NOTE TO USERS

This reproduction is the best copy available.

UMI®

u Ottawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Dhruv Biswas
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A System-on-a Chip Testing Methodology

TITRE DE LA THÈSE / TITLE OF THESIS

Sunil Das
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Emil Petriu
CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

V. Groza

T. Kwasniewski

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES / DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

A System-on-a Chip Testing Methodology

By

Dhruv Biswas

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirement for the degree of

Master of Applied Science
School of Information Technology and Engineering
Ottawa – Carleton Institute for Electrical Engineering
University of Ottawa

© Dhruv Biswas, Ottawa, Canada, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-11220-4
Our file Notre référence
ISBN: 0-494-11220-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

In this thesis, we present a system-on-a chip testing methodology. The system consists of a wrapper, test access mechanism and the cores under test. The cores include the ISCAS sequential and combinational benchmark circuits. At the gate level, stuck at fault model is used to detect faults. The wrapper separates the circuit under test from other cores. The test access mechanism transports the test patterns or test vectors to the desired circuit under test and then transports the responses back to the output pin of the SOC. The faults are then injected using the fault simulator that generates test for the circuit under test. Out of the many TAM design methods, we implemented the TAM as a plain signal transport medium, which is shared by all the cores in the system-on-chip. Once the dedicated TAM lines are set to the circuit under test, fault simulation is done. Each circuit in an SOC is independently tested for its fault coverage. The isolation of the circuit under test from the others is taken care by the program running in the background. We were able to simulate the whole SOC testing and get satisfactory fault coverage for the circuits under tests.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my supervisor **Dr. Sunil.R.Das** for his patient guidance, numerous help and continued encouragement during the progress of my research work. His overly enthusiasm and integral view on research and his mission for providing quality work, has made a deep impression on me. I owe him lots of gratitude for having me shown this way of research.

I would also like to thank my co-supervisor **Dr. Emil M Petriu**, Dean of School of Information Technology and Engineering, University of Ottawa, for providing me excellent computing, simulation environments and research grants towards my thesis work, to attend seminars and ultimately pursue knowledge. It gave me a lot of motivation while working in this techno-equipped and sophisticated environment during my two years of research. It gives me immense pleasure to reflect this in my thesis work.

I am particularly thankful to **Dr. Mansoor Assaf,** who has been constantly helping me out throughout the two years. I was able to get his support whenever I was stuck at a problem or needed clarification on any related research topics. He also helped me to understand the various simulating and compiling tools used in my thesis work. At the outset, I would also like to thank **Dr. Voicu Groza** and **Dr.Amiya Nayak**. By taking their graduate courses, I was able to have a very strong base and clear concept in the Embedded Systems and System on Chip field, which helped me enormously to support and implement my thesis work. My special thanks to my colleague **Rami Abielmona,** who was always available to help me with design software tools and providing feedback on various research topics. I owe special thanks to colleagues **Liwu Jin** and **Chuan** of the Systems Science department for their help in the fault injection techniques.

Finally I would like to thank my parents who are the root cause of my success and for providing me constant encouragement and inspiration. Without their love and blessing nothing would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES......iv
LIST OF TABLES......vii
ACRONYMS......ix

1. INTRODUCTION......1

1.1 THESIS MOTIVATION......1
1.2 THESIS OBJECTIVE AND HYPOTHESIS......3
1.3 ORGANIZATION OF THESIS......4

2. SYSTEM ON CHIP (SOC) TESTING: A CHALLENGE......6

2.1 RISE OF SYSTEM ON CHIP......6
2.2 SOC RELATED PROBLEMS......8
2.3 SOC TESTING......11

3. WRAPPER AND TEST ACCESS MECHANISM (TAM)......14

3.1 WRAPPER......14
3.2 SCALABLE ARCHITECTURE......17
3.2.1 WRAPPER CONTENTS......18
3.2.2 WRAPPER MODES OF OPERATION......19
3.3 TEST ACCESS MECHANISM (TAM)......19
3.3.1 TAM ARCHITECTURES......22
3.3.2 PREVIOUS WORK......30
3.3.3 OUR APPROACH......34
3.4 SUMMARY......41

4. CORE TEST GENERATION......42

4.1 FAULT SIMULATION......42

4.1.1 FAULT SIMULATION DESIGN PROCESS......44

4.2 DEFINITIONS AND TYPE OF FAULTS......46

4.3 FAULT MODELS......47

4.4 FAULT INJECTION TECHNIQUE......50

4.4.1 HARDWARE IMPLEMENTED FAULT INJECTION......50

4.4.2 SOFTWARE FAULT INJECTION......52

4.4.2.1 EXAMPLE SEQUENTIAL CIRCUIT......53

4.4.2.2 EXAMPLE COMBINATIONAL CIRCUIT......57

4.5 SUMMARY......60

5. SYSTEM TEST ENVIORNMENT......61

5.1 SYSTEM OVERVIEW......61

5.2 PARSING......64

5.3 ALTERA MAXPLUS II DESIGN AND DEVELOPMENT SOFTWARE...70

5.3.1 ALTERA MAXPLUS II DESIGN ENVIORNMENT......71

5.3.2 DESIGN FLOW UNDER ALTERA MAXPLUS II......73

5.3.3 MAXPLUS II APPLICATIONS......74

5.4 INPUT VECTOR FILE THROUGH MAXPLUS II......76

5.5 GRAPHIC DESIGN FILE: COMBINATIONAL SOC (SOC_COMB) 80

5.6 GRAPHIC DESIGN FILE FOR SEQUENTIAL SOC (SOC_SEQ)......80

5.7 GRAPHIC DESIGN FILE FOR MIXED SOC (SOC_MIXED)......80

5.8 SYSTEM CONFIGURATION......84

5.9 RUNNING THE PROGRAMS......84

5.10 SUMMARY......88

6. EXPERIMENTAL RESULTS..89

6.1 SEQUENTIAL CIRCUIT RESULTS (SOC_SEQ)..............................89
6.2 COMBINATIONAL CIRCUIT RESULTS (SOC_COMB)........................94
6.3 DETERMINISTIC TEST GENERATION FOR COMBINATIONAL CIRCUITS (MINTEST VECTORS USED)......................................98
6.4 COMBINATIONAL & SEQUENTIAL CIRCUITS RESULTS (MIXED_SOC)..101
6.4.1 MIXED_SOC_1...101
6.4.2 MIXED_SOC_2...107
6.4.3 MIXED_SOC_3...111
6.4.4 MIXED_SOC_4...116

7. CONCLUSION..119

7.1 SUMMARY OF THE THESIS..119
7.2 THESIS CONTRIBUTION...120
7.3 SUGESSIONS FOR FUTURE RESEARCH...................................120

8. REFERENCES..122

9. PAPERS PUBLISHED AND SUBMITTED IN REFERED JOURNALS AND CONFERENCES.. 125

APPENDIX – C PROGRAM SOURCE CODE......................................126

LIST OF FIGURES

FIGURE 2.1 EVOLUTION OF SOC....7
FIGURE 3.1 IEEE P1500 SECT WRAPPER....17
FIGURE 3.2 P1500 SCALABLE ARCHITECTURE....18
FIGURE 3.3 TAM IN SOC TEST ARCHITECTURE....20
FIGURE 3.4 OUR SCABALE ARCHITECTURE....21
FIGURE 3.5 SERIAL BASED DIRECT ACCESS TAM....23
FIGURE 3.6 SERIALLY CONNECTED TAP....24
FIGURE 3.7 TAM WITH P1500 WRAPPER....24
FIGURE 3.8 NETWORKED INDIRECT AND MODULAR ARCHITECTURE (NIMA)....26
FIGURE 3.9 TIME DOMAIN MULTIPLEXED TAM [11]....27
FIGURE 3.10 CONNECTING CORES WITH TAM [11]....28
FIGURE 3.11 ADDRESSABLE TEST PORT ARCHITECTURE [15]....28
FIGURE 3.12 TEST ACCESS PORT (TAP) INTERFACE....29
FIGURE 3.13 ARCHITECTURE WITH TAM....30
FIGURE 3.14 ARCHITECTURE WITH TEST PORT....30
FIGURE 3.15 MULTIPLEXING OF THE CORES....31
FIGURE 3.16 TESTING ONE CORE AT A TIME....31
FIGURE 3.17 TEST RAIL CONNECTIONS....33
FIGURE 3.18 TEST ARCHITECTURE....35
FIGURE 3.19 OUR APPROACH TAM DESIGN....36
FIGURE 3.20 TAM CONNECTIONS IN MAXPLUS II....37
FIGURE 3.21 SYMBOL FOR SOC....38
FIGURE 3.22 OUR APPROACH OF WRAPPER DESIGN....40
FIGURE 4.1 FAULT SIMULATION DESIGN PROCESS....44
FIGURE 4.2 AN ALFSR STRUCTURE WITH $P(X) = X^4 + X + 1$....45
FIGURE 4.3 SINGLE STUCK-AT FAULTS....48
FIGURE 4.4 SENSITIZED PATH....50

FIGURE 4.5 HARDWARE FAULT INJECTION TECHNIQUE........................51
FIGURE 4.6 SCHEMATIC DIAGRAM FOR SEQUENTIAL CIRCUIT S27..........53
FIGURE 4.7 VERILOG CODE FOR SEQUENTIAL CIRCUIT S27....................54
FIGURE 4.8 DESIGN FLOW OF COMBINATIONAL AND SEQUENTIAL CIRCUIT TESTING METHOD..56
FIGURE 4.9 VERILOG CODE FOR COMBINATIONAL CIRCUIT C17.............57
FIGURE 4.10 LOG FILE FOR C17 COMBINATIONAL CIRCUIT (C17.LOG) 59
FIGURE 5.1 FLOW DIAGRAM FOR SOC TEST GENERATION.....................63
FIGURE 5.2 VERILOG FILE FOR THE SYSTEM-ON-CHIP, SOC_COMB..........67
FIGURE 5.3 VERILOG FILE FOR THE SYSTEM-ON-CHIP, SOC_SEQ.............69
FIGURE 5.4 MAX PLUS II DESIGN ENVIRONMENT.................................71
FIGURE 5.5 MAX PLUS II APPLICATIONS...74
FIGURE 5.6 MINTEST VECTOR FILE FOR COMBINATIONAL CIRCUIT C432 (C432.VEC)..79
FIGURE 5.7 GRAPHIC DESIGN FILE FOR SOC_COMB (SOC_COMB.GDF) 81
FIGURE 5.8 GRAPHIC DESIGN FILE FOR SOC_SEQ (SOC_SEQ.GDF)............82
FIGURE 5.9 GRAPHIC DESIGN FILE FOR SOC_MIXED (SOC_MIXED.GDF)....83
FIGURE 5.10 SCREEN SHOT OF SOC_COMB.EXE..................................85
FIGURE 5.11 SCREEN SHOT OF COMB_C499.EXE..................................86
FIGURE 5.12 SCREEN SHOT OF COMB_C3540.EXE WITH TEST VECTOR C499.VEC..86
FIGURE 5.13 SCREEN SHOT OF SOC_SEQ.EXE......................................87
FIGURE 5.14 SCREEN SHOT OF SOC_MIXED.EXE...................................87
FIGURE 6.1 NO. OF DFFS, INPUTS AND OUTPUTS OF SEQUENTIAL CIRCUITS...90
FIGURE 6.2 NO. OF WIRES, VECTORS AND TESTING TIME OF SEQUENTIAL CIRCUITS..92
FIGURE 6.3 NUMBER OF INJECTED FAULTS, DETECTED FAULTS AND FAULT COVERAGE OF SEQUENTIAL CIRCUITS.....................93
FIGURE 6.4 NO. OF INPUTS AND OUTPUTS OF COMBINATIONAL CIRCUITS..95

FIGURE 6.5 NO. OF TEST VECTORS, TESTING TIME AND FAULT COVERAGE OF COMBINATIONAL CIRCUITS..96
FIGURE 6.6 NO. OF COLLAPSED, DETECTED AND UNDETECTED FAULTS OF COMBINATIONAL CIRCUITS..97
FIGURE 6.7 NO. OF TEST VECTORS, TESTING TIME AND FAULT COVERAGE OF COMBINATIONAL CIRCUITS (DETERMINISTIC TEST).........99
FIGURE 6.8 NO. OF COLLAPSED, DETECTED AND UNDETECTED FAULTS OF COMBINATIONAL CIRCUITS (DETERMINISTIC TEST)............100
FIGURE 6.9 NO. OF WIRES, INPUTS AND OUTPUTS OF SEQUENTIAL & COMBINATIONAL CIRCUITS...103
FIGURE 6.10 NO. OF TEST VECTORS AND TESTING TIME OF SEQUENTIAL & COMBINATIONAL CIRCUITS...104
FIGURE 6.11 NO. OF INJECTED FAULTS, DETECTED FAULTS AND FAULT COVERAGE OF SEQUENTIAL & COMBINATIONAL CIRCUITS..106
FIGURE 6.12 NO. OF WIRES, INPUTS AND OUTPUTS OF SEQUENTIAL & COMBINATIONAL CIRCUITS...108
FIGURE 6.13 NO. OF TEST VECTORS AND TESTING TIME OF SEQUENTIAL & COMBINATIONAL CIRCUITS...109
FIGURE 6.14 NO. OF INJECTED FAULTS, DETECTED FAULTS AND FAULT COVERAGE OF SEQUENTIAL & COMBINATIONAL CIRCUITS..110
FIGURE 6.15 NO. OF WIRES, INPUTS AND OUTPUTS OF SEQUENTIAL & COMBINATIONAL CIRCUITS...112
FIGURE 6.16 NO. OF TEST VECTORS AND TESTING TIME OF SEQUENTIAL & COMBINATIONAL CIRCUITS...113
FIGURE 6.17 NO. OF INJECTED FAULTS, DETECTED FAULTS AND FAULT COVERAGE OF SEQUENTIAL & COMBINATIONAL CIRCUITS..115
FIGURE 6.18 NO. OF WIRES, INPUTS AND OUTPUTS OF SEQUENTIAL & COMBINATIONAL CIRCUITS...117
FIGURE 6.19 NO. OF INJECTED FAULTS, DETECTED FAULTS AND FAULT COVERAGE OF SEQUENTIAL & COMBINATIONAL CIRCUITS..118

LIST OF TABLES

TABLE 6.1.1 BASIC SPECIFICATIONS OF 10 ISCAS 89 SEQUENTIAL BENCHMARK CIRCUITS...90

TABLE 6.1.2 TESTING TIME AND NUMBER OF DFFS, GATES, INVERTERS, AND WIRES OF BENCHMARK SEQUENTIAL CIRCUITS............91

TABLE 6.1.3 NUMBER OF DFFS, WIRES AND DETECTED FAULTS...............92

TABLE 6.1.4 FAULT COVERAGE, NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS..................................93

TABLE 6.2.1 BASIC SPECIFICATIONS OF 10 ISCAS 85 COMBINATIONAL BENCHMARK CIRCUITS...94

TABLE 6.2.2 TESTING TIME AND NUMBER OF GATES, INVERTERS, AND WIRES OF BENCHMARK COMBINATIONAL CIRCUITS............95

TABLE 6.2.3 NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS...96

TABLE 6.2.4 FAULT COVERAGE, NUMBER OF COLLAPSED FAULTS, DETECTED FAULTS, AND UNDETECTED FAULTS...................97

TABLE 6.3.1 TESTING TIME AND NUMBER OF GATES, INVERTERS, AND WIRES OF BENCHMARK COMBINATIONAL CIRCUITS............98

TABLE 6.3.2 NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS...99

TABLE 6.3.3 FAULT COVERAGE, NUMBER OF COLLAPSED FAULTS, DETECTED FAULTS, AND UNDETECTED FAULTS..................100

TABLE 6.4.1.1 BASIC SPECIFICATIONS OF 2 ISCAS 89 SEQUENTIAL & COMBINATIONAL BENCHMARK CIRCUITS..........................102

TABLE 6.4.1.2 NUMBER OF WIRES, TEST VECTORS AND TESTING TIME....103

TABLE 6.4.1.3 FAULT COVERAGE, NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS……………………………..105

TABLE 6.4.2.1 BASIC SPECIFICATIONS OF 2 ISCAS 89 SEQUENTIAL & COMBINATIONAL BENCHMARK CIRCUITS……………………..107

TABLE 6.4.2.2 NUMBER OF DFFS, WIRES AND DETECTED FAULTS………...109

TABLE 6.4.2.3 FAULT COVERAGE, NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS……………………………...110

TABLE 6.4.3.1 BASIC SPECIFICATIONS OF 2 ISCAS 89 SEQUENTIAL & COMBINATIONAL BENCHMARK CIRCUITS……………………..111

TABLE 6.4.3.2 NUMBER OF DFFS, WIRES AND DETECTED FAULTS………...112

TABLE 6.4.3.3 FAULT COVERAGE, NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS……………………………...114

TABLE 6.4.4.1 BASIC SPECIFICATIONS OF 2 ISCAS 89 SEQUENTIAL & COMBINATIONAL BENCHMARK CIRCUITS…………….……..116

TABLE 6.4.4.2 NUMBER OF WIRES, TEST VECTORS AND TESTING TIME ...117

TABLE 6.4.4.3 FAULT COVERAGE, NUMBER OF INJECTED FAULTS, TEST VECTORS, AND DETECTED FAULTS…………………………....118

ACRONYMS

ALFSR	Autonomous Linear Feedback Shift Register
AHDL	Altera Hardware Description Language
AMS	Analog/Mixed Signal
ATPG	Automatic Test Pattern Generation
ASIC	Application Specific Integrated Circuit
BIST	Built in Self Test
CTL	Core Test Language
CUT	Circuit Under Test
DFF	D-Flip Flop
DFT	Design For Testability
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
IR	Instruction Register
NIMA	Networked Indirect and Modular Architecture
SECT	Standard for Embedded Core Test
SOC	System-on-a-Chip
TAM	Test Access Mechanism
TAP	Test Access Port
VHDL	Verilog Hardware Description Language
VLSI	Very Large Scale Integrated Circuit

Chapter 1
INTRODUCTION

1.1 THESIS MOTIVATION

The technological development is enabling production of increasingly complex electronic systems. All those systems must be verified and tested to guarantee correct behavior. As the complexity grows, testing is becoming one of the most significant factors that contribute to the final product cost. The testing complexity is mainly due to the reduction in the ratio of externally accessible points (primary inputs and outputs) to internal inaccessible points in the circuits [1]. Test development is now seen as a major bottleneck in SOC design, and test challenges are a major contributor to the widening gap between design and manufacturing capability [2]. In all SOC designs, predesigned cores are the most essential components. Today's System-On-Chip (SOC) devices are targeting complex multimedia applications where there is a need for significant amount of Digital Signal Processor (DSP) computing power in order to achieve the various data processing tasks which could include both video and audio. In order to complete these tasks various Intellectual Property (IP) cores, which are optimized for the application in hand are utilized. A wide range of circuits comes within the definition of cores. These circuits may be on-chip functions such as microprocessors, large memory arrays, audio and video controllers, modems, Internet tuner, 2D and 3D graphic controllers, DSP functions, etc. These cores are usually available in synthesizable high-level description language like Verilog or VHDL. There are three forms of cores: soft, firm and hard. **Soft cores** (mergeable and changeable) are reusable blocks of a synthesizable RTL description or a netlist of generic library elements. They are responsible for actual implementation and layout. **Firm cores** (mapped to gates but still changeable) are reusable cores that have been structurally and topologically optimized for performance and area through floor planning and placement. **Hard cores** (mapped to silicon) are reusable blocks that have been optimized for performance, power and size and mapped to a specific process technology. The modern and emerging SOC design techniques are based on the use of

cores. The designer's job is to connect together the various cores in a single chip. The major problem arises to test these multiple cores individually and sometimes the logic to separate the cores. Testing of SOC is mostly done at speeds lesser than the core's operating speed. The core can pass the low-speed test but usually fails when tested at full speed. This problem is even greater in case of SOCs [3]. With the increase in narrow lines and denser circuitry, the chances of defects increase exponentially which affects the circuit delays. Some of the test issues that surround the implementation of the SOC as a whole are: timing verification, lack of scan and BIST, getting at embedded pins, at-speed test, test pattern reliability, access, controllability, observability, reuse of test, reuse of DFT, mixed signal test and synchronization. In SOC testing, isolation of the cores involves electrically detaching the input and output ports of the core from the chip logic (or other core) connected to theses ports. Isolation may be required in two cases: during internal testing of the core and during the testing of the logic outside of the core. During the testing of the core, it may be necessary to ensure that the glue logic or other cores do not interfere with the test of the core. During the testing of the glue logic, it may be necessary to ensure that the core inputs are protected from the effects of these tests. The isolation can be input isolation, output isolation or both.

