concept or specification but also as a finished design before the prototype can be constructed. For this reason, the implementation of a prototype for logic validation is only meaningful rather late in the design process. Prototype manufacture can be a costly process considering ASIC fabrication [Benn82] [NaBi88], as well as construction time involved. Since it is difficult to make alterations to a prototype as many errors as possible should be detected early in the design process in order to avoid incurring new costs for the manufacture of multiple prototypes.

2.1.5 Comparison

Software simulation as described in Section 2.1.1 allows for levels of circuit evaluation that are difficult to achieve with a prototype, where access to the internal states of the circuit is almost impossible and their values can not be easily altered. With software simulation, the values of the internal states are able to be displayed as well as altered, resulting in easily observable circuit behavior. Because this is not usually possible with prototypes it becomes relatively difficult to locate and isolate circuit errors. Although the production of prototypes is relatively expensive and their use difficult, prototyping is the more accurate method of logic validation. Today, evaluating a prototype as a first silicon is a necessary part of the logic validation process, although it occurs relatively late in the design process. Locating and correcting most circuit errors is handled by other validation techniques.

Logic emulation combines the flexibility of software simulation with the speed of design prototyping. Measured against software simulation, higher execution speeds can be achieved with emulation. However, software simulation is the better method for evaluating circuit abstractions or a small amount of test vectors. The operational simplicity as well as flexibility of the simulation overrides its negative performance results, e.g. long runtimes. Simulation is usually not an issue in the area of software development for the target system because simulation of a circuit can not be executed within an acceptable time period. When compared to a prototype, an emulation is easier and faster to create; circuit behavior can also be observed, controlled and modified better than with a prototype. Moreover, the evaluation of the circuit is possible while running with hardware and software target systems. Emulation is the more effective tool for the location and eventual detection of system errors because it can be implemented earlier in the design process using a high-level circuit specification, whereas a prototype can only be activated at the end of the design phase. As with prototypes, it is possible to give an emulation to an end-user for evaluation purposes before completion of the design process. Finally, emulation saves much of the time and material that would otherwise be devoted to the manufacture of multiple prototypes.

Logic emulation as well as software simulation are not to be disregarded once a prototype has been developed. For instance, when an error is detected in the prototype it is often difficult this late in the design process to isolate the error in the circuit. An emulator can be used to reproduce the error since it can execute nearly as many cycles as the prototype. The emulator's observability can also be utilized to isolate the error.

In an ideal validation environment, the validation methodology would utilize the strengths of each approach while combining multiple approaches to overcome individual weaknesses. The design phase begins with the specification of the circuit to be designed. Next, software simulation attempts a quick evaluation of the circuit and formal verification techniques are added for the detection of errors in the specification. During this process, software simulation detects the simple errors, further errors are detected by using a large quantity of test vectors. At this point, emulation, rather than software simulation, becomes the better validation method. Early in the design phase logic emulation provides a platform for parallel software development for the target hardware. When designers determine that the design is relatively fault-free, the time-consuming and costly manufacture of a prototype can commence. The prototype is actually a fabrication of the circuit that can be analyzed in a real operating environment. When failures have been detected by the prototype, emulation as well as software simulation can be applied to isolate and remove the errors.

It can be concluded then, that when the benefits of software simulation, emulation, formal verification, and prototyping are combined the result is an ideal validation methodology. Software simulation is for designers particularly useful as a tool for the detection of errors in small circuits. Emulation takes this one step further by enabling designers and end-users to observe the entire system in operation. Formal verification techniques prove whether two circuit descriptions are identical at different abstraction levels. The final check is provided with prototyping, where real system behavior is not affected by errors caused by incorrect modeling or abstractions.

2.2 FPGA-based Logic Emulation Design Flow

After the previous discussion of the advantages and disadvantages of the various approaches to logic validation, this section focuses on mapping a circuit into an FPGA-based logic emulator. Figure 2.3 illustrates the typical design flow of an FPGA system including the three steps generally required for the preparation of a circuit for emulation.

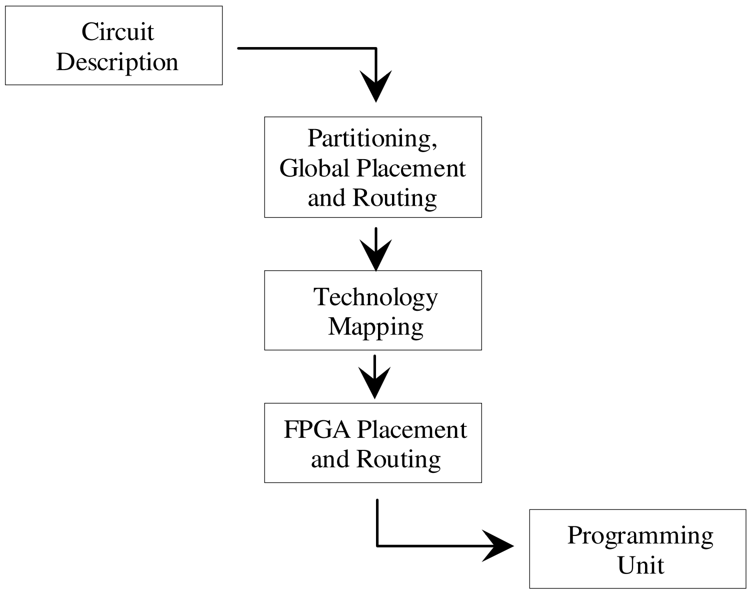

Circuit Description
Partitioning, Global Placement and Routing
Technology Mapping
FPGA Placement and Routing
Programming Unit

Fig. 2.3: A typical design flow of FPGA-based emulation system

Circuits are usually described in a hardware description language like Verilog or VHDL. For instance, at gate level the logic is represented by primitives such as ANDs, ORs, flip-flops etc. or at register transfer level (RTL) by adders, substractors, multipliers, counters etc. The compilation process of converting a structural circuit description to FPGAs includes partitioning, technology mapping, FPGA-system placement and routing, which involves the placement and routing of single FPGAs in the logic emulator and FPGA placement and routing. FPGA placement and routing are implemented in the same manner with single FPGAs. The compilation process is explained in more detail in Chapter 5.

Partitioning is generally the first step in the mapping process and involves dividing the circuit description into sections which then fit into the individual FPGAs of the logic emulator. The tool that divides the logic into partitions is called a partitioner. Routing between FPGAs must be accommodated within the board's routing topology. FPGA-system placement, combined with the partitioning process, is a procedure which allocates partitions to individual FPGAs in the logic emulator. The next step is FPGA-system routing, a method for routing signals between partitions, i.e. FPGAs within the emulator.

After completion of the partitioning procedure technology mapping reorganizes the logic for an optimal fit in the CLBs [HeRo94]. For the most part, smaller gates are brought together to form larger functions for the best possible utilization of individual configurable unit resources. In order to attain optimum results fanout gates may need to be split, logic resynthesized, and functions duplicated.

The final steps in the mapping procedure include the placement and routing of each FPGA in the system. Configuration files are then created and subsequently downloaded to the FPGA system, providing a thorough realization of the circuit's desired functionality.

3. Fault Simulation

During the manufacturing process manufacturing defects are introduced since no manufacturing process can guarantee 100% yield. The actual type of defect is technology dependent. Types of defects common to various technologies are open interconnections, bulk shorts, and missing transistors [TiBu83]. The larger the circuit in terms of area, the greater the probability of defects.

Fault detection and fault diagnosis are two important aspects of testing. Fault detection detects the presence of a fault, whereas the exact location of a fault is identified through fault diagnosis. By applying test vectors to a circuit during the testing process the response of the circuit can be compared to an expected response computed using logic simulation tools. Differing results indicate an fault, the cause of which is referred to as a physical fault [TiBu83]. When dealing with digital circuits, physical faults can be categorized as logic or parametric [TiBu83]. A logic fault can affect a change in the logic function of the circuit. By altering the magnitude of a circuit parameter, parametric faults cause changes in the circuit, such as circuit speed, current, or voltage levels. In this work, only logic fault detection is considered and will be described in greater detail in the following sections.

3.1 Fault Models

Faults in digital circuits can generally be divided into two groups [Muth75]: design errors and physical faults. It will be assumed here that design errors are no longer present in the circuit at the end of the design process. Therefore, only physical faults will be considered in this section. A physical fault can occur, for example, as a result of dust, contamination, or mask faults during the production process. Effects of physical faults can take the form of static, dynamic, or intermittent faults. Examples of static faults are defective connections between transistors or defective transistors. A dynamic fault on the other hand is, for example, the dynamic coupling between wires in the IC. Intermittent faults are those which do not occur permanently, such as loose connections, voltage breakdowns, or temporary internal warming up of a specific area of the IC.

Fault Modeling at Various Abstraction Levels

While the design of a circuit usually begins at the system level and ends at the mask level, the procedure involved for fault modeling begins at the mask level and ends at the system level. Of interest in the fault modeling process are the physical faults that occur in manufacturing and lead to faulty electrical behavior. Because simulation is too complex and time consuming at the lower levels, the faults are modeled at a higher level. When using a higher abstraction level of the fault model some faults of the lower levels are combined, as illustrated by the example in Fig. 3.1. Here an INVERTER is represented at the logic and electrical level. While the INVERTER can be modeled at the electrical level with 16 stuck-at faults (described in next section), at the logic level these faults can be reduced, i.e. combined to 4 stuck-at faults.

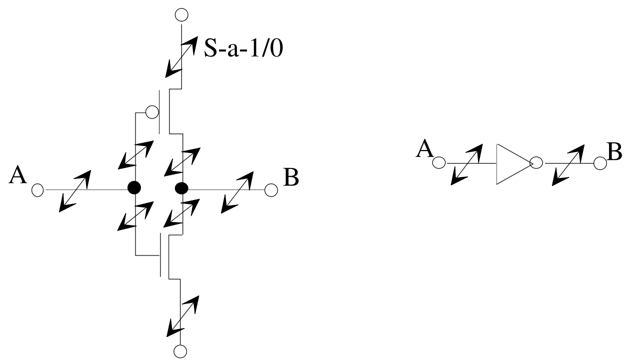


Fig. 3.1: CMOS INVERTER at electrical and logic level

With the goal of reducing the quantity of faults, many faults at the lower level are mapped to a fault at a higher level. Limited accuracy results during the transition from a fault model at a lower level to a model at a higher level (Fig. 3.1). A differentiation between the various faults is not possible at the higher levels, hence the possibilities for exact fault location are restricted. The more accurate the abstraction levels, the higher the magnitude of the circuit description. Thus, the accuracy of a fault model is dependent on the abstraction level where the model is defined. Due to this dependency, the complexity for fault simulation and test vector calculation, which will be described in Sections 3.4 and 3.5, increases with the accuracy of the fault model. A suitable compromise between accuracy and complexity is the stuck-at fault model, defined at logic level, which was introduced in [Eldr59].

The effects of physical faults on the behavior of the modeled circuit can be represented as logical faults [TiBu83]. By modeling physical faults as logical faults the fault analysis problem becomes a logical rather than a physical problem. The complexity involved in the analysis of the logical faults is reduced greatly since a logical fault can model many different physical faults. Additionally, due to the technology independence of some logical fault models, it is possible to apply the same fault model to various technologies [TiBu83]. The stuck-at fault model models various types of physical faults at logic level. It is the first and most widely used model and is also referred to as the standard or classical fault model. In a faulty circuit, it is assumed that an input or output of a logical gate is always set to logical "1" for stuck-at-1 or logical "0" for stuck-at-0. As illustrated in Fig. 3.2, for example, a stuck-at-1 fault is modeled at input I_1 of the NAND gate. Therefore, input I_1 has the value "1" and causes at output O_1 the faulty value "0" with the input vector $(I_1, I_2)=(0, 1)$. The input vector (0, 1) is a test vector for the stuck-at fault at $I_1(s\text{-}a\text{-}1)$.

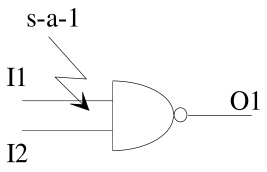


Fig. 3.2: Stuck-at fault model

In addition to the classical stuck-at fault model, bridging faults [Mei74] as well as various delay fault models are defined at logic level. Bridging faults result from shorts between two or more signals in the circuit. The nodes involved in a short become equipotential, i.e. they all have the same logic value. The characteristics of shorts between nodes at logic level for bipolar technologies have been examined in [Mei74]. For example, shorts are simulated either as "wired-AND" or "wired-OR". A bridging fault, then, can be modeled as a logical AND or OR connection of the shorted wire, as shown in Fig. 3.3. These models are not valid however, for CMOS technologies, where electrical resistance conditions resulting from the short-circuit must be considered in order to determine the logical behavior in the presence of a bridging fault [Wads71].

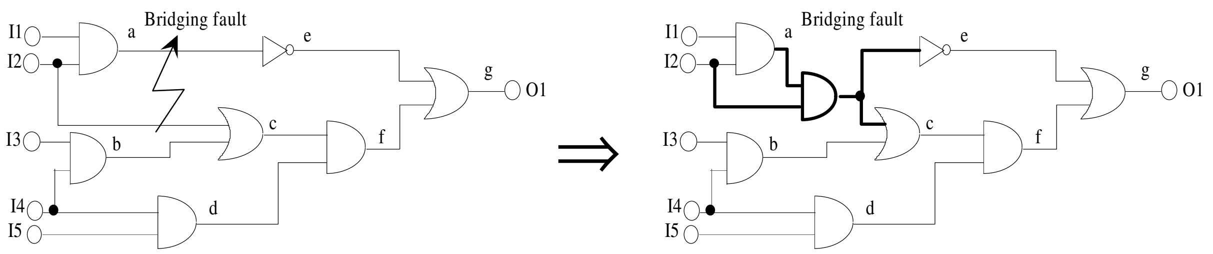


Fig. 3.3: Modeling of bridging fault as wired-AND

The stuck-at fault model can also be used to model other types of faults, such as delay faults f_d [AbBF90a]. A model of a delay fault can be applied to a single gate [Wund91a] or to paths from the inputs to the outputs of the circuit [Wund91b]. It should be noted that these gates and paths within the circuit must be described logically with timing characteristics. A gate- or path-delay-fault can be an arbitrary deviation from the predefined timing for the gates and paths, which is dependent on the applied technology, such as CMOS, TTL etc. A delay fault is specified by a pair of input vectors (initialization and test vector) and a delay δ, and indicates when an additional delay δ exceeds the slack ΔDel of the sensitized path as shown in Fig. 3.4.

Slack ΔDel is defined as the difference between the nominal time T_N, which is the signal value steady-state, and the propagation delay *Del* of a signal along a path [HiSC82][Mahl95]. A specific type of delay fault is the transition fault f_t. A fault of this type can also be interpreted as a delay fault f_d of the magnitude $\delta = \infty$. For this reason, the transition fault model can be simulated with the same algorithms that are applied to delay faults.

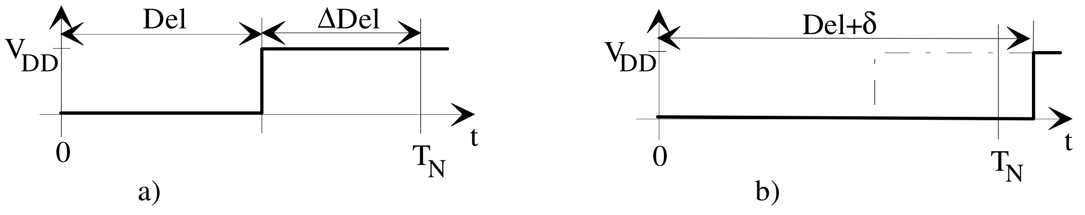


Fig. 3.4: Fault free (a) and faulty (b) signal delay

In addition to stuck-at faults, bridging faults, and delay faults, a variety of other fault models developed for various abstraction levels also exist, such as functional fault models [Haye72] at the algorithmic or system level, stuck-open and stuck-on fault models [Wade71] [Haye72], and hard and soft fault models [DuRa79] [Plic79] at the electrical level.

3.2 Fault Redundancy, Equivalence, and Dominance

An evaluation of the test vector set V containing the test vectors $v_1, v_2, ..., v_n$ is followed by a comparison of the actual output response of the faulty circuit $Bf(v)$ to the precomputed output response of the fault-free circuit $B(v)$. A fault is defined as being detectable if a test vector or a test vector sequence exists for detecting the fault. In other words, a test vector v detects a fault f if the function of the fault-free circuit $B(v)$ differs from the function of the faulty circuit Bf(v) with fault f. Assuming that a circuit has a single output, a test vector that detects a fault f causes $B(v)=0$ and Bf(v)=1 or vice versa. Thus, all test vectors that detect f are represented by the equation $B(v) \oplus Bf(v)=1$.

When the behavior of the fault-free circuit $B(v)$ and the faulty circuit $Bf(v)$ is identical for all possible test vectors, the injected fault is undetectable. In this case, there is no test vector for the creation of a sensitized path, i.e. for the propagation of this fault to the primary output of the circuit. The goal of test vector calculation (test generation) for a circuit is to create a complete detection test vector set capable of detecting all detectable faults.

As illustrated in the following example, when an undetectable fault is present in the circuit, a complete test vector set may be insufficient for detecting all detectable faults [Frie67]. Fig. 3.5

shows how the fault *s-a*-0 at net *a* is detected by (I_1,I_2,I_3)=(1,1,0). This fault is no longer detected by the test vector (1,1,0) if the undetected fault *s-a*-1 at the net *b* is also present.

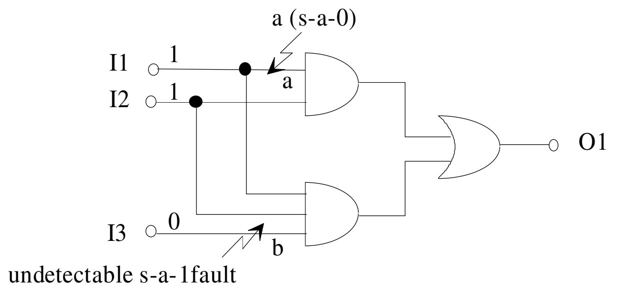
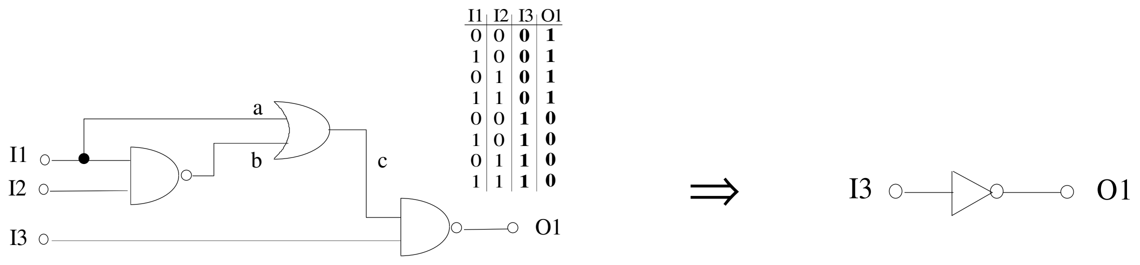


Fig. 3.5: Undetectable fault

Fault Redundancy

A combinational circuit containing an undetectable fault is referred to as redundant because the circuit can always be simplified by removing a subcircuit. For example, in Fig. 3.6 an *s-a*-1 fault is modeled in net *c*. In order to detect *s-a*-1 in net *c*, the inputs *a* and *d* of the OR gate must be set to 0. No test vector exists, however, that would make this possible. In this example, the net *c* is permanently set to 1. Thus, the behavior of the circuit is identical to that of an INVERTER, i.e. the circuit can be simplified or reduced to an INVERTER with input I_3, which results in net *c* being redundant.



I1	I2	I3	O1
0	0	**0**	**1**
1	0	**0**	**1**
0	1	**0**	**1**
1	1	**0**	**1**
0	0	**1**	**0**
1	0	**1**	**0**
0	1	**1**	**0**
1	1	**1**	**0**

Fig. 3.6 Redundancy

In practice, a complete test vector set cannot be generated for the detection of all faults in a large combinational circuit even if no redundant faults are present in the circuit. This is due to the fact that test generation for some faults may be too time-consuming and all existing test generation tools are designed to interrupt the test generation process for a fault when it becomes too time consuming, e.g. when fault detection becomes too costly. Therefore, an undetectable fault can not be differentiated from a detectable fault that has not been detected by an applied test vector set.

Fault Equivalence

A classical method for decreasing the quantity of modeled faults is the use of dominant [McCl71] and equivalent faults [ScMe72]. Two faults f_1 and f_2 are functionally equivalent when the functional behavior of the faulty circuit $Bf_1(v)$ with f_1 is equal to the functional behavior of the faulty circuit $Bf_2(v)$ with f_2, i.e. $Bf_1(v)=Bf_2(v)$. The two faults f_1 and f_2 are distinct when a test vector v is able to differentiate between them, i.e. $Bf_1(v)\neq\ Bf_2(v)$. No test vector however, can distinguish between two functionally equivalent faults. Faults which are functionally equivalent can be separated from the set of all possible faults and grouped into functional equivalence classes [McCl71][ScMe72]. Observing a single fault from each equivalence class is sufficient for fault analysis. Equivalence fault collapsing refers to the reduction of the set of faults to be analyzed based on their equivalence relations and is illustrated in Fig. 3.7 for a NAND gate.

The NAND gate with the inputs I_1 and I_2 and the output O_1 has four fault equivalence classes $\{I_1(s\text{-}a\text{-}0),\ I_2(s\text{-}a\text{-}0),\ O_1(s\text{-}a\text{-}1)\}$, $\{I_1(s\text{-}a\text{-}1)\}$, $\{I_2(s\text{-}a\text{-}1)\}$, $\{O_1(s\text{-}a\text{-}0)\}$ since each test vector that is able to detect $I_1(s\text{-}a\text{-}0)$ can also detect $I_2(s\text{-}a\text{-}0)$ and $O_1(s\text{-}a\text{-}1)$ and vice versa.

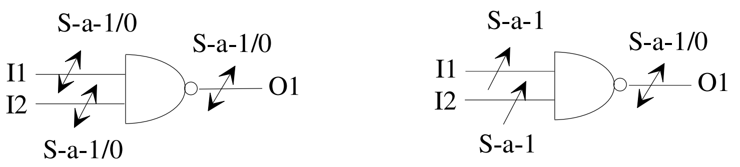


Fig. 3.7: Equivalence fault collapsing

Therefore, it is always sufficient to observe a single fault from each equivalence class. The n input gates of NOT, NAND, NOR, AND, OR is $n>1$ and have $2(n+1)$ single stuck-at faults, with $s\text{-}a\text{-}1$ and $s\text{-}a\text{-}0$ faults at the output and at all inputs. Using equivalence fault collapsing the set of faults is reduced to only $(n+2)$ faults for any n input gate.

Fault Dominance

Given that V_{f1} is the set of test vectors for detecting the fault f_1, fault f_2 dominates fault f_1 if f_2 and f_1 are functionally equivalent under V_{f2}. In other words, if f_2 dominates f_1, a test vector v that detects f_1, on the primary outputs can also detect f_2 on the same outputs since the functional behavior of the faulty circuit with f_1 is equal to the functional behavior of the faulty circuit with f_2. For purposes of fault detection, then, it is not necessary to consider the dominating faults. Dominance fault collapsing can be defined as a reduction of the set of faults to be analyzed based on dominance relations $V_{f1}\subseteq V_{f2}$. For example in the NAND gate in Fig. 3.7, the stuck-at faults $I_2(s\text{-}a\text{-}1)$ and $O_1(s\text{-}a\text{-}0)$ can be detected with the test vector v (I_1=1 and

I_2=0). In this case, O_1(*s-a*-0) dominates I_2(*s-a*-1). For other primitives such as an AND gate, the output *s-a*-1 dominates any input to the gate *s-a*-1 just as the output *s-a*-0 for an OR gate dominates any input *s-a*-0. Likewise, for NOR (NAND) gates the output *s-a*-0 (*s-a*-1) fault dominates any input *s-a*-1 (*s-a*-0).

For complete fault collapsing using both dominance and equivalent fault collapsing it is possible to considerably reduce the number of faults for an n input gate from $2(n+1)$ to $(n+1)$ faults.

3.3 General Approach to Fault Simulation

Fault simulation is the process of simulating a circuit in the presence of faults. Using a test vector set V, faults are detected by comparing the fault simulation results to the results from a fault-free simulation of the same circuit, i.e. good simulation. Fault simulation can be implemented to evaluate test vector set V. Generally, the grade or quality of V is determined by its fault coverage γ, defined by the ratio of the number of faults detected by V to the number of modeled faults.

$$\gamma = \frac{number\ of\ detected\ faults}{number\ of\ modeled\ faults} \tag{3.1}$$

The typical fault coverage curve is depicted in Fig. 3.8 by the evaluation of a test vector set. A linear slope is shown at the beginning of the fault simulation, which then ends in the area between 70-80% of the maximum fault coverage (100%). This linear area of the curve usually ends after just a small quantity of test vectors, with which the easily detectable faults are detected. Faults which are difficult to detect are processed in the saturation area, where only a few test vectors exist for the detection of these faults.

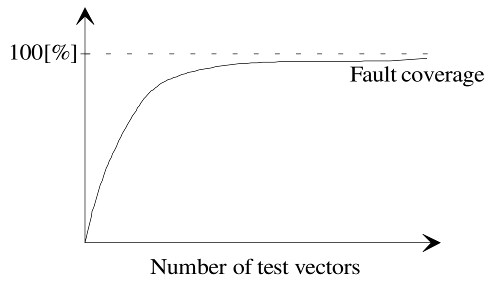


Fig. 3.8: Fault coverage curve

Fault simulators are also used to increase the efficiency of programs used for test vector calculation [ScTr87]. For each modeled fault a program for test vector calculation determines a test vector or, for sequential circuits, a sequence of test vectors. Due to the complexity of test generation, a single test vector or test vector sequence is calculated for the detection of a fault and can be used with a fault simulator to detect further faults. The total runtime for test generation is thereby greatly reduced. By evaluating randomly generated test vectors using a fault simulator, easily detectable faults can be detected, which leads to a reduction in the fault list before use of a test generator. Additionally, by combining the fault simulator with the test generator the quantity of test vectors can be kept to a minimum.

Another classical use of fault simulators is fault diagnosis [AbBF90b], which determines which faults are present by creating a fault dictionary with the help of a fault simulator. The fault dictionary contains information about the faults detected by a test vector. In order to attain the best possible results from the fault dictionary, each test vector must be evaluated for all faults. Therefore, fault dropping, i.e. the removal of faults from the fault list as soon as they are detected, cannot be applied to fault diagnosis.

3.4 Fault Simulation Techniques

For the fault simulation process both the compiler-driven and the table-driven approaches to logic simulation can be applied. A simulator carrying out a compiled-code model is referred to as a compiler-driven simulator or a compiled simulator. The compiled code can be produced, for example, from a structural model, or from a functional model written in a conventional programming language. A compiler-driven simulation has several advantages, such as the simplicity of the applied simulation algorithm and low memory usage. The disadvantages include the time-consuming compilation of the circuit into a processable code, which must occur before the simulation. Compiler-driven simulation deals mainly with functional verification rather than with the timing of the circuit.

A table-driven simulator interprets a model based on data structures that are generated, for example, from a structural model. Table-driven simulation, as opposed to compiler-driven simulation, allows for modeling the timing of the circuit as well as for a higher degree of flexibility. Complicated data structures and larger memory usage are, however, both consequences of this method of simulation.

The table-driven approach can usually be applied to event-driven simulation. Here, an alteration in the value of a signal is referred to as an event. The input of a logic element is activated by the presence of an event. The generation of new events resulting from changes in the output values by the activated logic elements is referred to as event-driven simulation. In order for events to be propagated along the interconnections among the logic elements of a circuit, a structural model of the circuit is required by the event-driven simulator. Because of this, event-driven simulation is usually table-driven.

Serial Fault Simulation

The simplest method of fault simulation is known as serial fault simulation. Using this method, the simulation process is repeated for each fault. Serial fault simulation is based upon a comparison between the simulation results attained from faulty circuits $Bf(v)$ and fault-free circuits $B(v)$. A disadvantage of serial fault simulation is that for a circuit with a set of faults F, $|F|+1$ simulation runs must be executed, one fault-free run plus one for each fault. Fault collapsing is a technique which can be applied to reduce the number of faults to be simulated and is described in more detail in the previous section 3.2. In addition, fault dropping can be used to remove faults from the fault list. For large circuits, however, serial fault simulation is impractical due to the amount of computation required for the simulation runs and for the comparisons of faulty and fault-free signal values.

Parallel Fault Simulation

Using the parallel fault simulation method [Sesh65], a fault-free circuit and a predetermined quantity of faulty circuits are simulated simultaneously. The signal values of the fault-free circuit and those of the corresponding faulty circuits are simulated in one or more words *W*.

As illustrated in Fig. 3.9, if a 2-valued logic and an 8-bit word is used each of the bits contained in the word is associated with a signal value in different circuits. Bit *0* usually symbolizes a signal value from the fault-free circuit. Given an AND gate with the inputs I_1 and I_2, a logical AND instruction is used between the words associated with I_1 and I_2. The AND gate can then be evaluated in parallel for the fault-free as well as for each of the 7 faulty circuits.

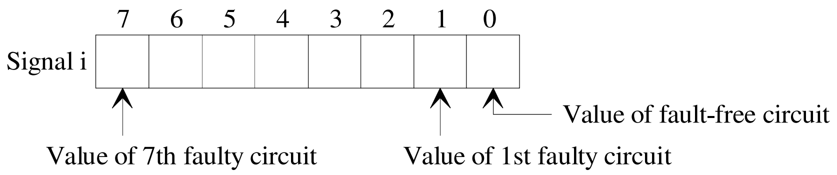

7 6 5 4 3 2 1 0
Signal i
Value of fault-free circuit
Value of 7th faulty circuit
Value of 1st faulty circuit

Fig. 3.9: Parallel fault simulation

The equation

$$val = val \cdot \overline{mask1_{[i]}} + mask1_{[i]} \cdot mask2_{[i]} \qquad mask1_{[i]} = \begin{cases} 1 & \text{active fault} \\ 0 & \text{inactive fault} \end{cases} \tag{3.2}$$

represents the process of fault injection, where *val* is the value of an arbitrary signal. The simulation process is carried out in parallel using two mask words that store the values *mask*1 and *mask*2 in the bit position associated with fault *f*. *Mask*1 corresponds to a signal and specifies if, and at which bit positions, faults are to be injected. The stuck-at fault values of these faults are defined by *mask*2. A fault *f* on bit position *i* of *mask*2 can only then be detected at a word of the signal *val* when the value on bit position *i* of $val_{[i]}$ differs from the value on the first bit position in the same word ($val_{[1]}$), i.e. $val_{[1]} \oplus val_{[i]}=1$ and if the fault *f* is active on bit position *i* of *mask*1, i.e. $mask1_{[i]}=1$.

Fig. 3.10 shows an AND gate which is part of the circuit and the masks used for fault injection on output *h*. Before evaluating *h*, fault injection is executed for inputs *c* and *d* of the AND gate. When the evaluation of signal *h* by $h=c\cdot d$ is completed, the effect of fault insertion on *h* can be calculated by

$$h = h \cdot \overline{mask1_{[h]}} + mask1_{[h]} \cdot mask2_{[h]} \tag{3.3}$$

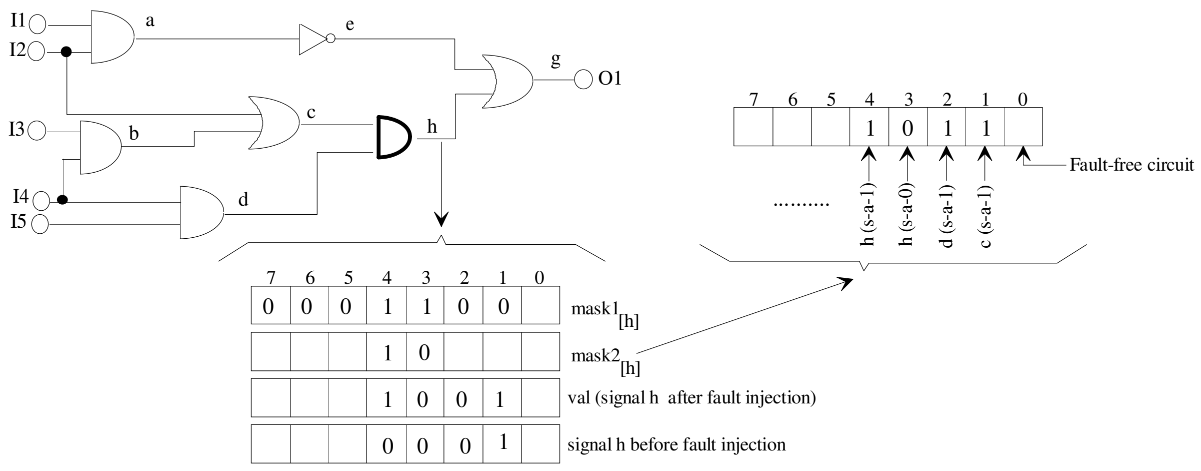


Fig. 3.10: Fault injection on signal *h*

The signal for a 3-valued logic 1, 0, u (unknown logic value) is represented by two words w_1 and w_2, which are encoded. The values 1,0, and u are coded respectively as "1:=11" (w_1=1, w_2=1), "0:=00" (w_1=0, w_2=0), and "*u*:=01" (w_1=0, w_2=1). Therefore, the equations $y_1=a_1\cdot b_1$ and $y_2=a_2\cdot b_2$ are used rather than $y= a\cdot b$ when evaluating an AND gate with inputs *a* and *b* and output *y*. Evaluation methods become more complex with an increase in the quantity of logic values. Therefore, for a more than 3-valued logic parallel fault simulation becomes impractical for large circuits.

Deductive Fault Simulation

Deductive fault simulation [GoVo71] is based on the algorithm that in order to observe faults and their effect on a circuit, the simulation process is not repeated for a circuit which has already been simulated. Instead of the calculation of many faulty circuits in parallel, all faults are calculated in one simulation run. Using the deductive method of fault simulation the fault-free circuit is simulated and the behavior of all faulty circuits is deduced. In practice, the deduction of all faulty circuits depends on the amount of memory available for this purpose. A data structure referred to as a fault list, F_i, represents the fault effects and corresponds to each signal *i*. During the simulation, F_i is the set of all faults, which are responsible for changes in the values of signal *i* in the fault-free circuit and the faulty circuit at the prevailing moment in the simulation.

The computation of fault lists is the basic task in deductive simulation and involves the calculation of the fault-free output value from the given fault-free input values, as well as the calculation of the output fault list from the given fault lists of the inputs of the logic elements. This procedure is referred to as fault-list propagation. In addition to the propagation of logic events, representing alterations in signal values, list events resulting from additions or deletions of faults from a fault list are also propagated by a deductive fault simulator.

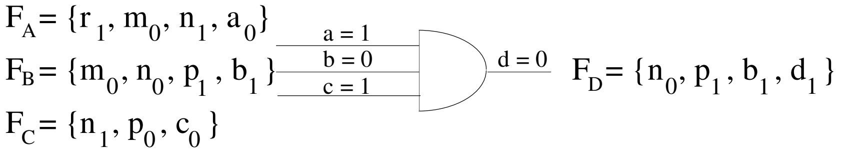

$F_A = \{r_1, m_0, n_1, a_0\}$
$F_B = \{m_0, n_0, p_1, b_1\}$
$F_C = \{n_1, p_0, c_0\}$
a = 1
b = 0
c = 1
d = 0
$F_D = \{n_0, p_1, b_1, d_1\}$

Fig. 3.11: Fault list propagation

Illustrated above is an AND gate *d* with inputs *a*, *b*, and *c* as well as the corresponding fault lists (F_A , F_B , F_C). The input values are *a*=1, *b*=0, and *c*=1, therefore *d*=0. The value of *d* changes for each fault which causes a change in the value of *b*, but which does not affect the values of *a* and *c*. The faults in F_B that are not in F_A and F_C are

$$F_D = F_B \cap (\overline{F_A} \cup \overline{F_C}) \cup \{d_1\} = F_B - (F_A \cup F_C) \cup \{d_1\}$$

where $\overline{F_A}$ and $\overline{F_C}$ are the set of all faults not in F_A and F_C. The fault lists from Fig. 3.11 contain the following faults, with letters referring to nodes and an index representing the stuck-at value.

$$F_A = \{r_1, m_0, n_1\} \quad ; \quad F_B = \{m_0, n_0, p_1\} \quad ; \quad F_C = \{n_1, p_0\}$$

The faults, added to the nodes *a*, *b*, and *c*, affect a change at the nodes from which the following lists are generated:

$$F_A = \{r_1, m_0, n_1, a_0\} \quad ; \quad F_B = \{m_0, n_0, p_1, b_1\} \quad ; \quad F_C = \{n_1, p_0, c_0\}$$

From these values it can be concluded that

$$F_D = \{n_0, p_1, b_1\}$$

The faults n_0 , p_1, and b_1 are propagated to the output of the logic element. Now d_1 must be added to the fault list since these faults influence the value of *d*. It follows then for *d* that

$$F_D = \{n_0, p_1, b_1, d_1\}$$

The deductive fault simulation algorithms require a huge memory capacity for simulation. A further disadvantage is the propagation of list events, which occur even when additional input to the circuit does not change the logical values at the logic element's inputs, i.e. when a fault is added or removed from the fault list. Therefore, the propagation of list events results in many long fault list computations.

Concurrent Fault Simulation

The most common fault simulator is the concurrent fault simulator [UlBa74]. Simulated are only the specific logic elements in the faulty circuit which differ from the corresponding ones in the fault-free circuit. These differences for every logic element in the fault-free circuit are maintained in a concurrent fault list.

With concurrent fault simulation, a replication is produced for each fault in the circuit which causes a faulty signal or its propagation through a logic element (gate). Here, the actual logic elements as well as the replications are simulated. The basic idea of concurrent fault simulation is that a simulation should only be executed at the gates or replications where the events occur. The gate replications are represented in the concurrent fault list as follows: *GT* is a gate with *n* inputs $a_1^f, \ldots, a_n^f$ and the output *c*. F_{GT} represents the set of faults which affect *GT*. During the simulation of each fault f with a test vector, the gate *GT* has the values $a_1^f, \ldots, a_n^f$ on its inputs and c^f on its output. Additionally, f=0 refers to the fault-free gate. The concurrent fault list for *c* consists of a list of entries in the form of

$$f; a_1^f, \ldots, a_n^f; c^f. \tag{3.4}$$

In Fig. 3.12 output *c* is contained in the concurrent fault list. The first replicated gate in the fault list is f_1; 1, 1;0 (In other words, fault f_1 has the input values 1, 1, and the output value 0). All gates in the fault list are arranged in a fault index. When the output values for a fault-free and a faulty gate are different, a fault *f* is said to be visible on this output. In Figure 3.12, for example, fault f_1 is a visible fault.

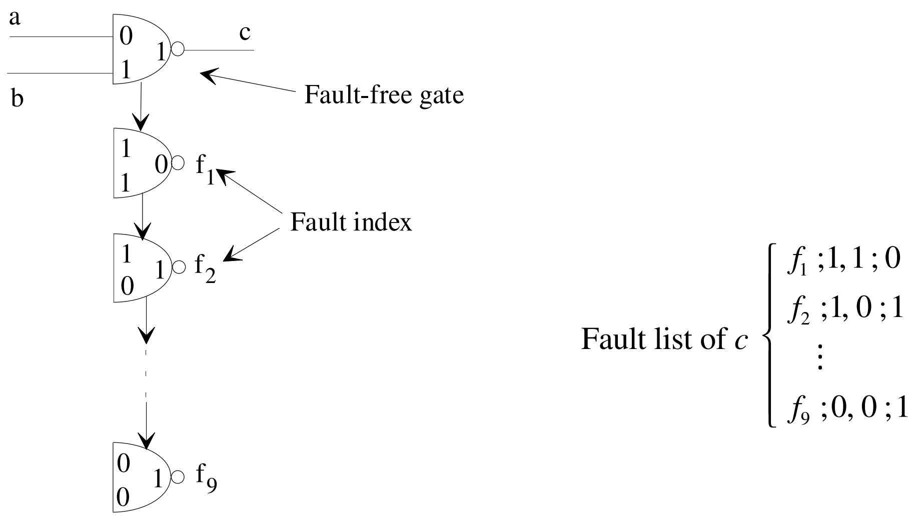


Fig. 3.12: Concurrent fault simulation

The concurrent fault simulation method can be explained using the example in Fig. 3.13, which represents a circuit as well as the influence of the faults f_1:*b*(*s*-*a*-0), f_2:*k*(*s*-*a*-0), and f_3:*c*(*s*-*a*-0) on the circuit. The fault status associated with each fault is represented by a replicated gate under the fault-free gate. The local fault f_1 affects a change in the value of all gates on the path to the primary output, therefore, the fault is included in the corresponding fault lists. Using the selected test vector, this fault can be detected at the output *p*. Fault f_2 however, does not cause a change in the output of the gate *l*. Therefore, fault f_2 is contained in the fault list of gate *l* as a local fault and is not visible on the output of this gate. The faults f_1 and f_3 are visible on many gate outputs but only f_1 can be detected at the output *p*.

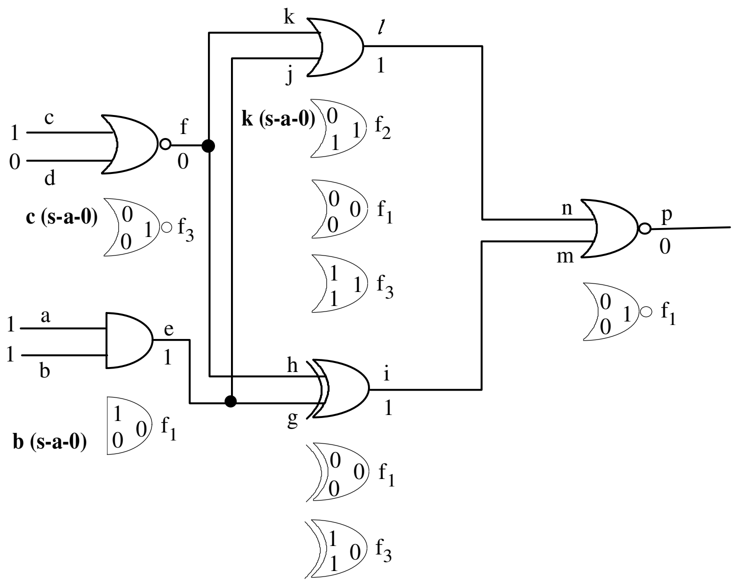


Fig. 3.13: Fault propagation of f_1 to output *p*

Parallel Pattern Single Fault Propagation (PPSFP)

PPSFP is an important and frequently applied method of simulation, which takes advantage of the word-oriented operations in a computer [WaEi85]. Simulation efficiency is dependent on the word length *i*, which is usually 2^4 to 2^8 bit. A word with length *i* can simulate parallel *i* test vectors in one simulation run and therefore the total number of simulation runs *SIM* is reduced to

$$SIM_{PPSFP} = \frac{n}{i} \tag{3.5}$$

for a test vector set containing *n* test vectors.

The simulation process is based on the logic operation of individual test vectors, whereby each vector can represent, for instance, one word. In general, the calculation for a logic element with two inputs *a* and *b* and an output *d* can be described as

$$vec(d) = vec(a)\ O\ vec(b) \tag{3.6}$$

where *O* represents an arbitrary logical operation for every bit of the input word. The i-th bit of the first input word *vec*(*a*) is always logically operated to the i-th bit of the second input word *vec*(*b*) and the result can be found in the i-th bit of the output word *vec*(*d*). The example in Figure 3.14 uses an AND gate to illustrate a two-valued fault-free simulation using PPSFP. Assuming that the word length is 4 bit *i*=4, then *vec*(*a*)=(0,1,0,1) and *vec*(b)=(0,0,1,1) can be selected. The AND-operation for every bit of the vectors *vec*(*a*) and *vec*(*b*) gives the result *vec*(*d*)=(0,0,0,1).

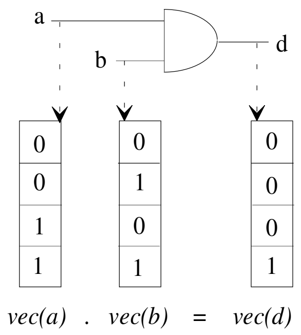


Fig. 3.14: Principle of two-valued Parallel Pattern Single Fault Propagation

The examination process involves injecting a fault at the fault location in a manner similar to parallel fault simulation. In the case of parallel fault simulation, the mask for a fault affected only a single word *w,* whereas two words are used with Parallel Pattern Single Fault Propagation. One of these words represents the fault-free value in each bit and is produced by a good simulation while the other word represents the value of the faulty circuit and is referred to as a fault-word. A fault-word will only be stored at the output of the gate when it differs in at least one bit from the word produced by a good simulation. The two-valued logic method can be easily expanded to a more-valued logic when further words are used in parallel.

The simulation runtime T_{PPSFP} is derived from the number of gate calculations per simulation run $\hat{E}$ (average activity), the value g, which indicates how many gates are evaluated per second (GEPS), and $SIM = SIM_g + SIM_f$, the number of simulation runs for fault-free $SIMg$ (Equation (3.5)) and faulty SIM_f circuits. The size of $\hat{E}$ depends on the evaluated circuit, the test vector set, and the modeled faults, whereas value g is hardware-dependent (workstation).

$$T_{PPSFP} = \frac{\hat{E} \cdot SIM}{g} \tag{3.7}$$

Using PPSFP techniques for sequential circuits an average number of iterations $\hat{I}$ must be executed for each simulation run. The size of $\hat{I}$ indicates how often a storage element must be calculated per simulation run. Further events are generated with each iteration at the combinational part of the circuit. Therefore, the number of gates that must be calculated for each simulation run increases, resulting in a runtime calculated with Equation (3.8). Factor $\tilde{n}$ depends on average circuit activity, the average number of iterations $\hat{I}$, word length i,and the average number of events produced per iteration.

$$\hat{T}_{PPSFP} = \frac{\tilde{n} \cdot SIM}{g}, \quad \tilde{n} = f(\hat{E}, \hat{I}, i) \tag{3.8}$$

When applied to combinational circuits this approach represents the fastest method of simulation. For sequential circuits, additional events are generated at the combinational part of the circuit where new storage elements can be activated through the iterations. Hence, an increase in $\tilde{n}$ leads to a significant increase in runtime. Because this approach is not appropriate for sequential circuits the use of other methods is required. For example, the PPSFP method has been used successfully for the evaluation of a large test vector set with 1 million test vectors.

3.5 Test Generation

The process of test generation involves determining the stimuli necessary to test a digital circuit. The two fundamental steps in generating a test vector for a fault *s-a-f* are first, the activation of the fault *f*, and second, the propagation of the resulting error to a primary output. When activating a fault, the values of the primary input are set to cause a signal with the value $\overline{f}$ at the fault location.

Test generation can be carried out manually or automatically. The most common methods of automatic test generation for digital circuits are classified in Fig. 3.15.

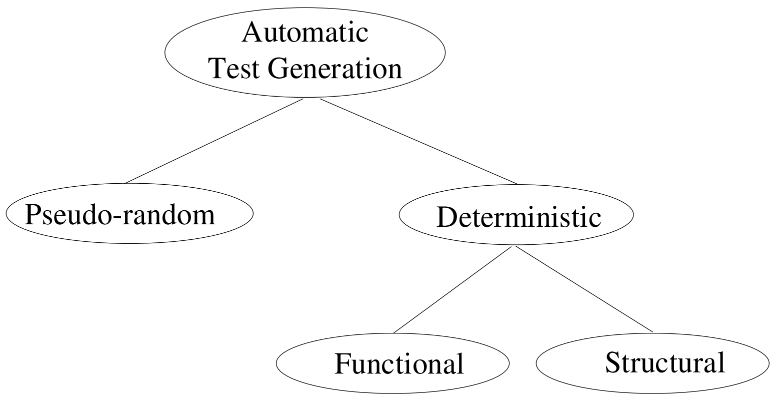


Fig. 3.15: Classification of test generation

In general, test generation can be divided into two specific approaches: deterministic and random. The first approach can be classified as functional for the evaluation of the boolean function of a circuit, or structural, for the extraction of test vectors from the topology of the circuit to sensitize specific paths. With the second approach, test vectors are generated pseudo-randomly. Hence, test vectors generated in this manner must be evaluated with fault simulation. Random test generation does not take into account the function or the structure of the circuit. While the deterministic method is on the one hand extremely time-consuming, the generated test vectors are on the other hand of a higher quality (fault coverage) than those generated by random test generation.

In this work hardware-based random generation of test vectors by a logic emulator is used. A detailed discussion of pseudo-random test vector generation as well as the corresponding hardware realization is included at the end of this section.

Deterministic Test Vector Generation

Deterministic test vector generation methods use a logical description of a circuit or a circuit structure for the determination of test vectors. The functional methods of test generation are based on the calculation of a node's function using boolean differences or fault matrix. The simplest approach for generating test vectors consists of constructing a fault matrix. The fault matrix technique developed by Kautz [Kaut68] [Chan65] uses a boolean matrix, called an *F*-matrix, to represent the relation between the set of all possible test vectors of the primary inputs for a given circuit, to their associated faults. The output values resulting from the application of the given test vectors under specified fault conditions are then entered into the *F*-matrix containing the results of all primary outputs for each fault with all input combinations. An evaluation of the matrix determines which faults are detected with which input combinations. The boolean difference [Selle68] is a formal method for the analysis of the operation of a circuit when faults occur at its primary inputs. The method is based on the exclusive OR operation between two boolean functions, one representing a faulty circuit and the other a fault-free circuit. A disadvantage of this technique is the highly complex calculation of the boolean difference. Since the calculation of the boolean difference and fault matrix becomes more complex with an increase in circuit size, the application of functional methods for increasingly complex digital circuits is becoming less frequent.

Structure-based approaches to test generation, referred to as path sensitization, are characterized by the attempt to find paths through a circuit so that a primary output is dependent on at least one primary input of the circuit, as well as by the evaluation of all nodes on this sensitive path. The basic path sensitization method [Arms66] is based on the activation of a specific fault within the circuit structure as illustrated in Fig. 3.16. The propagation of this fault to the primary output of the circuit along a sensitive path is referred to as forward trace, whereas the propagation of a fault along the sensitive path to the primary input is referred to as backward trace.

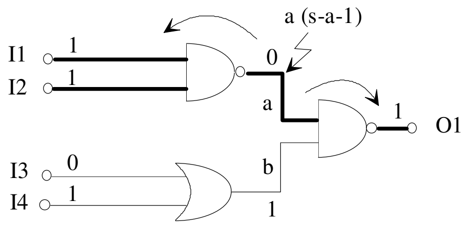


Fig. 3.16: Backward trace and forward trace along a sensitive path

The D-Algorithm [Roth66] is a standard method serving as the basis for the development of more efficient algorithms such as the PODEM-Algorithm. Test generation belongs to the class of NP-complete problems. In the case of combinational circuits, the complexity of the deterministic method grows quadratically to cubically [Goel80] with circuit size.

Testability is defined by [AbBF90c] as a design characteristic that affects various costs associated with testing. "Design for testability" are methods implemented during the design phase to establish that a circuit is testable. Two attributes related to testability are controllability and observability [Rutm72] [StGr76]. Controllability is the ability to establish a specific value at each node in the circuit by setting values on the circuit's inputs while observability is the ability to drive the specific value of a node to the circuit's primary outputs. These values are intended to represent the relative degree of difficulty for computing an input test vector or sequence for setting node *a* to the value 1, i.e. 1-controllability or to the value 0, i.e. 0-controllability, and propagating a fault to the primary output of the circuit. Controllabilty and observability measures are used with the goal of estimating the difficulty of generating test vectors for specific faults or for the entire circuit [AgMe82].

Random Test Vector Generation

A large set of random vectors is required to attain a test vector set of high quality, i.e. a high degree of fault coverage. Recall from Section 3.3 that fault coverage can be calculated by a fault simulator. Although random test vectors may be easily and rapidly generated, the use of fault simulation to determine the quality of random test vectors may be a time-consuming process dependent on the complexity of the circuit and the fault simulation algorithm.

One method of pseudo-random test generation uses a Linear Feedback Shift Register (LFSR) for the generation of test vectors [AbBF90]. With this technique, the number of inputs to the circuit determines the number of stages in the shift register. The feedback connections are selected with the goal of attaining maximum length. The *r*-stage LFSR is examined in Fig.

3.17, which displays the *r*-stage LFSR with feedback connections from all *r*-stages using a modulo-2 sum circuit (exclusive-OR). At each stages of the register the sequence is delayed by a single time interval compared to the previous stage. Depicted in Fig. 3.17 are the feedback connections m_1, m_2, ..., m_r and the storage elements FF_2 to FF_r, which receive their values from the preceding storage elements.

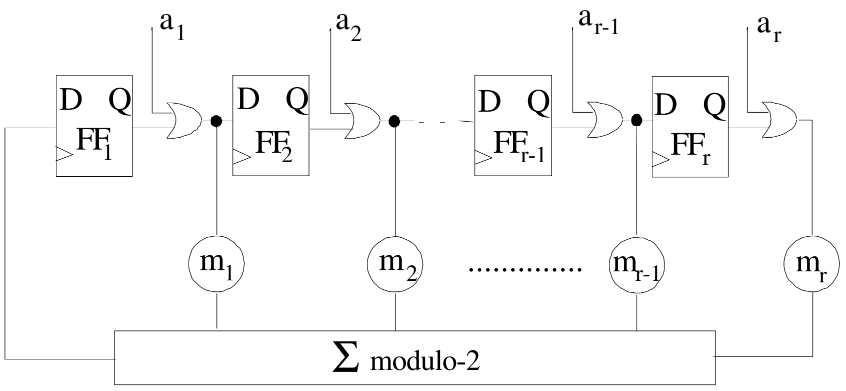


Fig. 3.17: Conceptual Linear Feedback Shift Register (LFSR)

The LFSR behavior is determined by the sequence of values generated at the inputs a_j and the feedback connections m_j and is expressed as a characteristic polynomial

$$P(a) = 1 + \sum_{j=1}^{r} m_j a^j \tag{3.9}$$

An all-zero state generates an all-zero sequence occurring with every LFSR regardless of the feedback connections. The maximum number of additional sequences possible is reduced by the zero sequence to 2^r-1.

Hardware Realization

Figure 3.18 illustrates an LFSR as a pseudo-random test generator with a length of 4 bit, which can be built in hardware by flip-flops for the *r*-stages (*r*=4) and an exclusive-OR for the modulo-2 sum. Note, that the outputs of LFSR are the value of flip-flops (Q).

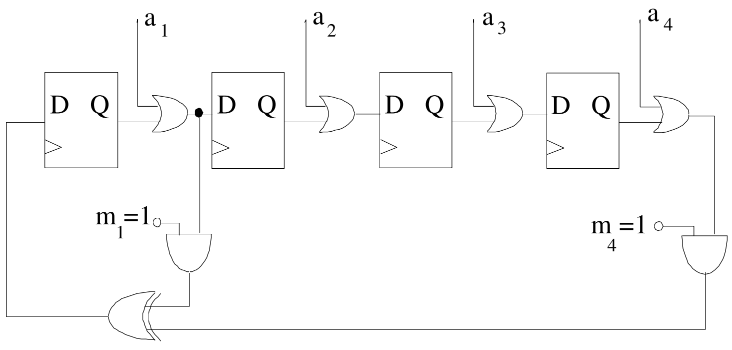


Fig. 3.18: Hardware realization of a LFSR as a pseudo-random test generator

The corresponding polynomial is indicated below as

$$\begin{aligned} P(a) &= m_4a^4 + m_3a^3 + m_2a^2 + m_1a + 1 \\ &= a^4 + a + 1 \end{aligned} \tag{3.10}$$

The sequence corresponding to this pseudo-random test generator is illustrated in Fig. 3.19, where the initial state is depicted as $\{a_1, a_2, a_3, a_4\}$={1,0,0,0} and the feedback connections as $\{m_1, m_2, m_3, m_4\}$={1,0,0,1}.

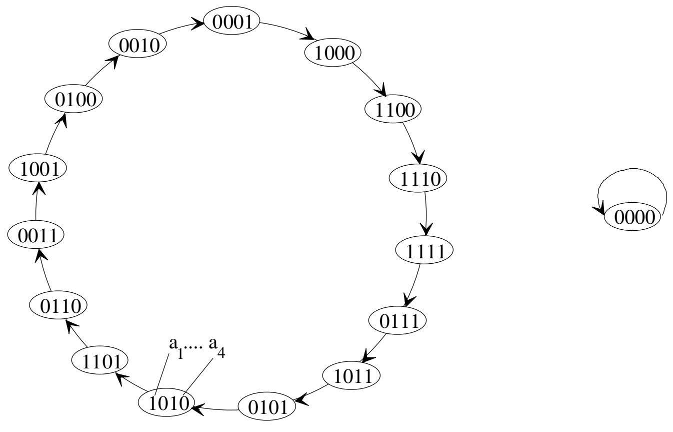


Fig. 3.19: Test generation cycle of Fig. 3.18

4. Fault Emulation

4.1 State of the Art

Traditional software-based approaches to fault simulation as well as hardware-based methods are introduced in Chapter 1. In the past years various methods have been presented for the generation of a hardware-based faulty circuit, each with its own approach to fault injection. Timoc [Timo79] and Cheng&Dai [ChHu95], for example, insert additional functions at the fault location, while SFE [BuRe96] and Krone [Kron96] utilize the reconfigurability of FPGAs. A common feature of each of these methods is the execution of hardware-based fault simulation at gate level and the modeling of faults using a stuck-at fault model. Different techniques utilized for the activation of faults in the circuit, such as a circular shift register and the reprogrammability of FPGAs, are discussed in greater detail in the next sections.

For fault detection and analysis processes Timoc applies a standard LSI-Tester, while the approaches of Cheng&Dai, SFE, and Krone are FPGA-based approaches, which use a logic emulator for the analysis of detected faults. Each method of hardware-based fault simulation, however, must handle the following tasks:

1. Fault injection
2. Fault activation
3. Fault detection and analysis.

The motivation behind hardware-based fault simulation is the reduction of the runtime of software-based fault simulation described in Chapter 3 in order to attain a speedup. Each hardware-based fault simulation technique achieves a different speedup, which is dependent on the applied technology, such as breadboarding or FPGA. A description and comparison of the various hardware-based fault simulation approaches is the focus of the following sections.

4.1.1 Timoc Approach

The first hardware-based approach to fault simulation was introduced in 1979 by Timoc [Timo79] and consists of expanding the circuit with fault simulation capabilities implemented by breadboarding and wire-wrapping techniques. In order to execute a hardware-fault simulation, the circuit must be prepared in the following manner:

Phase 1: Construction of a fault-free breadboard to execute the function of the circuit.

Phase 2: Addition of fault simulation capabilities to the fault-free breadboard.

Phase 3: Evaluation of the stuck-at-fault detection using a commercial LSI-Tester.

In order to model stuck-at-faults, the appropriate fault locations in the circuit are expanded by additional functions, such as OR and NOR. The following example in Fig. 4.1 illustrates a hardware fault simulator for a two-input AND gate with expanded functions on its primary inputs *X1* and *X2* [Timo79]. With fault collapsing the only faults to be injected are input *X1* stuck-at-one (*X1*-1), input *X2* stuck-at-one (*X2*-1), and input *X1* or *X2* stuck-at-0 (*X1*-0), (*X2*-0). The circuit represented below can simulate a good two-input AND gate as well as any of the single stuck faults associated with an AND gate.

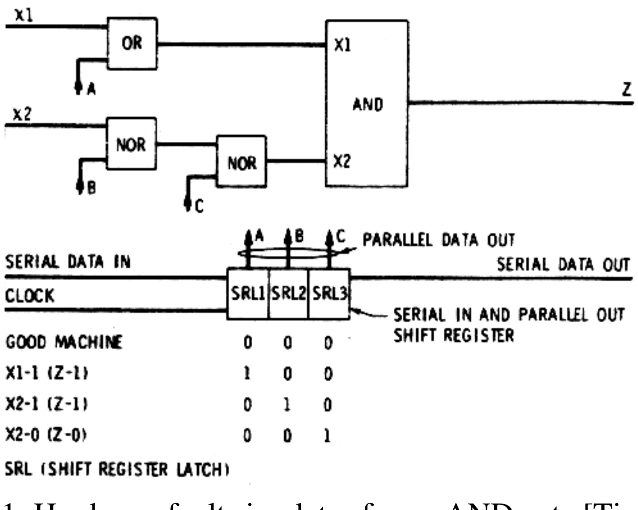


Fig. 4.1: Hardware fault simulator for an AND gate [Timo79]

The objective of Phase 2 is to enhance the fault-free breadboard constructed in Phase 1 with fault simulation capabilities by modeling a stuck-at fault for each primitive gate. Figure 4.1 illustrates the implementation of the fault simulation capability. After the substitution of fault simulation capabilities for all gates, the shift registers for these capabilities are cascaded, forming a long shift register. When a "1" is contained at the appropriate position of the shift register, the particular gate's inputs and outputs are selected and the stuck-at- fault can be simulated.

This approach was applied for the evaluation of an IBM microprocessor [Timo79] and resulted in a speedup of 2 in comparison to the IBM Deductive Fault Simulator [Cha76]. However, complicated construction as well as the high cost involved in the development of a breadboard are factors that must be considered before its implementation.

4.1.2 Cheng&Dai Method

The first FPGA-based method of fault simulation (fault emulation), was developed by Cheng&Dai in 1995 and does not differ greatly from the approach presented by Timoc. The Cheng&Dai method uses the reprogramming feature of the FPGAs and utilizes additional logic for the modeling of stuck-at faults. This fault injection and fault activation techniques in the expanded circuit lead to mapping problems in the emulator. The Cheng&Dai approach is described below, followed by a thorough discussion of the mapping problem. Cheng&Dai present a solution to this problem, which, however, results in a higher fault emulation runtime.

The Cheng&Dai method is presented in Fig. 4.2 and involves the serial injection and emulation of faults. Prerequisites to this process are recompilation, the preparation of new data resulting from the partitioning, technology mapping, placement, and routing processes for the reprogramming of FPGAs, as well as reconfiguration, involving the reprogramming of FPGAs. FPGAs are reprogrammed by downloading the bitstream files created during recompilation. The selected fault can be injected by reconfiguring the FPGA in order to convert a fault-free circuit into a faulty circuit. The test vector is then applied to the logic emulator that implements the faulty circuit and the results are compared to the expected output of the fault-free circuit. When the results differ, a fault has been detected and the process is reiterated until all faults have been injected. Before a fault emulation process the implementation of the fault-free design to the logic emulator as well as fault collapsing for the creation of a collapsed fault list is required.

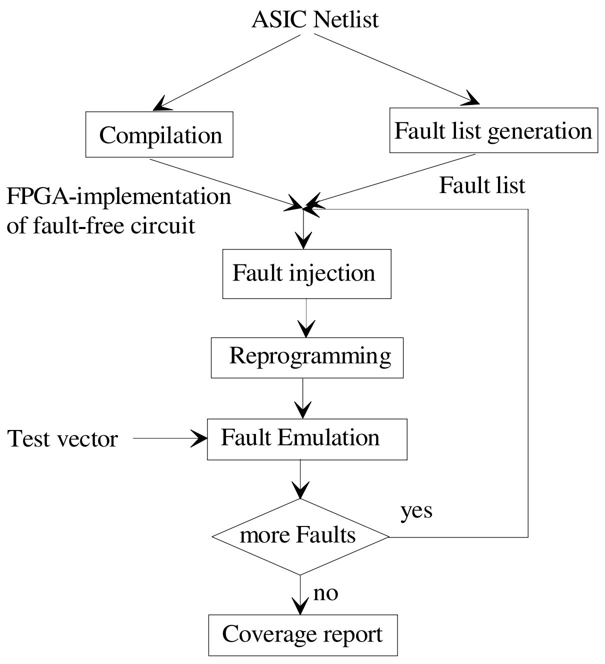


Fig. 4.2: Flow diagram of Cheng&Dai method [ChHu95]

An example of fault injection is shown in Fig. 4.3, with Fig. 4.3a displaying the mapping of a circuit section to an FPGA. An additional controller with two outputs, *x* and *y*, as well as an additional gate (*GT1 or GT2*) for each fault is added to the nets *a* and *g* in Fig. 4.3b. An OR gate, *GT1*, is added for the injection of the stuck-at-1 fault. When *x*=1, the fault is present, i.e. activated, whereas when *x*=0 no fault is activated and CLB1 is in a fault-free state. In the same manner, an AND gate, *GT2*, is added for the injection of the stuck-at-0 fault, which is active when *y*=1 and inactive when *y*=0, indicating the fault-free state for CLB2. Similar to the Timoc approach, a shift register (SR) is used for the activation of a single fault for each emulation process.

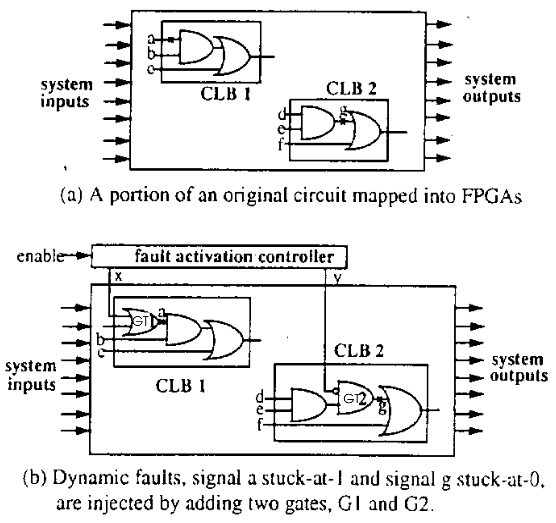

system inputs
CLB 1
CLB 2
system outputs
a
b
c
d
e
f
g
(a) A portion of an original circuit mapped into FPGAs
enable
fault activation controller
x
y
GT1
GT2
(b) Dynamic faults, signal a stuck-at-1 and signal g stuck-at-0, are injected by adding two gates, G1 and G2.

Fig. 4.3: Fault injection and activation of Cheng&Dai method [ChHu95]

The FPGA-system routing and FPGA routing problems are extremely complex due to the mapping problems caused by the shift register. When a large number of dependent faults are to be simulated in the circuit, extremely long shift register chains are required. Since each flip-flop of the shift register is mapped to flip-flops in the CLBs an increase in logic density and thus a decrease in flexibility in the FPGA is expected and therefore routing resources are insufficient for the mapping of complete circuits [BrFr92c].

As a solution, Cheng&Dai suggest that no more than 10% of all flip-flop resources of an FPGA are utilized. For instance, with an emulator consisting of Xilinx 4010 FPGAs, a maximum of 20 faults can be emulated in each FPGA. In order to map a circuit with a fault set *F* of 100k faults into an emulator with 100 FPGAs, 20 replications each with *D*=5000 faults are created from the original circuit. Because each circuit must be downloaded to the emulator, the reconfiguration time for the complete FPGA system amounts to approximately 1 minute. According to the manufacturer of the Xilinx 4000 series FPGA [Xili94], the reconfiguration time *R* required for each FPGA varies between 0.5 and 1 second. Thus, the reconfiguration time for all 100k faults amounts to 20 minutes, which must still be added to the fault emulation runtime *Rf* for the total runtime *Rt*. Hence, the total runtime can be calculated as:

$$Rt = \frac{F}{D} \cdot R + Rf \tag{4.1}$$

The additional functions necessary for fault injection, as well as for the control of the injection process, lead to the generation of a hardware overhead. As stated in [ChHu95], each of the replicated expanded circuits has an average CLB-overhead of 1.3 - 2 when compared to the original circuit. This technique was never fully implemented in a logic emulator due to the mapping problem and long emulation runtimes, therefore, emulation results are not available.

4.1.3 Serial Fault Emulation

Serial Fault Emulation (SFE), a further variation of hardware-based fault simulation was developed in 1996 by Meta Systems [BuRe96]. SFE was developed specifically for the logic emulator from Mentor-Meta Systems and represents a completely new method of fault emulation. The rapidly reconfigurable FPGAs from Meta Systems play a key role in the fault emulation process by allowing direct access to, as well as by changing the function of logic blocks (BLP) during an emulation process. The calculation of FPGA reconfigurations using the CAP (Computer-Aided Prototyping) software from Meta Systems is described on the left side of Fig. 4.4. A collapsed fault list is created by a fault generator using a netlist at gate level. The first step involves the generation of a fault specification file, which determines those logic blocks into which faults will be inserted for fault emulation.