1.2 THESIS OBJECTIVES

This thesis proposes a testing methodology for core-based digital system. The system is a system-on-a-chip consisting of a wrapper and test access mechanism. The cores include the ISCAS combinational and sequential benchmark circuits.

At the gate level stuck-at fault model is used to detect faults. The wrapper separates the circuit under test (CUT) from other cores, which is assumed to be IEEE P1500 compliant. The test access mechanism plays an important role of transporting the test patterns or test vectors to the desired circuit under test and then transporting the responses to the output pin of the SOC. The faults are injected using the fault simulator that generates test for the circuit under test. We tried to implement the test access mechanism (TAM) as a plain signal transport medium, which is shared by all the cores in the system-on-chip. Once the compilation of the cores is successful, the fault simulation is carried out where the test patters are fed to the cores through the TAM. The selection of the cores is taken care of by the program running in the background. The end result gives us the fault coverage of all the cores being tested. The whole process follows a turn-key approach without any intervention from the designer. A more detailed discussion about wrapper and test access mechanism is presented in Chapter 3.

1.3 ORGANIZATION OF THE THESIS

The thesis is organized as follows:

Chapter 2 discusses about the System-on-chip (SOC) background. We talk about the evolution and rise of SOC, with emphasis on previous work done related to its development. We then talk, in detail, about the SOC related problems and mention why testing SOC is a major challenge.

Chapter 3 discusses about Wrapper and Test Access Mechanism (TAM). It starts with the introduction of the IEEE P1500 standard Wrapper, which is followed by more detailed discussion about Scalable Wrapper Architecture, Wrapper contents, and their modes of operation. The second part of the chapter discusses about the Test Access Mechanism (TAM). We review the different TAM architectures done in previous work. We then talk about the design and implementation of our TAM and more about its architecture.

In **Chapter 4**, we present our core test generation technique. We start the chapter by describing Fault Simulation followed by the fault simulation design process. We briefly describe about the definition and types of faults. We then emphasize on the Fault Models and the Fault Injection Techniques. We explain our approach finally giving examples of the circuits used in the system.

Chapter 5 discusses about the testing environment of the whole system. We start with our system overview, providing flow diagram of the SOC test generation. We discuss in detail the tools we used and the interconnection of the different cores with the TAM. We talk about the three SOCs that we designed, the problems we faced and how we solved them. It is then followed by the test pattern generation technique (pseudorandom and deterministic) which we implemented. Finally, we presented some screen shots of the actual C program that invokes the other tools.

In **Chapter 6** we present all our experimental results, followed by conclusions and future work in this area in Chapter 7.

CHAPTER 2

SYSTEM ON CHIP (SOC) TESTING: A CHALLENGE

We start this chapter by discussing the rise of system on chip, the development which took place in the semiconductor industry from 1980s till now. We then talk in brief about the general problems related to system on chip. Lastly we focus on the main issue on which our work and this thesis are based: Testing of the SOC and related issues.

2.1 RISE OF SOC

As semiconductor technology continues to race down the path prescribed by Moore's Law, the resulting "digital revolution" has become a familiar and ubiquitous part of everyday life. Gordon Moore first postulated his theory on the rate of increasing chip complexity in 1965, and in the years since it has been popularly translated into an expected doubling in computing power every 18 months. This revolution has been marked by a progression from megabits in memory on a chip to gigabits, and the seemingly endless clock rate competition up to gigahertz speeds in microprocessors.

Purely digital chips no longer dominate the fastest growing segments of the semiconductor industry, as IC implementations driven by a convergence of complex digital and analog functions are now the industry's leading-edge designs. The increasing capacity for integration in submicron processes has led to the era of complete SOCs. This new era is now being marked by the combination of complex digital processors and memory plus high-performance analog and mixed-signal functions (in the form of IP cores) on a single chip to form SOCs.

The evolution of SOCs depicted in *Figure 1* shows that in the last 20 years IC design has progressed from discrete ICs, to application-specific ICs, to digital SOC ICs, to today's SOCs.

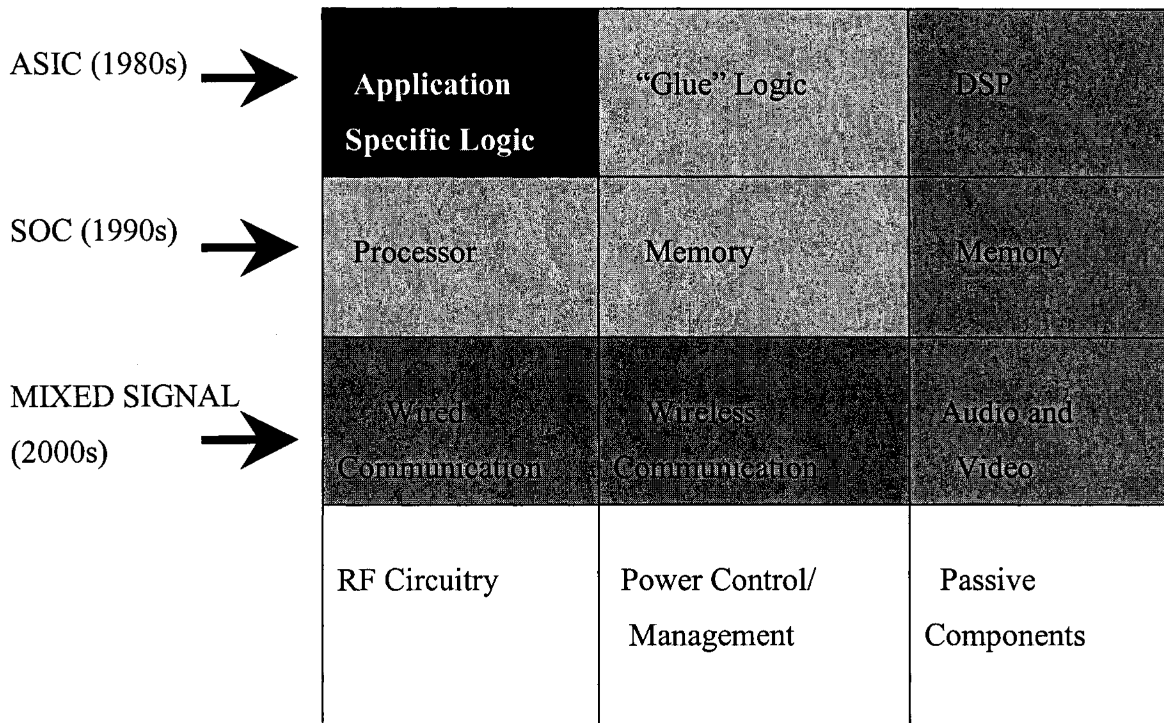


Figure 2.1: Evolution of SOC

The rapid move from digital semiconductors to SOCs has a profound impact on design. Up until the 1980s, there was only one IC design implementation methodology, with no distinction between analog and digital. With the emergence of digital synthesis tools a divergence occurred, leading to separate digital and analog design flows. That divergence continued from the introduction of gate-level synthesis in the 1980s to physical synthesis of digital designs in the 1990s, while analog design remained a mostly handcrafted art.
The top-down approach starting at the architectural level, which has typically been used in digital design, tends to keep process technology details hidden for a very long time. However, analog design has required a bottom-up approach based on the physical characteristics of the process technology. With the growth of mixed-signal and SOC

design, a reconvergence of design methodologies is taking place. Top-down Digital / Mixed Signal architectural design can now start in a converged hardware description language, with the analog/mixed-signal (AMS) extensions to both the Verilog and VHDL standards for digital behavioral and gate-level design. Analog and digital IP blocks can be modeled behaviorally before they are designed in detail.

With rapid advances in semiconductor processing technologies, the density of gates on the die increased in line with what Moore's law predicted. This helped in the realization of more complicated designs on the same IC. Over the last few years, with the advent of bleeding edge technology applications like HDTV and 3rd generation mobile devices, an increasingly evident need has been that of incorporating the traditional microprocessor, memories and peripherals - or in other words the whole system - on single silicon. This is what has marked the beginning of the SOC era.

2.2 SOC RELATED PROBLEMS

Today, the emergence of system-on-chip technology has brought with it a whole spectrum of opportunities and challenges. Opportunities come in the form of drastic reduction in the overall cycle time of the system with superior performance levels; challenges are the result of deep sub-micron complexities, testability issues and time-to-market pressures.

The exponential growth of transistor count with time, widely known as Moore's Law, has created a number of 'gaps' between the fundamental capability of silicon technology and engineer's capacity to exploit it. A longstanding 'gap' has existed, for example, between transistor counts and logic circuits where the regular architecture of the former allow greater components densities than the irregular and the more complex floorplans of the latter.

Atleast the gap between memory and logic densities has remained constant at about two technology generations for some time. The gap between transistor count and productivity

of designer continues to widen, however. Although the introduction of computer aided layouts, hardware description layouts and synthesis tools has helped designers keep pace with advancing technology, we are currently faced with the reality that chips comprising around 10million transistors take around a thousand designer years to complete. As complexity rises, an increasing proportion – currently 60-70% of design time has to be taken up in verifying that the chip stands a good chance of meeting its specification when fabricated, otherwise millions of dollars may be wasted in re-engineering. The productivity of the software engineers in not keeping pace with the silicon technology, despite the fact that ever greater quantities of code are needed to implement complex system control strategies, signal processing algorithms and communications stacks. And new approaches to testing chips are needed urgently if the cost of test is not to rise to such an extent that the final product becomes unaffordable. These are the most common challenges to be faced, from the industry point of view, while testing of a SOC.

Treating large functional blocks – the CPU, DSP, communications interfaces and software stacks - as reusable, preverified components that can be plugged into many different system-on-chips (SOCs) is one way of reducing the amount of low-level design work. Another is platform based design, where most of a carefully-selected basic system is carried forward over many product designs and their derivatives in a particular field of use. These approaches are strongly advocated in major industry players, although they bring their own challenges. The focus on silicon system-on-chip design is therefore moving from low-level implementation to the specific process and the architectural level, at which critical decisions need to be made at a relatively early stage in the design process. The objective is to use tools and languages to achieve a robust initial specification which can not only evolve as the design is processed through to implementation, but also generate system models against which the implementation can be verified and tested at the subsequent stages in the design process. Architectural decisions have a significant impact on system performance. In most applications, compromises have to be made between the all-hardware and all-software extremes and designer need tolls with which to explore and evaluate the options, not only in terms of silicon efficiency but also in terms of power consumption and chip area.

With the number of complex, multimillion-gate design, the IC design and manufacturing industry are forced to identify alternative methods to manage the increasingly large costs of test. We now briefly discuss the two common SOC tests:

Embedded test can help mitigate the need for more complex testers. With embedded test, designers are able to include with their design embedded functions that permit on-chip, at-speed structural test of their silicon devices. These tests initiated, controlled and evaluated on-chip, provide an internal test mechanism, which reduces the need to scale the big-iron testers to the growth curve of silicon device technology.

Memory test capability becomes a significant differentiator as the number of complex, embedded intellectual-property (IP) blocks grows on the silicon devices. Embedded test products, utilizing BIST techniques, can be made to easily scale with hierarchy and complexity with minimal (if any) impact on tester resource requirements. The SCAN/ATPG approach is challenged by increasing complexity and requires significant upgrades to tester resources to accommodate the increase in scan chains and the scan chain depth as the degree of complexity of SOC devices continues to scale.
As the use of such third-party IP cores as processors, embedded Field Programmable Gate Arrays (FPGA) and complex communications cores becomes more prevalent, the self-contained, embedded-test solution for these IP blocks benefits the IC manufacturing companies, testing the devices and the SOC designer integrating the blocks. Dealing with a self-contained, testable block that is usable in a scalable, hierarchical design environment accelerates the test implementation and design closure of these complex pieces of IP in the larger SOC.

2.3 SOC TESTING

Embedded IP cores are the basic building blocks in an SOC. These are pre-designed and pre-verified reusable building blocks, which are a replacement to the previous design styles of using, limited standard cell libraries containing basic logic blocks. Using embedded cores to design SOC has many advantages:

- Shorten the development time by design reuse.
- Enabling of importing external design styles.
- Divides the core community in 2 sectors – core providers and core users, each having their own duties and responsibilities.
- All the above eventually leads to improved design efficiency and time to market.

Today's industry is adapting to core-based design methodology to attain the above advantages. However, the lack of industry wide standards for many aspects of core-based designs, especially in the test, poses a big challenge for core integration. In order to reduce design time, a large number of embedded cores are often switched into SOCs. To facilitate the reuse of test patterns provided by the core vendor, an embedded core must be isolated from surrounding logic, and test access must be provided from the I/O pins of the SOC. Test wrappers form the interface between cores and test access mechanisms (TAMs), while TAMs transport test data between SOC pins and test wrappers. Effective test wrappers and TAMs are therefore important parts of an SOC test infrastructure [4]. In the following chapter we will be discussing in detail the wrapper and TAM. The cores are physically distinct blocks at the layout levels, and their test must be isolated from other cores and the user logic, which glues the cores. So the cores must therefore be logically isolated to be tested independently. This is the most important issue we tried to focus our thesis on. Another issue we focused is the interconnection of the cores. Since we are dealing with several cores the interconnection between the cores is also considered for testing. The test vectors, forming the test sets for each cores, are stored or created in some test 'source' and their test response are stored or analyzed in some test 'sink'. The Test Access Mechanism (TAM) is the connection between the test source, cores and the test

sink [5]. Through the wrapper and the TAM we perform the functional test that ensures that all the faults are detected in the core. This is not only difficult but also time consuming [6].

Some of the most common challenges while testing a core based SOC are [7]:

- Testing the core internally: the complexity of cores has drastically increased and their size reduced (with feature size of 100nm and smaller). The test to detect faults for these cores is, therefore a big challenge. Each core needs different fault models and test pattern generation techniques, which are not so easily available in the market, thus adding more complications.
- Test Strategy: the core provider has to provide sufficient information to the core user regarding the core test, i.e. the same core can be used in any situation without much difficulty.
- Test access: the cores are embedded deep into the chip. To provide test pattern to the desired core from the source and sending the response of the test to the sink is a big challenge. This is the part we need the test access mechanism to connect cores to the source and sink.
- Timing reverification, at-speed testing, test pattern reliability for hard cores, controllability, observability, Reuse of Test and synchronization are some other challenges.

External testers are very commonly used for core based SOC testing. But with the rise of complexity, the testers have to be more sophisticated. Because of the increased complexity of the cores, more sophisticated test equipments, long testing times and rise in cost of production, the hierarchical test methodologies are being adopted. One of the best example is the built in self-test (BIST), which is one of the emerging design for testability (DFT) technique. BIST has many advantages like

- The test pattern generation is on-chip which gives higher controllability,
- The response is also evaluated on-chip giving higher observability.
- Test can be run at circuit speed that is more realistic.
- External test equipment are greatly simplified or even totally eliminated.
- BIST technique can be easily adopted to engineering changes.

- And lastly, it has reduced the development and diagnostic effort at the IC, board and system levels.

In testing a system on a chip two issues comes into limelight. The first to find a method to sequence the test operations among the IP cores and the other to find a way optimize the connection between the cores and the chip input/output pins [6]. It all depends in the designing of the test architecture to get the issues resolved. In many cases a test controller is added which is shared among the cores. The main function of the controller is to select one core at a time and test. Another option is to allocate controllers to each IP blocks and add a common test bus. This option is complicated as each IP block can have its own test, timing requirements.

CHAPTER 3
WRAPPER AND TEST ACCESS MECHANISM (TAM)

In this chapter, we discuss the types of the standard IEEE wrappers. We emphasized on the one that we used in our thesis. We present in detail the standard wrapper modes, contents, architecture and its external and internal interfaces. We then talk about the role and design strategies of Test Access Mechanism (TAM) in SOC testing. We also present the four different TAM architecture followed by the previous work done in this area. We end this chapter by a detailed discussion of how we designed and implemented our wrapper and TAM, which being one of the most important part of the thesis.