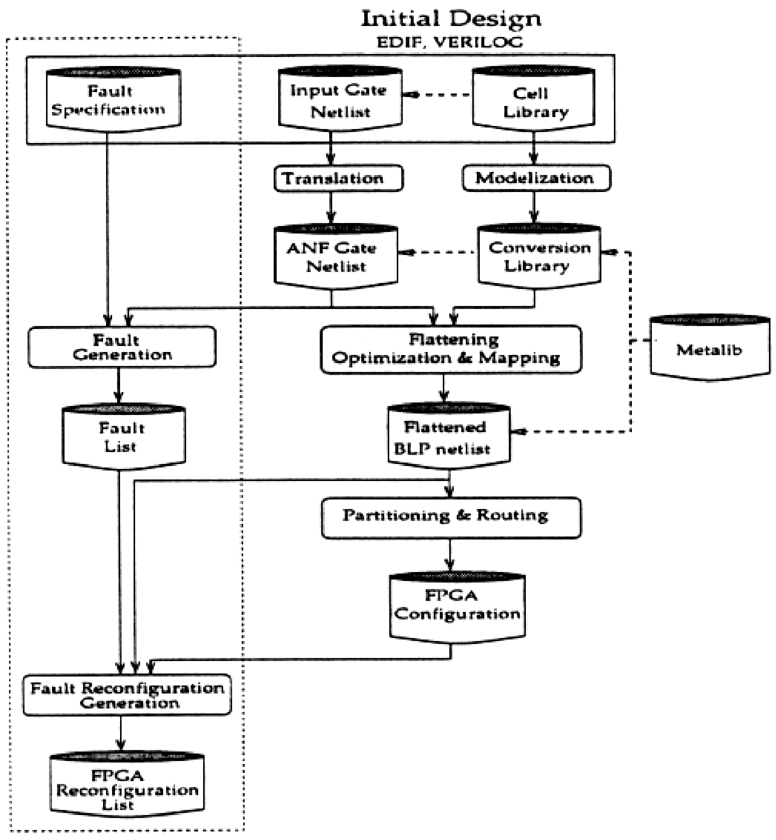


Fig. 4.4: SFE flowchart [BuRe96]

The next step in the reconfiguration process includes computing the FPGA reconfiguration corresponding to individual faults. This is a prerequisite for designing a hardware prototype of the faulty circuit. An FPGA reconfiguration is associated with a list of BLPs that are reprogrammed in order to generate a faulty circuit from a fault-free circuit. Each BLP in the reconfiguration corresponds to a list of the functions for each BLP for the fault-free and the faulty circuit as well as a logical address in the emulator (*bo, fp, bl*) which is a specific reference to the board number *bo* in the emulator, the FPGA number *fp* on the board, and the BLP number *bl* within the FPGA.

By reconfiguring an FPGA for single-stuck-at faults (SSF) at a gate only the 4-input function of the BLP is influenced. An example of FPGA reconfiguration is given in Fig. 4.5. The three gates *GT0, GT1 and GT2* of the circuit are mapped into a BLP with the logical address (0,0,0) (indicating board 0, FPGA 0 and BLP 0) and the fourth gate, *GT3*, to the BLP (0,0,1). The equation $Z=A\cdot B + C\cdot D$ represents the 4-input function of the BLP (0,0,0). When signal x in the circuit is stuck-at-0, the BLP must be reconfigured to implement the function $Y=C\cdot D$.

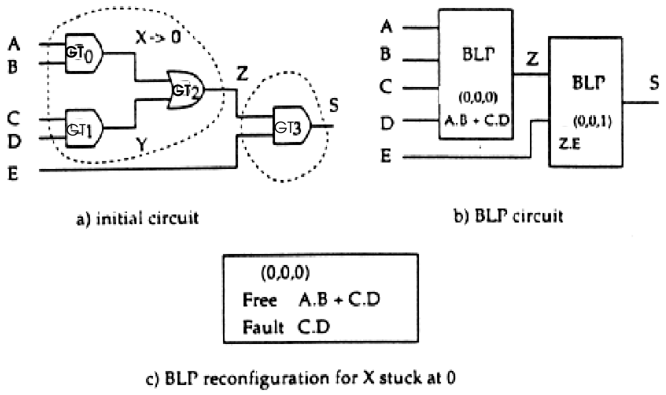


Fig. 4.5: An example of FPGA reconfiguration [BuRe96]

The runtime required for SFE is determined by the number of emulation cycles to be executed as well as the maximum clock speed (MCS), i.e. operating frequency, of the circuit in the emulator. Given that T_{reconf} represents the time required for reconfiguration of the FPGA-system and *AN* is the average number of test vectors necessary for fault detection, serial fault emulation (SFE) speed *SP* can be defined as:

$$SP_{SFE} = \frac{MCS}{AN + T_{reconf} \cdot MCS} \tag{4.2}$$

The time required for the reconfiguration of the BLPs is the most important factor in T_{reconf}. Reconfiguring a Xilinx FPGA involves reprogramming the complete FPGA, whereas direct access to the Meta FPGA allows individual sections of the FPGA to be read or modified. Generally, the reconfiguration time T_{reconf} is equal to 0.8 milliseconds [BuRe96]. Assuming that all faults are detected by the first test vector in a test vector set, i.e. $AN=1$, then a maximum of 1.200 faults can be emulated per second. The reconfiguration time is negligible when the average number of test vectors for each fault is over 10,000. However, experimental results [BuRe96] indicate that 90% of all faults are detected by the first hundred test vectors.

Results obtained with SFE have been compared to those from the fault simulator HOPE [LeHa92][LeHa93]. HOPE, which combines fault simulation approaches such as single fault propagation and parallel fault processing, was evaluated previously under similar conditions with an identical fault model and test vector set. A Sparc 10 workstation was also used to evaluate the performance of the HOPE simulator. During fault injection, i.e. reconfiguration of the emulator, an interruption in the emulation process, as described above, leads to a reduction in emulation speed. These SFE characteristics play an important role in the relatively low speedup from 8 to 20 of SFE over HOPE when fault emulation is carried out using the ISCAS´89 Benchmark circuits.

4.1.4 KRONE

The KRONE AG developed a reprogrammable prototyper in 1996 capable of modeling the timing of logic modules in the circuit as well as stuck-at faults [Kron96], which however was never actually constructed. Additional publications on this subject are not available, therefore, this discussion focuses only on the function blocks of the KRONE prototyper and the modeling of stuck-at faults. A Field Programmable Emulation Chip (FPEC) contains an array of 32 by 32 emulation cells (FPE-cells). These cells model the logic function of a gate-level netlist. Modeling a 4/1 Multiplexer, a D-flip-flop (Fig. 4.6) or a 4-input logic function is also feasible (Fig. 4.6). In addition, timing elements such as wiring- and load-delays are modeled by using adjustable delay lines(Fig. 4.6).

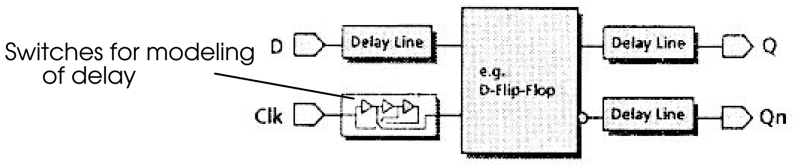


Fig. 4.6: Delay lines [Kron96]

As with simulation, during the emulation process all internal nodes of an FPE-cell are observable and controllable at any time during the emulation process. The controllability of the FPE-cell system allows the stuck-at fault to be modeled with an embedded switch (Fig. 4.7). Faults can be inserted and their effects analyzed due to the implementation of the reprogrammable prototyper as a fault emulator.

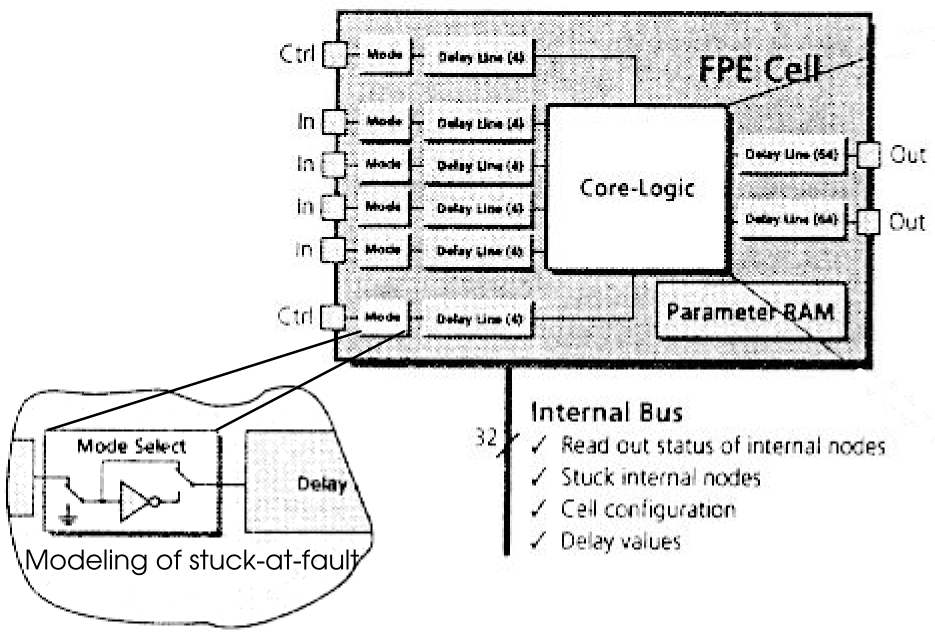


Fig. 4.7: FPE-cell [Kron96]

Comparison

The ability to model stuck-at faults is a common feature of all previously described hardware-based fault simulation methods. Differences, however, exist in the areas of fault injection, fault activation, evaluation of emulation results, hardware overhead, emulator technology dependency, and runtime. The fault injection methods of Timoc and Cheng&Dai involve expanding the circuit description by additional functions, such as AND for s-a-0 and OR for s-a-1. Both techniques also utilize a shift register (SR) for fault activation. For the evaluation of results, Timoc uses a commercial LSI-Tester, whereas Cheng&Dai propose utilizing a logic emulator as a comparator for good and fault emulation results. The consequence of both approaches is an extremely high hardware overhead.

The approaches presented in the previous sections are listed in Table 4.1 based on a comparison of the following aspects of hardware-based fault simulation:

- Method of fault injection at gate-level.
- Hardware implemented for the activation and deactivation of fault for fault-free and faulty circuits.
- Evaluation of emulation results; calculation of fault coverage; observation and analysis of the circuit after fault injection.
- Ramifications of additional hardware; hardware overhead including fault injection, fault activation and comparison modules.
- Dependency on specific hardware technology, such as breadboarding and FPGAs.
- Runtime.

Approach	Fault Injection	Fault Activation	Evaluation of Results	Hardware Overhead	Technology Dependency	Runtime
Timoc (1979)	additional logic functions OR / NOR/...	shift register	LSI-Tester	n/a	breadboarding and wire-wrapping	high
Cheng&Dai (1995)	additional logic functions AND / OR/...	shift register	emulator	1.3 - 2 for each replication	none	n/a
SFE (1996)	reconfiguration of logic blocks (BLP)	emulator software	additional hardware	n/a	Meta FPGAs	low-high
KRONE	switch modules embedded in FPE-cells	emulator software	emulator	n/a	KRONE FPEC	n/a

Table 4.1: Comparison of available hardware-based fault simulation approaches

4.4 Objectives of this Work

As explained in previous sections, the various hardware-based fault simulation methods differ in the areas of fault injection, fault activation, evaluation of emulation results, hardware overhead, emulator technology dependency, and runtime. Taking into account the above-mentioned aspects of hardware-based fault simulation, a new approach to fault emulation can be developed, which, for optimum fault emulation results, should fulfill the following list of requirements :

- Rapid fault injection.
- Aspects of fault injection and fault activation to be considered with the objective of minimizing the hardware overhead.
- Mapping of the faulty circuit with optimized partitioning, technology mapping, placement, and routing.
- Independence from specific emulator technology.
- Use of the logic emulator speed as a result of parallel execution of the logic blocks.
- Implementation quality to be increased, including minimal FPGA usage leading to improved circuit timing characteristics in the emulator resulting in optimized fault emulation runtimes.
- Runtime reduction of fault simulation in order to attain a speedup.

These are the requirements for a novel approach, which is the focus of this work. The next section introduces Fault Emulation System FES, which is characterized by its method of rapid fault injection approach including fault activation through a fault activator. Also shown is a calculation of fault coverage from the fault emulation results. Chapter 5 presents various procedures for mapping faulty circuits while focusing on optimal partitioning, technology mapping, placement, and routing.

4.2 Fault Emulation System (FES)

Two new approaches to fault emulation are presented, FES1/1 and FES/2. Although both approaches utilize the same method of fault injection and fault activation in the FPGAs, fault emulation FES/1 [SeBa97] [Seda97] uses the in-circuit mode and involves expanding the logic emulator by additional hardware modules for test generation and test analysis, whereas fault emulation FES/2 [SeBa98] [Seda97a] uses the acceleration mode and evaluates the test vector set without additional hardware. In contrast to the previously presented methods of fault emulation, these new approaches involve no reprogramming or reconfiguration of the FPGAs and allow for faster fault injection into any node of the circuit without dependency on a specific logic emulator technology.

4.2.1 FES/1

The new fault emulation approach FES/1 utilizes the in-circuit emulation mode of the logic emulator. Figure 4.8 displays the FES/1 flowchart beginning with a circuit represented as a netlist, which is expanded by additional functions for fault injection, referred to as Fault Injectors. The faultlist $F=\{f_1, f_2,..., f_i, ..., f_n\}$ for this circuit indicates the set of fault locations in the circuit $FLO=\{flo_1, flo_2,..., flo_i, ..., flo_n\}$, i.e. the nets where the Fault Injectors $FI=\{fi_1, fi_2,..., fi_i, ..., fi_n\}$ should be inserted. Each Fault Injector fi_i has a corresponding logical address and is controlled by the Fault Activator. The expansion of a circuit using Fault Injectors and a Fault Activator results in an overhead of FPGA resources in the logic emulator. The node assignment method described in the following chapter leads to an optimum usage of FPGA-resources for fault emulation, i.e. a reduction of FPGA overhead. Compilation, which is the process of partitioning, technology mapping, placement and routing precedes the mapping of the expanded circuit into the logic emulator.

After the expanded circuit has been mapped into the emulator both the faulty and the fault-free circuits are present in the emulator. If the Fault Injectors are deactivated the circuit is fault-free indicating a good emulation. A good emulation is performed for all test vectors of the test vector set $V=\{v_1, v_2, ..., v_j, ..., v_m\}$, and the good results are stored. The fault emulation involves the activation of Fault Injectors, therefore, the process begins with the activation of the first Fault Injector fi_1 with the address 0 and applies the first test vector v_1 at the primary inputs of the circuit.

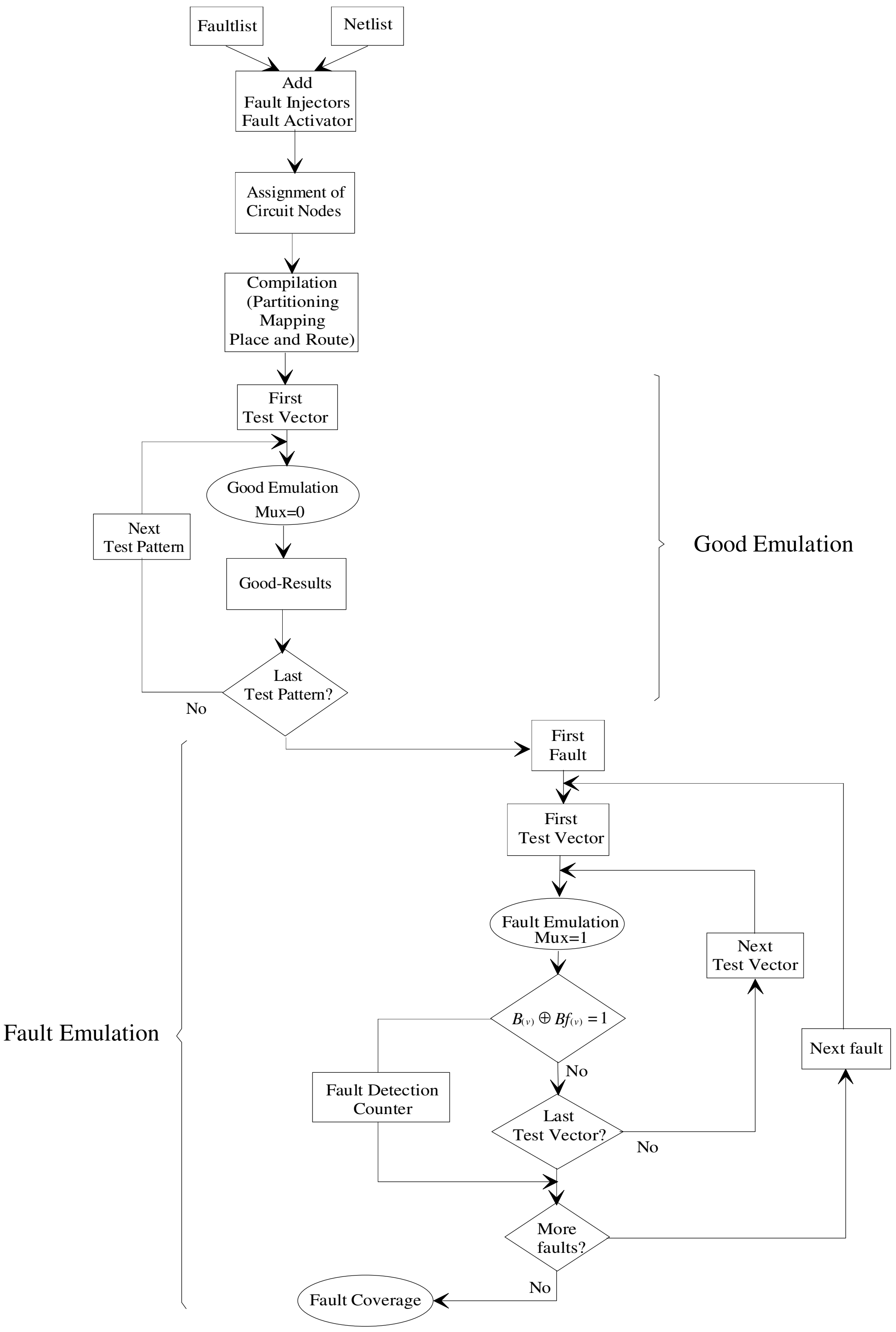


Fig. 4.8: FES/1 flow diagram

An internal and an external loop are included in the fault emulation process (Fig. 4.8). The purpose of the internal loop is to apply test vectors from the test vector set *V* to the primary inputs of the circuit for each activated fault, while the external loop injects, i.e. activates faults from the faultlist. A test vector v_j detects a fault f_i if the fault-free function of the emulated circuit $B(v_j)$, i.e. good emulation, differs from the faulty functions of the emulated circuit $Bf(v_j)$, i.e. fault emulation. If the fault emulation process for the test vector v_j results in $B(v_j)\oplus Bf(v_j)=0$, the next test vector v_{j+1} is applied to the primary inputs of the circuit with the same activated Fault Injector fi_i. This process is repeated until the fault is detected ($B(v_j)\oplus Bf(v_j)=1$), leading to an interruption of the internal loop, or until the test vectors are exhausted in the test vector set. At this point the fault is dropped and the next Fault Injector fi_{i+1} is activated. With the detection of a fault f the fault detection counter is incremented in the fault emulation process. The calculation of fault coverage follows once the fault set F has been processed.

Fault Emulation Hardware

The process of hardware-based fault emulation FES/1 is illustrated in Fig. 4.9. The complete system consists of the logic emulator hardware and two additional hardware modules, the test vector hardware module and the comparison hardware module. The former is connected to the primary inputs of the mapped circuit, i.e. the inputs of the emulator hardware, the latter to their outputs. The test vector hardware module can be used to load external test vectors or as a pseudorandom test generator (PRTG) [Hart96], as described in Section 3.5. The comparison hardware module evaluates the results of the good/fault emulation and indicates the number of detected faults, enabling the calculation of fault coverage.

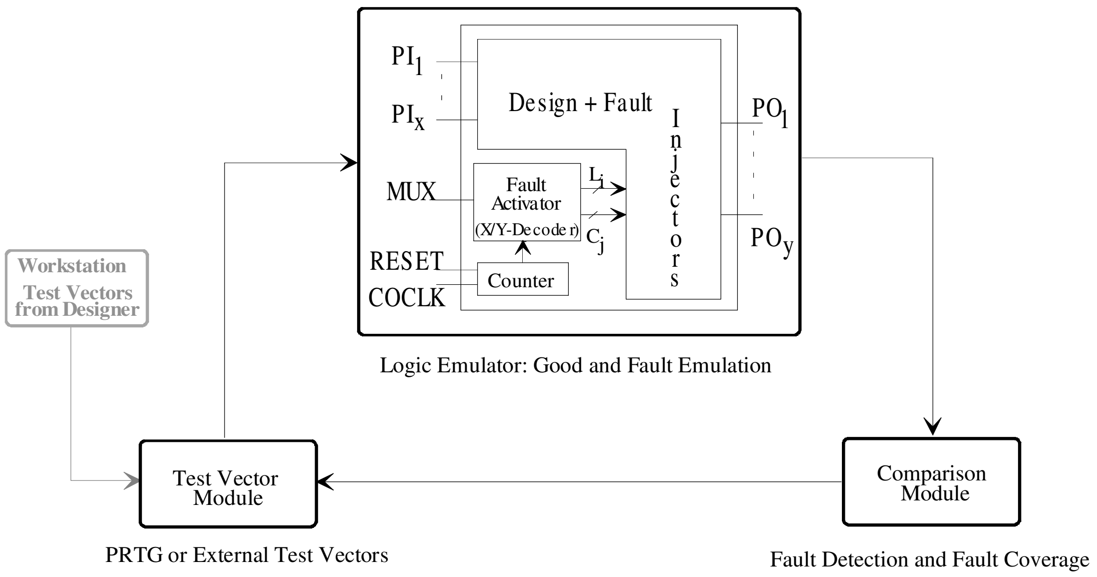

PI1
PIx
Design + Fault
Injectors
PO1
POy
MUX
Fault Activator (X/Y-Decoder)
Li
Cj
RESET
COCLK
Counter
Logic Emulator: Good and Fault Emulation
Workstation
Test Vectors from Designer
Test Vector Module
PRTG or External Test Vectors
Comparison Module
Fault Detection and Fault Coverage

Fig. 4.9: Fault emulation hardware

Test Vector and Comparison Hardware Modules

A 16-bit microcontroller is responsible for the control of all hardware modules, the loading of external test vectors [Hart96], and the generation of all fault emulation control signals, which involves expanding the primary input of the circuit (PI_1, ..., PI_x) by the following three control signals:

- The signal *MUX* controls the emulation process: good emulation (*MUX*=0), i.e. emulation of fault free-circuit, and fault emulation (*MUX*=1), i.e. emulation of faulty circuit.

- The faults are activated by a two-dimensional array of x-y address decoders. Driven by the *COCLK* signal, a synchronous binary counter is utilized to increment the addresses of Fault Injectors in the decoder.

- The *RESET* signal initializes all fault emulation hardware; the logic emulator, the test vector hardware module, and the comparison hardware module.

An important component of the test vector module is the PRTG, a multi-output device implemented using a linear feedback shift register (LFSR, described in Chapter 3) capable of generating up to 0.5 million test vectors with 64 bits with 4x16 bit word width memorys and can be configured with up to 128 bits with 8x16 bit word width memorys for 8 million test vectors.

The comparison hardware module [Seda97b] consists of memory and XORs and is connected to the outputs of the emulator, i.e. the mapped circuit. The good emulation results *B*(*v)* are stored in the memory. The XORs are used to compare the results of good/fault emulations. Detected faults are counted if $B(v) \oplus Bf(v) = 1$ as illustrated in Fig. 4.10.

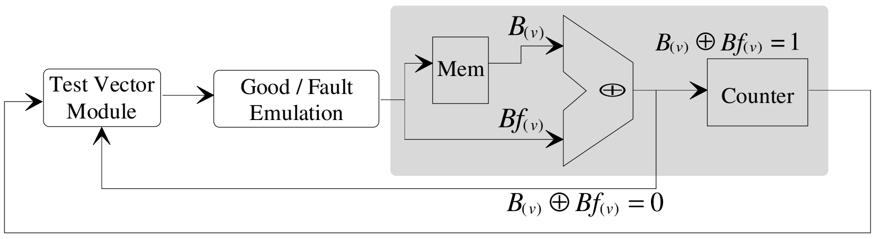


Fig. 4.10: Comparison hardware module

4.2.2 FES/2

The processes of expanding the circuit, node assignment, compilation, and good emulation are identical in the FES/1 and FES/2 approaches. In contrast to FES/1 however, FES/2 utilizes the acceleration mode of the logic emulator and the good emulation results from FES/2 are stored as a file in the host of the logic emulator rather than in the memory of a hardware module. Additionally, the fault emulation procedure for the FES/2 approach includes the external loop for applying test vectors to the primary input of the circuit and the internal loop for the activation of Fault Injectors.

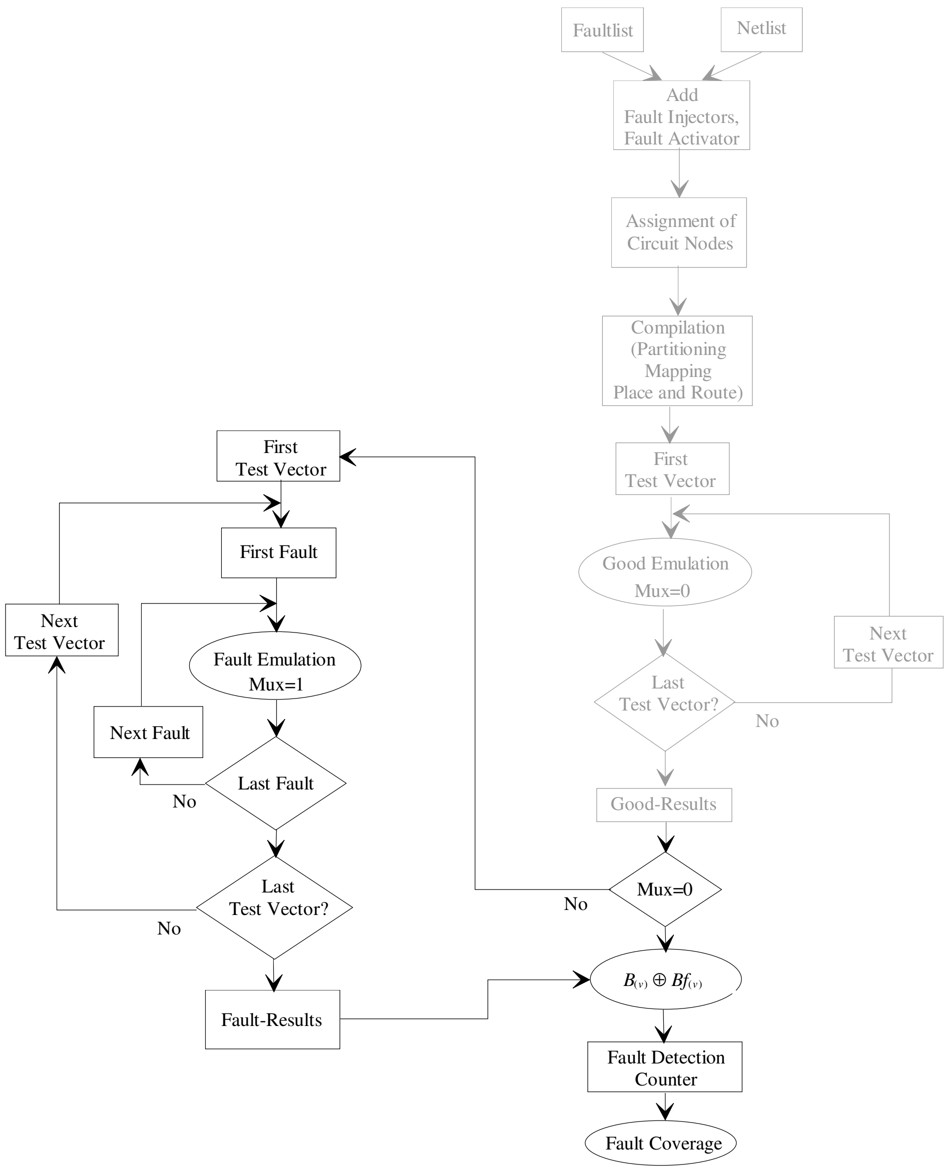


Fig. 4.11: FES/2 flow diagram

As shown in Fig. 4.11, at the beginning of FES/2 a test vector v_j is applied to the circuit's primary inputs (PI_1, ..., PI_x) and all faults in the internal loop are injected serially until the fault set F of the faultlist is completed. The next test vector v_{j+1} is then applied, followed by the serial injection of all faults of the faultlist. This process is repeated until the test vector set V has been evaluated. Fault emulation results are stored in a file in the host of the logic emulator. Fault coverage δ can now be calculated by comparing the good and faulty emulation results.

With the FES/2 approach it is feasible that a fault f will be detected several times in a test vector set, which results in an increase in fault emulation runtime when compared to FES/1. In acceleration mode the test vector set V will be compiled as a binary code[Quic96], which cannot be interrupted during the emulation process, therefore, fault dropping is not possible in accelerator mode. Therefore, the data of the control signals (*MUX*, *COCLK*, and *RESET*) for fault injection and fault activation are included in the test vector set.

4.2.3 Fault Injection

Fault injection in FPGAs is one of the main problems of fault emulation. Recall that a rapid hardware-based fault injection requirement was established in Section 4.5 in addition to the requirement that fault injection be independent of specific emulator technology. As shown by the fault injection techniques described in Section 4.1 the processes of reconfiguration and reprogramming of FPGAs are too time consuming. Therefore, the rapid fault injection requirement presented in Section 4.5 cannot be satisfied. Inherent in the processes of reconfiguration and reprogramming of FPGAs is the dependency on a particular technology, such as the SFE and Cheng&Dai approaches. Therefore, the requirement of technology independence also cannot be fulfilled.

In order to satisfy these requirements the fault injection process is enhanced by the addition of Fault Injectors. Fault Injectors expand the fault location in the circuit by additional boolean functions and are also used for modeling stuck-at-0 and stuck-at-1 faults. In general, every Fault Injector consist of two control inputs, a data input and a data output. The activation/deactivation of the Fault Injector, which causes the faulty/fault-free circuit, is controlled by signals L_i and C_j and determined in the Fault Injector by the signal *EN* ($EN=L_i \cdot C_j$). These signals are generated by a Fault Activator. The data input N_{in} and data output N_{out} of the Fault Injector are linked to the fault location in the circuit. The values of N_{in} and N_{out} are equal in a fault-free circuit, while in a faulty circuit the value of N_{out} is independent of N_{in}, but depends on the modeled fault, which is predefined by the Fault Injector.

Stuck-at-0 Fault Injector

To model a stuck-at-0 fault the fault location in the circuit is expanded by the function $N_{out} = \overline{L_i \cdot C_j} \cdot N_{in}$, which corresponds at gate level to two primitive gates, as shown in Fig. 4.12. If the control signal $\overline{EN} = \overline{L_i \cdot C_j} = 1$, the stuck-at-0 Fault Injector is deactivated causing $N_{in} = N_{out}$ and a fault-free circuit *B* results. In contrast, when the value of the control signal of the Fault Injector is $\overline{EN} = \overline{L_i \cdot C_j} = 0$, then $N_{out} = \overline{L_i \cdot C_j} \cdot N_{in} = 0$, i.e. the Fault Injector is in an active state, modeling the stuck-at-0 fault and resulting in the faulty circuit *Bf*.

$$N_{out} = \overline{L_i \cdot C_j} \cdot N_{in} \quad \Rightarrow$$

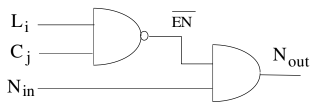


Fig. 4.12: Stuck-at-0 Fault Injector

Stuck-at-1 Fault Injector

Comparable to a stuck-at-0 Fault Injector, modeling a stuck-at-1 fault entails expanding the fault location in the circuit with the function $N_{out} = L_i \cdot C_j + N_{in}$. This corresponds at gate level to two primitive gates as shown in Fig. 4.13. In the case of fault-free circuit *B*, the control signal has the value $EN = L_i \cdot C_j = 0$. With *EN*=0 the Fault Injector is deactivated and leads to a fault-free circuit, therefore $N_{in} = N_{out}$. In the activated state the stuck-at-1 Fault Injector has the value $EN = L_i \cdot C_j = 1$, which results in $N_{out} = L_i \cdot C_j + N_{in} = 1$. Thus, the modeled stuck-at-1 fault generates a faulty circuit *Bf*.

$$N_{out} = L_i \cdot C_j + N_{in} \quad \Rightarrow$$

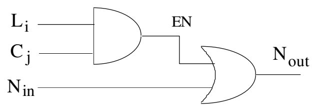


Fig. 4.13: Stuck-at-1 Fault Injector

Stuck-at-0/1 Fault Injector

The expansion of the fault location in the circuit by the function $N_{out} = EN \cdot SF + N_{in} \cdot \overline{EN}$ involves modeling both stuck-at faults, stuck-at-0 and stuck-at-1. Fig. 4.14 depicts the stuck-at-0/1 Fault Injectors at gate level, which have an additional input *SF*. The value of *SF* determines the function of the Fault Injector when $EN = L_i \cdot C_j = 1$. When the value of *SF* is set to logical 1, (*SF*=1) the Fault Injector models the stuck-at-1 fault, i.e. $N_{out} = (L_i \cdot C_j) \cdot SF + N_{in} \cdot (\overline{L_i \cdot C_j}) = 1$. In the case of input *SF*=0 the output of the Fault Injector models the stuck-at-0 fault and the value of $N_{out} = (L_i \cdot C_j) \cdot SF + N_{in} \cdot (\overline{L_i \cdot C_j}) = 0$.

$$N_{out} = EN \cdot SF + N_{in} \cdot \overline{EN} \quad \Rightarrow$$

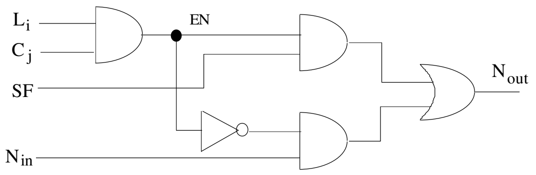


Fig. 4.14: Stuck-at-0/1 Fault Injector

A symmetrical FPGA (such as XILINX) consists of several CLBs, each of which contains two look up tables (LUT). A boolean function depending on four boolean variables can be mapped into each LUT. The function of Fault Injectors to model stuck-at-0 and stuck-at-1 faults depends on three or four boolean variables (L_i, C_j , *SF*, and N_{in}). Therefore, the mapping of Fault Injectors requires a complete LUT, which leads to higher FPGA usage, referred to in this work as FPGA-overhead.

4.2.4 Fault Activation

Fault activation presents an additional problem for fault emulation. The requirements for fault emulation presented in Section 4.5 include minimizing hardware overhead with optimized compilation of the expanded circuit, maximum fault emulation speed, and independence from specific emulator technology.

Fault activation through use of a shift register in an emulator [ChHu95] leads to difficulties mapping the circuit into the logic emulator. Cheng&Dai's solution to the mapping problem causes a high hardware overhead and results in a drastic incres in fault emulation runtime (Section 4.1.2). An alternative approach to fault activation can be executed with reconfiguration and reprogramming of the FPGAs, and is applied in the SFE and Cheng&Dai methods. However, these approaches are technology dependent as well as time consuming (Section 4.1.2 and 4.1.3) and thus do not satisfy the previously listed requirements for fault emulation.