3.1 WRAPPER

The IEEE Technical Council on Test Technology has established a working group: P1500 to develop a standard architecture that deals with the issues related to the test access and core isolation in core based SOCs. "Integrated circuits are increasingly designed by embedding pre-designed reusable cores. IEEE P1500 *Standard for Embedded Core Test* (SECT) is a standard-under-development that aims at improving ease of reuse and facilitating interoperability with respect to the test of such core-based ICs, especially if they contain cores from different sources" [7]. SECT focuses specifically on reusability and access of core tests where different cores are provided by different providers. The two main accomplishments of P1500 are the Core Test Language (CTL) and the Scalable Core Test Architecture. P1500 Scalable Architecture Task Force has standardised the test wrapper around the core and its interface to one or more TAMs. The IEEE P1500 proposes two different types of cores for core based testing of embedded systems:

1) IEEE 1500 Compliant Core (Wrapped): This refers to a core that incorporates an IEEE 1500 wrapper function, and comes with an IEEE 1500 CTL program. The

CTL program describes the core test knowledge, including how to operate the wrapper, at the wrapper's external terminals.

The ultimate goal of the core user is to get the compliant core since it provides the full benefits of the interoperability of the cores. In this case the wrapper is built according to the costumer's specification [7]. We assumed that the cores we used in our work are compliant cores.

2) IEEE 1500 Ready Core (Unwrapped): This refers to a core which does not have a complete IEEE 1500 wrapper, but does have a IEEE 1500 CTL program on the basis of which the core could be made '1500 Compliant', either manually or by using dedicated tools. The CTL program describes the core test knowledge at the bare core terminals [7]. This is an intermediate step towards getting a compliant core out of it. Here the core provider leaves the option of customizing the wrapper in the user's hand. So the advantage is that the user could instantiate the wrapper with particular and desired parameter values. Also it could be easily fitted into the system environment. The core provider provides a CTL program that describes the core's testability features, operation and control of these features, the method to apply the test vectors and the exact sequence of operation of the core testing. It is up to the user's discretion to use the CTL programs as they are not the exact requirements for the implementation of the wrapper.

The wrapper is a thin shell around the core, which allows the core to be tested as a stand-alone entity by shielding off from its environment. Also the wrapper allows the environment (the glue logic and the bus architecture) to be tested separately without the core's intervention. The P1500 Scalable Task Force defines the behavior of a standard wrapper to be used with every core that complies with P1500 and an interface between the wrapper and the test access mechanism. The objective of the wrapper is to facilitate core test, interconnect test, and isolation functions by providing switching between different test and diagnostic modes and normal functional mode. The wrapper of a core may interfere with three types of input/output signals:

1) Static control signals for wrapper modes.
2) Digital data and dynamic control signals (to be routed through wrapper cells).
3) Exceptional signals (to bypass wrapper cells), such as clocks, high-speed, asynchronous, analog, and other special-purpose signals.

Standard wrapper behavior can be implemented and provided by the core vendors or it can be added to the core at one of the design stages of SOC. The interface between the wrapper and the test access mechanism is also standardized, but a specific test access mechanism is not defined for SOC since this implementation depends on the test strategy used by the SOC designer. With the design, simulation and test vector files of intended functionality of the core, it is the core provider's responsibility to perform the following: augment the functionality of the wrapper into the core and provide an register transfer level (RTL) simulation model of its behavior, or provide a file of an RTL or gate level design with synthesis scripts and a text description of its operation, which if implemented with the core design according to the text description will perform the P1500 functions. The P1500 functions ensure the following:

- An isolation such that during the core-test mode, any activity in the core and the wrapper will not adversely affect any other section of the chip.
- Access to the core design to use the testability features, test vector application, and test response observation.
- Access to interconnects and user defined logic between the cores or between a core and primary I/Os, such that any testing of interconnects and UDL will not adversely affect the core circuitry.

The figure shows an example IEEE P1500 SECT Wrapper [8]:

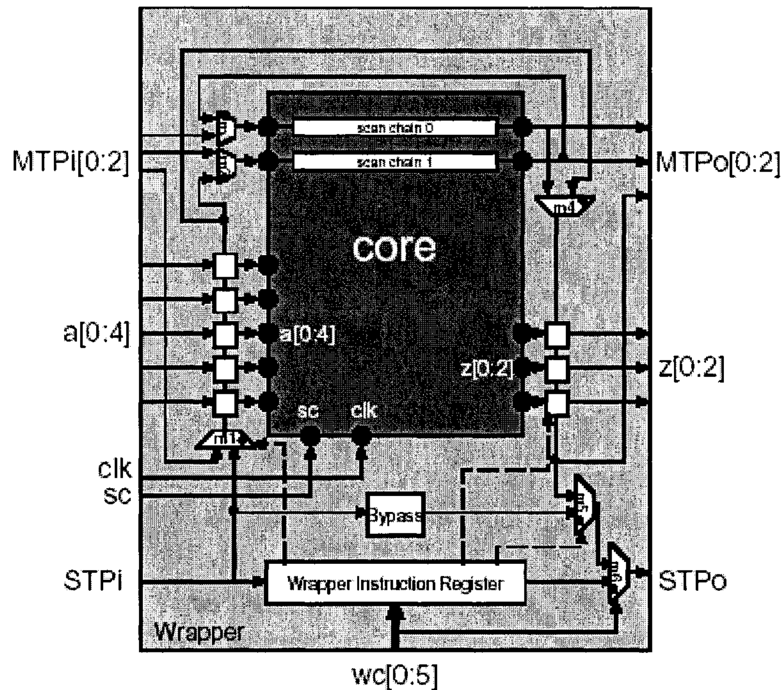


Figure 3.1: IEEE P1500 SECT Wrapper

3.2 SCALABLE ARCHITECTURE

Before we discuss the wrapper and core in detail we give a brief presentation of the P1500 Scalable architecture. This architecture is used for testing cores embedded in a SOC. The architecture is shown in the figure below. It consists of the following:

1) An off or on chip source that generates the test vectors.
2) A similar Sink that evaluates the test responses.
3) TAM: responsible for transporting test vectors from source to core and core to sink.
4) Wrapper: It is an interface that connects the TAM to the core.

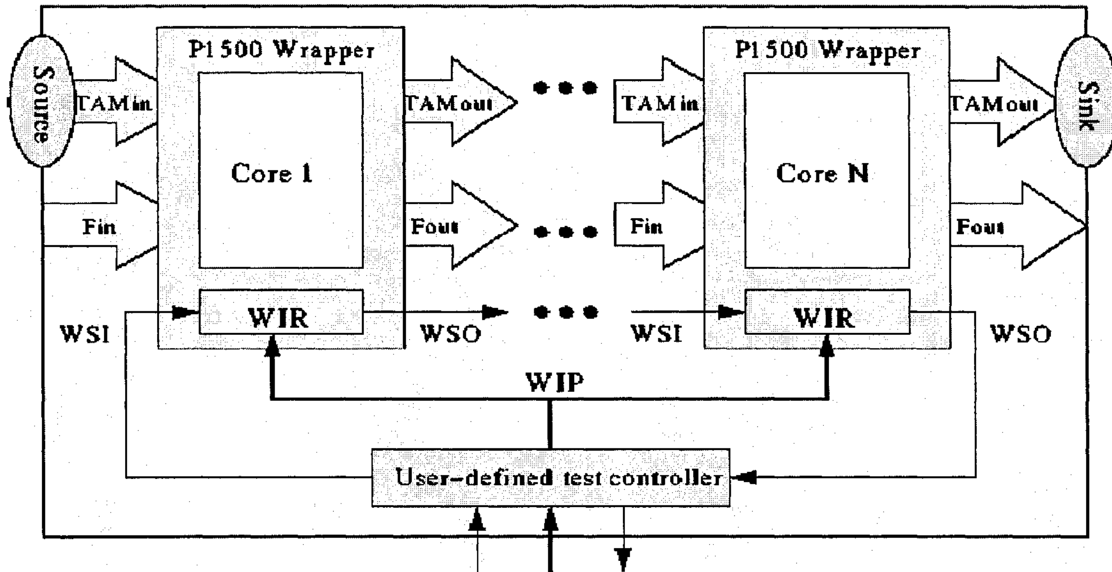


Figure 3.2: P1500 Scalable Architecture

3.2.1 WRAPPER CONTENTS:

The wrapper is usually made of the following elements [7] [8]:

1) *The Wrapper Instruction Register*: It is also termed as the Controller as it is responsible for the total control of the wrapper.

2) *Wrapper Cells*: These are the building blocks for the wrapper boundary register [8]. They provide controllability and observability for the core terminals. As these registers are placed on the boundary of the core, it checks the inputs to the core and the responses from the core.

3) *Wrapper Bypass Register*: When the core is not to be tested for test pattern injection, the bypass register is used [8]. By adding just small amount of hardware, it reduces a large amount of testing time.

3.2.2 WRAPPER MODES OF OPERATION:

1) *Normal or Functional Mode*: In this mode the core is not tested and does normal operation.
2) *Core Test Mode*: Here the CUT, within the wrapper, is tested for faults. The test inputs are injected from the source to the CUT and the responses are transported from the CUT to the sink. This operation is taken care by the wrapper.
3) *Interconnect Test Mode*: In this mode, the interconnections between the wrapper and the CUT are tested. This includes the user-defined logic that glues them together.
4) *Bypass Mode*: In a core based SOC many cores have to be tested. In order to save testing time the cores usually have an internal transparent mode such that data could be transported through them without affecting the core. However, if the core is not provided with such a mechanism the wrapper takes care of this by having a bypass mode. The wrapper makes sure that the test stimuli pass by without affecting the CUT. The bypass register is used for this purpose.

3.3 TEST ACCESS MECHANISM (TAM)

Once the wrapper design is completed or a wrapper is available for the core, we have to deal with two issues, namely test scheduling of the cores, and the design of test access architecture. The test access mechanism is responsible for the transportation of the test data from the system inputs to the core inputs and from the core outputs to the system outputs [9].

The figure below shows the role of TAM in the SOC test architecture.

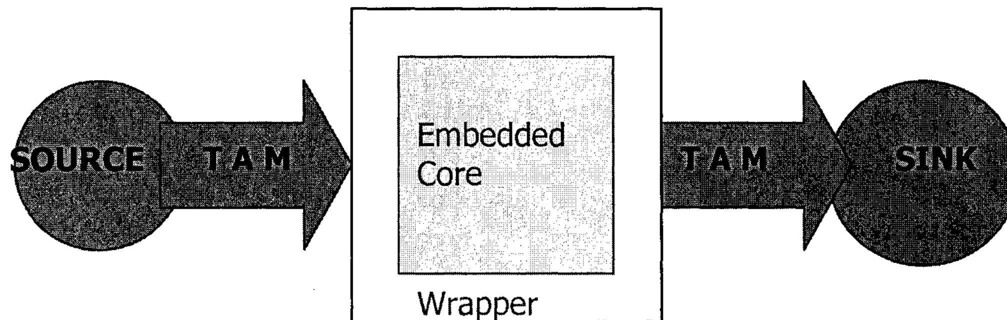


Figure 3.3: *TAM in SOC Test Architecture*

The test access mechanism transports test patterns. It can be used for on-chip transport of test stimuli from a test pattern source to the core-under-test, and for transport of test responses from the core-under-test to a test pattern sink [10].

There are two important concepts related to the TAM. The test pattern source and sink and the core wrapper (as shown in the Figure above). The **test pattern source** is responsible for generating the test vectors or test stimuli for the desired core under test. The test **pattern sink** compares the fault free response to the faulty response of the core under test. The test pattern source and sink can be built on-chip or off-chip. In our case, we implemented this by using the fault simulator, which generates the test vectors, and after getting the test response compares it with the fault free response. The figure below shows the scalable architecture concept we designed.

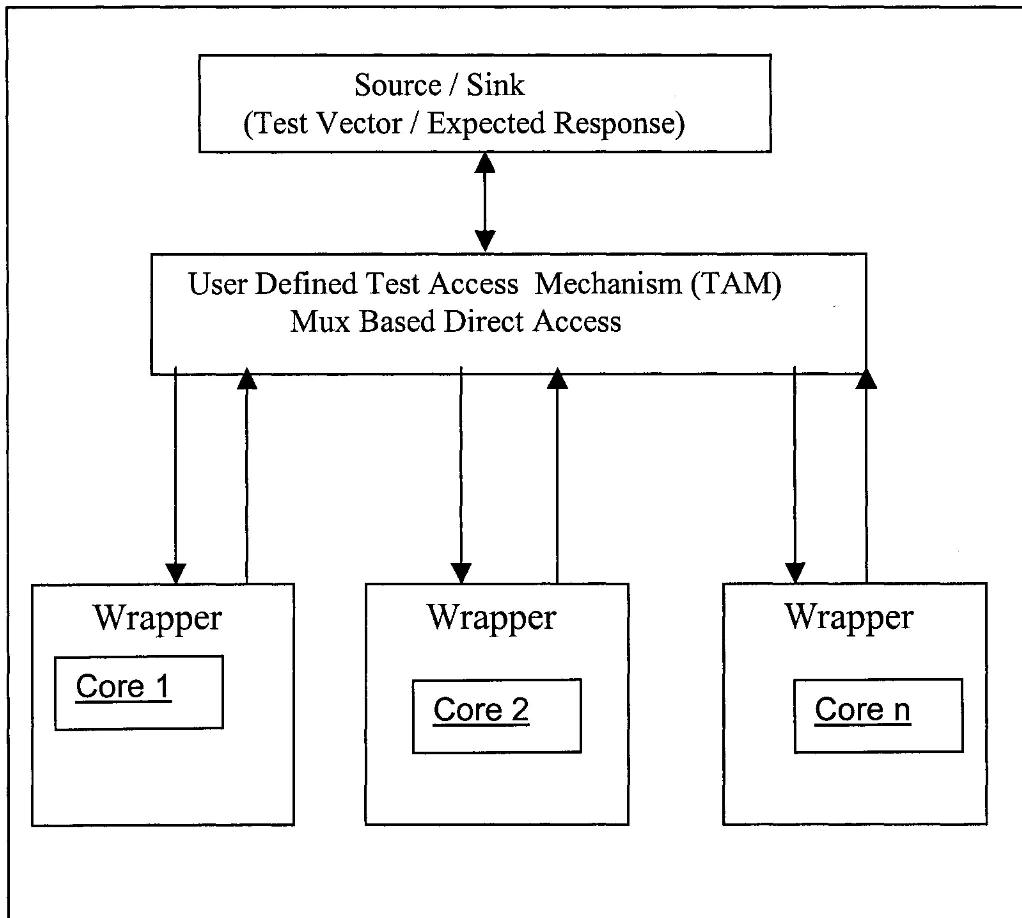


Figure 3.4 : *Our Scabale Architecture*

Test access mechanism refers to the application of the desired test stimulus at the input of the core and getting the response from the core under test. TAM design is of critical importance in SOC system integration since it directly impacts the vector memory depth required on the ATE as well as testing time, and thereby affects test cost [4]. One of the important tradeoff while designing the TAM is between the TAM bandwidth and the cost

of testing. The bandwidth is determined by the bandwidth of the source and the sink and the amount of silicon area spent on the TAM. The wider the TAM bandwidth, the costlier is the testing as it involves more wiring. However, the wider TAM width is not very advantageous if the source is an external ATE (like in our case). The TAM width also affects the test time, as the volume of test vectors transported from the source and the response from the sink depends on the TAM width.

The TAM is used to drive the test vectors from the test source i.e. the fault simulator to the desired core under test (CUT) and to transport to test response from the CUT back to the fault simulator. The selection of the core in the SOC has also been implemented as a part of the TAM. We determined the width of the TAM by the core, which has the maximum number of the input/output pins, within the SOC. There are a lot of issues to be considered at this phase like the bandwidth of the TAM versus the cost of extra wires to implement it, the total test time depending upon the bandwidth of TAM and the test vectors from the source and ultimately depending on the test data for the individual core.

The four general strategies [10] while designing a TAM are:

1. In order to transport the test patterns, the TAM can reuse the existing functionality of or it can be designed using dedicated hardware.
2. TAM can pass through other cores in the SOC or either bypasses them.
3. The SOC can have an independent TAM for every core or can share a TAM for all the cores.
4. TAM can be used as a simple signal transport medium (like assigning it as a bus in verilog) or by adding some control features to it such that it can have different modes of operation.

3.3.1 TAM ARCHITECTURES

There are many ways to design and implement the TAM. Some of the TAM architectures are multiplexed based, serial connection, indirect access and bus-based. In this section we describe in brief the above four architectures [11]:

- **Multiplexed Based Direct Access TAM**

  Multiplexing is done to access the cores in the SOC. To connect the test pins of the chip to the primary input/output pins of the cores, multiplexers are used. By using this method, the core can also be made transparent for testing. Detailed discussion is presented in [12]. Some of the issues related to this method are large overhead area, long test time, non-scalability of the architecture and limited scope of use for future complex SOCs.

- **Serial Based Direct Access TAM**

  This architecture uses the IEEE 1149.1 standard test access port [13]. The ***Figure 3.5*** below shows the architecture of the 1149.1 compliant design. This also includes the test access port (TAP) architecture.

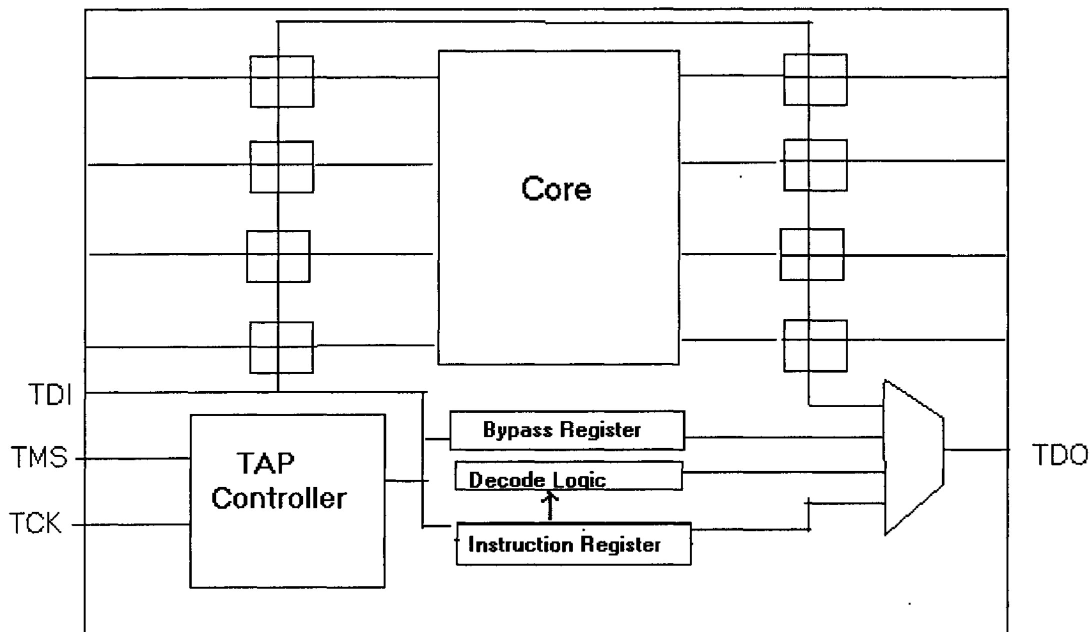


Figure 3.5: *Serial Based Direct Access TAM*

There are four major parts in this design: (1) a TAP controller, (2) an Instruction Register (IR), (3) Decoder and (4) registers to store data. Two input signals, the Test Clock (TCK) and the Test Mode Select (TMS) controls the TAP's operation. Also the Test Data Input (TDI) and the Test Data Output (TDO) are used by the TAP to serially move data into the registers. The Instruction Register (IR) is used to load instruction related to test actions. The Decoder decodes the IR contents to generate the control signals. The advantage of this architecture is that the internal scan, boundary scan, built in self test (BIST) and various industry related design for test (DFT) features can all be accessed and controlled by the TAP. The ***figure 3.6*** below shows the serially connected TAPS to access the individual cores in the SOC:

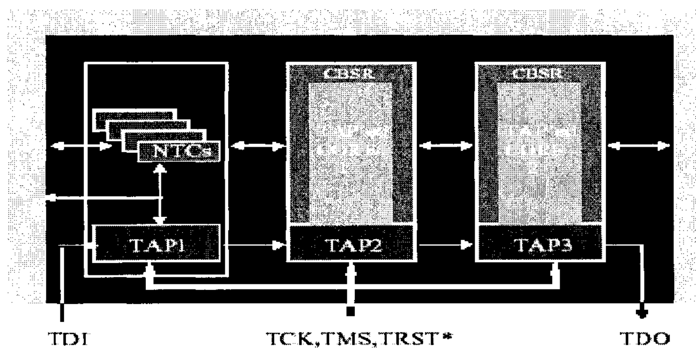


Figure 3.6: *Serially Connected TAP*

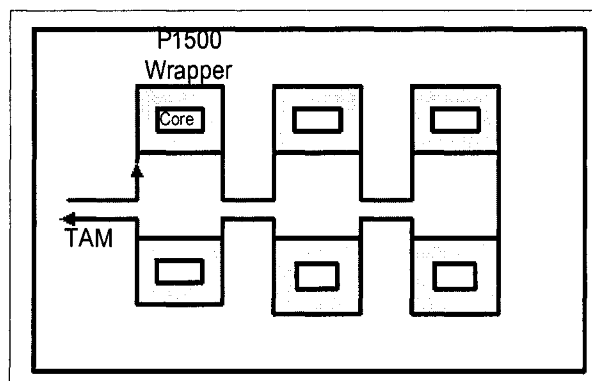


Figure 3.7: *TAM with P1500 wrapper*

In [13], they used the hierarchical test access port which performs the normal 1149.1 compliant TAP operation, coordinates the connection between the test bus and the TAPs of each core and observes the operation of the TAP while being used by the test bus. It is very useful for SOC having less number of embedded cores, as the time taken to serially transport the test vectors is possible. However, as the complexity of the core grows, the testing time becomes a major issue hence raises the cost of testing [11].

- **Indirect Access TAM**

  To emphasis on the modularity, generality and configurability of the TAM architecture, a Networked Indirect and Modular Architecture (NIMA) was proposed [14]. Here they tried to establish an indirect digital communication path through the use of packet switching connection. Both test vectors and test results are converted into packets while they are transferred from source to the destination core and core to sink respectively. On and off chip sources and sinks are used to provide/receive data to/from the cores. Each core has a specified DFT which may be scan-based, or full or partial Built-in-Self-Test (BIST). All test related communications destined/originated to/from the chip need to pass through the on-chip network.

  NIMA is designed such that the communication between source/sink and cores is achieved in a simple 3-layer hierarchical model, i.e., *Application Layer*, *Network Layer*, and *Physical Layer*. Each layer, except the *Physical Layer*, deals directly with its own peer through a virtual link. Moreover, the idea of source routing is used, so the routing path is pre-determined. To simplify the *Network layer*, packets are sent in order and assumed to reach their destinations in order.

  The ***figure 3.8*** below shows the NIMA TAM architecture:

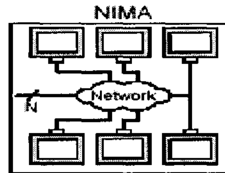


Figure 3.8: Networked Indirect and Modular Architecture (NIMA)[14]

The NIMA TAM requires that test data be pre-processed into packets. Therefore, before testing begins, test data (either test vectors for scan-based modules or commands for the BIST) is formatted into test packets and sent to the designated block. In order to translate the test packets into information understandable to the P1500 wrapper, a NIMA interface module must be developed and attached to each SOC block. A controller within the NIMA interface module generates the P1500 wrapper control signals. Since P1500 does not specify the implementation of such a controller, the TAP controller design from the JTAG standard was used because of its robustness and similarity between the standards. This controller is responsible for sequencing the P1500 control signals such that the P1500 wrapper can perform the operations outlined in the standard. In addition, the NIMA test network parallelizes the test data, so a data funnel is required between the test network and the NIMA interface module in order to down-convert the data width. This enables the packet switching capability of NIMA test network by allowing the network to have a higher bandwidth than required by a single core. Since the test data from the NIMA test network arrives at the NIMA interface in bursts, a buffering scheme is incorporated to ensure the P1500 wrapper gets the data only when it is ready to accept it [14].

- **Bus Based TAM**

The bus-based architecture are the most efficient TAM schemes in present times considering the complexity of the SOCs. The advantages of this architecture are scalability, efficiency in testing time and chip area, flexibility, reconfigurability and efficient handling of cores with BIST. Two proposed methods are the Time

Domain Multiplexed TAM (TDM TAM) [11] and an architecture using addressable test ports [15]. We briefly describe both the methods below.

Time Domain Multiplexed TAM (TDM TAM): This bus based architecture uses logic situated locally at each core to enable or bypass each core. Here the data to be sent to the TAM is divided into frames. Each core is assigned a specific mask enabling the cores to extract the appropriate data bits from the frame. The ***figure 3.9*** below shows the TDM TAM architecture concept:

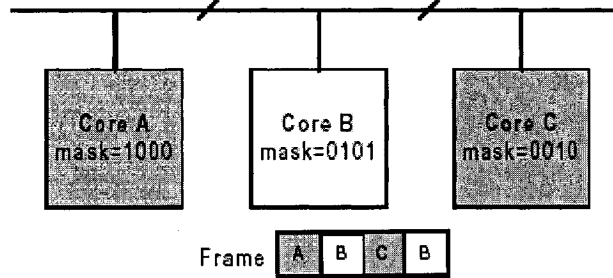
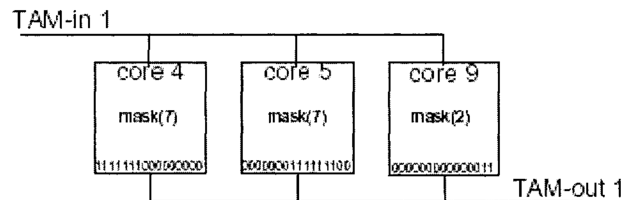


Figure 3.9: *Time Domain Multiplexed TAM [11]*

The figure explains that the frame consists of four bits. The first bit goes to core A, the second and fourth goes to core B and the third goes to core C. So by assigning proper mask and frame to cores, desired data bits can be extracted. Various cores are connected through the TAM as shown in the figure below:



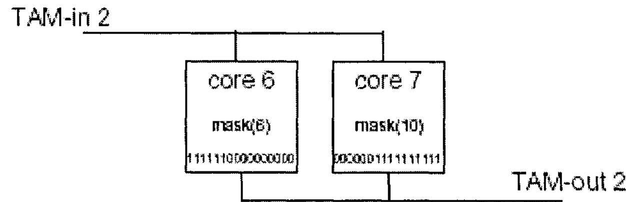


Figure 3.10: *Connecting Cores with TAM [11]*

This scheme uses the TDM method to effectively reduce the TAM area requirements while still achieving good test time performance.

Architecture Using Addressable Test Ports:

In [15], they implemented an addressable test port on each core, which provides the capability of directly addressing a core to be tested and once addressed, effectively testing the addressed core. The test port is scalable giving variations to the test capabilities. The figure below shows the architecture:

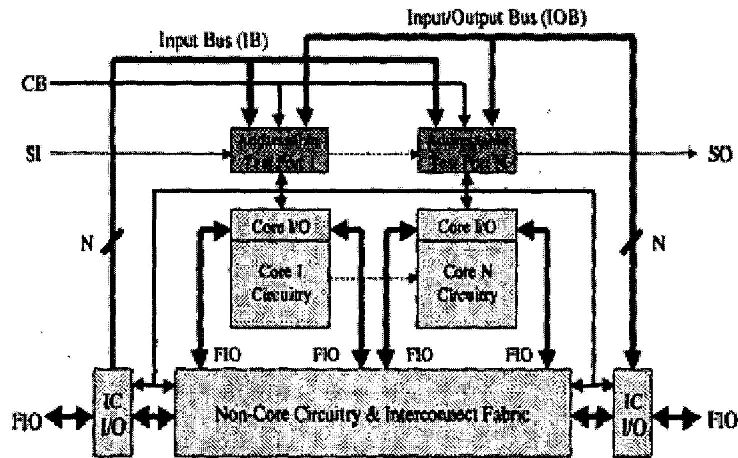


Figure 3.11: *Addressable Test Port [15]*

The architecture includes: (1) externally accessible functional input and output (FIO) signals, (2) IC I/O circuitry connected to the FIO, (3) an interconnect fabric consisting of wires and non-core circuitry connected to IC I/O, and (4) cores I-N each containing core circuitry and core I/O.

The figure below shows the interfacing of the port to the cores:

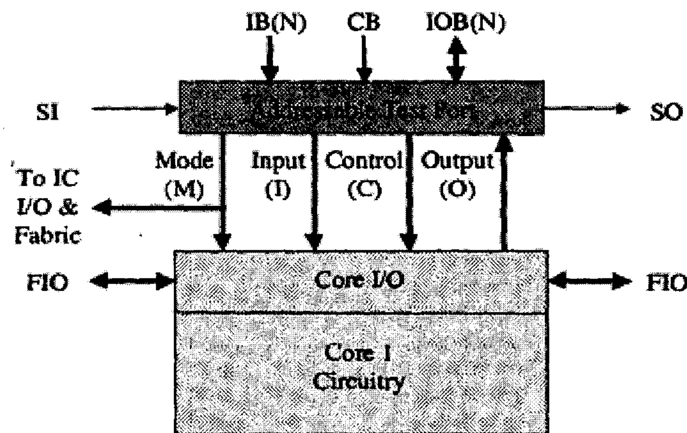


Fig 3.12: TAP interface[15]

The interface between an addressable test port and core is shown consisting of mode (M), input (I), control (C), and output (O) test signals. The addressable test port is connected to the IC I/O via an N-bit wide input bus (IB) and an N-bit wide input/output bus (IOB), and to the IC I/O and fabric via M signals. Further, the addressable test port is connected to serial input (SI), serial output (SO), and control bus (CB) signals which are dedicated for test and accessible external of the IC. The addressable test port is located close to the core to allow the M, I, C, and O test signal interfaces to the core I/O to be localized. In some instances, the addressable test port may be designed as part of the core, such that the test port and core form a completely predesigned unit.

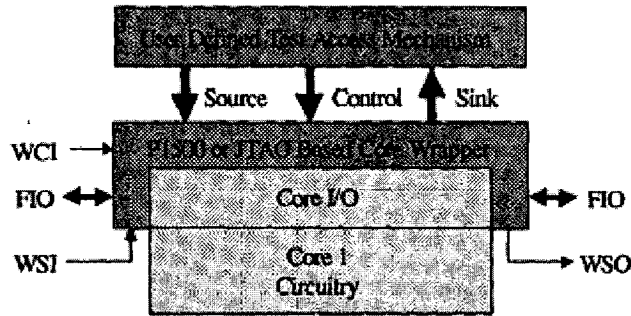


Figure 3.13: *Architecture with TAM[15]*

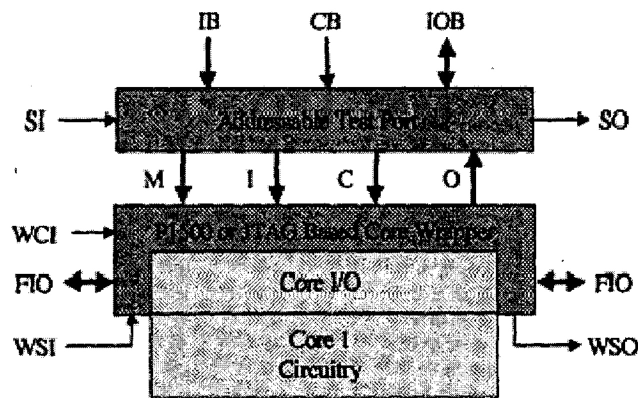


Figure 3.14: *Architecture with Test Port*

In figure 3.14 they replaced the TAM block (Figure 3.13) with their addressable test port. While the addressable test port is classified as a TAM, it differs from other TAMs in that it has local addressing and instruction capability, and resources for testing mixed signal and analog circuitry.

3.3.2 PREVIOUS WORK

In this section, we discuss the several proposed and currently used test access mechanism (TAM) for testing core based systems. The techniques are based on Macro test, transparency of cores, reusing the system bus, multiplexed access, dedicated test bus, boundary scan and least but not the last Test Rail [10].

Iyengar and Chakrabarty [9] also proposed solutions for determination of TAM width. They presented a new wrapper/TAM co-optimization methodology that overcomes the limitations of previous TAM design approach that have addressed TAM optimization and wrapper design as independent problems. The new design approach they presented improved upon previous approaches by minimizing the core testing time, as well as reducing the TAM width required for the core. They also considered the conflict that appears when sharing the test bus for test data transportation. They based their work on the assumption of the test bus model. According to the model, the TAMs on the SOC can operate independently of each other but also the cores on a single TAM are tested sequentially. This is implemented by multiplexing all the cores assigned to a TAM or by testing one core at a time on the TAM, while bypassing the others. This is shown in the figures below:

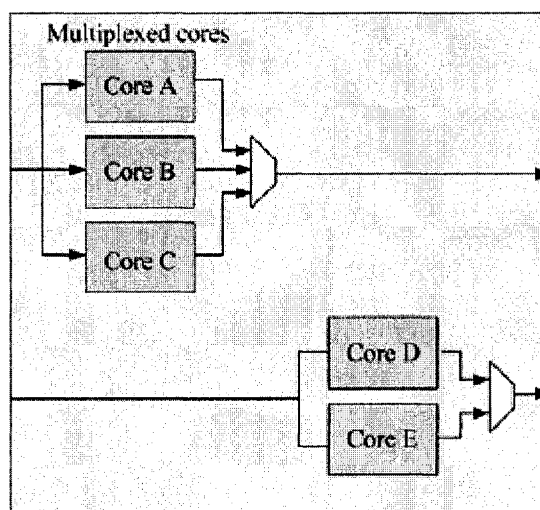


Figure 3.15: *Multiplexing of the cores.*

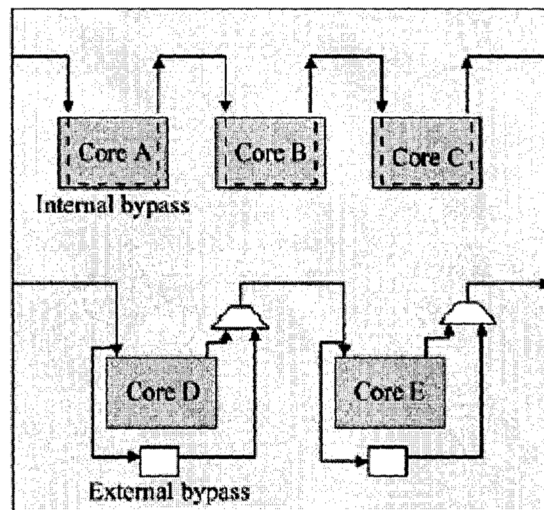


Figure 3.16: *Testing one core at a time*

The advanced microcontroller bus architecture (AMBA) is an approach where all test sets are scheduled in a sequence on a single bus [6].

Varma and Bhatia also proposed bus architecture [16]. They proposed a structured automated test bus methodology which addresses the test access, isolation, interconnect and shadow logic test issues. This methodology combines the testing of both the user-defined logic, also called the glue logic and the individual cores. It is based on the use of a test bus framework. The hardware components, which were used, are:

1) Test data buses
2) Test control bus
3) Test collar cells
4) Multiple Input Signature Registers (MISRs)
5) Test controller

The test buses used are of variable width and provide access to test collar cells at input/output ports. The buses could be input, output or bidirectional type. Since this methodology is a reuse testing method, the frequency of this application used for testing and the test time is determined by the width of the test bus. Their methodology is compatible with IEEE 1149.1 (JTAG) boundary scan technique.

Another important TAM designed proposed is the scalable bus-based architecture called the **Test-Rail** [9]. [17] Proposes a test data transport mechanism called the Test Rail. Three main factors determine their Test Rail width: host pins, test time and silicon area. This architecture is very flexible and can be implemented in many ways as shown in the figure below.

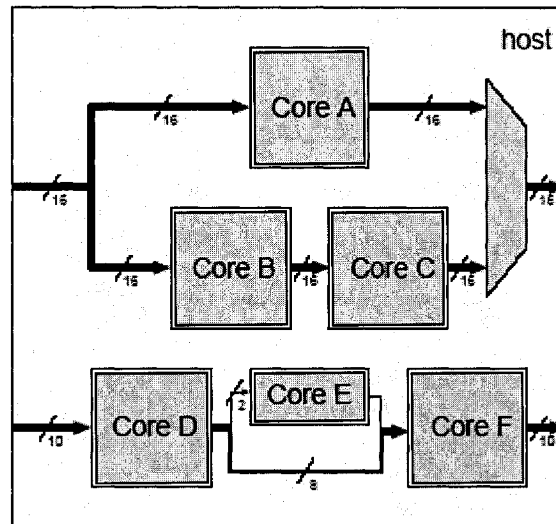


Figure 3.17: *Test Rail connections*

The Test Rail connects the different cores to the host (the system). A core can have a separate test rail assigned to it as in the case of core A. it has a 16 bit bus assigned only to itself. Two cores can share a common test rail as in the case of core B and core C. Theses two core have a common test rail of width 16. Test Rail can also be forked out as in core D. The 10-bit bus is forked to 2 bits and 8 bits. Similarly they can be merged together as done for core F. In case of more than test rails each of them can have access to separate test pins in the system. As shown in the figure core D, E and F have their own private pins. Also from the figure it is evident that the test rails can be multiplexed. Test rails from core A and from core B and C are multiplexed together.