Fault activation through use of the Fault Activator introduced in this work differs from the previous techniques in that neither a shift register nor reconfiguration of FPGAs is utilized. The Fault Activator arranges the fault locations in an addressable array. Thus, each addressed fault location, i.e. Fault Injector, is directly accessible, and in contrast to a shift register, processing a data sequence is not a prerequisite to fault injection. A two-dimensional address decoder can be utilized to address the fault location. Direct access to the fault location enables a faster fault injection. Furthermore, optimal mapping of the expanded circuit into the logic emulator can be attained using a symmetrical FPGA, which is also structured as a two-dimensional array. However, utilizing the two-dimensional address decoder as a Fault Activator leads to FPGA overhead.

The Fault Activator depicted in Fig. 4.15 addresses the Fault Injectors, which are then controlled by X- and Y-decoders. The address data $AD=\{A_1, ..., A_k, ..., A_n\}$ for the primary input of the Fault Activator is the binary representation of the address of each fault. In other words, the first fault is activated when the decoder input has the address 1, $\{A_1, A_2 ..., A_k, ..., A_n\}=\{1, 0, ..., 0, ...,0\}$. When the faults are arranged in a consecutive order, an $n+1$-bit counter can be used to control the address decoder and activate the Fault Injectors. The Fault Activator can be activated/deactivated with an additional input signal *MUX,* thereby controlling the fault and good emulation processes.

In order to generate the Fault Activator size the address area of the decoder must be calculated, which depends on the number of faults *Nf.* A decoder generator, DecGen, is developed in this work for the calculation and generation of the Fault Activator at gate level.

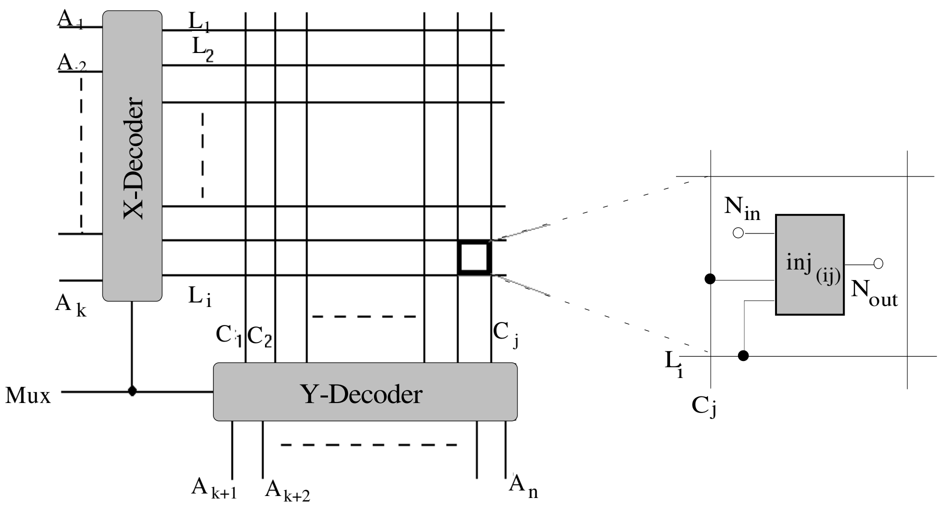


Fig. 4.15: Fault Activator

When calculating the decoder size the quadratic array structure of the Fault Activator must be considered for an optimized mapping of the expanded circuit into the array structure of the FPGAs. Therefore, the following calculation is suggested:

Given a number of faults Nf, the number of lines i in the fault activator is calculated as

$$i = \lceil \sqrt{Nf} \rceil \; . \tag{4.3}$$

The number of addressbits for the X-decoder $k+1$, which depends on the number of line decoders i in equation (4.3), is calculated as

$$k+1 = \lceil ld\ i \rceil \; . \tag{4.4}$$

The number of columns j in the fault activator is equal to i, $i=j$. The number of columns j is used to calculate the number of addressbits for the Y-decoder $n\text{-}k$ and is calculated as

$$n-k = \lceil ld\ \ j \rceil \; . \tag{4.5}$$

The address area of the X-decoder is 2^{k+1} and of the Y-decoder $2^{n\text{-}k}$. The total address area of the Fault Activator is then calculated as $2^{(k+1)+(n\text{-}k)}=2^{(n+1)}$. With equation (4.3), (4.4), and (4.5) the quadratic array structure of the Fault Activator can be generated.

4.2.5 Combinational Circuits

The process of fault emulation described in Sections 4.2.1 and 4.2.2 is implemented for the evaluation of combinational circuits. This section discusses fault injection for a combinational circuit in an FPGA. The function of a circuit can be mapped into several functional blocks of FPGAs, such as LUTs, i.e. combinational blocks in FPGAs. Only the combinational parts of the circuit are mapped into the LUTs. Since a LUT consists of four inputs and one output, only a boolean function dependent on four variables can be modeled. In order to enable fault injection, the function of the LUT is altered using a Fault Injector. Two inputs of the LUT are utilized to activate/deactivate the Fault Injector using the control signals L_i and C_j. Changing the function of a fault-free LUT to a faulty LUT involves setting the control signal $EN = L_i \cdot C_j = 1$.

An example of fault injection is illustrated in Fig. 4.16. Modeling a stuck-at-0 fault in net α, involves expanding the function of α with the corresponding function of the stuck-at-0 Fault Injector α (s-a-0). The circuit displayed below is mapped into a CLB with two LUTs. Gate GT_1 is mapped with four variables $\alpha = i_1 . i_2 . i_3 . i_4$ in LUT1 and gate GT_2 with two variables $o = i_5 + \alpha$ in LUT2.

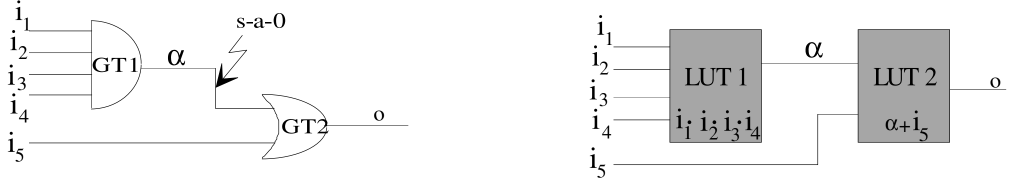


Fig. 4.16: Combinational circuit in a CLB

As a result of fault injection in net α, the function of LUT2 $o = i_5 + \alpha$ is affected such that net α is expanded, i.e. replaced, by the function $o = i_5 + \alpha \cdot \overline{L_i \cdot C_j}$, as shown in Fig. 4.17. In this case additional CLBs are not required for the mapping of the Fault Injector. Both the FES/1 and FES/2 approach can be utilized to detect the stuck-at-fault α (s-a-0). The Fault Injector controls the inputs of LUT2 with the values L_i=C_j=0 characterizing the fault-free circuit $B(v)$ in the good emulation process. Assume that the test vector set V contains a single test vector V={v}. An arbitrary test vector such as v=11110 is applied to the primary inputs of the circuit i_1, i_2, i_3, i_4, i_5. Storing the output value of LUT2 $o(v)$=1 from the good emulation precedes the fault emulation process, where the value of L_i=C_j=1 results in the faulty circuit $Bf(v)$ with the output value $o(v)$=0. If $B(v) \oplus Bf(v)$=1, the stuck-at-0 fault at α is detected by test vector v.

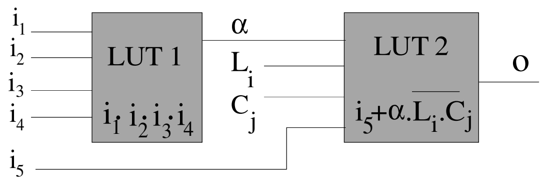


Fig. 4.17: Fault Injector s-a-0 in a CLB

4.2.6 Sequential Circuits

The basic characteristics that distinguishes a sequential circuit from a combinational circuit is that a particular set of outputs is dependent not only on the inputs, but also on the current state of the circuit. The state variables ξ_1, ξ_2, ..., ξ_k describe the previous states of the circuit's storage elements, which may vary with every clock period. A new state at clock period $t+r$,

r=1, ..., n, results from the state of clock period t and the input values, $\xi^{t+r} = \eta(I^t, \xi^t)$. The value η represents a function with which the dependency of the state variables ξ^{t+r} on the inputs I^t and the previous state is expressed in terms of ξ^t. In order to describe sequential circuits schematically, the state variables generated from η are delayed by time interval r and have feedbacks to the input of the combinational part of the circuit as illustrated in Fig. 4.18.

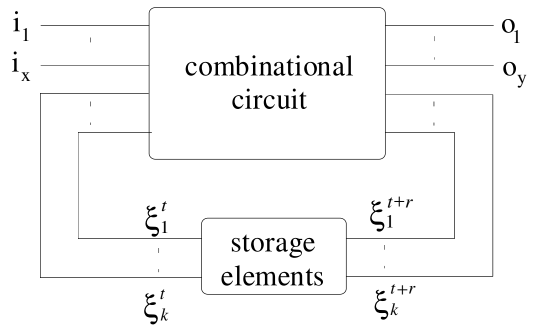


Fig. 4.18: Generic schematic of sequential circuits

The description of sequential circuits includes timing dependencies for registering the memory characteristics of the circuit. The primary outputs, O^t, of the circuit in Fig. 4.18 may be a function of the present state only, $O^t = F(\xi^t)$, or a function of the present states and the inputs, $O^t = F(I^t, \xi^t)$. Circuits meeting these conditions are known respectively as Moore machines [Moor56] and Mealy machines [Meal55].

There are two classes of sequential circuits, synchronous and asynchronous. The inputs of synchronous circuits are synchronized in predefined time periods. Sampling the input during each period, as in circuits where the storage elements (FFs) have a common clock, precedes the entering of a new state, which in turn produces the new output values. In contrast, asynchronous circuits change state in response to changes at the inputs of logic elements, such as circuits whose storage elements do not contain common clocks. The fault emulation approach for sequential circuits introduced here focuses on synchronous circuits.

One approach to the representation of a sequential circuit involves the use of a state transition table. The entries in the table are the next states reached by the circuit following any given change of the input. A sequential circuit should be developed implementing the characteristics in Table 4.2**.** The state table depicts the state transition of one primary input i and two storage elements with outputs Q_1 and Q_2, having 2^n=4 states. When the circuit is in the initial state A and receives the input i=1, a transition from state A to state D occurs producing an output Q_1=1, Q_2=1.

States of the circuit	Input of the circuit, *i* 0	1	Output of the storage elements *Q* , *Q*
A	A	D	0, 0
B	C	C	0, 1
C	B	A	1, 0
D	A	B	1, 1

Table 4.2: State transition table

Another approach to describing a sequential circuit uses a state transition diagram, a directed graph which represents the states of the circuit by nodes connected by directed edges to indicate the transition paths. Two commonly applied approaches are the Mealy model and the Moore model. Using the Mealy model of a state diagram, each transition path is labeled with the input affecting the transition and the resulting output state ξ^{t+r}. The actual state ξ^t is contained in the node. The Moore model also labels the paths with the inputs causing the transition, but differs in that the nodes contain the previous states ξ^t as well as the next states ξ^{t+r}. As a result, the output state ξ^{t+r} is solely a function of the previous state ξ^t.

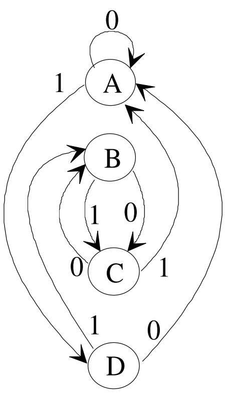


Fig. 4.19: State transition graph

The state transition graph in Fig. 4.19 is based on the Mealy model. Similar to the state table in Table 4.2, if the initial state is *A* with circuit input *i*=1, a transition path to *D* results. The initial state *A* is distinguished as the reset state in which sequential circuits are set before the beginning of operation. In practice, this state is easily attained by the activation of a *RESET* signal which sets the logical value of all outputs of the storage elements to logical 0. This *RESET* signal in the circuit is not always available, therefore, for the simulation of a sequential circuit, it is assumed that all outputs of the storage elements have the unknown value *U*.

Fault simulation for sequential circuits is executed in at least a 3-valued logic $\{0,1,U\}$. At the beginning of the simulation, all circuit nodes are in an unknown state and have therefore the logical value *U*. In contrast, the primary inputs of the circuit acquire either the defined logical value 0 or 1, dependent on the test vector. During the simulation of a sequential circuit the storage elements can prevent the internal signals from setting a defined value. Figure 4.20 displays a subcircuit exemplifying the storage element's inability to be initialized with a 3-valued logic.

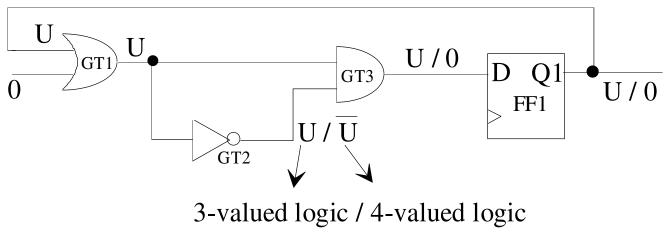


Fig. 4.20: Initialization of storage element with 3- and 4-valued logic

Here, the output of gate GT3 has the unknown value U, preventing the initialization of the storage element FF1 to a defined logical value. Initialization is possible, however, when the logical values are expanded to a 4-valued logic $\{0,1,U,\overline{U}\}$, the fourth value representing the negation of the unknown logical value *U*. It follows then, that the storage element FF1 can be initialized with the defined logical value 0.

At first it seems that the initialization problem can be solved by using the values *U* and $\overline{U}$ with the corresponding rules $U \cdot \overline{U} = 0$ and $U + \overline{U} = 1$. However, only when one storage element is set to *U* is this a correct solution. The use of *U* and $\overline{U}$ for more than one storage element may lead to incorrect results. A correct solution is to use several unknown values U_1, U_2, ...,U_n, one for each storage element, with $U_i \cdot \overline{U_i} = 0$ and $U_i + \overline{U_i} = 1$. When dealing with large circuits, this is a complex method using large boolean expressions of u_i variables.

The logical values in logic emulators are represented in terms of voltage levels, which are predefined for logic values. Therefore, logic emulators can model the circuit only with a 2-valued logic {0,1}. Due to this characteristic of logic emulators, the following concept of fault detection is introduced for fault emulation.

The evaluation process for sequential circuits involves the application of a 2-valued test vector sequence, i.e. test sequence $VS \in \{0,1\}$, as opposed to the single test vector *v* applied to combinational circuits. The test sequence consists of several test vectors, $VS=\{v_1, ..., v_i, ..., v_m\}$. The value of the circuit's output is a function of its initial state $IS \in \{0,1\}$, which may be the result of several individual test vectors. Generally, the initial state $IS=v_{Initial}$ is reached by one test vector $v_{Initial}$, such as the reset test vector, $IS=v_{RESET}$. Assume that VS is a test sequence for the detection of faults in a sequential circuit and $B(IS,VS)$ represents the function of a fault-free sequential circuit with test sequence VS and the initial state IS. Similarly, the function of a faulty circuit with the test sequence VS at the initial state IS can be represented as $Bf(IS,VS)$.

The complete evaluation process for fault detection is divided into two steps. The first involves the initialization of the fault-free and the faulty circuits. This step precedes the application of a test sequence VS. The fault f can be detected by a test sequence VS when, for some specific test vector v_i in the test sequence VS, $v_i \in VS$, the output sequence of the fault-free circuit $B(IS,v_i)$ differs from the output sequence of the faulty circuit $Bf(IS,v_i)$.

During circuit design consideration is given to an initialization by inserting a common *SET* and/or *RESET* signal to every storage element. Circuit initialization then requires only a single test vector, for example $IS=\{v_{RESET}\}$. Compared to fault emulation for combinational circuits, fault emulation for sequential circuits is more complicated. Using the 2-valued logic {1, 0} of the logic emulator and the definition of fault detection with a 2-valued logic, an example of fault emulation is given in Fig. 4.21.

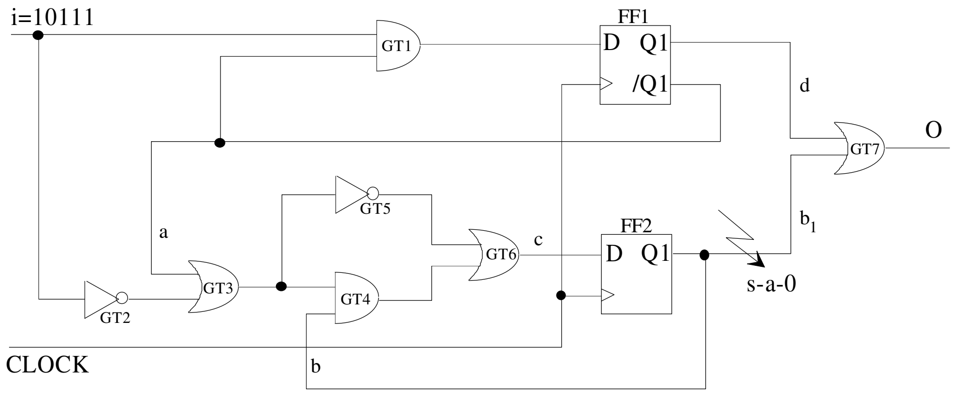


Fig. 4.21: Fault emulation for a sequential circuit

The circuit above with the state transition graph from Fig. 4.19 is mapped into two CLBs as shown in Fig. 4.22. Each CLB contains two combinational elements (LUTs) and two sequential elements (flip-flops) as depicted in the Fig. 4.22. In order to model a stuck-at-0 fault in net b_1, the corresponding Fault Injector is inserted, i.e. the corresponding function of LUT3 is expanded. Gates *GT*2, *GT*3, *GT*4, *GT*4 and *GT*6 are mapped into LUT2 with the function

$c=f_2(i, a, b)$, which is dependent on three variables. Gate $GT1$ with the function $e=f_1(i, a)$, dependent on two variables, is mapped into LUT1. Both flip-flops are mapped into the flip-flops of CLB1 and the last gate $GT7$ with the function $O= d+b_1$ is mapped into LUT3. As a result of fault injection in net b_1 the function of LUT3 $O = d + b_1 \cdot \overline{L_i \cdot C_j}$ is changed such that net b_1 is replaced by the function $b_1 \cdot \overline{L_i \cdot C_j}$.

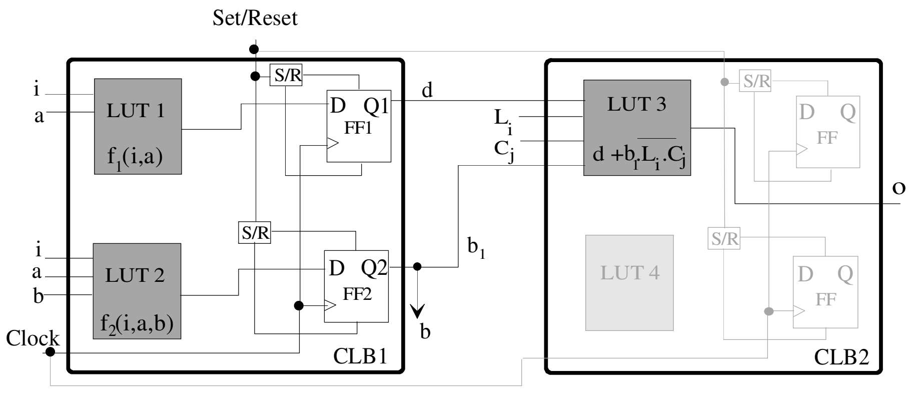


Fig. 4.22: Mapping of the sequential circuit into an FPGA

The first step of a good emulation is the initialization of the mapped circuit to a known state and is easily accomplished when all flip-flops have the same state. All flip-flops in the FPGAs can have common set and reset inputs and can therefore be initialized with a logical 1 or 0 using the initialization $IS=\{1\}$or $IS=\{0\}$. Thus, an initialization of individual storage elements with set and reset inputs is not possible without the addition of extra primary inputs for every set and reset input of each flip-flop in the circuit. Therefore, the initialization sequence must be calculated. However, the calculation of the initialization sequence is very complicated for large complex circuits. A technique is described at the end of this section, which allows for rapid initialization of the individual flip-flops at various states in the logic emulator.

Using the FES/1 approach a good emulation is executed by initializing the circuit by the initial vector $IS=\{0\}$, i.e. resetting all flip-flops so that the outputs of the flip-flops are $Q_1=Q_2=0$. In a good emulation the Fault Injector of stuck-at-fault b_1(s-a-0) is deactivated, $L_i=C_j=0$, resulting in the fault-free circuit $B(IS,VS)$. With the application of a test sequence VS to the primary input i of the circuit, the logical value of the circuit's primary output O is stored. The fault emulation process of FES/1 begins with the activation of the Fault Injector, i.e. $L_i=C_j=1$. Therefore, the function of LUT3 is changed, its output has the logical value 0, and the faulty circuit $Bf(IS,V_S)$ results. This faulty circuit is initialized by the same initial vector $IS=\{0\}$ from the good emulation, $Q_1=Q_2=0$. The stuck-at-0 fault at b_1 is detected if for some specific test vector v_i in the test sequence *VS*, $v_i \in VS$, the output sequence of the fault-free circuit $B(IS,v_i)$ differs from the output sequence of the faulty circuit $Bf(IS,v_i)$. At this point, the fault emulation process is interrupted and the next fault evaluated.

Table 4.3 shows the emulation results of the circuit in Fig. 4.22 including the output sequence of the fault-free and the faulty circuit with the stuck-at-0 fault b_1(*s-a*-0), obtained in response to the test sequence *VS*={1,0,1,1}.

Initial state Q_1, Q_2	Output sequences of Q_1, Q_2	Output *O* of fault-free circuit *B*	Output *O* of faulty circuit *Bf*
0, 0(A)		0	0
	1, 0(D)	1	1
	0, 0(A)	0	0
	1, 0(D)	1	1
	0, 1(B)	**1**	**0**
1, 0(D)		1	1
	0, 1(B)	**1**	**0**
	1, 1(C)	**1**	**0**
	0, 0(A)	1	1
	1, 0(D)	0	0

Table 4.3: Emulation results for initial states (A) and (D)

As indicated in Table 4.3, setting the common reset inputs of all storage elements to the logical value 0 initializes the circuit to a defined state $Q_1=\{0\}$, $Q_2=\{0\}$. The fault emulation process commences at the initial state *IS*={*A*}, leading to a transition to state *D* when the circuit's primary input is *i*=1. Every test vector $v_i \in VS$ causes a transition to the next state. The process is repeated until the emulation results of the fault-free circuit $B(v_4)$ differ from those of the faulty circuit $Bf(v_4)$ whereby the fourth test vector $v_4=1$ in the test sequence *VS* results in $B(IS,v_4) \oplus Bf(IS,v_4)=1$. At this point the fault emulation process is interrupted, as indicated by the last test vector of the test sequence in Table 4.3. Given the initial state *IS*={*D*}, the fault is detected by the second test vector $v_2=0$ in the test sequence *VS*.

Rapid Initialization of Individual Storage Elements

Various methods are available for the initialization of individual storage elements, one of which uses an initial sequence to set the circuit into a defined state. Generally, this approach is applied to circuits containing storage elements with reset and set inputs. A hardware-based approach can also be implemented to set the circuit into a defined state by setting and resetting the storage elements.

A hardware-based technique for the individual initialization of storage elements is presented in Fig. 4.23. All flip-flops in the FPGAs contain a reset and a set input, which can be expanded with additional logic for an individual initialization. Initializing the selected flip-flops to a defined state involves expanding the set input by a stuck-at-1 Fault Injector (Set-Fault Injector) for the initial state of logical value 1 and expanding the reset input by a stuck-at-0 Fault Injector (Reset-Fault Injector) for the initial state of logical value 0. The setting and resetting of the selected flip-flops involves setting the corresponding addresses of the Set-Fault Injectors and Reset-Fault Injectors in the Fault Activator. The initialization is divided in two phases;

- MUX=0, initialization of all flip-flops in a defined state
- MUX=1, initialization of selected flip-flops by setting the corresponding addresses

Figure 4.23 depicts a circuit with initialization state $IS=\{Q_1=0, Q_2=0, Q_3=1\}$. The first phase of the initialization process entails initializing all flip-flops in the FPGA with the common reset input with the reset signal $RESET$=0 and results in $Q_1=0$, $Q_2=0$, $Q_3=0$. The second phase completes the initialization process when the corresponding address of the Set-Fault Injector (Inj3) is set, leading to $Q_1=0$, $Q_2=0$, $Q_3=1$.

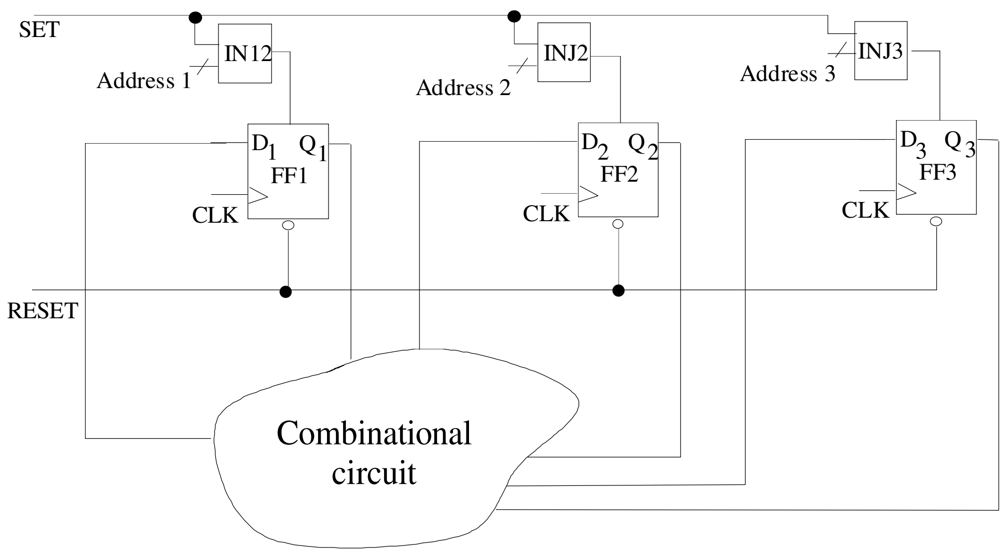


Fig. 4.23: Hardware initialization of individual storage elements

5. Node Assignment

The expansion of a circuit using Fault Injectors and a Fault Activator results in an overhead of FPGA resources in the logic emulator. The node assignment method described in the following chapter leads to an improved usage of FPGA-resources for fault emulation, hence a reduction of FPGA overhead [Seda98a]. Compilation, which is the process of partitioning, technology mapping, and placement and routing precedes mapping the expanded circuit into the logic emulator. An optimized partitioning and a routable placement is difficult to achieve due to the generation of additional connections between the fault locations by the control signals *L* and *C* of the Fault Injectors. Therefore, assigning the fault locations in the circuit to the Fault Injectors is an important aspect in the compilation process of the logic emulation.

5.1 Introduction

Beginning with a set of independent variables or parameters, an optimization problem generally includes conditions or restrictions that define acceptable values for the variables, and are defined as the constraints of the problem. Defining the cost function is another important aspect of the optimization problem. The set of values of the variables from which the cost function assumes an optimal value represents the solution to an optimization problem. Optimization involves maximizing or minimizing, e.g. maximizing yield or minimizing costs. The optimization problem can be expressed as:

$$\begin{aligned} &\min_{x \in \Re^n} && F(x) \\ &\text{subject to} && \phi_i(x) = 0, i = 1,2,\ldots,m'; \\ & && \phi_i(x) \geq 0, i = m'+1,\ldots,m. \end{aligned} \tag{5.1}$$

An optimization problem with *m* conditions can be expressed mathematically by a cost function *F*(*x*) and a constraint function $\phi_i(x)$. Depending on the type of cost and constraint function used, a differentiation is made between linear and non-linear optimization problems, the latter of which includes the quadratic optimization problem.

When all the constraints of equation (5.1) are satisfied, any point *x* is defined as feasible. A feasible region refers to the set of all feasible points, such as in a two-dimensional problem with the single constraint $x_1 + x_2 = 0$, representing a line. The feasible regions consists of all points of this line. Given the constraint $x_1^2 + x_2^2 \leq 1$, which represents a unit circle, the feasible regions consist of the interior as well as the boundary of this circle.

Before considering methods for solving optimization problems, the "solution" to a problem must first be defined for equation (5.1). Only feasible points may be an optimal solution. Furthermore, the relationship of a point x to its neighboring points defines the optimality of a point x, $F(x) \neq 0$. A set of points can be defined as the set of feasible points $\tilde{N}(x, \tilde{n})$ contained in a neighborhood $\tilde{n}$ of x where x indicates a feasible point for the problem in equation (5.1). In general, optimal feasible points can be defined as either a local or global minimum. The point x_{local} is a local minimum if $\tilde{n} > 0$ such that $F(x)$ is defined in $\tilde{N}(x_{local}, \tilde{n})$, and $F(x_{local}) < F(x)$ for all $x \in \tilde{N}(x_{local}, \tilde{n})$, $x \neq x_{local}$. In some applications it may be necessary to find the feasible point at which $F(x)$ assumes its minimal value. This point is referred to as the global minimum. The point x_{global} is a global minimum if $F(x_{global}) < F(x)$, $x_{global} \neq x_{local}$.

A minimization problem is assumed and, as shown in Fig. 5.1, an iterative algorithm examines the set of neighboring points $\tilde{N}(x, \tilde{n})$ for a minimum beginning at a initial state $x_{initial}$. A set of neighboring points $\tilde{N}(x, \tilde{n})$ of point $x_{initial}$ is reached after a minor alteration to point $x_{initial}$, which represents a cost calculated from equation (5.1). If the costs for the set of neighboring points $\tilde{N}(x, \tilde{n})$ are lower than a feasible point such as $x_{initial}$ in Fig. 5.1, then the algorithm has converged to a local optimum, such as local minimum x_{local}.

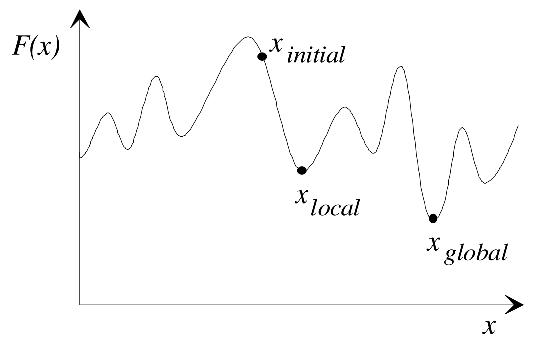


Fig. 5.1: Local versus global optima

The cost curve presented here is non-convex [Papa91] due to its multiple minima. Finding the global minimum x_{global} requires "climbing the hill" at the local minimum x_{local}. If an algorithm accepts only inferior costs it will not deviate from the local optimum x_{local}. As a result, the global minimum x_{global} can not be found.

In general optimization problems the feasible region includes x, with some of the variables restricted to being members of a set of values for minimizing the cost function. Many practical problems occur in which some of the variables are restricted to being members of a finite set of values. This type of limitation defines the combinatorial optimization problem. An example of such as the number of journeys made by a traveling salesman. Given the classical example of a traveling salesman, the problem consists of an number of cities $C=\{c_1,c_2,c_3,...,c_m\}$ and distances $d(c_i,c_j)$ between each pair of cities (c_i,c_j), where each city is visited only once. The solution to the problem is formed by the permutation $\{c_{\pi\ (1)},c_{\pi\ (2)},c_{\pi\ (3)},...,c_{\pi\ (m)}\}$ the cost function of which is given by the following equation:

$$F(\pi) = \sum_{i=1}^{m} d_{c_{\pi(i)},c_{\pi(i \bmod m)+1}} \tag{5.2}$$

5.2 Placement and Quadratic Assignment Problems

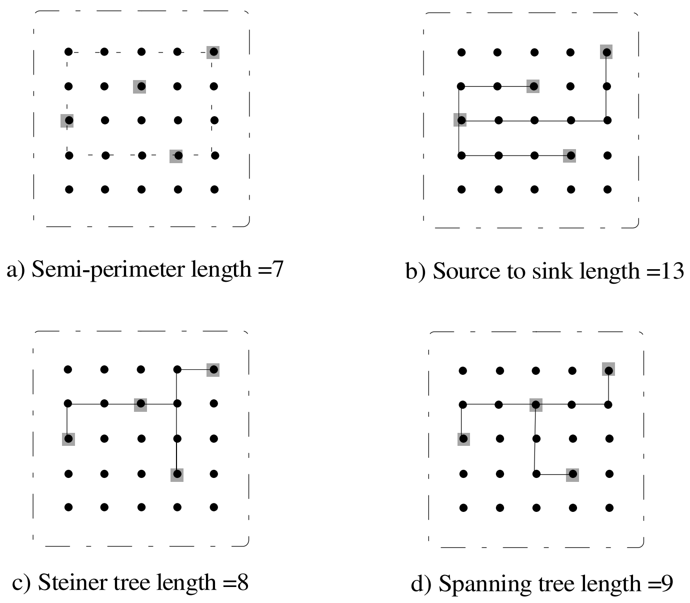
The placement problem can be defined as an optimization problem, which, for instance, involves optimizing the connection-cost between modules. The objective of connection-cost optimization is to minimize the amount of wiring required for the placement process. One of the main hindrances to an optimized placement is the difficulty in estimating the wiring requirements of subsequent routing phases. The wiring estimates included in the cost function for the placement may deviate from the wiring requirements during the ensuing routing phase. Consequently, the proper selection of the cost function minimized in placement is of utmost importance. Some of the prevailing cost functions are minimize maximum cut, and minimize maximum density, as well as minimize total wire length, a frequently applied cost function [SaYo95b] described in detail in this section.

Some commonly applied techniques [SaYo95b] for estimating the wirelength required by a given placement are semi-perimeter, source to sink, Steiner tree, and spanning tree (Fig. 5.2). The speed with which estimation is carried out is an important aspect of the performance of the placement algorithm. Hence, a good method of rapid estimation is key to any placement algorithm. When estimating total wirelength it is assumed that routing follows the Manhattan geometric model, running either horizontally or vertically. To connect module i to module j, the Manhattan length of the interconnection is

$$d_{lk(i)lk(j)} := \left|lk(i)^x - lk(j)^x\right| + \left|lk(i)^y - lk(j)^y\right| \tag{5.3}$$

Here, $lk(i) = [lk(i)^x, lk(i)^y]$ and $lk(j) = [lk(j)^x, lk(j)^y]$ are the locations of modules i and j on the coordinates x and y separated by the Manhattan distance $d_{lk(i)lk(j)}$.

One of the most widely used approximation methods for estimating the wirelength of a net is the semi-perimeter method, which attempts to find the smallest bounding rectangle enclosing all pins and nets. Half the perimeter of this bounding rectangle is the estimated wirelength of the interconnects. Wirelength is underestimated when the wiring area is heavily congested.


a) Semi-perimeter length =7 b) Source to sink length =13

c) Steiner tree length =8 d) Spanning tree length =9

Fig. 5.2: Wirelength estimation methods

Using the Steiner tree method a wire can branch from any point along its length to connect to other pins of the net. The problem of finding the optimal branching point, i.e. the minimum Steiner tree, is NP-complete [Leng90]. In contrast to the Steiner tree, a minimum spanning tree permits branching only at the module locations.

In a source to sink connection the output of a module is assumed to be connected to all the other module inputs by separate wires in a star configuration. This easily implemented method results in extremely long wirelength estimations. While providing a reliable approximation for a heavily congested wiring area, this type of connection is seldomly used for the estimation of wirelength in a lightly congested wiring area.

Formulation of placement as QAP

The quadratic assignment problem (QAP) is related to the placement problem and, beginning with the definition of the quadratic function, can be illustrated as follows:

A function F is quadratic when it has the matrix form in equation (5.4) for a matrix G, vector c, and scalar α. Multiplication by 1/2 is included in the quadratic term to avoid the occurrence of a factor of two in the derivatives. The matrix G is a Hessian matrix of F and is defined as a symmetric matrix.

$$F(\tilde{a}) = \frac{1}{2}\tilde{a}^T G\tilde{a} + c^T\tilde{a} + \alpha \tag{5.4}$$

The quadratic assignment problem QAP(E,H) with the two nxn matrices $E=(e_{ij})$, $H=(h_{ij})$ and the set $\{1,2,...,n\}$, can be stated as

$$\min_{\pi \in S_n} \hat{F}(E,H) = \sum_{i=1}^{n}\sum_{j=1}^{n} h_{ij} e_{\pi(i)\pi(j)} \tag{5.5}$$

An optimal solution to QAP(E,H) refers to a permutation $\pi_{opt} \in S_n$ which minimizes the cost function $\hat{F}(E,H)$ over S_n (Equation (5.5)). S_n is the set of permutations of $\{1,2,...,n\}$. Given a permutation $\pi \in S_n$ and an nxn matrix $E=(e_{ij})$, the matrix $E^{\pi} = (e^{\pi}_{ij})$ is obtained from E by permutating its rows and columns according to permutation π, i.e. $e^{\pi}_{ij} = e_{\pi(i)\pi(j)}$, for $1 \le i,j \le n$. The size of QAP(E,H) is determined by the size n of the coefficient matrices E and H.

An alternative formulation of the QAP was presented by Koopmans and Beckmann [KoBe57]. Here the relation between the set of permutations S_n and the set of all nxn permutation matrices Π_n is defined when $X = (x_{ij})$ is an nxn matrix. If the entries x_{ij} fulfill the following conditions in equation (5.6), then X is called a permutation matrix.

$$\begin{aligned} &\sum_{i=1}^{n} x_{ij} = 1, \qquad 1 \le j \le n \\ &\sum_{j=1}^{n} x_{ij} = 1, \qquad 1 \le i \le n \\ &x_{ij} \in \{0,1\}, \qquad 1 \le i,j \le n \end{aligned} \tag{5.6}$$

The relation described above is attained by associating a permutation $\pi_{\tilde{A}} \in S_n$ to each permutation matrix $X = (\mathrm{x}_{ij}) \in \Pi_n$, where $\pi_X(i)=j$ if and only if $x_{ij} = 1$. Thus, on the set of permutation matrices, QAP(E,H) is equivalent to the following minimization problem.

$$\begin{aligned} \text{min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} e_{ij}h_{kl}x_{ik}x_{jl} \\ \text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1, \qquad 1 \le j \le n \\ & \sum_{j=1}^{n} x_{ij} = 1, \qquad 1 \le i \le n \\ & x_{ij} \in \{0,1\}, \qquad 1 \le i,j \le n \end{aligned} \tag{5.7}$$

The similarity of the problems in equation (5.5) and (5.7) becomes more evident in the context of the assignment of two sets of elements. In other words, $x_{ij} = 1$ when element *i* is assigned to element *j*. Otherwise, $x_{ij} = 0$. The constraint that each element *i* should be assigned to exactly one element *j* formalizes the restrictions in (5.7). Conversely, to each element *j* should be assigned exactly one element *i*.

A simplified placement problem can be described as a QAP, which, when introduced in 1961 by Steinberg [Stein61], was termed "the backboard wiring problem". In 1972 Hanan and Kurtzberg [HaKu72] reevaluated the problem. The task here involves the placement of a quantity of modules on a board where each module pair is connected by a number of wires. The objective is to find an optimal placement of modules on the board so as to minimize the length of the connecting wires.

Given a set of modules $MD=\{md_1, md_2, ..., md_n\}$ and a set of signals $SG= \{sg_1, sg_2, ..., sg_k\}$, each module $md_r \in MD$ is associated with a set of signals SG_{md_r} where $SG_{md_r} \subseteq SG$. In the same manner, each signal $sg_r \in SG$ is associated with a set of modules MD_{sg_r}, where $MD_{sg_r} = \{md_s \mid sg_r \in SG_{md_s}\}$. $LK=\{lk_1, lk_2, ..., lk_n\}$ indicates a set of locations. The placement problem consists of assigning each $md_r \in MD$ to a unique location lk_s for the optimization of a cost function. The assignment of *n* modules *MD* to *n* positions *LK* on a board can be expressed mathematically by a permutation π of $\{1,2,...,n\}$. Given the number of signals connecting two modules *i* and *j* represented as w_{ij} and the distance between the two locations *k* and *l* on the board expressed as d_{kl}, $1 \le k, l \le n$, the wirelength needed to connect the modules *i* at $lk(i)$ and *j* at $lk(j)$ is given by $w_{ij}d_{lk(i)lk(j)}$. Furthermore, the total wirelength necessary to connect all module pairs is equal to $\hat{F}(W,D)$, where $W=(w_{ij})$ and $D=(d_{kl})$. Thus, determining the optimal assignment that minimizes total wirelength is equivalent to solving QAP(*E,H*).

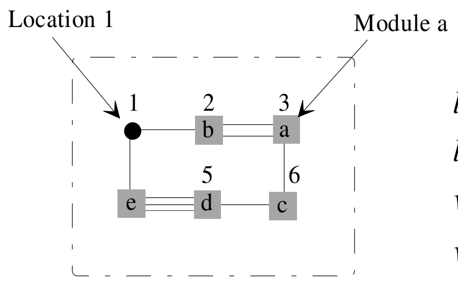
$$\min_{\pi \in S_n} \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} d_{lk(i)lk(j)} \tag{5.8}$$

An effective method for the elimination of long wires is to punish their use by a disproportionate factor. This is possible when the distance between the modules *i* and *j* in equation (5.3) is calculated quadratically and results in the cost function

$$\hat{F}(lk) = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(lk(i) - lk(j))^2 \tag{5.9}$$

Since *w* is symmetric ($w_{ij}=w_{ji}$) the quadratic function $\hat{F} = lk^T G lk$ results, where $G=T-W$ and T is a diagonal matrix of the row sum of W.

Figure 5.3 depicts the placement of five modules MD=$\{a,b,c,d,e\}$ to a set of locations LK=$\{lk_1, ..., lk_6\}$. The goal of placement consists of determining the function $\hat{F}(W,D)$ so that the total wirelength is minimized. Using equation (5.9) the wirelength of the given placement in Fig. 5.3 amounts to $\hat{F}(W,D)$=19.



$$lk(a) = 3,\ lk(b) = 2,\ lk(c) = 6$$
$$lk(d) = 5,\ lk(e) = 4$$
$$w_{a,b} = 2,\ w_{a,c} = 1,\ w_{b,d} = 1,$$
$$w_{c,d} = 1,\ w_{d,e} = 3$$

Fig.5.3: An example of the placement problem

When applied to the placement problem, Koopmans and Beckmann's definition of QAP (Equation (5.7)) formalizes the constraint that each module be assigned to exactly one location. Thus, module *i* is assigned to location *k*, $lk(i)=k$, and module *j* is assigned to location *l*, $lk(\mathrm{j})=l$, when

$$x_{ik} = \begin{cases} 1 & \text{if} \quad lk(i)=k \\ 0 & \text{else} \end{cases}$$
$$x_{jl} = \begin{cases} 1 & \text{if} \quad lk(j)=l \\ 0 & \text{else} \end{cases} \tag{5.10}$$

5.3 Fault Activator and Node Assignment

In the previous section the placement problem is described as an assignment problem. According to the definition of the assignment problem (Equations (5.5) and (5.7)), a specific logic element in a circuit can be assigned to several equivalent locations, for example to CLBs in an FPGA. Here, the task of the assignment problem is to minimize the cost function. Given this general description of the assignment problem, it is evident that an association exists between the node assignment problem and the quadratic assignment problem. The following description of symmetrical FPGAs is necessary before defining the node assignment problem.

A model of the general structure of a symmetrical FPGA [Rose91] is shown in Fig. 5.4, which depicts a conceptual structure of a typical FPGA consisting of a two-dimensional array of logic blocks (CLBs) connected by general interconnection resources. The routing channels are composed of wire segments, straight sections of wires varying in length that are used to form a part of a connection and programmable switch, which consist of two types of blocks, namely connection *C* blocks and switch *SW* blocks. The *C* blocks contain routing switches for connecting the logic block pins to the wire segments, whereas the switches housed in the *SW* blocks permit the connection from one wire segment to another. Logic circuits are implemented in the FPGA by mapping the logic into separate logic blocks and then interconnecting the blocks as necessary using programmable switches.

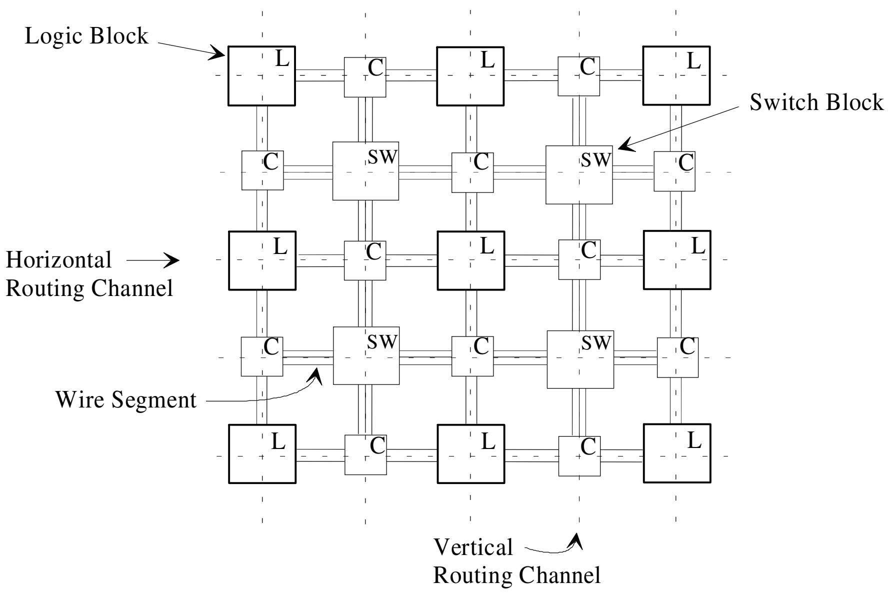


Fig. 5.4: Model of a symmetrical FPGA

The node assignment optimization problem is a simplified model of the placement problem in a symmetrical field programmable gate array [Seda97c]. The symmetrical FPGAs have a regular array of CLBs, similar to the Fault Activator with its regular array structure. The following model is defined for the node assignment optimization problem and assumes that the Fault Activator including the Fault Injectors is mapped into the FPGA. An additional assumption is that at least one Fault Injector can be mapped into each CLB, such that the array structure of the Fault Activator corresponds to the array structure of the FPGA as depicted in Fig. 5.5.

The fault location denotes a node in a circuit graph. This process of assignment of nodes, i.e. node assignment, has the objective of optimized mapping of the expanded circuit into the logic emulator.

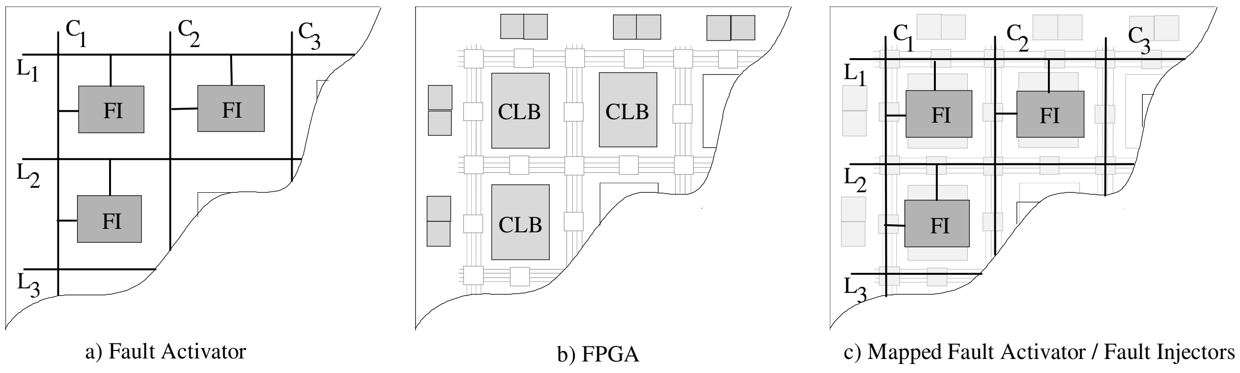


Fig. 5.5: Node assignment model

A circuit expanded by additional functions for fault injection and fault activation requires additional FPGA resources. Consequently, the task of node assignment is to minimize this overhead.

The compilation process in the logic emulator involves partitioning, technology mapping, and placement and routing, as described in Chapter 2. Discussed in the following sections are the consequences of a non-optimized node assignment on the compilation process. This results in a non-optimized partitioning, technology mapping, placement and routing of the expanded circuit, which leads to an FPGA-overhead or to the inability to map the expanded circuit into the logic emulator.

5.3.1 Partitioning of an Expanded Circuit

Generally, a single large circuit will not fit into a single FPGA and must therefore be divided into partitions that can then be fit to several FPGAs. As a result, the signals between FPGAs must be interconnected. However, connections between FPGAs are problematic since the amount of *I/O* resources on the FPGAs tends to be exhausted long before the CLBs are used up. Thus, the main objective of the partitioner becomes minimizing cutsize, i.e. connections between partitions.

The expanded circuit is modeled by an extra node graph [AlKa95]. The method of modeling a circuit by a graph with the use of extra nodes is based on the assumption of a fanout in an interconnect that connects more than two logic elements. Using the extra node model, a fanout is represented in a circuit graph as a node, which is termed an extra node. All interconnected logic elements are connected by the extra nodes through an edge. A connection of two logic elements does not constitute a fanout, however, an extra node is also generated. It is necessary to differentiate between three types of nodes in an extra node graph of a circuit. These nodes include those for the primary inputs, extra nodes for interconnections between logic elements, and instance nodes for the logic elements of the circuit. The extra node graph $G(U_{I/O}, U_N, U_I, E)$ can be described by a set of *I/O* nodes $U_{I/O}$, a set of extra nodes U_N, a set of instance nodes U_I, and a set of edges E. The circuit in Fig. 5.6 is modeled as an extra node graph with the logic elements represented by instance nodes U_I (square nodes) and the nets by extra nodes U_N (circle nodes).

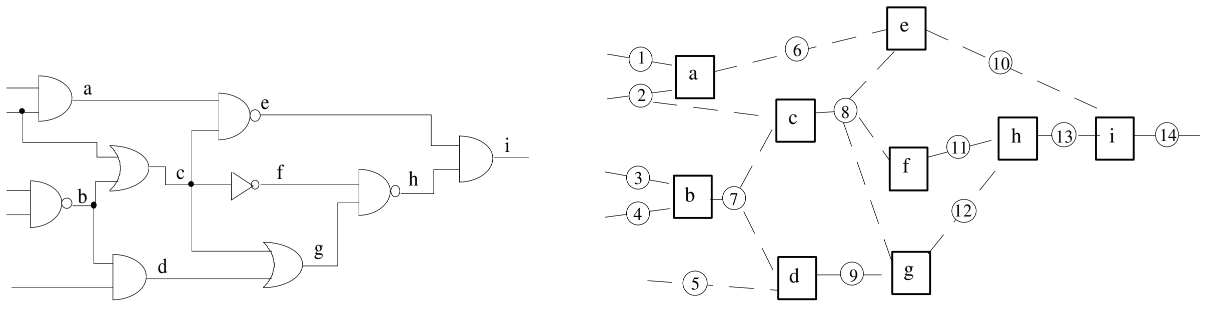


Fig. 5.6: Circuit without Fault Injectors (left) modeled as extra node graph (right)

The graph partitioning problem for an FPGA-based logic emulator is modeled by the extra node graph $G(U_{I/O}, U_N, U_I, E)$ in which each node $u_I \in U_I$ has a size $sz(u_I)$ and each edge $e \in E$ has a weight $w(e)$. The problem here becomes dividing the set of instance nodes U_I into k subsets U_{I1}, U_{I2}, ... ,U_{Ik}, such that a cost function is optimized. The size $sz(u_I)$ of instance node u_I represents the area of the corresponding circuit elements. If a circuit is divided into k subcircuits, the graph is partitioned into k subgraphs $G(j)(U_{I/O}(j), U_N(j), U_I(j), E(j))$, j =1, 2, ... ,

k, $G(j) \in G$, $G(j) \neq \emptyset$, $\bigcup_{j=1}^{k} G(j) = G$ and $|G(j)| \leq sz(u_I)$. The interconnections of the circuit elements are modeled as extra nodes U_N in this graph, and are used to divide the graph in subgraphs $G(j)$. The number of extra nodes to cut the graph is represented by the cutsize Φ. With an iterative improvement algorithm such as Kernighan-Lin [KeLi70], which addresses the two-way partitioning problem [ChWe91], the cutsize Φ between two partitions can be minimized.

The circuit in Fig. 5.7 is partitioned into two subcircuits, with primary inputs I_1, I_2, I_3, I_4, and I_5 and gates *a, b, c, d,* and *g* in one subcircuit and primary output O_1 and gates *e, f, h,* and *i* in the other. Two subgraphs, G_1 and G_2, result from the partition. Subgraph G_1 includes subset

$U_{I1} = \{u_{I(a)}, u_{I(b)}, u_{I(c)}, u_{I(d)}, u_{I(g)}\}$ and subgraph G_2 includes subset $U_{I2} = \{u_{I(e)}, u_{I(f)}, u_{I(h)}, u_{I(i)}\}$ resulting in sizes $sz(u_{I,G_1})$=5 and $sz(u_{I,G_2})$=4. Additionally subgraph G_1 includes I/O nodes $u_{I/O(I1)}$, $u_{I/O(I2)}$, $u_{I/O(I3)}$, $u_{I/O(I4)}$, and $u_{I/O(I5)}$, as well as extra nodes $u_{N(1)}$ to $u_{N(5)}$, $u_{N(7)}$, and $u_{N(9)}$. The subgraph G_2 includes the I/O node $u_{I/O(O1)}$ and the extra nodes $u_{N(10)}$, $u_{N(11)}$, $u_{N(13)}$, and $u_{N(14)}$. The three extra nodes $u_{N(6)}$, $u_{N(8)}$, and $u_{N(12)}$ are cut, thus Φ=3.

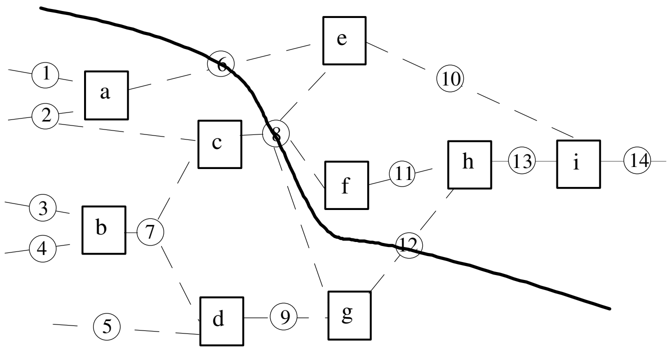


Fig. 5.7: Circuit partitioning of expanded circuit

Assigning the fault locations in the circuit to the Fault Injectors is an important aspect of an optimized partitioning of the expanded circuit in the compilation process of the logic emulation. An illustration of node assignment in the partitioning problem is followed by a discussion of the non-optimized and optimized node assignment of the fault location to the Fault Injectors. First, a non-optimized assignment is described. Second, the optimized node assignment is discussed as well as its influence on partitioning. The circuit from Fig. 5.6 shown in Fig. 5.8 as an expanded circuit with fault injection, is modeled with the stuck-at-0 fault (s-a-0) at the output of each gate.

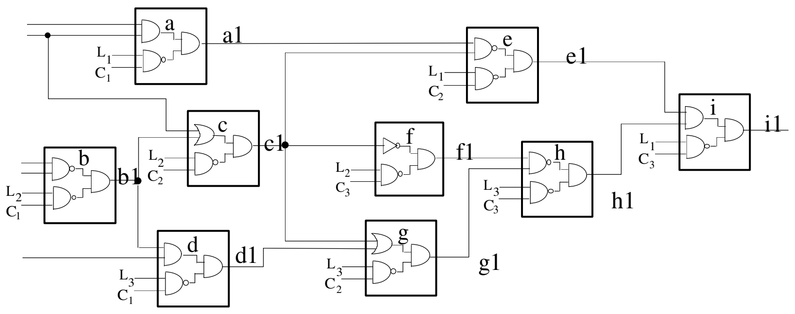


Fig. 5.8: Circuit with Fault Injectors

In the extra node graph of the expanded circuit every instance node is expanded by a Fault Injector and builds a new instance node, referred to here as an expanded instance node $u_{I(exp)}$. In the example below (Fig. 5.8) the Fault Injector is at the output of gate *a*. Hence, the function of gates in the circuit, i.e. instance node $u_{I(a)}$ in the extra node graph, generates a new expanded instance node $u_{I(a1)}$ with the function of a Fault Injector, thus the function of the expanded instance node is $a1 = \overline{L_1 \cdot C_1} \cdot a$.

The expanded circuit is modeled in Fig. 5.9 as an extra node graph $G(U_{I/O}, U_N, U_I, E)$ with the following sets:

I/O nodes: $U_{I/O}=\{u_{I/O(I1)}, u_{I/O(I2)}, u_{I/O(I3)}, u_{I/O(I4)}, u_{I/O(I5)}, u_{I/O(O1)}\}$,
extra nodes: $U_N=\{u_{N(1)}, ..., u_{N(14)}, u_{N(L1)}, u_{N(L2)}, u_{N(L3)}, u_{N(C1)}, u_{N(C2)}, u_{N(C3)}\}$.

The instance node $u_I \in U_I$ is expanded with a Fault Injector generating a new instance node $u_{I(exp)} \in U_{I(exp)}$, $U_{I(exp)}=\{u_{I(a1)}, u_{I(b1)}, u_{I(c1)}, u_{I(d1)}, u_{I(e1)}, u_{I(f1)}, u_{I(g1)}, u_{I(h1)}, u_{I(i1)}\}$.

The partitioning problem becomes one of dividing the set of instance nodes U_I into k=2 subsets U_{I1}, U_{I2}. In this example an optimized assignment of fault locations to the Fault Injectors results in the bipartitioning of the expanded circuit with a cutsize of Φ=7. Subgraph G_1 includes subset $U_{I1}=\{u_{I(a1)}, u_{I(b1)}, u_{I(c1)}, u_{I(d1)}, u_{I(g1)}\}$ while subgraph G_2 includes subset $U_{I2}=\{u_{I(e1)}, u_{I(f1)}, u_{I(h1)}, u_{I(i1)}\}$.

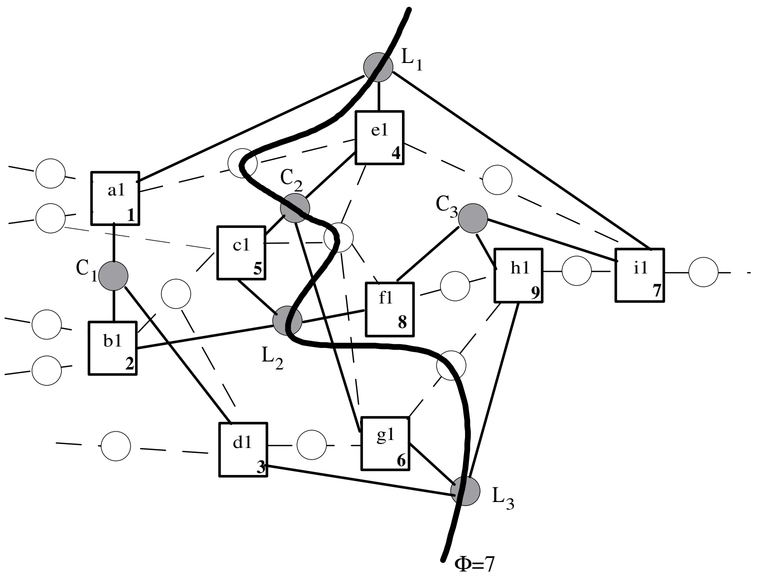


Fig. 5.9: Bipartition with optimized node assignment and cutsize Φ=7

Compared to an optimized node assignment, the cutsize resulting from the bipartitioning of the expanded circuit is always higher in a non-optimized node assignment of fault locations. The example below depicts a non-optimized node assignment of fault locations with cutsize Φ=9 through the extra nodes $u_{N(6)}$, $u_{N(8)}$, $u_{N(12)}$, $u_{N(L1)}$, $u_{N(L2)}$, $u_{N(L3)}$, $u_{N(C1)}$, $u_{N(C2)}$, $u_{N(C3)}$. The extra nodes cut in this process includes those from the circuit $u_{N(6)}$, $u_{N(8)}$, $u_{N(12)}$ and those from the interconnection of the control signals L_l and C_c of Fault Injectors, $u_{N(L1)}$, $u_{N(L2)}$, $u_{N(L3)}$, $u_{N(C1)}$, $u_{N(C2)}$, $u_{N(C3)}$. An optimized partitioning of this expanded circuit with $\Phi<9$ is difficult to attain due to the interconnection of control signals L_l and C_c.

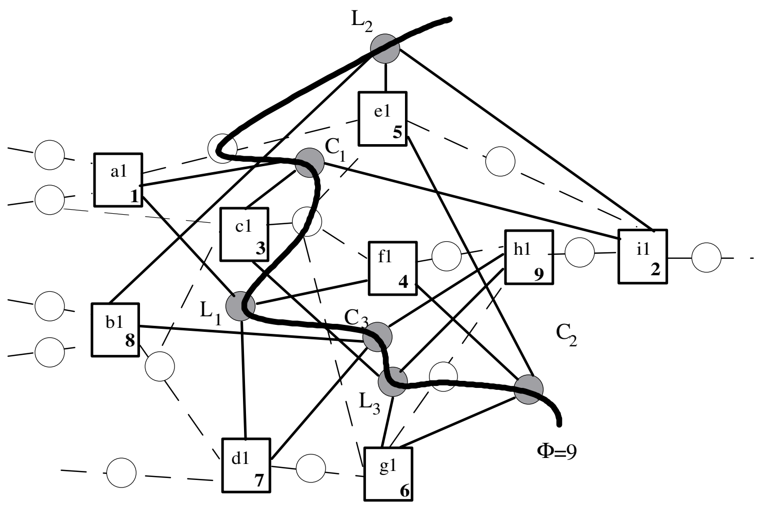


Fig. 5.10: Bipartition with non-optimized node assignment and cutsize Φ=9

In the optimized node assignment in Fig. 5.9, seven extra nodes are cut by the partitions. In this case the circuit is partitioned without cutting the extra nodes $u_{N(C1)}$, $u_{N(C3)}$ of the control signals C_1 and C_3. In contrast, with the non-optimized node assignment shown in Fig. 5.10 bipartitioning is not attainable without cutting all extra nodes of the control signals $u_{N(L_l)}$, $u_{N(C_c)}$. As a result of a non-optimized node assignment, FPGA usage increases and in many cases expanded circuits can not be mapped in the logic emulator. A solution to this problem, similar to the Cheng&Dai approach (Section 4.1.2), is found by dividing the set of faults in order to expand the circuit with only a subset of the faultlist. However, this solution leads to the compilation and emulation of several expanded circuits as well as an increase in fault emulation runtime as described in Section 4.1.2. Recall that the amount of *I/O* resources of the FPGAs tends to be exhausted long before the CLBs are used up. Thus, an optimized assignment of fault locations to Fault Injectors is essential for an optimized partitioning which minimizes cutsize Φ.

5.3.2 Technology Mapping of an Expanded Circuit

The second step in the mapping process, technology mapping converts the input netlist at gate or RT-level into FPGA logic blocks. The quantity of inputs of the functions may not exactly correspond to the those of the look-up table, which is carrying out those functions in the FPGAs. In the case of too many inputs, the logic elements must be split, or decomposed, into various smaller functions that can then be used by the LUTs. If, on the other hand, the number of inputs is smaller than those of the LUTs several logic gates are combined to build one LUT. Technology mapping reorganizes the logic for an optimal fit in the logic blocks of the FPGA. Various algorithms are available for the optimized technology mapping of circuits for FPGA implementation. While some methods focus on LUT count [Fran91a] [Fran91b], others seek to facilitate routing within the FPGA [ScKo94] [BhHi92] [ChWo94].

The logical functions of Fault Injectors are dependent on either three (s-a-1 or s-a-0) or four (s-a-1/0) variables. Given a 4-input LUT, the function of each expanded node can be mapped in a LUT, as illustrated in Fig. 5.11. Consequently, five CLBs are necessary to map the complete circuit. For instance, CLB1 consists of two functions $a1 = (I_1 + I_2) \cdot (\overline{L_1 \cdot C_1})$ and $c1 = (I_2 + b1) \cdot (\overline{L_2 \cdot C_2})$, each of which depends on four variables.

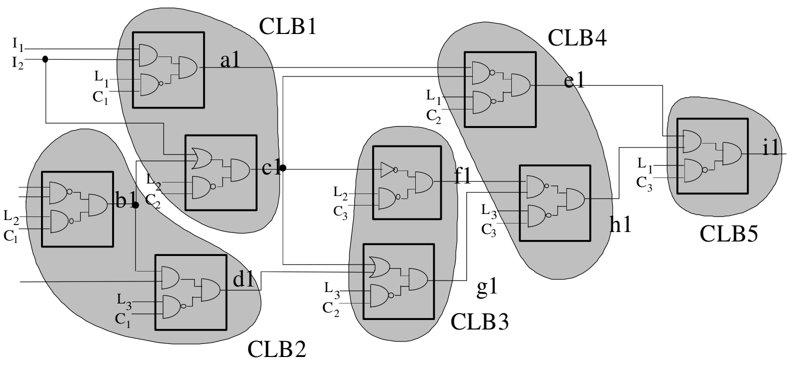


Fig. 5.11: Technology mapping of an expanded circuit

5.3.3 Placement and Routing of an Expanded Circuit

Given a set of logical functions generated by the technology mapping process, the appropriate locations (logic blocks) for the logical functions are those which minimize given cost functions, subject to certain constraints imposed by the implementation process. The constraints could include the requirement that cells fit into a predefined or prefabricated area, such as CLBs and routing resources in an FPGA. A difference between placement and routing for a single FPGA and for an FPGA system is that in an FPGA system signal delay resulting from the routing resources in the emulator must be minimal. In the case of a single FPGA, mapping quality, here the optimized usage of FPGA resources, has a high priority. Due to the interconnections in an FPGA system, changes to part of the circuit can affect a multitude of FPGAs.

The placement process for an FPGA involves assigning a set of logic functions formed by technology mapping to a set of logic blocks in the FPGA and can greatly influence the mapping quality and performance of the FPGA. Long routing paths between the logic blocks result in signal delay and the usage of additional resources. Therefore, the objective of placement becomes minimizing wirelength in the FPGA [SeLe87]. A placement is acceptable if routing between all logic blocks can be achieved with the available routing resources. Before an acceptable placement can be found, different solutions must be compared and evaluated. Performing actual routing for this purpose is impractical, because the placement problem is an NP-complete problem. Therefore, estimates are used. A technique [SaYo95b] is presented below for estimating the routability of a given placement.

A measure for estimating the routability of an FPGA placement ρ is the density $\tilde{d}(\rho)$. Within a given placement ρ, the number of nets that must pass through each side ε_i of a switch block can be estimated with $\eta\rho(\varepsilon_i)$. If $\psi\rho(\varepsilon_i)$ represents the capacity of the switch block side ε_i, then the density of the switch block side ε_i can be defined as:

$$\tilde{d}\rho(\varepsilon_i) = \frac{\eta\rho(\varepsilon_i)}{\psi\rho(\varepsilon_i)} \tag{5.11}$$

A measure of the routability of the placement, where the maximum is taken over all switch block sides ε_i of the routing resources, is given by equation (5.12) and must be $\tilde{d}(\rho) \le 1$:

$$\tilde{d}(\rho) = \max_i \left[\tilde{d}\rho(\varepsilon_i) \right] \tag{5.12}$$

Long routing paths between the logic blocks in the expanded circuit can be avoided by applying the technique of node assignment in the FPGA. The following examples of optimized and non-optimized node assignment illustrate the placement and routing problems in an FPGA.

Figure 5.12 illustrates the placement of the expanded circuit in the FPGA with non-optimized node assignment (Fig. 5.10). The portion of the circuit interconnections depicted is sufficient to explain the placement problem for an expanded circuit. In order to determine the routability of this placement the cost function (Equation (5.12)) defined for the minimization of maximum density can be applied. Given that the capacity of each switch block side amounts to four bidirectional inputs and outputs ($\psi\rho(\varepsilon_i)$ =4). Inasmuch as *max*[$\eta\rho(\varepsilon_i)$]=5, $\tilde{d}(\rho) > 1$ and the placement in Fig. 5.12 is not routable .

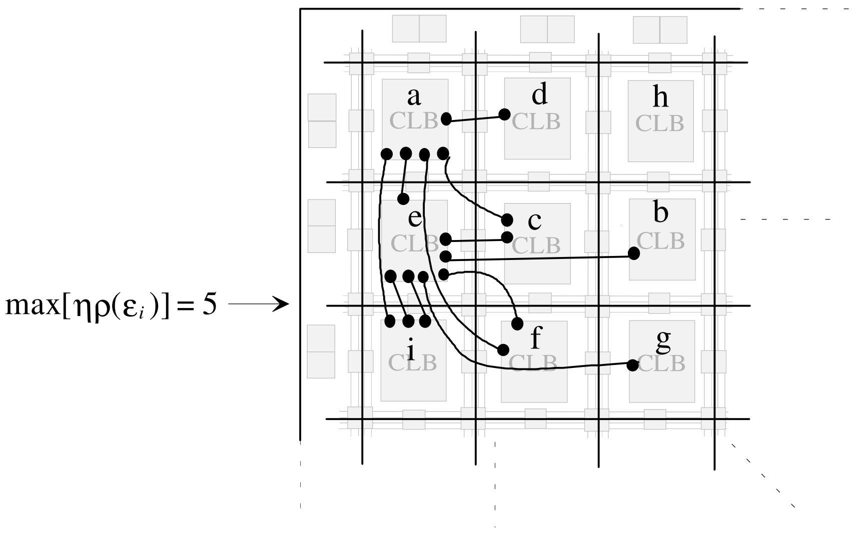


Fig. 5.12: Placement of circuit with non-optimized node assignment in an FPGA

A routable placement is difficult to achieve with non-optimized node assignment due to the generation of additional connections between the fault locations by the control signals L and C of the Fault Injectors. As depicted in Fig. 5.6, before circuit expansion a connection between nodes e and b does not exist, whereas $e1$ and $b1$ are connected by the control signal L_2 after circuit expansion (see Fig. 5.9).