The Test Rail can be connected to the core through the wrapper in three different ways:

1) Parallel Connection: one to one connection is done between the test rail and the core inputs.

2) Serial Connection: The Test Rail is connected to multiple core inputs / outputs by a shift register.
3) Compressed Connection: Here the single test rail is decompressed at the input of the core and the response from the core is compressed at the output of the core, to pass it back to the test rail.

During recent times, integrated TAM design and test scheduling has been attempted in [10]. We tried to implement the bus based TAM which is flexible and scalable and is most promising. However, we faced problem related to minimizing testing time under our TAM architecture.

3.3.3 OUR APPROACH

We implemented the Multiplexer Based Direct Access TAM Architecture. The TAM is used to drive the test vectors from the test source i.e. the fault simulator to the desired core under test (CUT) and to transport to test response from the CUT back to the fault simulator. The selection of the core in the SOC has also been implemented as a part of the TAM. We determined the width of the TAM by the core, which has the maximum number of the input/output pins, within the SOC. The ***Figure 3.18*** below shows the concept of our test architecture:

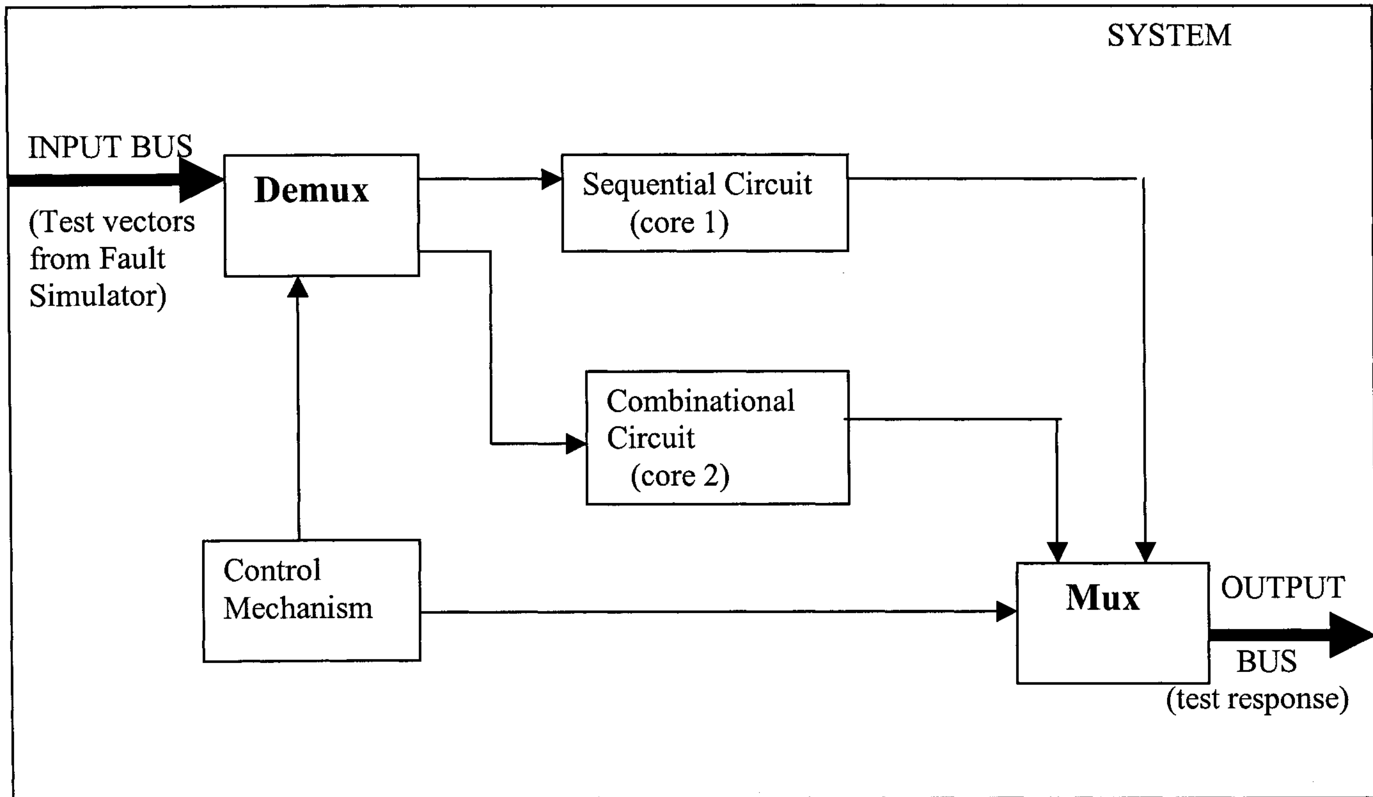


Figure 3.18: *Test Architecture*

The demultiplexer, multiplexer and the control mechanism are coded such that they provide a dedicated test mode for each core. The control signal connects the input bus to the specific sequential circuit (CUT) and the test vectors are passed to the circuit through the bus. Once the fault simulation is done the test response is channeled through the output bus. After all the sequential circuits are tested the same process is repeated for the combinational circuits. The width of the input (and output) bus is determined by the maximum number of the input (and output) pins contained in circuit out of all of the sequential and combinational circuits. The input and output bus uses only the bits required by the CUT at a time, the rest of them are set to high impedance (Z). The advantage of this approach is clear i.e. the embedded core is tested as if it were a stand-alone device.

The figure below shows the TAM architecture we designed:

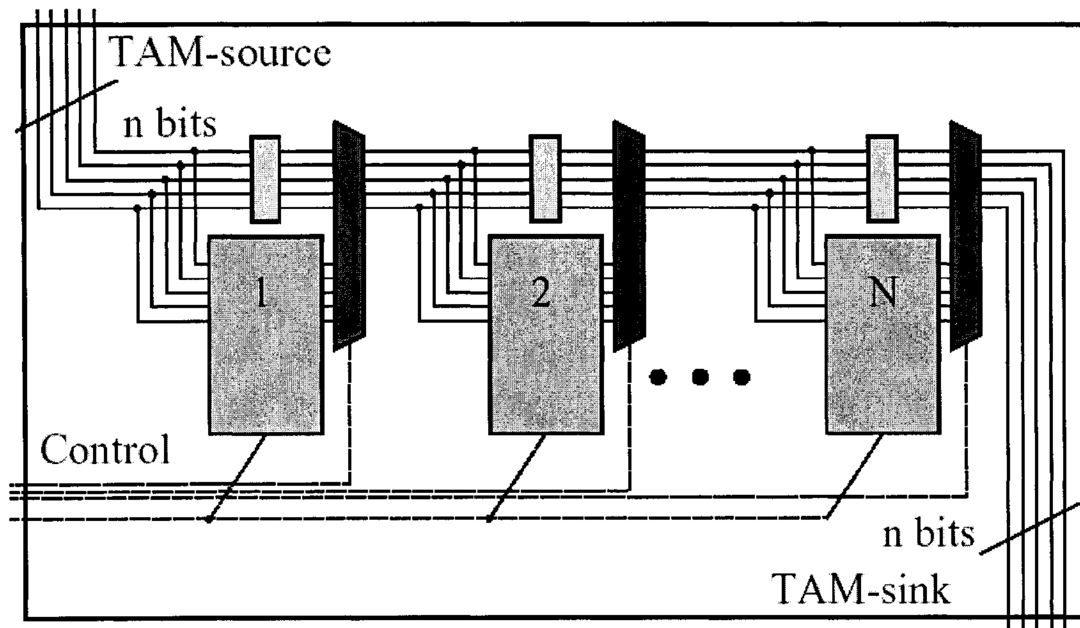


Figure 3.19: *Our Approach- TAM design*

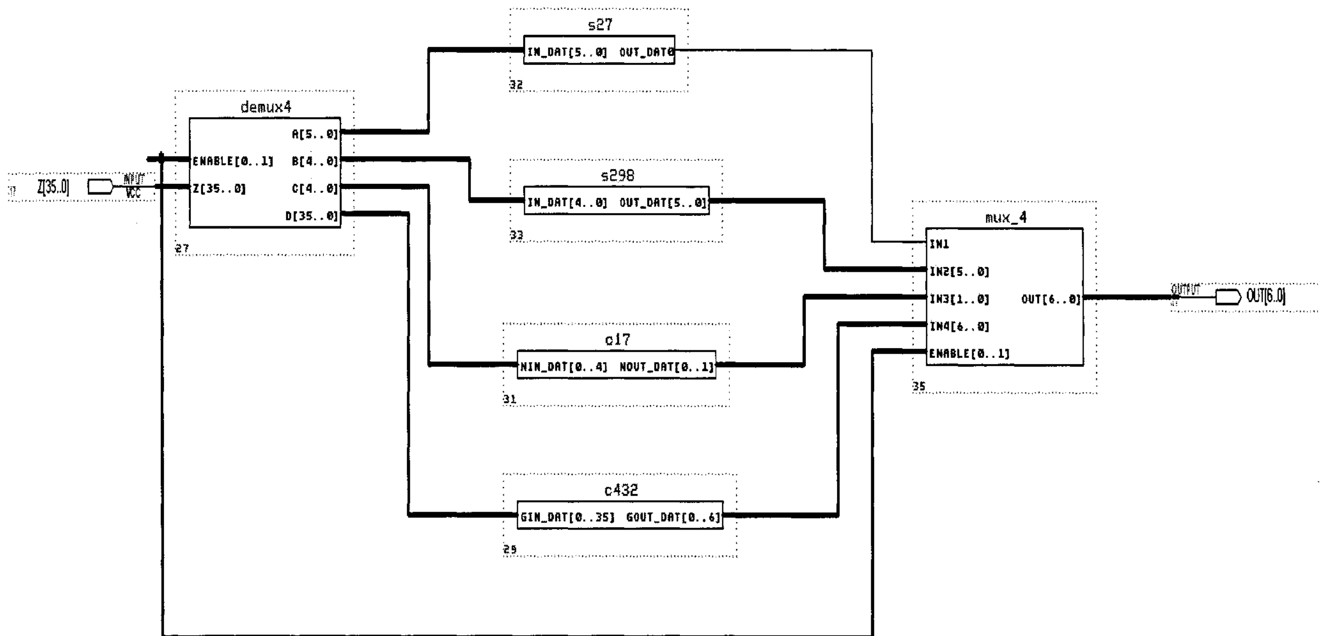


Figure 3.20: *TAM Connections in Maxplus II*

The n bits of the TAM bus depends on the maximum number of the inputs or outputs of the core, the TAM is connected. We used a demultiplexer to connect the n bits of the TAM bus to the core inputs. The multiplexer is used to connect the core outputs back to

the TAM bus. Based on the value of the enable the demultiplexer selects the core and similarly the multiplexer is also connected to the same selected core. In the above example we used 4 cores, 2 sequential circuits (s27 and s298) and 2 combinational circuits (c17 and c432). Out of the four circuits C432 has the maximum number of inputs (36) so the TAM width of the input bus (Z) is set to 36. Also out of the four circuits, C432 has the maximum number of outputs (6), so the TAM width for the output bus (OUT) is set to 6. We can see from the figure that we assigned 36 bits to the input bus (Z) and 7 bits to output bus (OUT). Based on the value of the Enable, Z [35..0] is connected to A, B, C or D. If Z [35..0] is connected to A [5..0], then the rest of the 30 bits are set to high impedance. Similarly the extra bits of Z are set to high impedance while connecting to B, C and D. The outputs of the demultiplexer are connected to the four circuits. The responses of each circuit are then connected to the multiplexer. Based on the same value of the Enable the selected circuit is connected at a time. The symbol file of the SOC looks like the figure shown below:

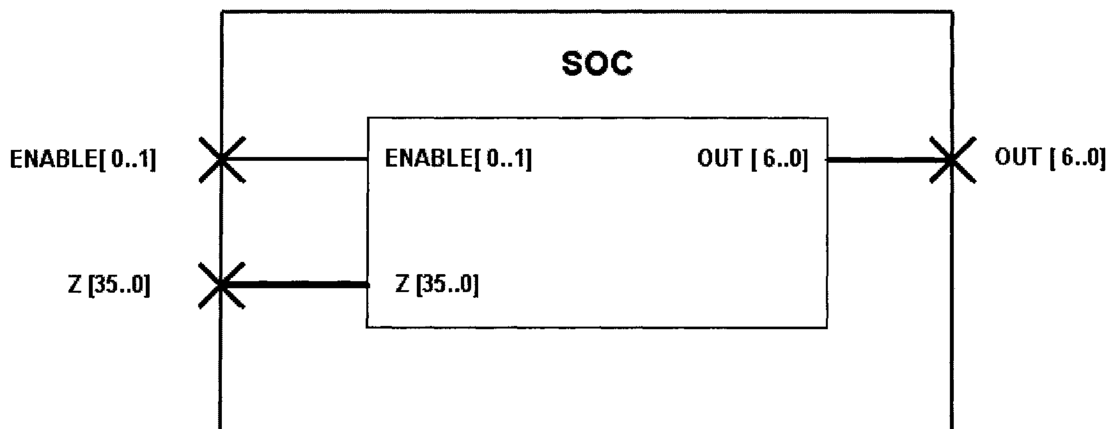


Figure 3.21: *Symbol for SOC*

We designed our wrapper based on the P1500 Compliant Core as shown in the figure below. We used two 2-1 multiplexers and demultiplexers. Our wrapper has two modes of operation:

- The normal mode where the core is not tested and normal input is given to the CUT. When the enable is 0 for both the multiplexers, the wrapper is in normal operation mode.
- The test mode, where once we give random test vectors as input vectors and during the second run we give deterministic test vectors as input vectors. This is performed when the enable of both the multiplexers is 1..

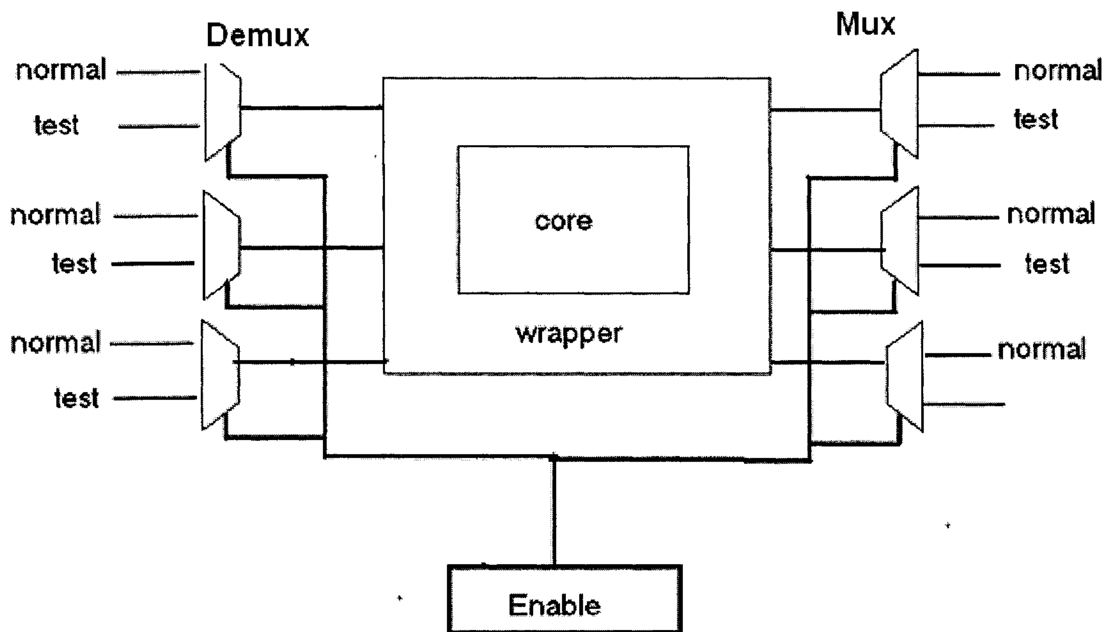


Figure 3.22: *Our approach of Wrapper design*

Depending on the value of the Enable either the normal signal is transported to the core or the test signal. Once the normal signal or test vectors reach the CUT, the responses are transported back to the test access mechanism. We used demultiplexers to channel the test vectors. All the multiplexers and the demultiplexers are enabled at the same time in-order to be at the same operating mode. We added a multiplexer at each input of the core and a demultiplexer at each output of the core. The two inputs to the multiplexer are the normal signal and the test signal. In test signal, we provide random test vectors at one time and deterministic test vectors the second time. We did not implement the bypass mode in the wrapper as the test access mechanism (TAM) takes care of that. The TAM selects one core at a time to test.

3.4 SUMMARY

In this chapter, we discussed the various TAM architectures, the wrapper operations and their modes of operation. We reviewed some previous work done in this area. We finally presented the approach we took in deigning the TAM and the Wrapper.

CHAPTER 4
CORE TEST GENERATION

In this chapter, we will discuss about fault simulation, its definition and its types. We then talk about the fault simulation design process we used. We present in brief the definition and types of faults followed by detailed discussion on fault models. We emphasized the single stuck-at fault model, which we implemented. We also define the basic terminologies used in this context like fault collapsing, etc. We conclude this chapter by presenting fault injection technique we used, being the most important part in the whole test generation process.

4.1 FAULT SIMULATION

Simulation refers to modeling of a design, its functions and performance. Some of the types of simulation are: logic or switch level, timing, circuit and fault. We will be discussing here about fault simulation. Fault simulation normally refers to obtaining the response of a system from a model. In general, simulating a circuit in the presence of faults is known as fault simulation. For electronic systems, both hardware and software models are used. While hardware models or breadboards are the traditional way of verifying designs, software models are becoming more popular due to their economy and accuracy derived largely from the advances in computing [1]. It is an integral part of a logic circuit design. Through fault simulation, we can determine the effects of physical failures on the behavior of the digital circuits. It also helps us to grade the quality of the test sets designed to detect the presence of such failures [18]. It is used in test generation to determine the fault coverage of a test set. If we have a circuit and a set of faults in the circuit together with a set of tests (input vectors or stimuli), then fault simulation is used to decide which faults will be detected by which tests. Based on the result, new test vectors are then generated to cover the undetected faults, and fault simulation is performed again to obtain better fault coverage. This cycle of test generation and

simulation is repeated until a satisfactory test set is obtained. A vector contains the values of all input signals. Whenever one or more inputs change, a new vector must be specified. In most circuits, input signal changes are synchronized with some periodically changing clock signal. Thus, the input stimuli are a sequence of vectors applied at specified periodic intervals. The simulator computes the response of the circuit [1].