A placement solution to guarantee routablity is high flexibility [BrFr92c], which is a measure of the connectivity within a routing architecture and is a function of all switch blocks and wire segments in the FPGA. A balance between flexibility, logic density and circuit speed must exist when designing a successful FPGA routing architecture. Configuring an FPGA with high flexibility is simple due to the large number of routing switches and wire segments involved. However, the area available for routing may be unnecessarily consumed by unused switches, resulting in less area for logic blocks thus, low logic density. Because each additional switch block causes a signal delay, high flexibility leads to reduced circuit speed. A further solution to the routability of a given placement is the optimized node assignment exemplified below, which depicts a placement of the circuit from Fig. 5.9 in an FPGA with $\psi\rho(\varepsilon_i) = 4$ and $max[\eta\rho(\varepsilon_i)] = 4$, hence $\tilde{d}(\rho) = 1$.

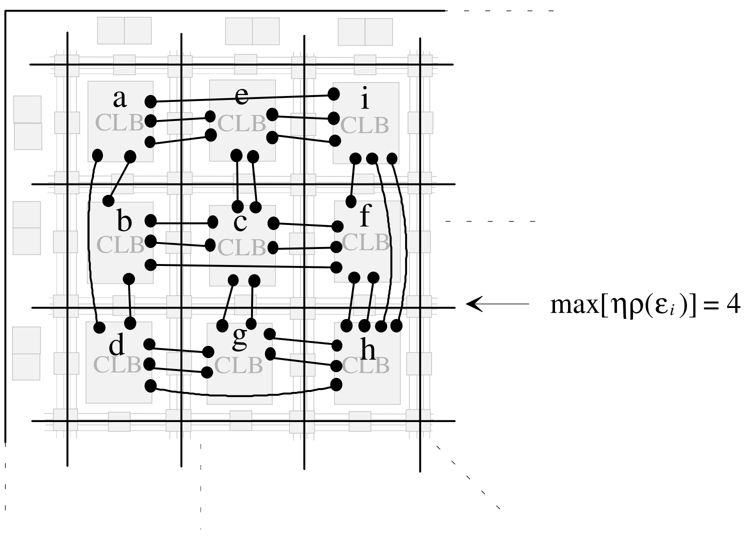


Fig. 5.13: Placement of circuit with optimized node assignment in an FPGA

The discussion in this section has shown the necessity of node assignment for the compilation of the expanded circuit. The routability of the expanded circuit in the logic emulator as well as a decrease in FPGA overhead can be achieved by optimized node assignment. The following problems illustrate the necessity of optimized node assignment:

- An optimal cutsize is difficult to attain with non-optimized node assignment due to the additional connections between fault locations caused by the control signals *L* and *C* of the Fault Injectors. Therefore, the *I/O* resources on an FPGA are used up long before CLBs are exhausted.

- The routability of the expanded circuit is not guaranteed with non-optimized node assignment due to the additional connections between the fault locations caused from control signals *L* and *C* of Fault Injectors. A possible solution to guarantee routablity is high flexibility. However, high flexibility leads to reduced circuit speed and increased FPGA usage.

5.4 Optimized Node Assignment

The previous section discussed the node assignment problem and its importance for the compilation process in the logic emulator. Various algorithms are used for the calculation of an optimized node assignment, which leads to routability and a reduction in FPGA-overhead. A new algorithm for optimized node assignment has been developed in this work and is presented in Section 5.4.1.3.

Defining the node assignment problem as a QAP entails assigning each fault location $flo_g \in FLO$, $g = 1,\dots,n$, $FLO = \{flo_1, flo_2, \dots, flo_n\}$ in the circuit to a Fault Injector $fi_g \in FI$, $FI = \{fi_1, fi_2, \dots, fi_n\}$ of the Fault Activator in order to optimize the cost function $\hat{F}(W, D)$.

$$\min_{\pi \in \Pi_n} \quad \hat{F}(W,D) = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} d_{\pi(i)\pi(j)} = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} (fi(i) - fi(j))^2$$

$$\text{subject to} \quad \tilde{d}(NA) \le 1 \qquad (5.13)$$

In order to connect node *i* to node *j*, their Manhattan length is calculated by $fi(i) = [fi(i)^x, fi(i)^y]$, $fi(j) = [fi(j)^x, fi(j)^y]$, which are the locations of node *i* and *j* on the coordinates *x* and *y* separated quadratically by the Manhattan distance $d_{fi(i)fi(j)} := (fi(i) - fi(j))^2$.

Node assignment is the problem of finding a permutation $\pi \in \Pi_n$ that minimizes the double sum in equation (5.13). An optimal solution to node assignment refers to the global minima X_{global} which is the permutation π_{opt}, $\pi_{opt} \in \Pi_n$ that minimizes the cost function over Π_n. Given are a number of interconnections between two fault locations *i* and *j* represented as w_{ij} and a distance between the two Fault Injectors *k* and *l* on the FPGA expressed as d_{kl}, $1 \le k,\ l \le n$. The wirelength needed to connect the fault location *i* at *fi*(*i*) and *j* at *fi*(*j*) is expressed as $w_{ij} d_{fi(i)fi(j)}$. Given a distance matrix $D=(d_{kl})$ and a permutation $\pi \in \Pi_n$, $D^{\pi} = d_{kl}^{\pi}$ denotes *D* by permutating, i.e. $d_{kl}^{\pi} = d_{fi(i)fi(j)}$, $1 \le k,\ l \le n$. Thus, determining the optimal solution of node assignment π_{opt} that minimizes total wirelength with the cost function in equation (5.13) amounts to solving the quadratic assignment problem QAP(*E*,*H*). The restrictions in (5.13) formalize the constraints with the routability condition $\tilde{d}(NA) \le 1$ of node assignment *NA*.

The fault locations in a circuit are defined as neighboring fault locations when they share a common interconnection, such as several branches of a fanout. In addition, the fault locations at the inputs and outputs of a logic element are also defined as neighboring fault locations. For an optimized node assignment the circuit nodes are assigned to all Fault Injectors so that the neighboring nodes are mapped to the neighboring Fault Injectors in the Fault Activator. The distance between two neighboring Fault Injectors in the Fault Activator is measured as one unit.

The circuit from Fig. 5.9 is shown in Fig. 5.14a with optimized node assignment in the structural array of the Fault Activator. Using the model of node assignment (see Fig. 5.5) the assignment problem is related to the simplified model of the placement problem of symmetrical FPGAs, termed here relative placement. Given the circuit from Fig. 5.9 it is assumed that connecting one fault location to another uses as many interconnections as the Manhattan distance between fault locations. For instance, nodes $d1$ and $g1$ below necessitate horizontal or vertical interconnections with a total sum of one unit. The node assignment in Fig. 5.14a has a total wirelength $\hat{F}(W,D)=114$ using equation (5.13).

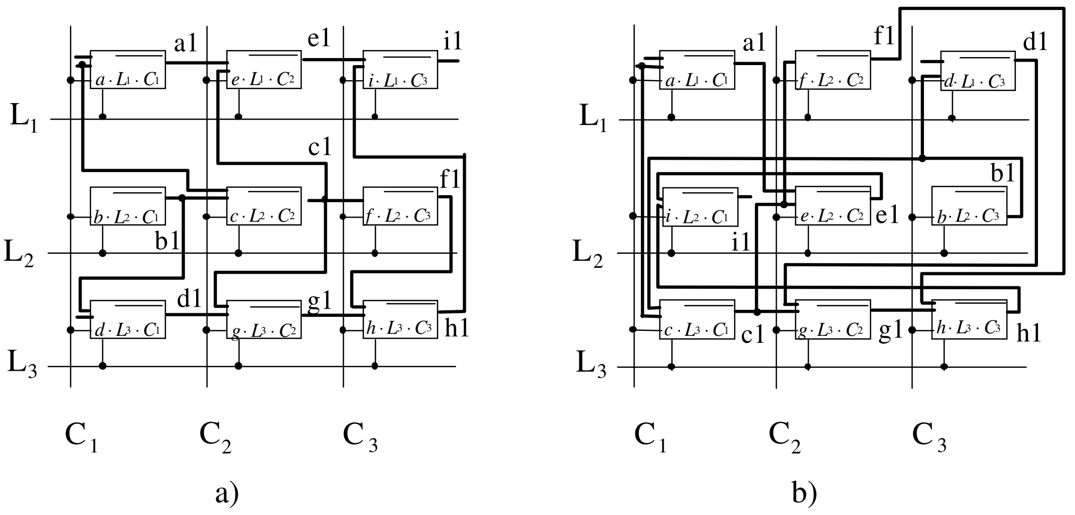


Fig 5.14: Optimized node assignment (a) compared to random node assignment (b)

The circuit from Fig. 5.10 is shown in Fig. 5.14b with a random node assignment in the structural array of the Fault Activator. The assignment of nodes *b*, *c*, *d*, *e*, *f*, and *i* to the Fault Injectors in Fig. 5.14b leads to an increase in the estimated total wirelength. For instance, nodes $d1$ and $g1$ require interconnections having a total sum of three units. The node assignment in Fig. 5.14b has a total wirelength $\hat{F}(W,D)=210$ using equation (5.13).

An optimized node assignment can be calculated by the optimized relative placement described above. The node assignment problem has the same complexity as the placement problem, which is NP-complete. A number of heuristic techniques have been developed recently for solving the QAP. Heuristics use local search to find a good solution (local minima), although not necessarily the best solution (global minimum). The time requirements of heuristic algorithms are modest, i.e. a polynomial function of the number of nodes.

5.4.1 Algorithms

In the last decade various polynomial time heuristics have been introduced, including the simulated annealing and min-cut algorithms, which provide suboptimal solutions, e.g. local optima. In addition, a new algorithm developed in this work for the calculation of the optimized node assignment is described in detail in Section 5.4.1.3. A comparison between the results of optimized node assignment for several circuits with simulated annealing, min-cut, and the new algorithm is presented in Chapter 6.

5.4.1.1 Simulated Annealing

A heuristic algorithm which attempts to overcome local optimality, i.e. local minima, in solving the combinatorial optimization problem is the simulated annealing approach [WoLe88]. The metropolis algorithm [MRRT53] can be used to simulate the behavior of a physical system and can also be applied as a heuristic method in combinatorial optimization problems [KGVe83]. A thermal process for obtaining lower energy states of a solid in a heat bath refers to the annealing process, which includes two phases. The first phase involves melting the solid by raising the temperature of the heat bath to a maximum value. In the next phase the temperature of the heat bath is slowly decreased to the minimum value of the temperature until the particles arrange themselves, which is characterized by a minimum of energy. The evolution of a solid in a heat bath in thermal equilibrium is simulated by the metropolis algorithm. By applying a small perturbation of the current state τ1 with $En_{\tau 1}$ to a subsequent state τ2 with energy $En_{\tau 2}$, an energy difference $\Delta En = En_{\tau 1} - En_{\tau 2}$ results. If ΔEn is negative, state τ2 is accepted as the next current state. When the energy difference is positive, state τ2 is accepted with a certain probability P given by $e^{\frac{\Delta En}{k_B Temp}}$. Here k_B is the Boltzmann constant and $Temp$ represents the temperature.

$$P(\Delta En) = e^{\frac{-\Delta En}{k_B Temp}} \tag{5.14}$$

Simulated annealing can be applied to any combinational optimization problem [BuRe83] when a neighborhood structure has been introduced on the set of feasible solutions (see Section 5.1). The metropolis algorithm uses as a new feasible solution $\hat{F}_{new}$ a neighbor of the current feasible solution $\hat{F}$.

The simulated annealing algorithm can be used for the node assignment problem [Kupk98]. The pairwise interchange is a simple neighbor function in which two Fault Injectors are selected and their nodes permutated. Other schemes to generate neighboring states include displacing a randomly selected node to a random Fault Injector or any interchange of nodes that may cause a change in estimated wirelength. Let $\Delta En = (\hat{F}_{new} - \hat{F})$ be the change, due to permutation, in the estimated wirelength of the cost function in equation (5.13), where $\hat{F}$ is the previously calculated wirelength and $\hat{F}_{new}$ the actual wirelength. If the new wirelength cost is lower, $\Delta En < 0$, i.e. $\hat{F}_{new} < \hat{F}$, then $\hat{F}_{new}$ is acceptable and the current state is set to

$\hat{F} = \hat{F}_{new}$. Otherwise, if the new solution $\hat{F}_{new} > \hat{F}$, the metropolis algorithm accepts the new solution on a probabilistic basis. A comparison between probability *P* and a random number *Random* generated in the range from 0 to 1 results in an acceptable solution $\hat{F}_{new}$ if $Random < e^{\frac{\Delta En}{k_B Temp}}$. This simulated annealing process is repeated until the freezing state of the solid, generally at *Temp*=0, is attained.

5.4.1.2 Min-cut

The min-cut partitioning procedure is used to generate an optimized node assignment [Li97] as shown in Fig. 5.15. Consider in an FPGA layout the horizontal line at $x=x_i$ dividing, i.e. cutting the FPGA into a top region *TO* and a bottom region *BO*. Let (x_i) denote the number of nets cut by the line x_i having at least one connection in *TO* and at least one connection in *BO*. The vertical line at $y=y_i$ cuts the FPGA into a left region *LF* and a right region *RI*. Furthermore, let (y_i) denote the number of nets cut by the line y_i that have at least one connection in *LF* and at least one connection in *RI*.

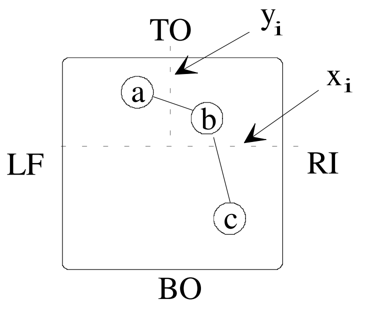


Fig. 5.15: Vertical and horizontal cut lines

Partitioning the circuit with the cutlines x_i and y_i, so as to minimize (x_i) and (y_i) can be accomplished with several algorithms, one of which is the KL (Kernighan and Lin) algorithm [KeLi70]. Similar to all versions of the min-cut heuristic algorithm, this heuristic algorithm generates the cut tree illustrated in Fig. 5.16. Each node on the cut tree represents a subgraph $G_m(U_m, E_m)$ of the graph *G*, and a section S_q to which G_m is assigned. The root of the cut tree represents the total graph *G* and the complete section *S*. A node of the cut tree is processed by selecting a horizontal or vertical cutline that subdivides *S* into two subsections S_1 and S_2. The subgraph represented by a node of the tree is then bipartitioned into two subsections whose sizes have the same ratio as the size of S_1 and S_2. This process continues until each subgraph consist of a single vertex of the graph *G*.

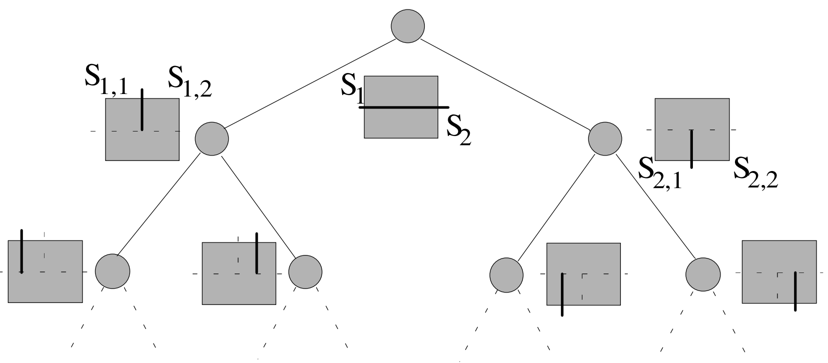


Fig. 5.16: Cut tree

The procedure for the selection of cutlines as well as the sequence in which they are processed is recommended by Breuer [Breu77a][Breu77b]. Three popular techniques for the generation of the cut tree are the Quadrature, Bisection, and Slice/Bisection procedures. In this work the Slice/Bisection procedure is implemented for the node assignment problem. The Slice/Bisection procedure selects the cutline that slices off a subgraph of the graph G resulting in an unbalanced bipartition and leading to unequal sizes $sz(u)$ of subgraphs G_1 and G_2, $|G_1| \neq |G_2|$.

A partitioning algorithm is applied to a given circuit graph for the generation of two subsections S_1 and S_2 in the FPGA as illustrated in Fig. 5.17a based on example in Fig. 5.6. The subgraph G_1 is assigned to subsection S_1 above the imaginary horizontal cutline x_1, and the subgraph G_2 to subsection S_2 below x_1. Using the Slice/Bisection procedure, n nodes of the circuit graph are divided by cutline x_1 into two set k and n-k nodes, such that (x_i) is minimized. The first k nodes obtained are assigned to the top most row, i.e. first slice of the FPGA. The procedure is then applied to the remaining n-k nodes dividing them into k and n-2k nodes of the circuit graph and continues until all nodes have been assigned to their respective rows (Fig. 5.17b).

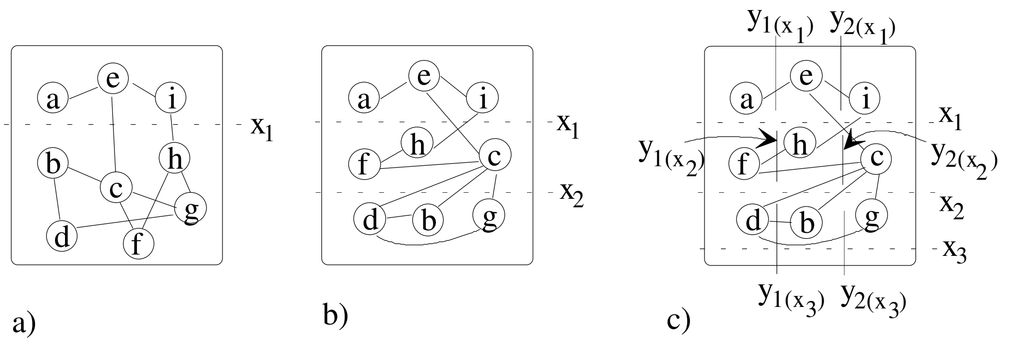


Fig. 5.17: Slice/Bisection procedure

The nodes are then assigned to columns using vertical bisection (Fig. 5.17c). Consider subsection S_1, which is separated into three subsections $S_{1,1}$, $S_{1,2}$, and $S_{1,3}$ using two vertical cutlines $y_1(x_1)$ and $y_2(x_1)$. Similarly, subsection S_2 is separated into three subsections $S_{2,1}$, $S_{2,2}$, and $S_{2,3}$ by vertical cutlines $y_1(x_2)$ and $y_2(x_2)$. The last cutline x_3 is a dummy cutline for separating the last subsection S_3. In every row the smallest section corresponds to a single node of the circuit graph.

The graph G in Fig. 5.17 is partitioned and assigned to three subsections S_1, S_2, and S_3. A typical limitation of all min-cut heuristics is that the assignment of the subgraphs G_m to the subsections is independent of the interconnections between the subgraphs [DuKe85]. For instance, if node c of section S_2 (Fig. 5.17c) is assigned to the left of cutline $y_2(x_2)$, a higher estimated total wirelength results than if assigned to the right of cutline $y_2(x_2)$.

5.4.1.3 Delta-Path

An algorithm for node assignment enabling an optimized compilation in the logic emulator is developed in this work. The compilation process for circuits with millions of gates is very time-consuming, therefore it is very important that the node assignment runtime for mapping the expanded circuit into the logic emulator is kept to a minimum. As illustrated in Section 6.1, compared with the Delta-Path and simulated annealing algorithms the min-cut algorithm has the lowest improvement of estimated wirelength in most of the circuits. The simulated annealing algorithm has the highest runtime and in most of the circuits highest improvement in estimated wirelength. Therefore, the motivation behind the Delta-Path algorithm is that this algorithm provides a compromise between runtime and optimized wirelength, and achieves an acceptable reduction of FPGA overhead as well as routability of the expanded circuit in an acceptable runtime.

The node assignment problem is represented by a directed graph $G(U,E)$, which models the combinational circuit. The graph $G(U,E)$ is a directed graph when E consists of the pairs $e=\{u,v\}$ with the nodes $u \in U$ and $v \in U$. The node u is referred to as the start-node of edge e and the predecessor node of node v. The node v is referred to as the end-node of edge e and the successor node of node u (Fig. 5.18). The algorithm consists of four subprocesses.

First Subprocess
The algorithm converts graph G(U,E) to a new graph $G'(U',E')$ by dividing $U=\{u_1, u_2, ..., u_n\}$ into k subsets $\Lambda_1, \Lambda_2, \ldots, \Lambda_k$. The subset $\Lambda_1 \subset U$ is defined as the set of source nodes u^s with empty predecessor set $\Gamma^-_{(u^s)}$ and nonempty successor set $\Gamma^+_{(u^s)}$. The source nodes u^s can be the primary inputs of the circuit and the subset Λ_1 is calculated as:

$$\Lambda_1 = \left\{ u^s \;\middle|\; (u^s \in U) \wedge (\Gamma^-_{(u^s)} = \emptyset) \wedge (\Gamma^+_{(u^s)} \neq \emptyset) \right\} \tag{5.15}$$

The subset $\Lambda_2 \subset U$ contains the set of successor nodes $\Gamma^+_{(u^s)}$ from subset Λ_1 and represents the union of the sets $\Gamma^+_{(u^s)}$.

$$\Lambda_2 = \left\{ u \;\middle|\; (u \in \Gamma^+_{(u^s)}) \wedge (u^s \in \Lambda_1) \right\}$$

$$= \bigcup_{u^s \in \Lambda_1} \Gamma^+_{(u^s)} \tag{5.16}$$

This process is repeated until the last set Λ_k contains only nodes with empty successor sets $\Gamma^+_{(v)} = \emptyset$. Since a node in Λ_k, k=1...s can be the successor node of several nodes in Λ_j, j=1...s-1 it can exist in more than one set $\Lambda_1, \Lambda_2, \ldots, \Lambda_k$. To prevent this occurrence it is defined here that each node of $U=\{u_1, u_2, \ldots, u_n\}$ can exist only once in a set Λ. In each step the nodes already contained in earlier sets Λ_j are eliminated from the newly generated set Λ_k.

$$\Lambda_k = \left\{ v \;\middle|\; (v \in \Gamma^+_{(u)}) \wedge (u \in \Lambda_{k-1}) \wedge (v \notin \bigcup_{j<k} \Lambda_j) \right\}$$

$$= \bigcup_{u \in \Lambda_{k-1}} \Gamma^+_{(u)} \setminus \bigcup_{j<k} \Lambda_j \tag{5.17}$$

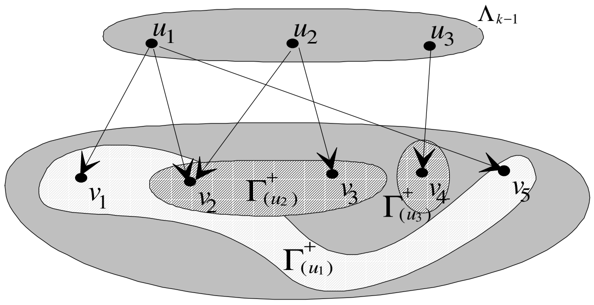


Fig. 5.18: Dividing of U into k subsets $\Lambda_1, \Lambda_2, \ldots, \Lambda_k$,

In each Λ_k a node occurs only once. The grouping of nodes from U into subsets Λ_k is performed in order to construct a new graph $G'(U',E')$. The goal is to obtain a directed graph G' in which a node $v \in \Lambda_k$ with k>1 has exactly one predecessor $u \in \Lambda_{k-1}$. In $G(U,E)$ a node

$v \in \Lambda_k$ may have several edges to the predecessor in Λ_{k-1}. The question arises which edges from *E* are to be kept in *E'*. The following solution is presented:

The first subprocess starts with an empty edge set *E'* and each time when a new set of nodes Λ is built *E'* is augmented by a corresponding set $E_k^{'}$ of edges. $E_k^{'}$ contains for each *v* only one edge from a predecessor $u \in \Lambda_{k-1}$ to $v \in \Lambda_k$. This edge is selected as follows:

Let $\deg(u)^+ = \left|\Gamma_{(u)}^+\right|$ represent the number of successor nodes of *u*. The predecessor nodes $\Gamma_{(v)}^-$ are arranged into a list $(u_1^{(v)}, u_2^{(v)}, \ldots, u_n^{(v)})$ with

$$\deg(u_1^{(v)})^+ \le \deg(u_2^{(v)})^+ \le \ldots \le \deg(u_n^{(v)})^+ \tag{5.18}$$

and only the first edge $(u_1^{(v)}, v)$ is kept, which is expressed as

$$E_k^{'} = \left\{(u_1^{(v)}, v) \,\middle|\, v \in \Lambda_k\right\}$$

and the set of all edges is

$$E^{'} = \bigcup_{k=2}^{s} E_k^{'} \tag{5.19}$$

If the degrees $\deg(u^{(v)})^+$ between $u_1^{(v)}$ and several nodes $(u_2^{(v)}, \ldots, u_n^{(v)})$ are equal all but one edge are deleted randomly.

Second Subprocess

The purpose of the second subprocess is to generate from *G'* a series of simple paths Δ_z with maximum length *h* where *h* is the number of columns in the Fault Activator and is calculated using the equation (4.3). The subprocess first generates those paths $\Delta_z = (U_z \subset U^{'}, E_z)$ which start with a node $u_z^m \in \Lambda_m$, *m*=0. U_z and E_z are initialized by $U_z = \{u_z^m\}, E_z = \emptyset$. Node sets and edge sets are augmented gradually by

$$U_z := U_z \cup \{u_z^{m+1}\} \text{ where } u_z^{m+1} \in \Gamma_{(u_z^m)}^+$$

$$E_z := E_z \cup \{u_z^m, u_z^{m+1}\} \text{ until } m = h-1 \tag{5.20}$$

Here Γ^+ denotes the successor set with respect to *G'*. Each node $u \in \Lambda_k$ used in the construction of a path Δ_z is removed from Λ_k. Thus each node can occur exactly once in a path. If $\Gamma^{+}_{(u_z^m)}$ contains more than one node then the node with maximum successor degree $\deg(u_i^m)^{+} = \left|\Gamma^{+}_{(u_i^m)}\right|$ is selected from $\Gamma^{+}_{(u_z^m)}$. If set $\Gamma^{+}_{(u_z^m)}$ contains several nodes *u'*, *u''*, ... with maximum successor degree then one of them is selected randomly. If *m*<*h*-1 and $\Gamma^{+}_{(u_z^m)} = \varnothing$ then an arbitrary node from the set Λ_m is selected although there is no edge in *E'* connecting these nodes. If further nodes are not available in Λ_m a node is selected from Λ_{m+1} to generate the subgraphs Δ_z as described above. This random selection of nodes is a drawback of the algorithm, which, when applied to the optimization of the assignment problem results in a higher value of the cost function when the number of nodes that are randomly selected increases. Note that all the paths Δ_z are subgraphs of *G'*. Some paths Δ_z will connect nodes which are not connected in *G'*.

Third Subprocess

The set of Δ is referred to as a super node set. When represented as a graph *G*(*SU,SE*) the nodes are represented by $su \in SU$ and the edges by $se \in SE$. With the application of this algorithm to the node assignment problem, the model of a two-dimensional symbolic placement can be reduced to a one-dimensional symbolic placement as shown in Fig. 5.19.

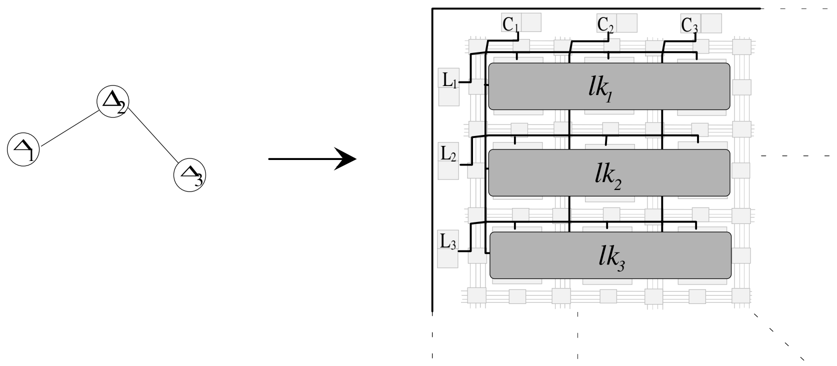


Fig. 5.19: One-dimensional symbolic placement

The assignment problem entails assigning the set of super nodes *SU*={su_1, su_2, ...,su_z} to the set of locations *LK*={lk_1, lk_2, ..., lk_z}, $Location\colon SU \to LK$, $lk_f := Location(\Delta_i)$, such that in a solution space $S = \left\{ s = (\Delta_{i_1}, \ldots, \Delta_{i_p}) \middle| (i_1, \ldots, i_p = \pi(1, \ldots, p) \right\}$ the cost function $\hat{F}\colon S \to \Re$ is minimized. As stated in Section 5.4, the assignment of *p* nodes to *p* locations *LK*={lk_1, lk_2, ..., lk_n} can be expressed mathematically by a permutation π of {1,...,*p*}. As illustrated in Fig. 5.19, determining the optimal permutation $\pi_{opt} \in S$ with $\forall \pi \in S\colon \hat{F}(\pi) \geq \hat{F}(\pi_{opt})$ that minimizes total wirelength amounts to solving the QAP in equation (5.13). Thus, the super nodes are assigned to the locations as a one-dimensional symbolic placement problem. Each π describes a possible assignment. However, the S built by the Delta-Path algorithm does not contain all the possible node assignments *NA*, $S \subset NA = \{possible\ node\ assignment\}$. S is a restricted solution space where it is hoped that the restriction is a plausible one and that S contains a solution not too far from the best solution of *NA*. Each *lk* corresponds to one line in the Fault Activator in the FPGA. In Fig. 5.19 above, the Fault Activator consists of three lines. Hence, the model above contains three locations LK=$\{lk_1, lk_2, lk_3\}$, each of which contains three Fault Injectors *fi*, $lk_1=\{fi_1, fi_2, fi_3\}$, $lk_2=\{fi_4, fi_5, fi_6\}$, and $lk_3=\{fi_7, fi_8, fi_9\}$.

Fourth Subprocess

Every Δ_i with a maximum of h nodes is assigned to a location lk with h Fault Injectors and each circuit node $u \in \Delta$ is assigned to a Fault Injector $fi \in lk$, $u \mapsto fi$. In the subgraph Δ_i the first node $u_i^m \in \Lambda_m$, m=1 is assigned to the first Fault Injector $u_i^m \mapsto fi_1$, $fi_1 \in lk_1$ and the second node $u_i^{m+1} \in \Lambda_{m+1}$ is assigned to the second Fault Injector $u_i^{m+1} \mapsto fi_1$, $fi_2 \in lk_1$, as shown in Fig.5.20. This assignment continues until all nodes in the subgraph Δ_i have been assigned to all Fault Injectors in lk_1, i.e. $u_i^h \mapsto fi_h$ and is repeated for each subgraph Δ_i in $G(SU,SE)$.

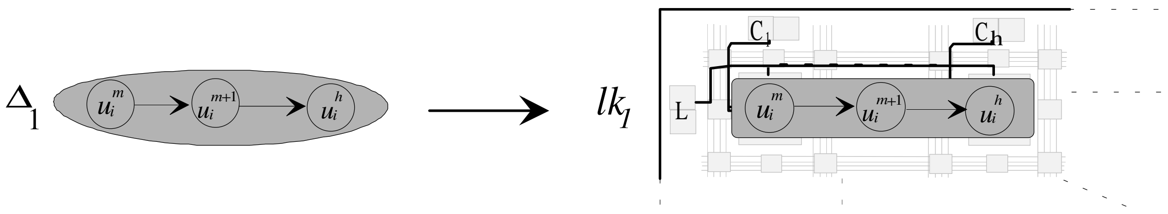


Fig. 5.20: Assignment of *u* to *fi*

Example

The graph in Fig. 5.21a contains nine nodes representing the fault locations $U = \{u_1, ..., u_9\}$ in a circuit that are assigned to nine Fault Injectors $FI=\{fi_1, ..., fi_9\}$. The set of source nodes $U^S=\{a,b\}$ has as successor nodes the sets $\Gamma^+_{(a)} = \{e\}$ and $\Gamma^+_{(b)} = \{c,d\}$, and generates $\Lambda_2 = \{e,c,d\}$. In addition, the sets of successor nodes $\Gamma^+_{(e)} = \{i\}$, $\Gamma^+_{(c)} = \{e,f,g\}$, and $\Gamma^+_{(d)} = \{g\}$ build the subset Λ_3. Since as the node e already exists in the subset Λ_2, it is eliminated in Λ_3. Furthermore, the node g has several edges to the predecessor nodes in Λ_{k-1} and by generation of the edges E' the edge to node d is selected due to the lower degree of node d, $\deg(d)^+ \le \deg(c)^+$, based on equation (5.18). Thus, Λ_3 contains the nodes $\Lambda_3 = \{i,f,g\}$, as shown in Fig. 5.21b.

The sets of successor nodes $\Gamma^+_{(i)} = \varnothing$, $\Gamma^+_{(f)} = \{h\}$, and $\Gamma^+_{(g)} = \{h\}$ generate the set Λ_4. Node *h* is present in two sets. Since both predecessors of node *h* have an equal degree $\deg(f)^+ = \deg(g)^+ = 1$ the randomly selected edge (*g*,*h*) is kept and (*f*,*h*) is deleted. The successor node $\Gamma^+_{(h)} = \{i\}$ builds Λ_5 and since node *i* already exists in the subset Λ_3, it is eliminated in Λ_5, $\Lambda_5 = \varnothing$.