Two common fault simulations are the gate level and the behavioral fault simulation. In our thesis we are using the gate level fault simulation. The difference lies in the modeling technique used and the level of abstraction applied. The hardware description languages like the VHDL and Verilog are supported by simulation tools and are inputs to both behavioral and gate level fault simulation blocks. The fault simulation technique is described in Figure 4.1 below.

4.1.1 FAULT SIMULATION DESIGN PROCESS

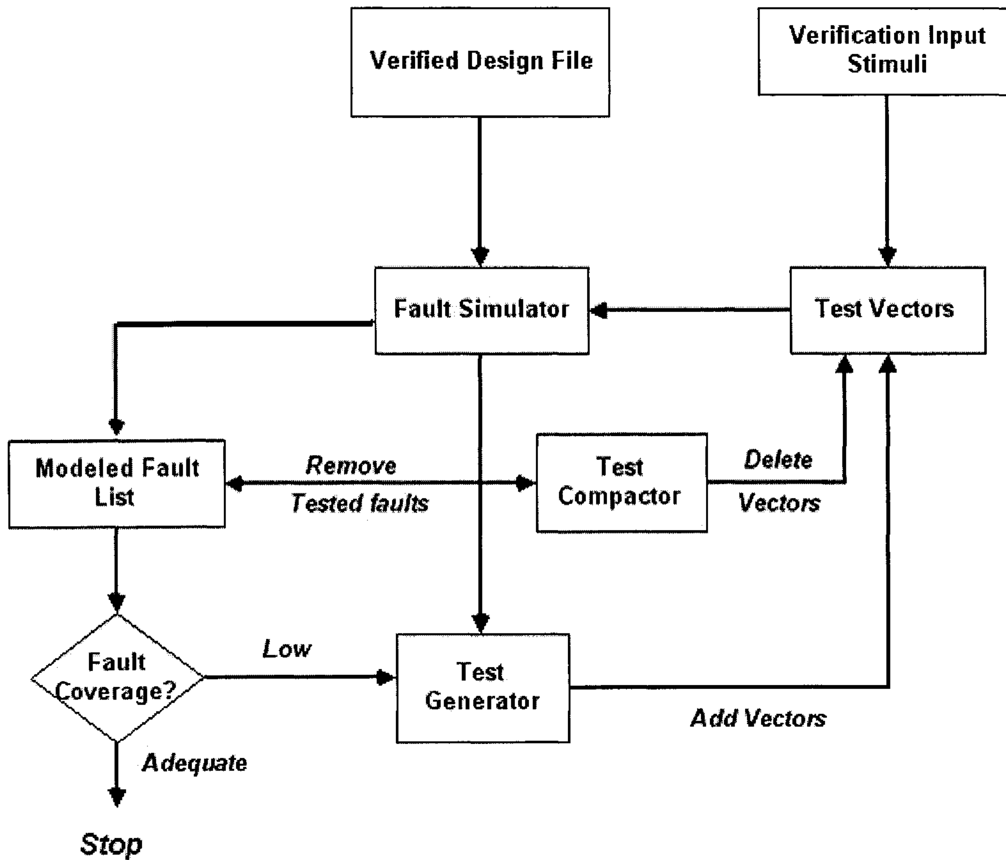


Figure 4.1: *Fault Simulation Design Process*

The figure above shows our fault simulation process.

Verified design file: We have this file designed in hardware description language Verilog, for ISCAS 89 sequential and ISCAS 85 combinational benchmark circuits.

Fault Simulator: The fault simulator is written in C language that is the key component in the whole test generation process. The working of the fault simulator is discussed in detail later in the chapter.

Test Vectors: We provide test vectors to CUT that are specific for each CUT. We performed deterministic test pattern generation (MINTEST vectors were used) and random test pattern generation (pseudorandom test vectors were used). The deterministic test vectors we used are the MINTEST vectors which are developed by the IGATE research group of *The University of Illinois at Urbana-Champaign* [19] [26].

Test Generator: The pseudorandom test vectors are generated by test generator. In BIST techniques, theses vectors are used because of the ease of their generation and ease of on-chip storage [29]. Many practical circuits need few test vectors for full coverage of single stuck faults. Using practical algorithms (such as PODEM, FAN), these test sets can be generated on-chip at a low hardware cost with high fault coverage. For hardware implementation of random signal generator, the autonomous linear feedback shift register (ALFSR) can be used to generate pseudorandom test vectors. An ALFSR is a serial connection of D flip-flops with no external inputs and Exclusive-OR (XOR) gates providing the feedback. A four-stage ALFSR is shown below in Figure 4.2.

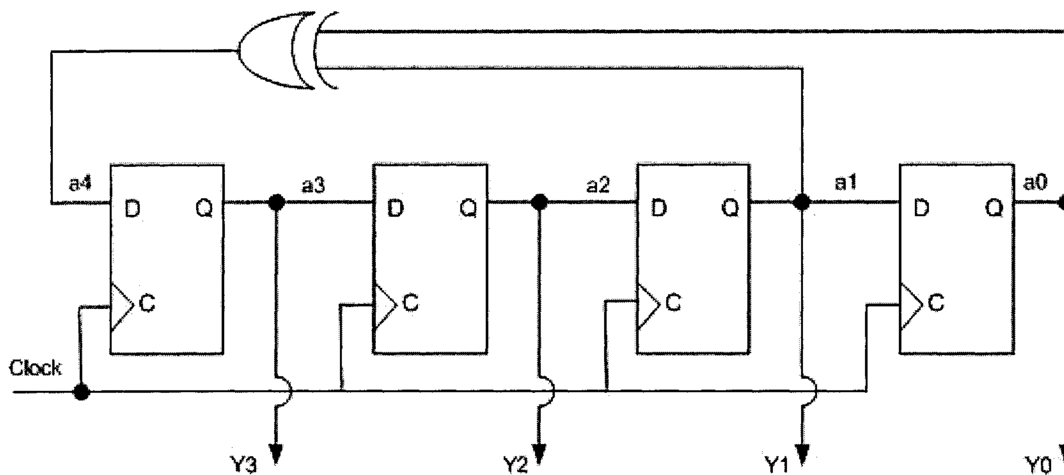


Figure 4.2: *An ALFSR structure with* $p(x) = x^4 + x + 1$.

In our thesis, however, we implemented the random signal generator by software simulation rather than using the hardware technique. According to the specific sequential circuit selected, the random number is saved in the vector file for that circuit in a format compatible to the development software we used (Altera Max Plus II). The random signal generator is developed in C programming language. The random_number() function is a fast **pseudorandom** number generator. When we need a random number, we simply call the random_number() functions. Initializing the function with a certain seed produces exactly the same series of random numbers on all platforms. The random_number() functions then returns high quality equally distributed random numbers [20].

Test Compactor: The test compactor is inbuilt in the fault simulator.

Fault List or Table: Every time we inject a fault into the CUT, it is updated in the fault list if it is detected. We also store a list of faults injected. When all the faults are injected into the CUT, the fault coverage is calculated, which is the total number of faults detected (from the fault list) upon the total number of faults injected multiplied by 100. After the fault coverage is calculated, the fault simulation process is stopped.

4.2 DEFINITION AND TYPES OF FAULT

At the defect level, an enormous number of different failures could be present and it is almost impossible to analyze them as such. Thus failures are grouped together with regards to their logical fault effect on the functionality of the circuit and this leads to the construction of logical fault models as the basis for testing algorithms. More precisely, a *fault* denotes the physical failure mechanism, the *fault effect* denotes the logical effect of a fault on a signal carrying net, and an *error* is defined as the condition (or state) of a system containing a fault (deviation from correct state). Faults can be further divided into classes as *permanent* faults, that is, fault is in existence long enough to be observed

at test time, as opposed to *temporary* faults (transient or intermittent), which appear and disappear in short intervals of time, or *delay* faults which affects the operating speed of the circuit.

Permanent faults are further classified into catastrophic (open and short) faults and parametric (due to disturbance in the process parameters) faults. When a catastrophic fault occurs, the topology of the circuit is changed. Parametric faults are also tested, because processing/layout disturbance is inherent in any manufacturing process. The performance parameters of each manufactured circuit will be deviated from the nominal one due to this and therefore corresponds to a different point in each parameter space. If it is within fault-free space, then circuit is treated as fault-free, otherwise it is considered as faulty circuit.

4.3 FAULT MODELS

A fault model is an abstraction that captures the behavior of the original system. It must be simple and lead to accurate conclusion. A key component of test generation process is the fault modeling, which abstractly describes the expected faulty behaviors. The fault model provides fault detection goals for the automatic test generation (ATG) process and enables evaluation of a test sequence's fault detection qualities [21]. All fault models are extrapolations of some presumed type of defect behavior. A stuck-at fault model, for example, is an extrapolation of a short to power or ground. Fault models are used differently in test generation and fault diagnosis. In test generation, it is important that the tests that cover the target faults are able to detect the defective parts. Single stuck-at tests have been successful in this area. Although defects cannot be modeled directly and the fault models cannot mimic all possible defects, it is nevertheless possible to successfully identify defects using fault simulation. To achieve this, fault models are made as realistic as possible, i.e. fault behavior must conform closely to defect behavior [22].

Typical fault models are: functional faults model, single stuck-at fault model, transistor open and short fault model, memory fault model, delay fault model and analog fault model. The single stuck-at fault model is the most common fault model. Inserting shorts to power or ground into fault-free circuits usually creates stuck-at defects. We implemented the single stuck-at fault model to detect faults in our CUT. In the sequel of the chapter we will be discussing in brief about our model and the terms used during our fault simulation process.

Single Stuck-at-faults: Three properties define a single stuck-at fault:

1) Only one line is faulty.

2) The faulty line is permanently set to 0 or 1.

3) The fault can be at an input or output of a gate.

Figure 4.3 below is an XOR circuit that has 12 fault sites (•) and 24 single stuck-at faults.

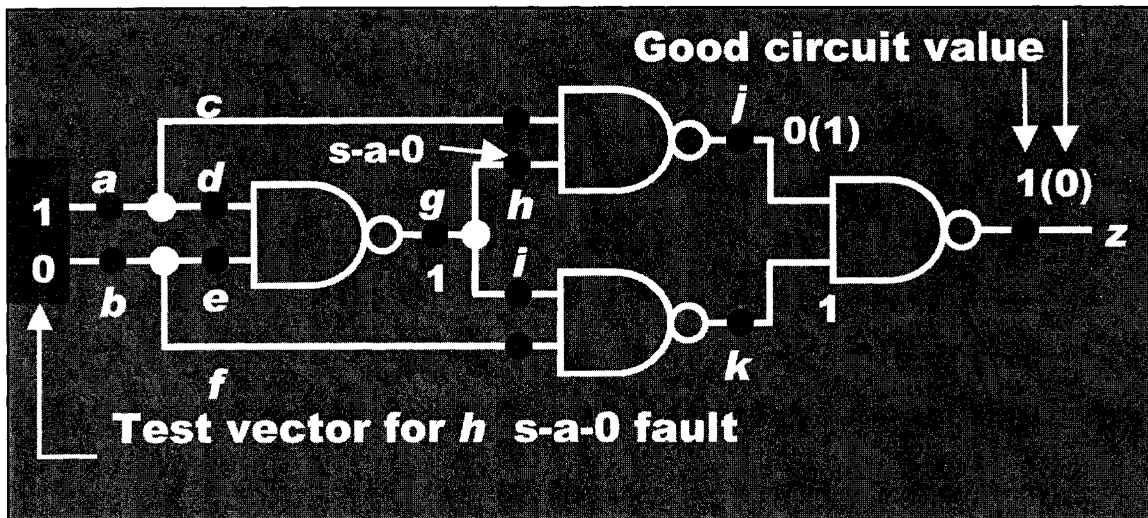


Figure 4.3: Single Stuck-at Faults.

Fault Equivalence: Two faults F1 and F2 are equivalent if all tests detecting F1 also detect F2. Equivalent fault are also called indistinguishable faults and have exactly the same set of tests.

Fault Dominance: Given two faults F1 and F2, if all tests of fault F1 detect fault F2, then F2 is said to dominate F1. If fault F2 dominates F1, then F2 is removed from the fault list. This process is called dominance fault collapsing.

Fault Collapsing: All single faults of a logic circuit can be divided into disjoint equivalence subsets, where all faults in a subset are mutually equivalent. A collapsed fault set contains one fault from each equivalence subset [30].

Two faults are called equivalent if their effect is indistinguishable at the outputs of a circuit, which means that any test detecting one of them will also detect the other. Selecting one representative fault from each class of equivalent faults is called equivalence fault collapsing. Computationally, this is an intractable problem in its general form. In practice, incomplete yet substantial fault collapsing may be possible with little computational effort. A good example is the collapsing of faults associated with the inputs and output of a logic gate. A three-value logic simulator is effective in finding equivalent faults that may be separated by several gates in a circuit. The three values used in simulation are 0, 1, and X (an unknown or don't care). In equivalence fault collapsing, we only collapse the faults that are indistinguishable. If we are prepared to give up on diagnostic resolution (ability to distinguish between faults), more collapsing is possible. This is accomplished by using the concept of fault dominance. It has been shown that it is sufficient to consider single faults on checkpoints in a circuit as long as all such faults are detectable. In actual circuits, however, there can be a small number of faults, referred to as undetectable or redundant faults, which are not detected by any test. The presence of such faults, by definition, does not cause malfunction in the circuit [1].

Path Sensitization

A line whose value in the test X changes in the presence of the fault j is said to be sensitized to j by X. A path composed of sensitized lines is called a sensitized path.

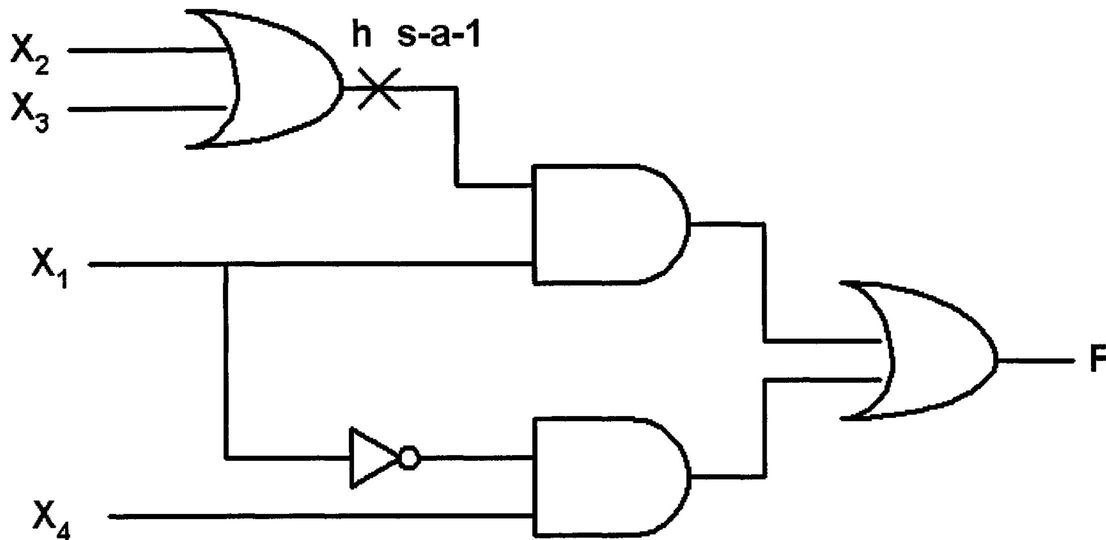


Figure 4.4: Sensitized Path

In order for an input vector **X** to detect a fault a s-a-j, j=0,1, the input **X** must cause the signal a in the normal (fault-free) circuit to take the value j. For example, to detect **h** s-a-1, we need to satisfy the condition X2 + X3 = 0. The error signal must be propagated to the output.

4.4 FAULT INJECTION TECHNIQUE

Fault injection is the most important part in the whole test generation process. It is important for evaluating the dependability of the computer systems. Fault injection can be done in both hardware and software levels.

4.4.1 HARDWARE-IMPLEMENTED FAULT INJECTION

It uses additional resources to introduce faults into the target system's hardware. Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories:

Hardware fault injection with contact: the injector has direct physical contact with the target system, producing voltage or current changes externally to the target chip.

Hardware fault injection without contact: the injector has no direct physical contact with the target system. Instead, an external source produces some physical phenomenon, such as heavy-ion radiation and electromagnetic interference, causing spurious currents inside the target chip [23].

The hardware fault injection technique is imperative in order to iteratively inject faults to every mutually exclusive wire, and to test for both the stuck-at-0 and stuck-at-1 faults. For that matter, a plan was devised that is based on Figure 4.5 below.

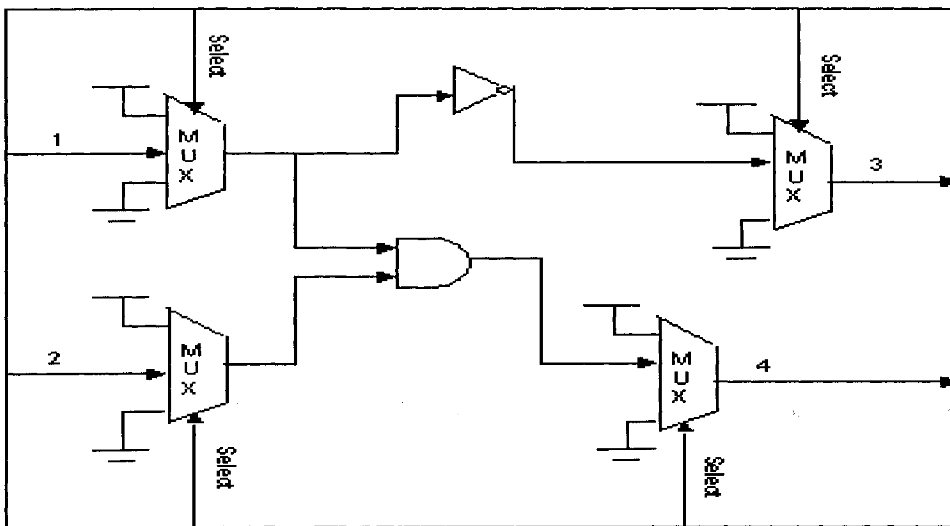


Figure 4.5: *Hardware Fault Injection Technique.*

In fault injection scheme, every mutually exclusive wire now has a multiplexer (MUX) introduced within it, which allows us to either run the wire as it is (normal operation), or inject stuck-at-0 or stuck-at-1 faults (test mode). If the value of the select signals of any multiplexer are at "00", then we run the wire as it is, while if the values are at "01", then we inject a stuck-at-1 fault indicated by the logical 1 value coming into the multiplexer, and if the values are at "10", then we inject a stuck-at-0 fault indicated by the logical 0 value coming into the multiplexer. Finally, if the values are at "11", then we again assume normal operation of the wire [20].

4.4.2 SOFTWARE FAULT INJECTION

These techniques are attractive because they do not require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection. In our thesis we implemented the above hardware fault injection scheme through entire software method. Since the cores we are dealing with are ISCAS 85 combinational and ISCAS 89 sequential benchmark circuits, the fault injection in both kinds of circuits are done in three phases. The first phase injects faults in the input wires of the circuit. The second phase injects faults in the internal wires of the circuit and finally, in the third phase, faults are injected in the output wires of the circuit. The pseudocode description of the algorithms that perform the input, internal and output wire testing for a sequential circuit is given in [20]. The compilation and simulation of the circuits is done in Altera Max-Plus II development software. Likewise, the pseudocode description of the algorithms that perform the input, internal and output wire testing for a combinational circuit is given in [24] [25]. It requires a sequence of vectors to detect a single stuck-at fault in a sequential circuit. Also, due to the presence of the memory elements, the controllability and observability of the internal signals in a sequential circuit are, in general, much worse than those in a combinational circuit. These factors cause the complexity of sequential circuit test generation to be much higher than that for combinational circuits [28]. Consequently, the testing time for sequential circuits is also much higher than that of the combinational circuits.

4.4.2.1 EXAMPLE SEQUENTIAL CIRCUIT

s27 is a basic sequential circuit in ISCAS 89 benchmark circuits. The gate level schematic for the circuit is show in Figure 4.6.

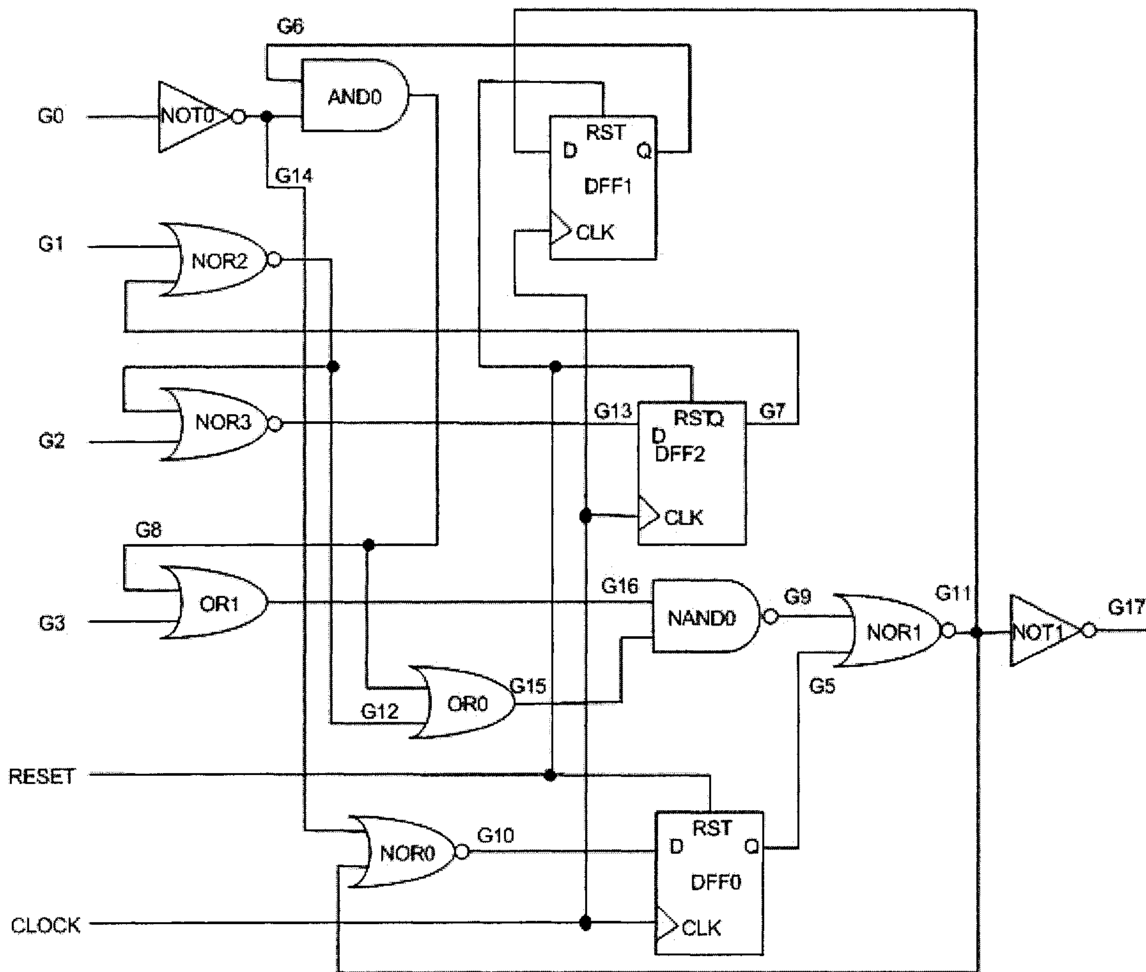


Figure 4.6: Schematic Diagram for Sequential Circuit s27.

The HDL Verilog code for the circuit is shown in *Figure 4.7*.

```
//# inputs 6
//# outputs 1
//# wires 12
//# 3 D-type flipflops
//# 2 inverters
//# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)


module ndff(q,d,rst,clk);
  input d,rst,clk;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
  if (rst==1) q<=0;
  else q<=d;
endmodule


module s27(in_dat, out_dat);
input [5:0] in_dat;
output [1:0]  out_dat;

  //wire G5,G10,G6,G11,G7,G13,G14,G8,G15,G12,G16,G9;


//new input wire
wire G0,G1,G2,G3,rst,CK;
//new out wire
wire G17;

//assign inputs
assign {G0,G1,G2,G3,rst,CK} = in_dat;
//assign output
assign  out_dat = {G17};

  ndff DFF_0(G5,1,rst,CK);
  ndff DFF_1(G6,1,rst,CK);
  ndff DFF_2(G7,G13,rst,CK);
  not NOT_0(G14,G0);
  not NOT_1(G17,1);
  and AND2_0(G8,G14,1);
  or OR2_0(G15,G12,G8);
  or OR2_1(G16,G3,G8);
  nand NAND2_0(G9,G16,G15);
  nor NOR2_0(G10,G14,1);
  nor NOR2_1(G11,G5,G9);
  nor NOR2_2(G12,G1,G7);
  nor NOR2_3(G13,G2,G12);
```

Figure 4.7: *Verilog Code for Sequential Circuit s27.*

As seen from the schematic and HDL Verilog code, the s27 circuit has

- Six inputs which are CK, rst, G0, G1, G2 and G3,
- One output G17 and
- 12 wires G5, G10, G6, G11, G7, G13, G14, G8, G15, G12, G16 and G9.

In the first phase of the fault injection process, the six input wires are injected with s-a-1 and s-a-0 faults. The second phase injects fault s-a-1 and s-a-0 in G17 output wire. In the third phase, stuck-at faults are injected into the twelve internal wires. After every stuck-at fault has been injected, the circuit verilog file (s27.v) is modified and compiled and simulated in Altera Max-Plus II software. The table file (s27.tbl) generated each time the circuit is simulated is also modified accordingly. All the table files generated during the fault injection process are stored in the application's working directory. The original fault free circuit verilog file and table file were stored as backup in the same working directory. Once the fault injections of all the input, internal and out wires are complete, we have the table files generated with injected faults. They are then compared with the fault-free table file to calculate the number of faults detected. We then calculate the fault coverage by the following formula:

$$\textbf{Fault Coverage} = \frac{\textbf{Total number of Faults detected}}{\textbf{Total number of Faults Injected}} \times 100$$

where:

The total number of faults injected = 2 x total number of (input +output + internal wires), which is 38 (2 x (6+1+12)) for this circuit. The results are then stored in a report file (s27.rpt).

The design flow of the Sequential and Combinational Circuits Test method is shown in ***Figure 4.8***. We will discuss in detail about the Max Plus II design environment in Chapter 5. From the design flow, we can see that once the CUT is loaded and the vector file is provided, the Max Plus II software does the compilation and simulation to generate a fault-free table file. The fault simulator then does the fault injection in three phases,

generating table files with injected faults. The comparison of the tables is done to determine the fault coverage, thus generating the final report file for the CUT.

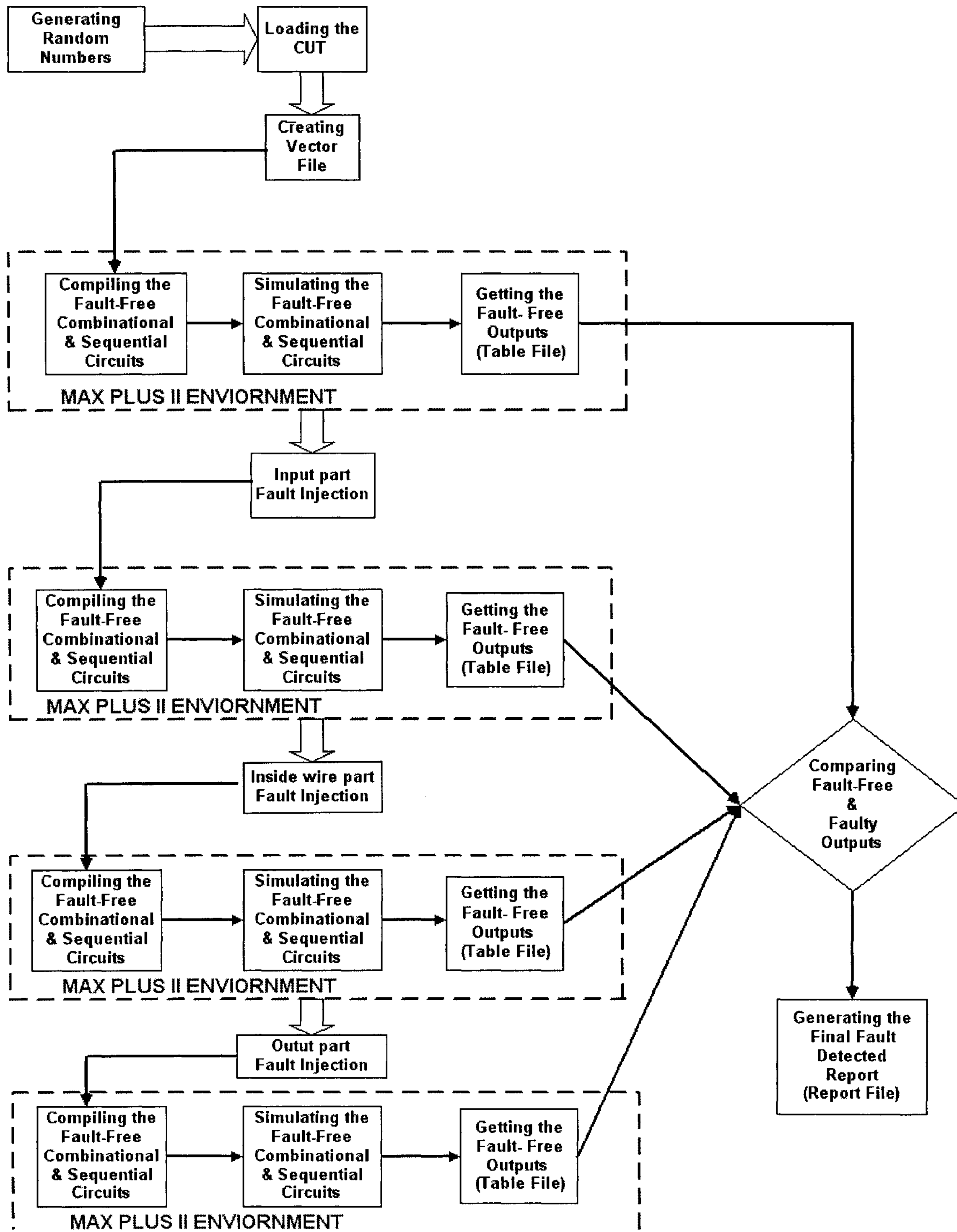


Figure 4.8: *Design flow of combinational and sequential circuit testing method.*

4.4.2.2 EXAMPLE COMBINATIONAL CIRCUIT

C17 is the basic combinational circuit in ISCAS 85 benchmark circuits. The HDL Verilog code for the circuit is shown in Figure 4.8 below.

```
// Verilog
// C17
// Ninputs 5
// Noutputs 2
// Wire 4
// NtotalGates 6
// NAND2 6

module C17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;

output N22,N23;

wire N10,N11,N16,N19;

nand NAND2_1 (N10,N1,N3);
nand NAND2_2 (N11,N3,N6);
nand NAND2_3 (N16,N2,N11);
nand NAND2_4 (N19,N11,N7);
nand NAND2_5 (N22,N10,N16);
nand NAND2_6 (N23,N16,N19);

endmodule
```

Figure 4.9: *Verilog code for Combinational Circuit C17.*

As seen from the HDL Verilog code, the C17 has:

- Five inputs which are N1, N2, N3, N6 and N7,
- Two outputs N22 and N23.
- Four wires which are N10, N11, N16 and N19.

The fault injections for the combinational circuits are done in similar fashion as for sequential circuits. The first phase injects stuck-at faults in the 5 inputs. The second phase injects stuck-at faults at the 2 outputs, followed by the 4 internal wires. After every stuck-at fault has been injected, the circuit verilog file (C17.v) is modified and compiled and

simulated in Altera Max-Plus II software. The table file (C17.tbl) generated each time the circuit is simulated is also modified accordingly. All the table files generated during the fault injection process are stored in the application's working directory. The original fault-free circuit verilog file and table file were stored as backup in the same working directory. Once the fault injections of all the input, internal and output wires are complete, we have the table files generated with injected faults. They are then compared with the fault-free table file to calculate the number of faults detected. We then calculate the fault coverage by the same formula as used for sequential circuits. The results are then stored as a log file (C17.log). The log file gives us the following information:

1. The number of test patterns applied and the faults detected by each test pattern.
2. The circuit structure which says about the number of input, outputs, wires, gates and the depth of the circuit.
3. Simulation parameters, if it is a random test pattern generation or deterministic (Mintest vectors used) test pattern generation.
4. Simulation results consisting of total number of test patterns being applied, fault coverage, number of collapsed faults, number of detected and undetected faults.
5. The total run time for the fault simulation.

Sample C17 log file is shown in Figure 4.10 below.

```
            C17.log
* Log file for the circuit C17.bench.
Number of faults detected by each test pattern:

test 1      :9 faults detected      0 1 1 1 0 = 0 0
test 2      :2 faults detected      0 0 0 1 0 = 0 0
test 3      :0 faults detected      0 1 1 1 0 = 0 0
test 4      :4 faults detected      0 0 0 0 1 = 0 1
test 5      :4 faults detected      0 1 0 1 0 = 1 1
test 6      :3 faults detected      1 0 1 0 1 = 1 1
test 7      :0 faults detected      1 0 0 0 0 = 0 0
test 8      :0 faults detected      0 0 0 1 0 = 0 0
test 9      :0 faults detected      1 0 1 0 1 = 1 1
test 10     :0 faults detected      1 0 0 0 0 = 0 0
test 11     :0 faults detected      1 1 1 1 0 = 1 0
test 12     :0 faults detected      1 1 1 0 1 = 1 1
test 13     :0 faults detected      0 1 1 0 1 = 1 1
test 14     :0 faults detected      0 1 0 1 1 = 1 1
test 15     :0 faults detected      0 1 0 1 0 = 1 1
test 16     :0 faults detected      1 1 1 1 1 = 1 0
test 17     :0 faults detected      1 1 0 1 1 = 1 1
test 18     :0 faults detected      0 0 0 0 0 = 0 0
test 19     :0 faults detected      1 0 0 0 0 = 0 0
test 20     :0 faults detected      0 1 1 0 1 = 1 1

End of fault simulation.

********    SUMMARY OF FAULT SIMULATION RESULTS ********

1.    Circuit Structure
      Name of the circuit                  :     C17
      Number of gates                      :     11
      Number of primary inputs             :     5
      Number of primary outputs            :     2
      Depth of the circuit                 :     4

2.    Simulation parameters
      Simulation mode                      :     random
      Initial random number generator seed       :     1050420308

3.    Simulation results
      Number of test patterns applied            :     20
      Fault coverage                       :     100.000 %
      Number of collapsed faults           :     22
      Number of detected faults            :     22
      Number of undetected faults          :     0

4.    Total run time                 :     22 seconds
```

Figure 4.10: *Log file for C17 combinational circuit (C17.log).*

4.5 CONCLUSIONS

In this chapter we discussed about various fault models and fault simulation design process. We also presented the fault injection scheme. We emphasized on the software fault injection that we implemented in our experimentation. Example sequential and combinational circuits were given to illustrate some features of the proposed techniques.

CHAPTER 5
SYSTEM TEST ENVIRONMENT

This chapter talks about our whole test generation system environment and how we implemented each part. We start with the system overview giving a conceptual view of the whole system test process. We defined our two main programs that make the whole process like a turnkey approach. In the second part, parsing, we discuss in depth how our C program interacts with the hardware description language, HDL Verilog. We then present sufficient information about the Max Plus II software, which is the design and development tool, we used for our compilation and simulation of SOCs. It is followed by a brief discussion about deterministic test pattern generation using test vectors. We then present the graphic design files for the three SOCs, which we designed and implemented. Through the design files we try to show the internal connection between the cores in an SOC. Details of system configuration is provided. We conclude our chapter by presenting the steps in running the programs. We supported our explanation by the screen shots of the program execution.

5.1 SYSTEM OVERVIEW

This thesis deals with system-on-chip testing. Here we implemented core selection and core (gate-level combinational and sequential circuits) test generation and fault simulation targeted on single stuck-at faults. The whole system test generation is carried out through two main programs. The first program is the core selection and the second in the fault simulation. The whole SOC test generation is like a turn key process where the user is asked to select a core out of the different cores in the SOC. Once the user gives the core name the first part of the program isolates the core and connects the core inputs to the TAM input bus and the core outputs to the TAM output bus. At this point, the core has been assigned a dedicated bus from the system input pins to the system output pins. The second program is now invoked within the first program i.e. the fault simulation.