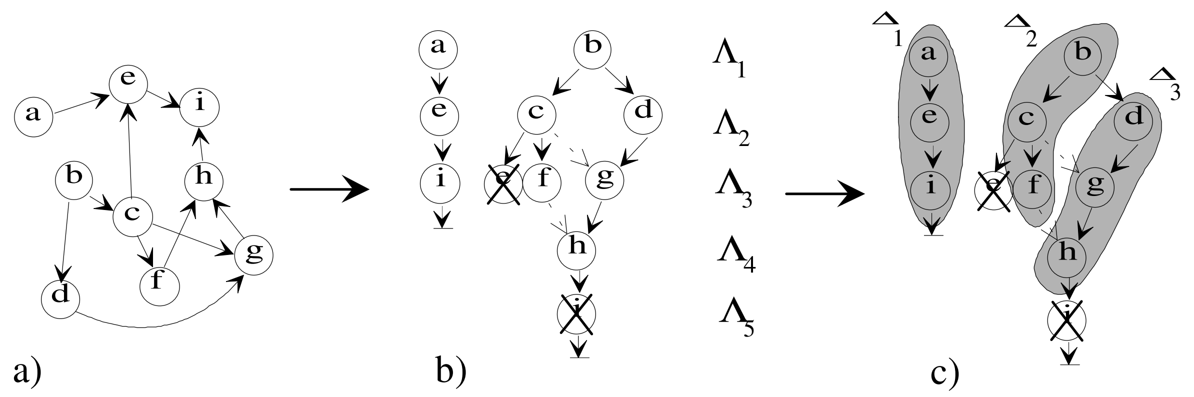


Fig. 5.21: Node assignment with Delta-Path algorithm

The set of Δ is generated for a one-dimensional symbolic placement and contains the maximum number of nodes, $\max\limits_{\Delta_z \subset \Delta} |\Delta_z| = h$. In order to generate Δ_1, a node in subset Λ_1 for instance, node *a* is selected as well as its successor node $e \in \Lambda_2$. The subgraph Δ_1 is complete with the selection of the successor of node *e*, $\Gamma^+_{(e)} = \{i\}, i \in \Lambda_3$, as shown in Fig. 5.21c, $\Delta_1 = \{a, e, i\}$. The subgraph Δ_2 is generated in a similar manner, but differs in that node *b* has two successor nodes in Λ_2, $\Gamma^+_{(b)} = \{c, d\}$. By definition node *c* is selected for inclusion in the subgraph Δ_2 based on its higher degree $\deg(c)^+ = \left|\Gamma^+_{(c)}\right|$ in comparison with the degree of node *d* ,$\deg(d)^+ \leq \deg(c)^+$. With the inclusion of the successor of node *c*, $\Gamma^+_{(c)} = \{f\}, f \in \Lambda_3$, the subgraph $\Delta_2 = \{b, c, f\}$ is generated. When the subgraph $\Delta_3 = \{d, g, h\}$ with the remaining nodes of *U'* has been generated the generation process of set Δ is complete.

The set Δ, referred to here as super nodes, is assigned to the locations $LK = \{lk_1,\ lk_2,\ lk_3\}$, $Location\colon SU \to LK$, $lk_f := Location(\Delta_i)$, such that the cost function in equation (5.13) is optimized. The process begins with a random assignment of super nodes to locations $\Delta_2 \mapsto lk_1$, $\Delta_1 \mapsto lk_2$, and $\Delta_3 \mapsto lk_3$ as shown in Fig. 5.21b. Since the distance between two neighboring locations is defined as one unit, eight units of wirelength are required for the random assignment. The new permutation π of super nodes to locations costs a wirelength of 6 units, which is the best possible permutation of symbolic placement for this example. The simulated annealing heuristic algorithm can be applied for finding the optimal solution.

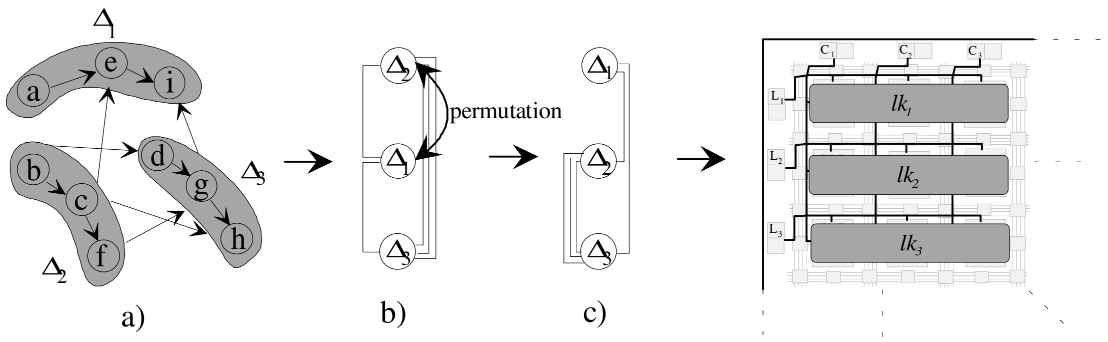


Fig. 5.22: One-dimensional symbolic placement

Every super node contains three circuit nodes, which are assigned to the three Fault Injectors in the first location $lk_1 = \{fi_1, fi_2, fi_3\}$. The process commences by assigning the first node $a \in \Lambda_1$ to the first Fault Injector $a \mapsto fi_1$, $fi_1 \in lk_1$, as depicted in Fig. 5.22. The second node $e \in \Lambda_2$ is the successor node of node $a \in \Lambda_1$, $\Gamma^+_{(a)} = \{e\}$, and is assigned to the second Fault Injector $e \mapsto fi_2$, $fi_2 \in lk_1$. This assignment continues until all nodes in the subgraph Δ_1 have been assigned to all Fault Injectors in lk_1. With the assignment of the circuit nodes $\{b,c,f\} \in \Delta_2$ and $\{d,g,h\} \in \Delta_3$ to the Fault Injectors in $lk_2=\{fi_4, fi_5, fi_6\}$ and $lk_3=\{fi_7, fi_8, fi_9\}$ the node assignment is completed.

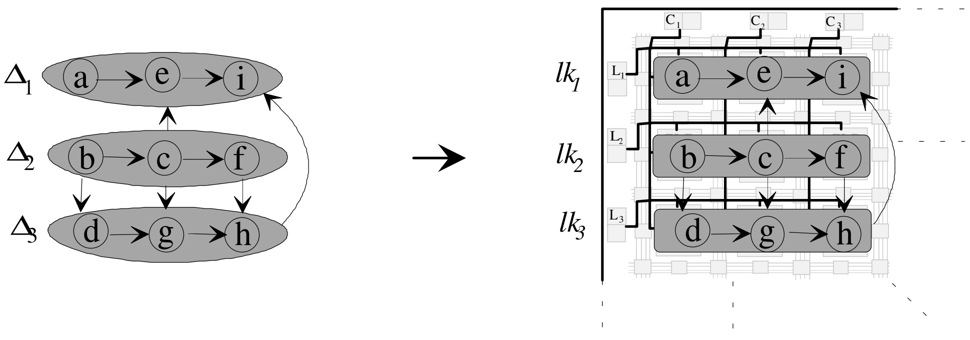


Fig. 5.23: Super node splitting

6. Experimental Results

6.1 CLB-Overhead

The previous section discussed the node assignment problem and its importance for the compilation process in the logic emulator. The node assignment approach is used in order to reduce FPGA-overhead and guarantee routability of the expanded circuit. Three different algorithms have been utilized for the calculation of an optimized node assignment. Table 6.1 lists the benchmark-85 circuits with their stuck-at-0 faults (#S-a-0), as well as the calculation of estimated wirelength with the simulated annealing (sim), min-cut (min), and Delta-Path (dp) algorithms, which provide suboptimal solutions, e.g. local optima. A comparison between the results of random (rdm) and optimized node assignment is presented as well as the percentage of improvement of the estimated wirelength with optimized node assignment relative to the estimated wirelength with random node assignment. For all circuits except circuits c3540 and c6288 the simulated annealing algorithm results in the best improvement in estimated wirelength. The wirelength for the circuits with stuck-at-1 faults (Table 6.2) is also calculated using node assignment, which when used with the Delta-Path algorithm gives a better improvement in estimated wirelength for smaller circuits than the other algorithms. However, experimental results using simulated annealing show that a slow temperature decrease corresponds to a better improvement in estimated wirelength [Kupk98], which results in a higher runtime for optimized node assignment .

Circuit	# S-a-0	Wirelength				Improvement [%]		
		(rdm)	(min)	(sim)	(dp)	(min)	(sim)	(dp)
c1908	288	622	129	121	135	79.30	80.55	78.3
c2670	705	1614	153	133	148	90.52	91.76	90.83
c3540	980	21596	2933	4043	2797	86.42	81.28	87.05
c5315	1353	15402	3423	2128	2531	77.78	86.18	83.57
c6288	5744	413006	43212	61127	21364	89.54	85.2	94.83
c7552	1646	15742	3261	1683	2959	79.28	89.31	81.2
c32k	36569	5712598	1707649	1575878	1891063	70.11	72.41	66.9
c65k	52974	187707361	35883754	5657101	31186059	80.88	96.99	84

Table 6.1: Comparison of node assignment results for stuck-at-0 faults

Circuit	# S-a-1	Wirelength (rdm)	(min)	(sim)	(dp)	Improvement [%] (min)	(sim)	(dp)
c1908	288	27203	3032	5018	3156	88.85	81.55	88.4
c2670	705	36352	5324	4760	3998	85.35	86.91	89
c3540	980	44163	6004	6464	6012	86.4	85.36	86.39
c5315	1353	128145	27439	23302	15624	78.59	81.82	87.81
c6288	5744	12512	2843	1232	3016	77.29	90.15	75.9
c7552	1646	201702	34983	31006	22025	82.66	84.63	89.08
c32k	36569	5735512	194624	1512978	1857228	66.07	73.62	67.62
c65k	52974	187707361	25889781	6016115	31158861	86.21	96.79	83.4

Table 6.2: Comparison of node assignment results for stuck-at-1 faults

The compilation process for circuits with millions of gates is very time-consuming, therefore it is very important that the node assignment runtime for mapping the expanded circuit into the logic emulator is kept to a minimum. As shown in Fig. 6.1, compared with the Delta-Path and simulated annealing algorithms the min-cut algorithm has the lowest improvement of estimated wirelength in most of the circuits. The simulated annealing algorithm has the highest runtime and in most of the circuits highest improvement in estimated wirelength. The Delta-Path algorithm provides a compromise between runtime and optimized wirelength, and achieves an acceptable reduction of FPGA overhead as well as routability of the expanded circuit in an acceptable runtime. However, experimental results indicate that the FPGA overhead factor remains almost constant with respect to circuit size because the additional functions required for the fault injection and fault activation are proportional to the number of emulated faults.

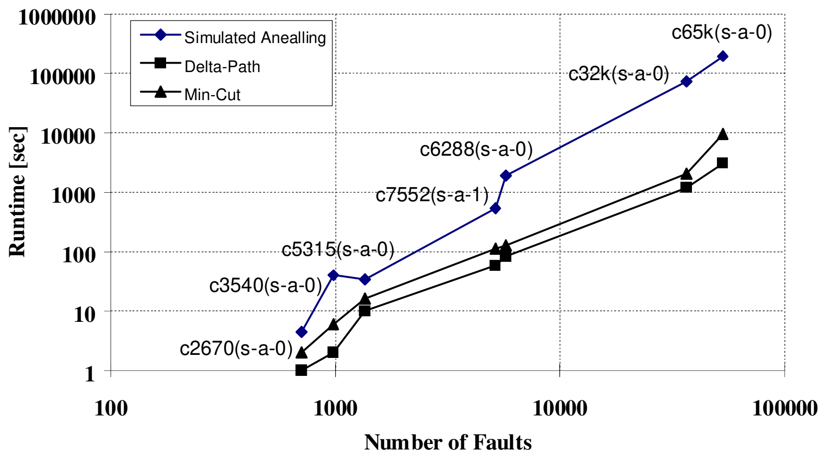


Fig. 6.1: Comparison of node assignment runtime for different algorithms

The calculated wirelength for circuit c65k (Table 6.1) with 150k nodes is 13 percent higher using the Delta-Path algorithm than with simulated annealing and the runtime for this circuit is 85 percent lower than with simulated annealing (Fig. 6.1). In spite of its higher calculated wirelength the Delta-Path algorithm satisfies the requirements of circuit mapping in the logic emulator and is therefore preferred over simulated annealing.

An experimental fault emulator has been developed (Fig. 4.9) and is described in Section 4.2. The hardware emulation component of the experimental fault emulator is a commercial logic emulator (Quickturn M250) consisting of 80 Xilinx XC4013-FPGA chips and compilation software to compile the ISCAS-85 benchmark circuits and evaluate the fault emulation approaches FES/1 and FES/2. The first step of the evaluation process involves mapping the fault-free circuits into the emulator. In addition to the ISCAS-85 benchmark circuits, Table 6.3 also includes the circuits c32k (multiplexer) and c65k (multiplexer), which are generated in this work.

Circuit	# Gate	# CLB	Pin	Net	Freq [MHz]
c1908	880	387	2442	916	11.12
c2670	1192	598	3660	1435	11.12
c3540	1669	732	4684	1721	11.12
c5315	2307	978	7016	2496	11.12
c6288	2416	1047	7280	2448	11.12
c7552	3512	1562	10045	3756	11.12
c32k	32754	7423	102192	7728	11.12
c65k	65535	10602	210367	12132	11.12

Table 6.3: Logic emulation of circuits without Fault Injectors

Table 6.4 depicts the benchmark circuits expanded by stuck-at-0 faults. Here, the number of faults (#s-a-0) represents the number of Fault Injectors. The number of CLBs (#CLB) used to map the expanded circuit is contingent on the results of node assignment. An optimal node assignment is particularly important when the capacity limit of the emulator has been reached. For example, the expanded circuit c65k can not be mapped into the logic emulator with a random assignment of fault locations to Fault Injectors. The CLB-overhead of the expanded circuits is represented by the ratio of the number of CLBs (expanded circuits) divided by the number of CLBs (original circuit) and is contingent on the number of Fault Injectors, i.e. the number of faults. With the simulated annealing algorithm the CLB-overhead has the lowest value with an average of 2.1 for stuck-at-0 and 2.7 for stuck-at-1.

The number of the CLBs required by the mapping process is also dependent on the structure of the circuit. The number of gates that can be mapped into the emulator depends on the number of pins (inputs and outputs of gates), which reflects the hardware cost. The number of extra pins required in the expanded circuit is constant, i.e. each Fault Injector needs two extra pins, which leads to a linear increase of CLB-usage in relation to the number of Fault Injectors. Compared to the original circuit, a circuit expanded by the Fault Activator uses FPGAs more efficiently due to the regular structure of CLBs in the FPGAs.

Circuit	# S-a-0	# CLB				CLB-Overhead			
		(rdm)	(min)	(sim)	(dp)	(rdm)	(min)	(sim)	(dp)
c1908	288	539	525	409	423	1.39	1.35	1.06	1.09
c2670	705	954	842	662	701	1.59	1.4	1.11	1.17
c3540	980	3097	2436	1492	1503	4.2	2.49	2	2.05
c5315	1353	2354	1824	1982	1936	2	1.29	2	1.98
c6288	5744	5015	4754	4351	4398	4.7	4.1	4.1	4.2
c7552	1646	2907	2752	2238	2205	1.86	1.7	1.43	1.41
c32k	36569	22123	21053	20012	21242	3.6	3.27	2.7	2.86
c65k	52974	38984*	30843	29051	29321	3.7*	3	2.74	2.77
Average						2.87	2.5	2.1	2.1

* larger than emulator capacity

Table 6.4: Fault emulation with stuck-at-0 Fault Injectors

Circuit	# S-a-1	# CLB				CLB-Overhead			
		(rdm)	(min)	(sim)	(dp)	(rdm)	(min)	(sim)	(dp)
c1908	1396	1309	1258	1107	1101	3.38	3.25	2.86	2.84
c2670	1781	1667	1471	1283	1298	2.78	2.45	2.15	2.17
c3540	2040	3234	2868	2765	2801	4.4	3.9	3.78	3.83
c5315	3550	3435	3407	3309	3382	3.51	3.48	3.38	3.46
c6288	560	1337	1224	1201	1230	1.27	1.16	1.15	1.17
c7552	5149	6649	6294	5409	5386	4.25	4	3.46	3.45
c32k	36555	21053	20193	20098	20153	3.44	3.1	2.71	2.71
c65k	53168	40437*	31981	30102	30423	3.81*	3	2.84	2.87
Average						3.3	3.16	2.79	2.81

* larger than emulator capacity

Table 6.5: Fault emulation with stuck-at-1 Fault Injectors

The relative improvement in CLB-overhead for random and optimized node assignment using the various algorithms is illustrated clearly in Figures 6.2 and 6.3 for all indicated circuits. The results are based on a random initial node assignment for each circuit. Substantial improvements can not be attained with the presented algorithms for node assignment when the initial state is already optimal. With the simulated annealing algorithm, the best reduction in CLB number for most circuits is associated with a high runtime of node assignment. The improvement in CLB-overhead with the Delta-Path algorithm comes close to matching the simulated annealing results. However, an even higher reduction in CLB number is achieved for circuits c1908(s-a-1), c5315(s-a-0), and c7552(s-a-0)(s-a-1). Note that better results are attained using the simulated annealing algorithm due to the limitations of the Delta-Path algorithm (Section 5.4.1.3) although in both cases the number of CLBs in circuit c3540 can be reduced by 50%. The min-cut algorithm results in a better reduction in CLB number only for circuit c5315(s-a-0). As a result of random node assignment the number of CLBs required by circuit c65k exceeds the capacity of the emulator, hence the circuit can not be mapped into the logic emulator. In contrast, optimized node assignment leads to a decrease in CLB number by almost 25 percent. Therefore, the circuit is routable and can be mapped into the logic emulator.

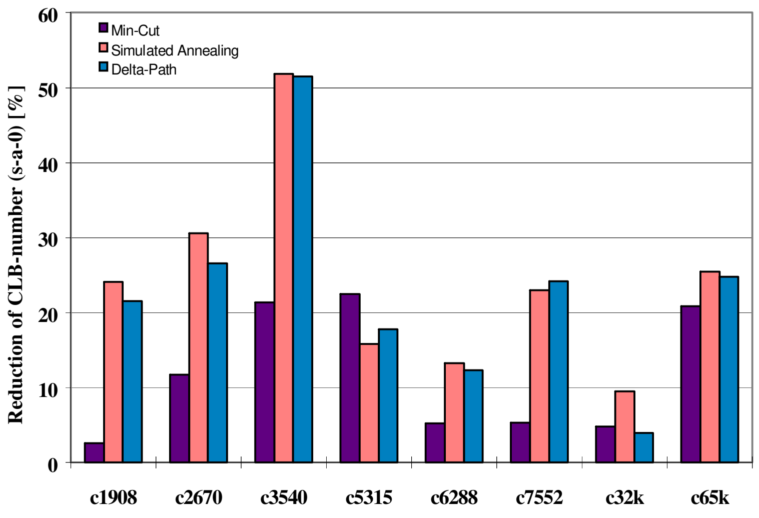


Fig. 6.2: Reduction of CLB number for s-a-0

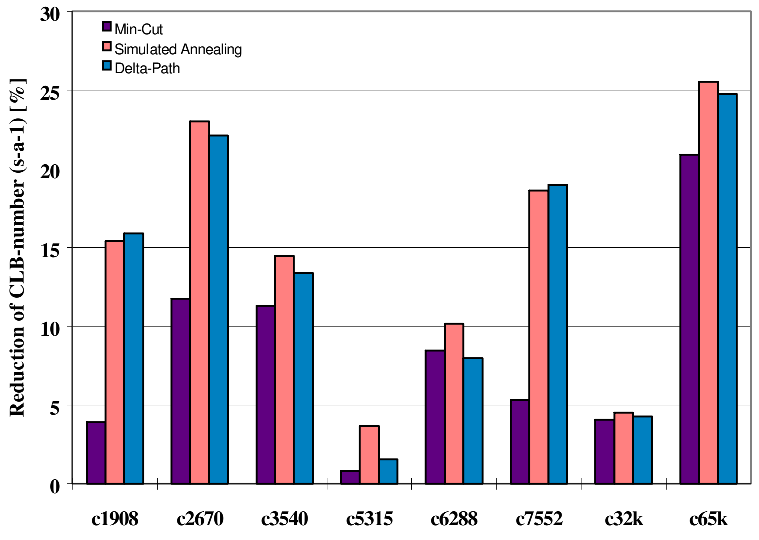


Fig. 6.3: Reduction of CLB number for s-a-1

6.2 Fault Emulation Runtime

Two approaches to fault emulation, FES/1 and FES/2, have been presented in Chapter 5. Generally, the runtime required for fault emulation is lower for FES/1 than for FES/2. This arises from the use of specific hardware modules, which interrupt the emulation process. Due to this characteristic of FES/1, fault dropping is feasible.

The fault emulation runtime for FES/1 $RT_{FE}^{FES/1}$ is determined by the number of faults *Nf* and the average number of test vectors necessary to detect a fault P_{avg}. Good emulation runtime RT_G is calculated from the number of test vectors $\hat{O}$ in a test vector set and the emulation clock speed *Freq*. Thus the total runtime $RT_{total}^{FES/1}$ is defined as follows:

$$RT_{total}^{FES/1} = RT_{FE}^{FES/1} + RT_G = \frac{(P_{avg} \cdot Nf) + \hat{O}}{Freq} \tag{6.1}$$

In contrast, FES/2 uses the vector debugger of the emulator software to control the test vectors. The test vector set is compiled as binary code, which can not be interrupted during the emulation process. Fault dropping is therefore not possible. Inasmuch as faults can be detected several times by a test vector set, fault emulation runtime increases relative to FES/1.

For FES/2, fault emulation runtime $RT_{FE}^{FES/2}$ is determined by the number of faults *Nf*, the number of test vectors $\hat{O}$, and the emulation frequency *Freq*. Runtime for a good emulation RT_G is the same as for FES/1. Total runtime $RT_{total}^{FES/2}$ is therefore defined as

$$RT_{total}^{FES/2} = RT_{FE}^{FES/2} + RT_G = \frac{(1 + Nf)\hat{O}}{Freq} \tag{6.2}$$

The Equations (6.1) and (6.2) vary in the number of test vectors needed for fault emulation. Usually, the average number of test vectors P_{avg} necessary to detect a fault is much smaller than the number of test vectors $\hat{O}$. Experience with the listed circuits has shown that almost 85 percent of all faults are detected by the first ten test vectors of a test vector set. Therefore, the runtime of FES/1 is in general much smaller than the runtime of FES/2 and a higher speedup of FES/1 compared to FES/2 can be achieved over fault simulation.

In order to evaluate this speedup, the FES results are compared with the fault simulators Comsim [MaAl93] and VED [DaKC91]. The fault simulator Comsim, a vectorized event driven Parallel Pattern Single Fault Propagation simulator developed at the University of Hannover was used on a Sun Workstation (Sparc 10 with 512 MByte RAM). The Comsim method was developed to increase the accuracy of fault modeling at gate level and deals with non-classical faults. In order to determine the fault effects for each library cell an analysis was performed beginning with a low level description of a standard cell library and a corresponding low level fault model. The resulting fault effects are mapped into gate level faults. Accurate modeling of the fault effects involves implementation of gate level fault models including stuck-at, bridging, transition, and function conversion fault models. The VED fault simulator is a Vectorized Event Driven Parallel Pattern Single Fault Propagation simulator. As presented in [DaKC91], the vectorization of parallel pattern fault simulation together with an adaptive method for controlling the vector length leads to improved simulator performance. Compared to compiled fault simulators VED is reported to be forty times faster for combinational circuits.

Comsim and VED represent state-of-the-art fault simulators although they were published in 1993 and 1991 respectively. Only Comsim was available for practical experiments and was therefore chosen for a detailed comparison using the ISCAS-85 benchmark circuits and two larger circuits generated in this work. Table 6.6, Fig. 6.4, and Fig. 6.5 illustrate the experimental results using 20k pseudo-random test vectors, a quantity which is sufficient to attain an average of 95 percent fault coverage (FC) for the evaluated circuits with the fault simulator Comsim and the fault emulators FES/1 and FES/2. Note that some circuits can reach this degree of fault coverage with a smaller number of test vectors. For example, 99,4 percent fault coverage is attained for the circuit c6288 with only 96 test vectors while 89 percent fault coverage is attained for the circuit c32k after 10000 random test vectors. The low fault coverage for the circuit c2670 results because many faults are redundant (Chapter 3). Note that the fault emulation runtimes are the sum of the s-a-0 and s-a-1 runtimes.

Circuit	# Gates	# Nodes	# Faults (*Nf*)	FC [%]	FES/2[1] [sec]	FES/1[1] [sec]	Comsim [sec]	Speedup FES/2 over Comsim	Speedup FES/1 over Comsim
c2670	1192	2678	2486	84.2	4,9	0,66	184	37.55	278.79
c3540	1669	3643	3020	95.4	6	0,75	201	33.5	268.00
c5315	2307	5115	4903	98.9	9.8	1,61	440	44.89	273.29
c6288	2416	6398	6303	99.5	12.6	1,90	629	49.90	331.05
c7552	3512	7882	6795	94.9	13.5	2,12	682	50.51	321.70
c32k	32754	75432	73124	93.6	121	23,2	6902	57.02	297.41
c65k	65535	150132	106142	94.7	176	33	12901	73.30	390.91

1) Quickturn Logic Emulator M250

Table 6.6: Comparison of fault emulation and Comsim

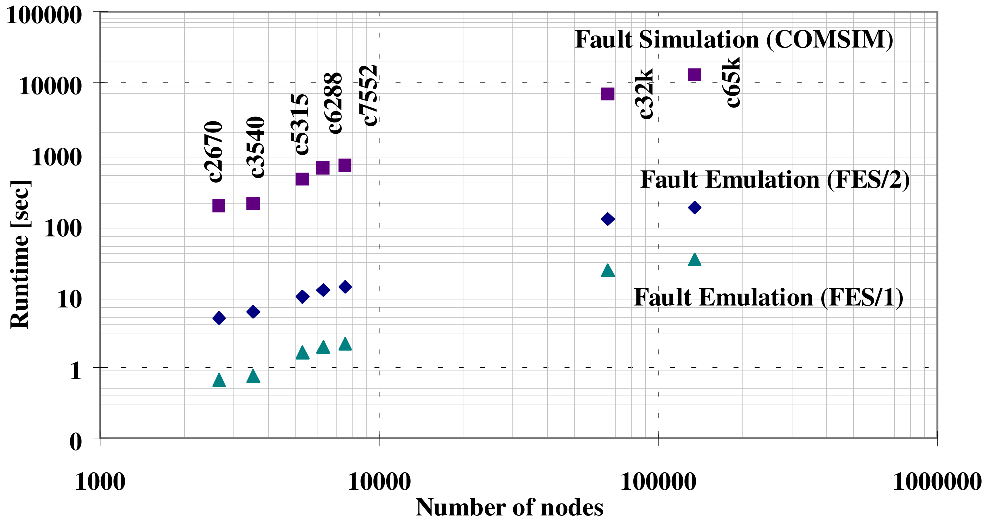


Fig. 6.4: Runtimes of fault emulation FES/1 and FES/2 and Comsim

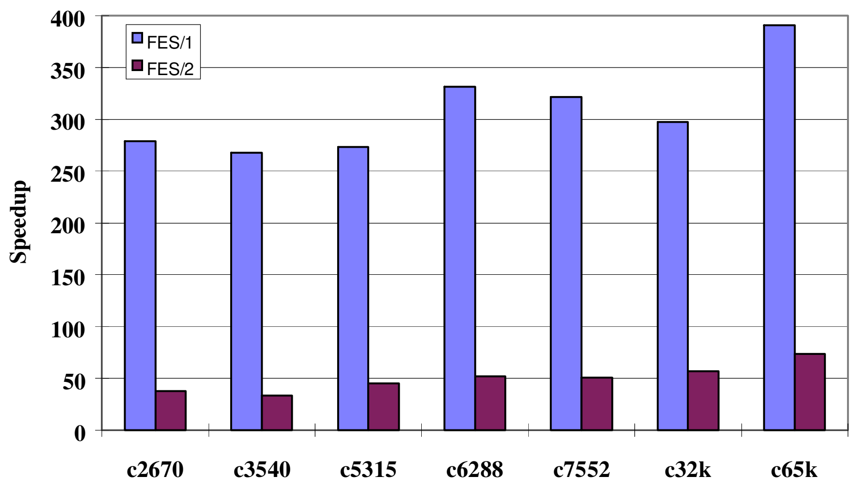


Fig. 6.5: Speedup of fault emulation FES/1 and FES/2 over fault simulation with Comsim

The experimental results confirm the expected behavior. FES/1 is approximately five times faster than FES/2 and both deliver a considerable speedup against Comsim. Note, however, that Comsim is able to model non-classical faults and therefore should be slower than a stuck-at fault simulator. To take this into account a comparison is made with the results published for VED in [DaKC91]. For comparison with fault emulation ISCAS-85 and -89 benchmark circuits [BBKo89] are used. However, only the combinational parts of the sequential ISCAS-89 benchmark circuits are simulated. The experimental results of the FES/1 emulation shown in Table 6.6 indicate that for the selected benchmark circuits a P_{avg} value of 3k-5k test vectors is appropriate. Therefore, P_{avg}=5k and a fault emulation speed of Freq=10 MHz have been used to calculate expected runtimes. Note that although FES/1 runtimes already exist for the circuits c5315, c6288, and c7552 in Table 6.6, a pessimistic calculation of runtime with P_{avg}=5k is made in Table 6.7 and results in a higher runtime for the circuits when compared to the runtime in Table 6.6. The VED runtimes have been taken from [DaKC91] for the random test vector set size of 500k and from [Daeh97] for the random test vector set size of 1000k. The results are presented in Table 6.7, Fig. 6.6, and Fig. 6.7.

Circuit	# Gates	# Nodes	# Faults (*Nf*)	FES/1[1)] [sec]	VED[2)] [sec]	Speedup FES/1[1)] over VED[2)]	FES/1[3)] [sec]	VED[4)] [sec]	Speedup FES/1[3)] over VED[4)]
c5315	2307	5115	4903	2.5	253	101.2	-	-	-
c6288	2416	6398	6303	3.2	345	107.8	-	-	-
c7552	3512	7882	6795	3.4	653	192	3.5	48	13.7
s9234	2900	8840	6100	3.1	1403	452.5	-	-	-
s13207	4700	12909	9000	4.5	1220	271.1	-	-	-
s15850	5400	15148	10600	5.3	1430	269.8	5.4	126	23.3
s38584	12600	35210	32000	16	3783	236.4	16.1	322	20
s38417	13200	36148	28700	14.4	4285	297.5	-	-	-
s35932	13200	36292	34000	17	6916	406.8	-	-	-

1) Calculated expected values for 500k test vectors
2) Values from [DaKC91] on Apollo DN2500 for 500k test vectors
3) Calculated expected values for 1000k test vectors
4) Values from [Daeh97] on HP735 for 1000k test vectors

Table 6.7: Comparison of fault emulation FES/1 and VED

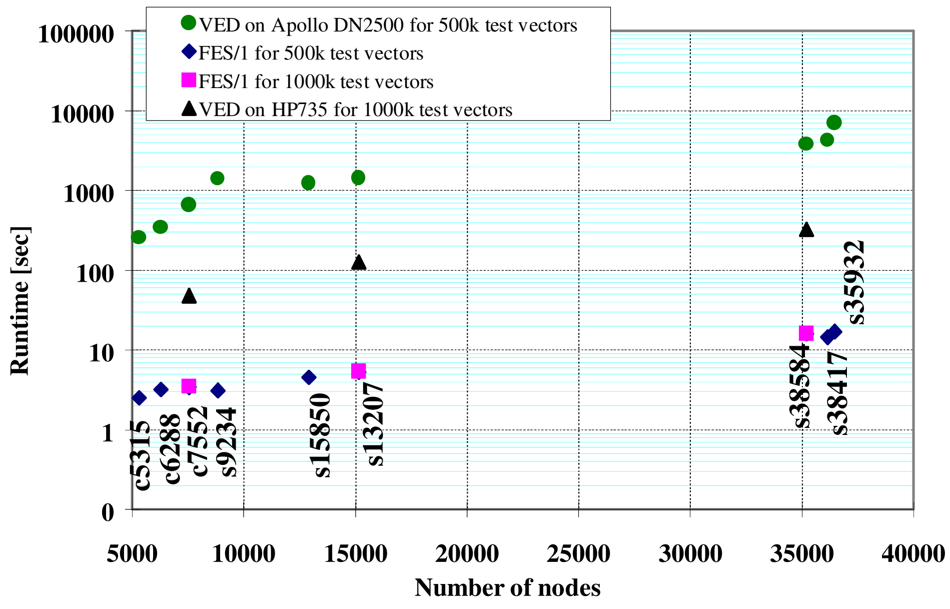


Fig. 6.6: Simulated runtime of VED and calculated runtime of fault emulation FES/1

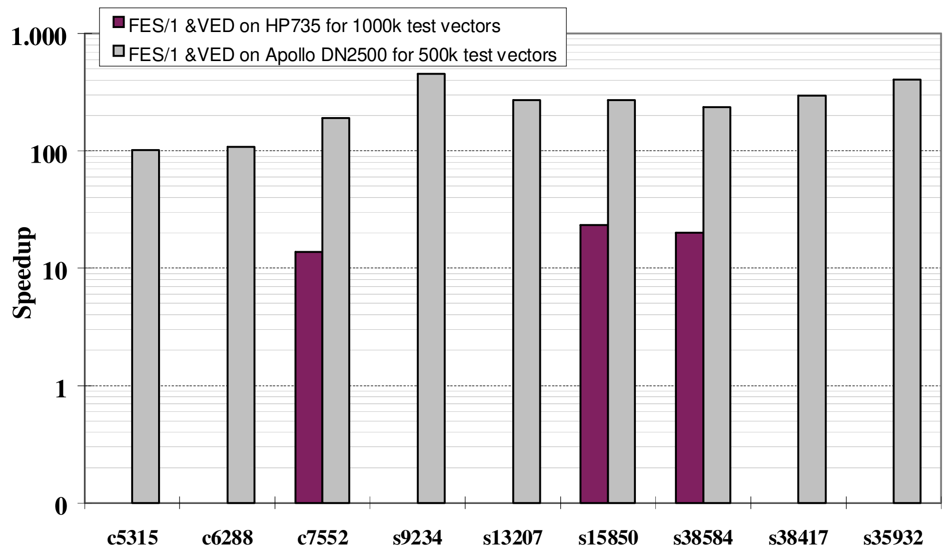


Fig. 6.7: Speedup of fault emulation FES/2 over fault simulation with VED

The speedup of FES/1 over VED is in the range of 10 to 20 at the least. Taking into account that the VED results have been obtained on Apollo DN2500 and HP735 computers, it can be assumed that for the examined circuits on today's workstations the speed of fault emulation will still outperform VED. Moreover, the following arguments support the claim that the potential of fault emulation is much larger.

1) The runtime of event-driven PPSFP depends on the average activity of the circuit per simulation run (Equation (3.7)). Low runtimes for benchmark circuits may be the result of their low average activity [Alt95]. For circuits with higher activity VED runtime increases (Equation (3.7)) and a higher speedup relative to VED can be expected with fault emulation.

2) When dealing with sequential circuits, an average number of iterations must be executed for each simulation run. With each iteration additional events are generated at the combinational part of the circuit where new storage elements can be activated. The activation of new storage elements results in an additional number of iterations, which in turn causes new events. The average circuit activity increases and a significant increase in the runtime of VED can be expected for sequential circuits.

3) One of the major disadvantages of fault simulation is that runtime increases linearly to quadratically with the number of circuit elements [KiHa87][Goel80] due to the fact that the number of faults grows proportionally with circuit size and the effect of each fault is propagated throughout the circuit. Advanced approaches to parallel fault simulation minimize the number of simulation passes by processing faults or test vectors

simultaneously. However, the circuit elements must still be processed sequentially in order to simulate the complete circuit. With fault emulation, all circuit elements are processed in parallel by the emulation hardware. Thus, emulation runtime is based solely on the number of faults and the number of test vectors and, in contrast to fault simulation, increases only linearly with circuit size. Hence, for fault emulation the speedup increases for larger circuits as shown in Figures 6.5 and 6.7. Note that Fig. 6.7 is scaled logarithmically.

4) Not only workstations but also emulators are increasing in performance. All measurements and estimations have been made using a Quickturn System Realizer M250 with FPGAs built in 1993. FPGA technology, capacity, and routing resources have been enhanced since then and an emulation speed of approximately 25MHz for an emulator capacity of 20 million gates is now possible.

Unfortunately, the speedup for circuits larger than 65k gates could not be measured due to limited hardware resources. The emulator M250 has a nominal capacity of 250k gates. Considering a hardware overhead of a factor of nearly 2 to 4 (see Tables 6.4 and 6.5) for fault emulation, circuits larger than 65k can not be fault emulated with the M250 system. The FES/1 and FES/2 approaches to fault emulation are implemented and evaluated only for combinational circuits (Table 6.6). As discussed in Chapter 4, these approaches can also be used for a two-valued logic fault emulation for sequential circuits. Due to the implementation of the fault emulator as a two-valued logic, the fault emulator can not handle potentially detected faults, which are unknown values at the primary outputs of the circuit. In other words the faults may or may not be detected, depending on the initial values of the flip-flops. The percentage of potentially detected faults in sequential circuits is usually very low [ChHu95]. Table 6.8 compares FES/1 and FES/2 to previously published fault emulation approaches and is based on Table 4.1 in Chapter 4. Similar to the Cheng&Dai approach, both FES/1 and FES/2 require additional logic functions for fault injection. In contrast to the Cheng&Dai approach, however, FES/1 and FES/2 use an x- and y-decoder for fault activation. This leads to a faster fault injection without reconfiguration of the emulator hardware, a requirement of the SFE approach. The solution to the mapping problem presented by Cheng&Dai implements the fault grading method to generate several expanded circuits with a subset of faults. Because each expanded circuit must be reconfigured in the emulator, the total reconfiguration time increases significantly. For a large number of faults fault emulation runtime (Equation (4.1)) is negligible when compared to the total reconfiguration time for the complete FPGA system. Neither the fault emulation runtime for different circuits nor a comparison with a software-based fault simulator is indicated in [ChHu95]. Rather, for a circuit with 100k gates and 100k faults the estimated total fault emulation runtime *Rt* (Equation (4.1)) is calculated with 1 MHz and 50k test vectors. This results in *Rt*=81 minutes with a reconfiguration time of *R*=20 minutes and a fault emulation runtime of *Rf*=61 minutes.

	Timoc (1979)	**Cheng&Dai**	**SFE**	**KRONE**	**FES/1**	**FES/2**
Fault Injection	additional logic functions OR / NOR/...	additional logic functions AND / OR/...	reconfiguration of logic blocks (BLP)	switch modules embedded in FPE-cells	additional logic functions AND / OR/...	additional logic functions AND / OR/...
Fault Activation	shift register	shift register	emulator software	emulator software	X-Y Decoder	X-Y Decoder
Evaluation of Results	LSI-Tester	emulator	additional hardware	emulator	additional hardware	emulator
Hardware Overhead	n/a	1.3 - 2 for each replication	n/a	n/a	2-3	2-3
Technology Dependency	breadboarding and wire-wrapping	none	Meta FPGAs	KRONE FPEC	none	none
Runtime	-	n/a	low-high	n/a	low	low-high

Table 6.8: FES/1 and FES/2 approaches based on the comparison of Table 4.1

The reconfiguration time with the SFE approach is negligible when the average number of test vectors for each fault is over 10000. However, experimental results [BuRe96] indicate that 90% of all faults are detected by the first hundred test vectors. For benchmark-89 circuits a speedup from 8 to 20 in fault emulation runtime over the fault simulator Hope [LeHa93] is reported. For most of the circuits, the average number *AN* of test vectors needed to detect a fault is less than 10000, e.g. *AN*=4531 for the circuit s38417. Therefore, the reconfiguration time T_{conf} as calculated with the SFE approach is a considerable part of the total runtime of SFE. In contrast, the FES/1 and FES/2 approaches involve no reprogramming or reconfiguration of the FPGAs. However, FES/2 runtime increases significantly when a large number of test vectors exists. The calculation of FES/1 runtime is similar to the SFE approach with the difference that the reconfiguration time is inapplicable in the FES/1 approach. Due to its required reconfiguration time, the total runtime of the SFE approach is by a factor of $T_{conf} \cdot Nf$ greater than the FES/1 runtime. The calculation of FES/1 runtime with Equation (6.1) for the largest circuit (s38417) in [BuRe96] with *AN*=4531, emulation speed of 1.5 MHz, and 57448 faults results in a runtime of 172 seconds. In contrast, the SFE approach requires in addition to 172 seconds runtime a reconfiguration time of 45 seconds, which is derived from the 0.8 millisecond average reconfiguration time [BuRe96] for each fault. The reconfiguration time for this circuit constitutes over 20 percent of the total fault emulation runtime. SFE speed [BuRe96], calculated with Equation (4.2), indicates how many faults can be executed per second and depends on the average number of test vectors, reconfiguration time, and the operating frequency of the fault emulation. In order to compare the number of faults per second that can be executed using the SFE approach with those that can be executed with FES/1 the Equation (6.1) is used for $RT_{total}^{FES/1}$=1 second and results in

$$Nf = \frac{Freq}{P_{avg}} \qquad (6.3)$$

A comparison of the Equations (4.2) and (6.3) shows that, due to its required reconfiguration time, the SFE approach executes a factor of $\frac{P_{avg}}{P_{avg} + T_{conf} \cdot Freq}$ fewer faults per second than the FES/1 approach. A considerable advantage of the FES approaches over previous fault emulation approaches is represented by the absence of reprogramming and reconfiguration times as well as the low-cost implementation of fault injection into the existing reprogrammable gate arrays.

Table 6.9 concludes the discussion of fault emulation by illustrating the basic characteristics of the existing approaches. Logic emulation systems exist today with a capacity of 20 million gates requiring a compilation runtime of 2 million gates per hour, with feasible clock speeds of up to 25 MHz. This impressive basic speed will translate into large speedups for large circuits. An additional advantage of fault emulation is the ability to plug the circuit into the target system as a reconfigurable prototype with the emulator taking the place of a chip, such as a board with several chips, where one or more chips are emulated for the evaluation of the complete system under real operating conditions. However, the long compilation runtime required by fault emulation is a disadvantage. Fault emulation and fault simulation using hardware accelerators necessitate specific hardware, whereas a fault simulator requires only a von-Neuman machine. Furthermore, the timing of the circuit can not be modeled in a logic emulator although the possibility exists of arbitrarily delaying each signal in the emulator.

	Fault Simulation	**Hardware Accelerator**	**Fault Emulation**
Implementation Time	Low	High	High
Basic Sim/Emulation Speed	Low(Hz)	Low(Hz)	High(MHz)
Hardware Requirement	Low	High	High
Timing Modeling	Yes	Yes	No
Runtime with Respect to Circuit Size	Linear to Quadratic	Linear to Quadratic	Linear
Integration into Target System	No	No	Yes

Table 6.9: Overview of existing approaches

7. Conclusion and Future Work

Conclusion

Due to the increasing complexity of large circuits fault simulation requires extremely long computing times. Chapter 3 discusses various methods which attempt to minimize simulation time by processing faults concurrently. If a large number of faults is to be considered, however, these methods do not lead to an acceptable reduction in computing time for complex circuits. A new approach to design verification, logic emulation, uses a reprogrammable prototype of a digital circuit for hardware-based fault simulation, which attains a speedup over the software fault simulator through a reduction in runtime.

Two new approaches to fault emulation, FES/1 and FES/2, were developed to satisfy the requirements of rapid fault injection including fault activation, emulator technology independence, optimal fault emulation runtime (Chapters 4 and 6), minimal hardware overhead, and optimized mapping (Chapter 5). Although identical methods of fault injection and fault activation in the FPGAs are used by FES/1 and FES/2, FES/1 [SeBa97] [Seda97] uses the in-circuit mode and involves expanding the logic emulator by additional hardware modules for test generation and emulation analysis, whereas fault emulation FES/2 [SeBa98] [Seda97a] uses the acceleration mode and evaluates the test vector set without additional hardware. In contrast to the previously presented methods of fault emulation, these new approaches allow for faster fault injection into any node of the circuit without dependency on a specific logic emulator technology.

When using benchmark-85 circuits (combinational circuits) with 800-65k gates a speedup factor of 15-390 is acquired compared to Comsim, whereas a calculated speedup factor of 13-23 is reached with 800-13k gates when compared to VED. As presented in Chapter 6, the resulting fault emulation runtime increases linearly with circuit size in contrast to fault simulation. Therefore, for fault emulation a higher speedup can be obtained for very large circuits with millions of gates. In comparison to the FES approaches, fewer faults per second can be executed using the SFE approach due to its required reconfiguration time (Equation 6.3). A considerable advantage of the FES approaches over previous fault emulation approaches is represented by the absence of reprogramming and reconfiguration times as well as the low-cost implementation of fault injection into the existing reprogrammable gate arrays.

FES/1 and FES/2 expand the circuit using Fault Injectors. Each Fault Injector has a corresponding logical address and is controlled by the Fault Activator. Fault activation through use of the Fault Activator utilizes neither a shift register nor reconfiguration or reprogramming of FPGAs and therefore differentiates itself from the previous techniques. In addition, processing a data sequence is not a prerequisite to fault injection as with a shift register. Fault locations are arranged in an addressable array by the Fault Activator. Thus, each addressed fault location, i.e. Fault Injector, is directly accessible and enables a faster fault injection. The fault location is addressed using a two-dimensional array of an address decoder. Furthermore, a symmetrical FPGA, which is also structured as a two-dimensional array, is used for the optimal mapping of the expanded circuit into the logic emulator. The expansion of a circuit by Fault Injectors and a Fault Activator constitutes an overhead of FPGA resources in the logic emulator. However, the node assignment method leads to an improved usage of FPGA-resources for fault emulation, hence a reduction of FPGA overhead [Seda98a]. Hardware-based fault injection is considered with the objective of minimizing the hardware overhead, and includes mapping the faulty circuit with optimized partitioning, technology mapping, and placement and routing.

The compilation process precedes mapping the expanded circuit into the logic emulator and is very time-consuming for circuits with millions of gates. It is therefore very important that the node assignment runtime for mapping the expanded circuit into the logic emulator be kept to a minimum. Simulated annealing, min-cut algorithms and a new algorithm Delta-Path, which is developed in this work, are utilized for the calculation of an optimized node assignment. The Delta-Path algorithm provides a compromise between runtime and optimized node assignment, and achieves an acceptable reduction of FPGA overhead as well as routability of the expanded circuit in an acceptable runtime. Experimental results show that optimized node assignment reduces the FPGA-overhead by 10% to 54%.

Future Work

In order to develop dependable systems the validation of their fault tolerance properties is required [CMSi98]. Fault tolerant circuits are designed in such a way that although parts of the system have failed, the system is still able to deliver correct outputs using functional duplication. Here, the outputs are computed individually by two, or often three, separate systems. A fault is indicated when system outputs differ. Fault injection is commonly used to evaluate the dependability of a system and can be performed in two ways: system simulation-based fault injection and hardware-based fault injection. The simulation-based approach involves the injection of faults into an accurate simulation model of the system and can be very time-consuming for complex systems. An advantage of this technique is that it can be applied earlier in the design phase than a hardware prototype. Hardware-based fault injection entails injecting physical faults into the target system hardware with the advantage of realistic fault modeling. Methods which implement this technique include electromagnetic interferences [KFAC95], pin-level fault injection [Arla90], power supply disturbances [MKGT92], and heavy-ion radiation [KLDJ94]. The main problem however, is not the injection of faults, but is related to the difficulties of controlling and observing the fault effects inside the system.

The dependability of a system can be evaluated using a logic emulator for hardware-based fault injection. An important application of fault emulation is real time fault injection into a target system hardware for the evaluation of system behavior. Here, faults are injected into the system in order to identify the dependability deficiencies of the system, observe system behavior with the given faults, as well as determine the degree of fault coverage. Then, data regarding the faulty behavior of the system can be gathered. The logic emulator has the advantage of being used early in the design phase similar to a simulation-based approach. Figure 7.1 illustrates the process of system evaluation with a fault emulation approach. The target hardware and software are connected to the logic emulator, which contains the expanded design with Fault Injectors and the Fault Activator. A data analyzer gathers data for the faulty behavior of the system and analyzes the results of an in-circuit fault emulation. The controller is responsible for the control of fault injection during system operation.

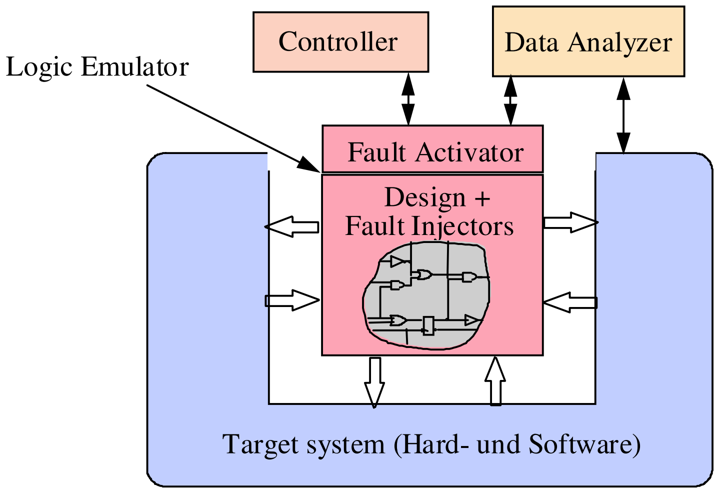


Fig. 7.1: Evaluation of system dependability using logic emulation

Various methods and an algorithm have been introduced in this work, which provide improved evaluation of future digital circuits with increasing design complexity. The results presented here indicate that, due to the acceleration of test vector evaluation, a large number of faults can be examined within a short evaluation time. This is a necessity for current and future circuit complexity. Fault emulation uses advanced debugging and performance monitoring features of a logic emulator to inject hardware-based faults in a realistic operating environment, as well as to monitor the activation of the faults and their impact on the target system behavior in detail. Therefore, for critical applications, such as traffic control, aerospace, and medical life support, this work makes an important contribution.

8. References

[AbBF90a] Abramovici, M., Breuer, M. A., Friedman A. D., "Digital Systems Testing and Testable Design", New York, W.H. Freeman and Company, 1990, pp. 111

[AbBF90b] Abramovici, M., Breuer, M. A., Friedman A. D., "Digital Systems Testing and Testable Design", New York, W.H. Freeman and Company, 1990, pp. 541

[AbBF90c] Abramovici, M., Breuer, M. A., Friedman A. D., "Digital Systems Testing and Testable Design", New York, W.H. Freeman and Company, 1990, pp. 343

[AlKa95] Alpert, C.J., Kahng, A.B., "Recent Directions in Netlist Partitioning: A Survey", The VLSI Journal, Vol. 19, 1995, pp. 1-93

[Alt95] Alt, J., "Fehlersimulation synchroner Schaltungen unter Berücksichtigung nicht-klassischer Fehler", University of Hannover, PhD Thesis, 1995, pp. 114

[Arla90] Arlat, J., "Fault Injection for Dependability Validation: A Methodology and Some Applications", IEEE Transactions on Software Engineering, Vol. 16, No. 2, 1998, pp. 166-182.

[Arms66] Armstrong, D. B., "On finding a nearly minimal set of fault detection tests for combinational logic nets ", IEEE Transactions on Electronic Computers, Vol. EC-15, 1966, pp. 66-73

[BARZ87] Barzilai, Z., "HSS-A High Speed Simulator", IEEE Transactions on Computer Aided Design, Vol. 6, No. 4, July 1987, pp. 601-617

[BBKo89] Brglez, F., Bryan, D., Kozminski, K., "Combinational Profiles of Sequential Benchmark Circuits", International Symposium on Circuits and Systems, ISCAS, 1989, pp. 1929-1934

[Benn82] Bennetts, R.G., " Introduction to Digital Board Testing", Edward Arnold, 1982

[BhHi92] Bhat, N., Hill, D., "Routable Technology Mapping for LUT FPGAs", International Conference on Computer Design, ICCD, 1992, pp. 95-100

[Bott85] Bottorff, PP., "Test Generation and Fault Simulation", VLSI Testing, North Holland, 1985, pp. 29-33

[Breu77a] Breuer, M.A., "A Class of Min-Cut Placement Algorithms" Proceedings of 14th Design Automation Conference, October 1977, pp. 284-290

[Breu77b] Breuer, M.A., "Min-Cut Placement", Journal of Design Automation and Fault Tolerant Computing, 1, October 1977, pp. 343-382

[Brew77] Brewer, B.A., "Digital System Design Automation: Languages, Simulation and Database", Pitman, 1977

[BrFr92a] Brown, S.D., Francis, et al., "Field-Programmable Gate Arrays", Kluwer Academic Publishers, 1992

[BrFr92b] Brown, S.D., Francis, et al., "Field-Programmable Gate Arrays", Chapter 3: Technology Mapping for FPGAs, Kluwer Academic Publishers, 1992, pp. 50-61

[BrFr92c] Brown, S.D., Francis, R.J., et al., "Field-Programmable Gate Arrays", Chapter 4: Flexibility of FPGA Routing Architecture, Kluwer Academic Publishers, 1992, pp. 147-166

[BRYA87] Bryant, R.E., "COSMOS: A Compiled Simulator for MOS Circuits", Proc. 24th Design Automation Conference, DAC, 1987, pp. 87-92

[BuBa90] Butts, M., Bacheler, J., "An Efficient Logic Emulation System", Proc. of International Conference on Computer Aided Design, ICCAD, 1990, pp. 138-141

[BuRe83] Burkard, R.E., Rendl, F., "A Thermodynamically Motivated Simulation Procedure for Combinatorial Optimization Problems", European Journal of Operational Research 17, 1983, pp. 169-174

[BuRe96] Burgun, L., Reblewski, F., Fenelon, G., Barbier, J., Lepapa, O., "Serial Fault Emulation", Proc. of the 33rd Design Automation Conference, DAC, 1996, pp. 801-806

[Cha76] Cha, C. W., "Deductive Simulator", IBM Technical Disclosure Bulletin, Vol. 19, No. 6, 1976, pp. 2352-2353

[Chan65] Chang, H. Y., "An Algorithm for Selecting an Optimum Set of Diagnostic Tests", IEEE Transactions on Electronic Computers, Vol. EC-14, 1968, pp. 706-711

[ChHu95] Cheng, K., Huang, S., Dai, W., "Fault Emulation: A New Approach to Fault Grading", International Conference on Computer Aided Design, ICCAD, 1995, pp. 681-686

[ChWe91] Cheng, C., Wei, Y.A., "An Improved Two-Way Partitioning Algorithm With Stable Performance", IEEE Transactions on Computer Aided Design, August 1991, pp. 1502-1511

[ChWo94] Chang, S., Woo, N., "Layout Driven Logic Synthesis for FPGAs", 31st ACM/IEEE Design Automation Conference, DAC, 1994, pp. 308-313

[CMSi98] Carreira, J., Madeira, H., Silva, J., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computer", IEEE Transactions on Software Engineering, Vol. 24, No 2, 1998, pp. 125-136.

[Coel89] Coelho, D.R., "The VHDL Handbook", Kluwer Academic Publishers, 1989

[DaKC91] Daehn W., Kannemacher, D., Castagne, J., "Vector Length Control for Control for Compiled Code Event Driven Pattern Parallel Fault Simulation", European Test Conference, ETC, 1991, pp. 165-170

[Dona88] Donath, W.E., "Physical Design Automation of VLSI Systems", Benjamin Cummings, 1988, pp. 33-62

[DuKe85] Dunlop, A.E., Kernighan, B.W., "A Procedure for Placement of Standard-Cell VLSI Circuits", IEEE Transactions on Computer Aided Design, 4, January 1985, pp. 92-98

[DuRa79] Duhamel, PP., Rault, J. C., "Automatic Test Generation Techniques for Analog Circuit and Systems: A Review", IEEE Transactions on Circuits and Systems, Vol. CAS-26, No. 7, 1979, pp. 411-440

[EIA87] Electronic Industries Association, "EDIF Electronic Design Interchange Format Version 2.0.0", EDIF Steering Committee, 1987

[Eldr59] Eldre, R. D., "Test Routines Based on Symbolic Logical Statement", Journal ACM, Vol. 6, No. 1, 1959, pp. 33-36

[Evek91] Eveking, H., "Verifikation digitaler Systeme", Teubner, 1991

[Fran90] Francis, R.J., et al., "Chortle: A Technology Mapping Program for Lookup Table-Based Field-Programmable Arrays", Proc. 27th Design Automation Conference, DAC, 1990, pp. 613-619

[Fran91a] Francis, R.J., et al., "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs", Proc. 28th Design Automation Conference, DAC, 1991, pp. 227-233

[Fran91b] Francis, R.J., et al., "Technology Mapping of Lookup Table-Based FPGAs for Performance", Proc. International Conference on Computer-Aided Design, ICCAD, 1991, pp. 133-138

[Frie67] Friedman, A.D., "Fault Detection in Redundant Circuits", IEEE Transactions on Electronic Computing, Vol. EC-16, 1967, pp. 99-100

[Goel80] Goel, PP., "Test Generation Cost Analysis and Projections", Proc. IEEE Design Automation Conference, DAC, 1980, pp. 77-84

[Goel81] Goel, PP., "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Transactions on Computers, Vol. C-30, 1981, pp. 215-222

[GoVo71] Godoy, H.C., Vogelberg, R.E., "Single Pass Error Effect Determination (SPEED)", IBM Technical Disclosure Bulletin, Vol. 13, 1971, pp. 3343-3344

[HaKu72] Hanan, M., Kurtzberg, J.M., "A Review of the Placement and Quadratic Assignment Problems", SIAM Review 14, 1972, pp. 324-342

[Hart96] Hartong, W., "Entwicklung eines Hardwaremoduls zur Verarbeitung von Testvektoren für ein Fehleremulationssystem", Studienarbeit, Institute of Microelectronic Systems, University of Hannover

[Haye72] Hayes, J. PP., " Fault Modeling", IEEE Design & Test, 1985, pp. 88-95

[HeRo94] Heusinger, PP., Ronge, K., Stock, G., "Handbuch der PLDs und FPGAs", Franzis Publishers, 1994, pp. 160-169

[HiSC82] Hitchcock, R., Smith, G.L., Cheng, D.D., "Timing Analysis of Computer Hardware", IBM Journal of Research and Development, 1982, Vol. 26, No. 1, pp. 100-105

[Kaut68] Kautz, W. H., "Fault Testing and Diagnosis in Combinational Digital Circuits", IEEE Transactions on Electronic Computers, Vol. EC-17, 1968, pp. 352-366

[KeLi70] Kernighan, B.W., Lin, S., "An Efficient Heuristic Procedure to Partition Graphs", Bell System Technical Journal, September 1970, pp. 291-307

[KFAC95] Karlsson, J., Folkesson, PP., Arlat, J., Crouzet, Y., et al.,"Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications, 1995, pp.150-151

[KGVe83] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.PP., "Optimization by Simulated Annealing", Science 220, 1983, pp.671-680

[KhHu93] Khan, U. R., Owen, H. L., Hughes, J. L., "FPGA Architectures for ASIC Hardware Emulator", Proc. 6th IEEE ASIC Conference, 1993, pp. 336

[KLDJ94] Karlsson, J., Liden, PP., Dahlgren, PP., Johansson, R., et al., "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms", IEEE Micro, Vol. 14, No. 1, 1994, pp. 8-32

[KoBe57] Koopmans, T.C., Beckmann, M.J., "Assigned Problems and the Location of Economic Activities", Econometrica 25, 1957, pp.53-76

[KrHa87] Krishnamurthy, B., Harel, D., "Is There Hope for Linear Time Fault Simulation?", Fault Tolerant Computing Symposium, FTCS, 1987, pp. 30-35

[Kron96] Krone AG, "The Prototyper: A Breakthrough in ASIC Emulation", Report 1996

[Kupk98] Kupka, H., "Anwendung und Implementierung des Simulated Annealing zur optimalen Plazierung von Fehlerinjektoren", University of Hannover, 1998

[LeHa92] Lee, H.K., Ha, D.S., "HOPE: An Efficient Parallel Fault Simulation for Synchronous Sequential Circuits", Proc. of the 29th Design Automation Conference, DAC, 1992, pp. 336-340

[LeHa93] Lee, H.K., Ha, D.S., "New Techniques for Improving Parallel Fault Simulation in Synchronous Sequential Circuits", Proc. of International Conference on Computer-Aided Design, ICCAD, 1993, pp. 10-17

[Leng90] Lengauer, T., "Combinatorial Algorithms for Integrated Circuit Layout", Teubner, 1990, pp. 251-302

[Li97] Li, R., "Implementierung des Min-Cut Plazierungsalgorithmus zur Verteilung von Fehlerinjektoren", University of Hannover, 1997

[Lips89] Lipsett, R.,"VHDL: Hardware Description and Design", Kluwer Academic Publishers, 1989

[MaAl93] Mahlsteht, U., Alt, J., "Simulation of Non-Classical Faults on the Gate-Level Fault Simulator COMSIM", Proc. International Test Conference, ITC, 1993, pp. 883-892

[Mahl95] Mahlsteht, U., "Deterministische Testgenerierung für Gatterverzögerungsfehler unter Berücksichtigung der minimal erkennbaren Fehlergröße", University of Hannover, PhD Thesis, 1995, pp. 31-32

[McCl71] McCluskey, E. J., Clegg, F. W., "Fault Equivalence in Combinational Logic Networks", IEEE Transactions on Computers, Vol. C-20, No 11, 1971, pp. 1286-1293

[Meal55] Mealy, G.H., "A Method for Synthesizing Sequential Circuits", Journal of Bell Systems, Vol. 34, 1955, pp. 1045-1079

[Mei74] Mei, K. C. Y., "Bridging and Stuck-at Faults", IEEE Transactions on Computers, Vol. C-23, No 7, 1974, pp. 720-727

[Micz89] Miczo, A., "Digital Logic Testing and Simulation", Wiley, 1989

[MKGT92] Miremadi, G., Karlsson, J., Gunneflo, U., Torin, J., "Two Software Techniques for Online Error Detection", Proc. 22nd Fault Tolerant Computing Symposium, FTCS, 1992, pp. 328-335

[Moor56] Moore, E. F., "Gedanken-experiments on Sequential Machines", Automata Studies, Princeton University Press, 1956, pp. 129-153

[MoTh94] Moorby, PP. R., Thomas, D. E., "The Verilog Hardware Description Language", 1994, Kluwer Academic Publishers

[MRRT53] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., "Equations of State Calculations by Fast Computing Machines", Journal of Chemical Physics 21, 1953, pp.1087-1092

[Murg90] Murgai, R., et al., "Logic Synthesis for Programmable Gate Arrays", Proc. 27th Design Automation Conference, DAC, 1990, pp. 620-625

[NaBi88] Naish, PP. and Bishop, PP., "Designing ASICs", Ellis Horwood, 1988

[OnLL89] Ong, C., Li, J., Lo, C., "An Automatic Cell Synthesis Tool", 26th Design Automation Conference, DAC, 1989, pp. 239-244

[Papa91] Papageorgiou, M., "Optimierung", Oldenburger Publishers, 1991, pp. 116-117

[Plic79] Plice, W. A., "Overview of Current Automated Analog Test Design", International Test Conference, ITC, 1979, pp. 128-136

[Quick96] Quickturn Data Book, "PTRUN", 1996

[Quick96a] Quickturn Data Book, "m250", 1996

[Quick98] Quickturn CoBALT Plus Brochure, "CoBALT, CL20000", 1998

[Rose90] Rose, J., Brown, S., "The Effect of Switch Box Flexibility on Routability of Field-Programmable Gate Arrays", Custom Integrated Circuits Conference, CICC, 1990, pp. 27.5.1-27.5.4

[Rose91] Rose, J., Brown, S., "Flexibility of Interconnection Structures in Field-Programmable Gate Arrays", IEEE Journal of Solid State Circuits, Vol. 26, No. 3, 1991, pp. 277-282

[Roth66] Roth, J.P., "Diagnosis of Automata Failures. A Calculus and a Method", IBM Journal of Research and Development, Vol. 9, No. 2, 1966

[RuSa89] Russel, G. and Sayers, I.L., "Advanced Simulation and Test Methodologies for VLSI Design", Van Nostrand Reinhold, 1989

[Rutm72] Rutman, R., "Fault Detection Test Generation for Sequential Logic Heuristic Tree Search", IEEE Computer Repository Paper, No. R-72-187, 1972

[SaYo95a] Sait, S.M., Youssef, H., "VLSI Physical Design Automation - Theory and Practice", McGraw-Hill, 1995, pp. 37-79

[SaYo95b] Sait, S.M., Youssef, H., "VLSI Physical Design Automation - Theory and Practice", McGraw-Hill, 1995, pp. 141-205

[ScKo94] Schlag, M., Kong, J., Chan, PP., "Routability-Driven Technology Mapping for Lookup Table-Based FPGAs", IEEE Transactions of Computer Aided Design of Integrated Circuits and Systems, Vol. 13, No. 1, 1994, pp. 13-18

[ScMe72] Schertz, D.R., Metze, G., "Representation for Faults in Combinational Digital Circuits", IEEE Transactions on Computers, Vol. C-21, 1972, pp. 858-866

[ScTr87] Schulz, M.H., Trischler, E., Sarfert, T.M., "Socrates: A Highly Efficient Automatic Test Pattern Generation System", International Test Conference, ITC, 1987, pp. 1016-1026

[SeBa97] Sedaghat-Maman, R., Barke, E., "A New Approach to Fault Emulation", Proc. of the 8th International Workshop of Rapid System Prototyping, RSP, 1997, pp. 173-179

[SeBa98] Sedaghat-Maman, R., Barke, E., "Real Time Fault Injection Using Logic Emulators", Proc. of Asia and South Pacific Design Automation Conference, ASP-DAC 1998, pp. 475-480

[Seda97a] Sedaghat-Maman, R. "Fehleremulation mit Logikemulationssystemen", SICAN - Herbsttagung Mikroelektronik - Mikrosysteme, 1997, pp.221-225

[Seda97b] Sedaghat-Maman, R., DFG-Research Report, Report period: 15.3.1995-15.3.1997, No. Ba 812/3-1, 1997

[Seda97c] Sedaghat-Maman, R., DFG-Research Report, Report period: 15.3.1997-15.3.1998, No: Ba 812/3-2, 1998

[Seda98] Sedaghat-Maman, R., "Eine Neue Methode zur Fehleremulation", ITG-Fachtagung Mikroelektronik für die Informationstechnik, 1998, pp. 77-81

[Seda98a] Sedaghat-Maman, R. "Fault Emulation with Optimized Assignment of Circuit Nodes to Fault Injectors", Proc. IEEE International Symposium on Circuits and Systems, ISCAS, 1998, pp. 135-138

[SeLe87] Sechen, C., Lee, K., "An Improved Simulated Annealing Algorithm for Row-Based Placement", Proc. IEEE International Conference on Computer-Aided Design, ICCAD, 1987, pp.478-481

[Selle68] Sellers, F., "Analyzing Errors with the Boolean Differences", IEEE Transactions on Electronic Computers, Vol. EC-17, 1968, pp. 678-683

[Sesh65] Seshu, S., "On an Improved Diagnosis Program", IEEE Transactions on Electronic Computers, Vol. EC-12, 1965, pp. 76-79

[Spir85] Spiro, H., "Simulation integrierter Schaltungen", R. Oldenburg Publishers, 1985

[Stein61] Steinberg, L., "The Backboard Wiring Problem: A Placement Algorithm", SIAM Review 3, 1961, pp. 37-50

[StGr76] Stephenson, J., Grason, J., "A Testability Measure for Register Transfer Level Digital Circuits", Proc. International Symposium on Fault Tolerant Computing, FTCS, 1976, pp. 101-107

[Thor92] Thorpe, T.W., "Computerized Circuit Analysis With Spice: A Complete Guide to Spice With Applications", John Wiley & Sons, 1992

[TiBu83] Timoc, C., Buehler, M., "Logical Models of Physical Failures", Proc. IEEE International Test Conference, ITC, 1983, pp. 545-553

[Timo79] Timoc, C.C., Lawrence, M.H., "Fault Simulation: An Implementation into Hardware", Proc. IEEE International Test Conference, ITC, 1979, pp. 291-295

[UlBa74] Ulrich, E.G., Baker, T.G., "Concurrent Simulation of Nearly Identical Digital Networks", Computer, Vol. 8, No. 4, 1974, pp. 39-44

[Wads78] Wadsack, R. L., "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits", BELL System Technical Journal, Vol. 57, No. 5, 1978, pp. 1449-1474

[WaEi85] Waicukauski, E. B., Eichelberger, E. B., "Fault Simulation for Structured VLSI", VLSI System Design, Vol. 6, No. 12, 1985, pp. 20-32

[Waxm89] Waxman, R., "VHDL Links Design, Test, and Maintenance", IEEE Spectrum, May 1989, pp. 40-45

[Wilk94] Wilkins, B. R., "Testing Digital Circuits", Chapman & Hall, 1994, pp. 162-165

[WoLe88] Wong, D., Leong, H., Liu, C., "Simulated Annealing for VLSI Design" Kluwer Academic Publishers, 1988

[Wund91a] Wunderlich, H. J., "Hochintegrierte Schaltungen: Pruefgerechte Entwurf und Test", Springer Publishers, 1991, pp. 110

[Wund91b] Wunderlich, H. J., "Hochintegrierte Schaltungen: Pruefgerechte Entwurf und Test", Springer Publishers, 1991, pp. 182

[Xili94]	XILINX Data Book, " The Programmable Logic", 1994
[Zycad94a]	Paradigm RP, Fremont, Zycad Corporation, 1994
[Zycad94b]	Paradigm ViP, Fremont, Zycad Corporation, 1994
[Zycad94c]	Paradigm XP, Fremont, Zycad Corporation, 1994