Also at this phase, the user is asked if random test pattern generation or deterministic test pattern generation is to be performed. If random test patters is selected the fault simulator generates random input vectors and faults are injected to the CUT through the dedicated bus. In case of deterministic test, Mintest [19] vectors are provided as input vector file. This second program also invokes the Max-Plus development software to do the compilation and simulation after every fault is injected. Hence the Max-Plus II software runs in the background as long as all the faults are injected by the fault simulator. The Max-Plus II saves the fault list by generating table files every-time. The table files are compared with the fault free table file generated at the beginning of fault injection process. The fault simulator then transports the response of fault simulation of the CUT through the output TAM bus to the first program. The first program then halts the second program and saves the CUT response in a report file. The program then again asks the user to select a core to repeat the whole test generation process. ***Figure 5.1*** shows the flow diagram for the SOC test generation.

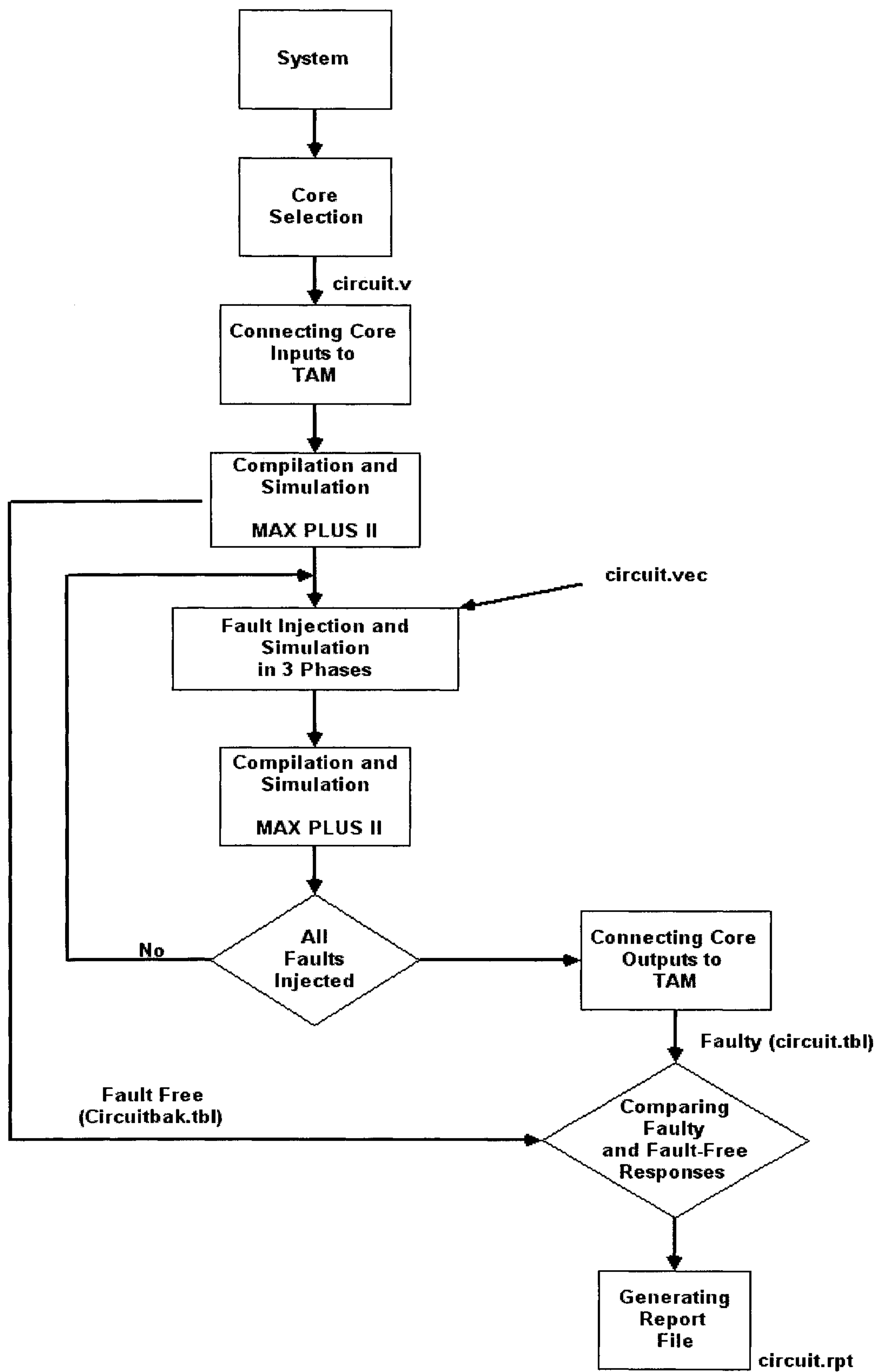


***Figure:* 5.1** *Flow Diagram for SOC test generation*

In the flow diagram, circuit.v is the filename provided by the user and circuitbak.tbl, circuit.tbl and circuit.rpt are files generated by the program. Circutbak.tbl file is the fault free table file generated by Maxplus II before fault injection. Circuit.tbl is table file generated with faults injected into the CUT. Circuit.vec is the vector file given to the fault simulator when the used selects the deterministic test generation option. Circuit.rpt is the report file generated at the end of fault simulation by the fault simulator. The report file contains the details of the faults injected, faults collapsed, faults detected, fault coverage, time taken, etc. A sample report file for combinational circuit c17 is presented in the previous chapter.

We designed three different SOCs. The first SOC, called **SOC_COMB**, consists of ten ISCAS-85 combinational circuits. They are c17, c432, c499, c880, c1355, c1908, c2670, c3540, c6288 and c74181. The second SOC, called **SOC_SEQ**, consists of ten ISCAS-89 sequential circuits. They are s27, s298, s344, s349, s382, s400, s420, s444, s820 and s1196. The third SOC, called **SOC_MIXED**, consists of two combination circuit c17 and c432 and two sequential circuits s27 and s298.

5.2 PARSING

As mentioned earlier two main programs do the whole SOC test generation process. Both the programs are developed in C programming language. When the first program is run, it asks the user to select the core from the SOC. Here it reads the Verilog file called the

SOC.v as shown in the figure below. The figure shows the Verilog file of the system SOC_COMB that consists of all combinational circuits.

```
// Copyright (C) 1991-2004 Altera Corporation
// Any megafunction design, and related netlist (encrypted or decrypted),
// support information, device programming or simulation file, and any other
// associated documentation or information provided by Altera or a partner
// under Altera's Megafunction Partnership Program may be used only
// to program PLD devices (but not masked PLD devices) from Altera. Any
// other use of such megafunction design, netlist, support information,
// device programming or simulation file, or any other related documentation
// or information is prohibited for any other purpose, including, but not
// limited to modification, reverse engineering, de-compiling, or use with
// any other silicon devices, unless such use is explicitly licensed under
// a separate agreement with Altera or a megafunction partner. Title to the
// intellectual property, including patents, copyrights, trademarks, trade
// secrets, or maskworks, embodied in any such megafunction design, netlist,
// support information, device programming or simulation file, or any other
// related documentation or information provided by Altera or a megafunction
// partner, remains with Altera, the megafunction partner, or their respective
// licensors. No other licenses, including any licenses needed under any third
// party's intellectual property, are provided herein.

module soc_comb(
      ENABLE,
      Z,
      OUT
);

input [0:3] ENABLE;
input [0:156] Z;
output      [63:0] OUT;

wire  [0:35] SYNTHESIZED_WIRE_0;
wire  [0:4] SYNTHESIZED_WIRE_1;
wire  [0:1] SYNTHESIZED_WIRE_28;
wire  [0:7] SYNTHESIZED_WIRE_29;
wire  [0:6] SYNTHESIZED_WIRE_12;
wire  [0:31] SYNTHESIZED_WIRE_13;
wire  [0:25] SYNTHESIZED_WIRE_14;
wire  [0:31] SYNTHESIZED_WIRE_15;
wire  [0:24] SYNTHESIZED_WIRE_16;
wire  [0:63] SYNTHESIZED_WIRE_17;
wire  [0:21] SYNTHESIZED_WIRE_18;
wire  [0:30] SYNTHESIZED_WIRE_19;
wire  [0:40] SYNTHESIZED_WIRE_20;
wire  [0:59] SYNTHESIZED_WIRE_21;
wire  [0:40] SYNTHESIZED_WIRE_22;
wire  [0:32] SYNTHESIZED_WIRE_23;
wire  [0:156] SYNTHESIZED_WIRE_24;
wire  [0:49] SYNTHESIZED_WIRE_25;
wire  [0:31] SYNTHESIZED_WIRE_26;
wire  [0:13] SYNTHESIZED_WIRE_27;




c432  b2v_29(.Gin_dat(SYNTHESIZED_WIRE_0),.Gout_dat(SYNTHESIZED_WIRE_12));

c17   b2v_31(.Nin_dat(SYNTHESIZED_WIRE_1),.Nout_dat(SYNTHESIZED_WIRE_28));
```

```
mux_10
      b2v_37(.enable(ENABLE),.in11(SYNTHESIZED_WIRE_28),.in10(SYNTHESIZED_WIRE_28
),.in107(SYNTHESIZED_WIRE_29),.in106(SYNTHESIZED_WIRE_29),.in105(SYNTHESIZED_WIRE
_29),.in104(SYNTHESIZED_WIRE_29),.in103(SYNTHESIZED_WIRE_29),.in102(SYNTHESIZED_W
IRE_29),.in101(SYNTHESIZED_WIRE_29),.in100(SYNTHESIZED_WIRE_29),.in2(SYNTHESIZED_
WIRE_12),.in3(SYNTHESIZED_WIRE_13),.in4(SYNTHESIZED_WIRE_14),.in5(SYNTHESIZED_WIR
E_15),.in6(SYNTHESIZED_WIRE_16),.in7(SYNTHESIZED_WIRE_17),.in8(SYNTHESIZED_WIRE_1
8),.in9(SYNTHESIZED_WIRE_19),.out(OUT));

demux10
      b2v_39(.enable(ENABLE),.z(Z),.a(SYNTHESIZED_WIRE_1),.b(SYNTHESIZED_WIRE_0),
.c(SYNTHESIZED_WIRE_20),.d(SYNTHESIZED_WIRE_21),.e(SYNTHESIZED_WIRE_22),.f(SYNTHE
SIZED_WIRE_23),.g(SYNTHESIZED_WIRE_24),.h(SYNTHESIZED_WIRE_25),.i(SYNTHESIZED_WIR
E_26),.j(SYNTHESIZED_WIRE_27));

c499  b2v_40(.Gin_dat(SYNTHESIZED_WIRE_20),.Gout_dat(SYNTHESIZED_WIRE_13));

c880  b2v_41(.Gin_dat(SYNTHESIZED_WIRE_21),.Gout_dat(SYNTHESIZED_WIRE_14));

c1355 b2v_42(.Gin_dat(SYNTHESIZED_WIRE_22),.Gout_dat(SYNTHESIZED_WIRE_15));

c1908 b2v_43(.Gin_dat(SYNTHESIZED_WIRE_23),.Gout_dat(SYNTHESIZED_WIRE_16));

c2670 b2v_44(.Gin_dat(SYNTHESIZED_WIRE_24),.Gout_dat(SYNTHESIZED_WIRE_17));

c3540 b2v_45(.Gin_dat(SYNTHESIZED_WIRE_25),.Gout_dat(SYNTHESIZED_WIRE_18));

c6288 b2v_46(.Gin_dat(SYNTHESIZED_WIRE_26),.Gout_dat(SYNTHESIZED_WIRE_19));

c74181      b2v_47(.Gin_dat(SYNTHESIZED_WIRE_27),.Gout_dat(SYNTHESIZED_WIRE_29));


endmodule
```

Figure 5.2: *Verilog File for the System-on-Chip, SOC_COMB*

The program parses this file for the circuit which the user selected as core to be tested. For example, if the user selects the core "c1908", the program looks for the circuit and sets the circuit as the CUT. As we can see from the figure, the *named mapping* of the input and output pins is already done with the SOC input and outputs. Also the multiplexer (mux_10) and the demultiplexer (demux_10) are used to dedicate the Tam bus to the selected CUT. Once the CUT is found through mapping the TAM bus is assigned to the CUT as shown in ***Figure 3.19*** in chapter 3. Then the second program is invoked and the input to this second program is "c1908.v". The fault simulation is then

carried on. We similarly do the parsing for the other two systems, SOC_SEQ and SOC_MIXED.

We present the Verilog files for the other two SOCs.

```
// Copyright (C) 1991-2004 Altera Corporation
// Any megafunction design, and related netlist (encrypted or decrypted),
// support information, device programming or simulation file, and any other
// associated documentation or information provided by Altera or a partner
// under Altera's Megafunction Partnership Program may be used only
// to program PLD devices (but not masked PLD devices) from Altera.  Any
// other use of such megafunction design, netlist, support information,
// device programming or simulation file, or any other related documentation
// or information is prohibited for any other purpose, including, but not
// limited to modification, reverse engineering, de-compiling, or use with
// any other silicon devices, unless such use is explicitly licensed under
// a separate agreement with Altera or a megafunction partner. Title to the
// intellectual property, including patents, copyrights, trademarks, trade
// secrets, or maskworks, embodied in any such megafunction design, netlist,
// support information, device programming or simulation file, or any other
// related documentation or information provided by Altera or a megafunction
// partner, remains with Altera, the megafunction partner, or their respective
// licensors. No other licenses, including any licenses needed under any third
// party's intellectual property, are provided herein.

module soc_seq(
      ENABLE,
      Z,
      OUT
);

input [0:3] ENABLE;
input [0:20] Z;
output      [18:0] OUT;

wire  SYNTHESIZED_WIRE_0;
wire  SYNTHESIZED_WIRE_1;
wire  [13:0] SYNTHESIZED_WIRE_2;
wire  [5:0] SYNTHESIZED_WIRE_3;
wire  [10:0] SYNTHESIZED_WIRE_4;
wire  [10:0] SYNTHESIZED_WIRE_5;
wire  [5:0] SYNTHESIZED_WIRE_6;
wire  [5:0] SYNTHESIZED_WIRE_7;
wire  [5:0] SYNTHESIZED_WIRE_8;
wire  [18:0] SYNTHESIZED_WIRE_9;
wire  [0:5] SYNTHESIZED_WIRE_10;
wire  [0:4] SYNTHESIZED_WIRE_11;
wire  [0:10] SYNTHESIZED_WIRE_12;
wire  [0:10] SYNTHESIZED_WIRE_13;
wire  [0:4] SYNTHESIZED_WIRE_14;
wire  [0:4] SYNTHESIZED_WIRE_15;
wire  [0:19] SYNTHESIZED_WIRE_16;
wire  [0:4] SYNTHESIZED_WIRE_17;
wire  [0:20] SYNTHESIZED_WIRE_18;
wire  [0:15] SYNTHESIZED_WIRE_19;
```

```
mux_10
      b2v_37(.in1(SYNTHESIZED_WIRE_0),.in7(SYNTHESIZED_WIRE_1),.enable(ENABLE),
.in10(SYNTHESIZED_WIRE_2),.in2(SYNTHESIZED_WIRE_3),.in3(SYNTHESIZED_WIRE_4),
.in4(SYNTHESIZED_WIRE_5),.in5(SYNTHESIZED_WIRE_6),.in6(SYNTHESIZED_WIRE_7),
.in8(SYNTHESIZED_WIRE_8),.in9(SYNTHESIZED_WIRE_9),.out(OUT));

demux10
      b2v_39(.enable(ENABLE),.z(Z),.a(SYNTHESIZED_WIRE_10),.b(SYNTHESIZED_WIRE_11),
.c(SYNTHESIZED_WIRE_12),.d(SYNTHESIZED_WIRE_13),.e(SYNTHESIZED_WIRE_14),.f(SYNTHESI
ZED_WIRE_15),.g(SYNTHESIZED_WIRE_16),.h(SYNTHESIZED_WIRE_17),.i(SYNTHESIZED_WIRE_18
),.j(SYNTHESIZED_WIRE_19));

s27   b2v_49(.in_dat(SYNTHESIZED_WIRE_10),.out_dat(SYNTHESIZED_WIRE_0));

s298  b2v_50(.in_dat(SYNTHESIZED_WIRE_11),.out_dat(SYNTHESIZED_WIRE_3));

s344  b2v_51(.in_dat(SYNTHESIZED_WIRE_12),.out_dat(SYNTHESIZED_WIRE_4));

s349  b2v_53(.in_dat(SYNTHESIZED_WIRE_13),.out_dat(SYNTHESIZED_WIRE_5));

s382  b2v_54(.in_dat(SYNTHESIZED_WIRE_14),.out_dat(SYNTHESIZED_WIRE_6));

s400  b2v_56(.in_dat(SYNTHESIZED_WIRE_15),.out_dat(SYNTHESIZED_WIRE_7));

s420  b2v_57(.in_dat(SYNTHESIZED_WIRE_16),.out_dat(SYNTHESIZED_WIRE_1));

s444  b2v_58(.in_dat(SYNTHESIZED_WIRE_17),.out_dat(SYNTHESIZED_WIRE_8));

s820  b2v_61(.in_dat(SYNTHESIZED_WIRE_18),.out_dat(SYNTHESIZED_WIRE_9));

s1196 b2v_62(.in_dat(SYNTHESIZED_WIRE_19),.out_dat(SYNTHESIZED_WIRE_2));


endmodule
```

Figure 5.3: *Verilog File for the System-on-Chip, SOC_SEQ*

5.3 ALTERA MAXPLUS II DESIGN AND DEVELOPMENT SOFTWARE

Altera MAX PLUS II development software is a fully integrated programmable logic design environment. This easy-to-use software supports the Altera ACEX, FLEX, MAX, and Classic programmable device families, and works in both PC and UNIX environments. Altera MAX PLUS II development software offers flexibility and performance, and allows seamless integration with industry-standard design entry, synthesis, and verification tools. Altera MAX PLUS II software is a comprehensive tool for the design, compilation, and simulation of digital circuit designs. Today, lots of designers like to develop, compile, verify, and program their digital circuit designs using Altera MAX PLUS II development software, which operates on personal computers (PCs) and engineering workstations. Altera MAX PLUS II development software gives designers the flexibility to enter a design using all the major design entry methodologies, including:

- VHDL
- Verilog HDL
- Schematic capture
- Design entry utilizing megafunctions and the library of parameterized modules (LPM)
- Altera hardware description language (AHDL)
- Waveform

By giving designers control over the entry format and the ability to mix and match design entry methodologies, the Altera MAX PLUS II development software minimizes development time.

5.3.1 ALTERA MAX PLUS II DESIGN ENVIRONMENT

The Altera multiple array matrix programmable logic user system (MAX PLUS II) provides a multi-platform, architecture-independent design environment that easily adapts to any specific design need. MAX PLUS II offers easy design entry, quick processing, and straightforward device programming.

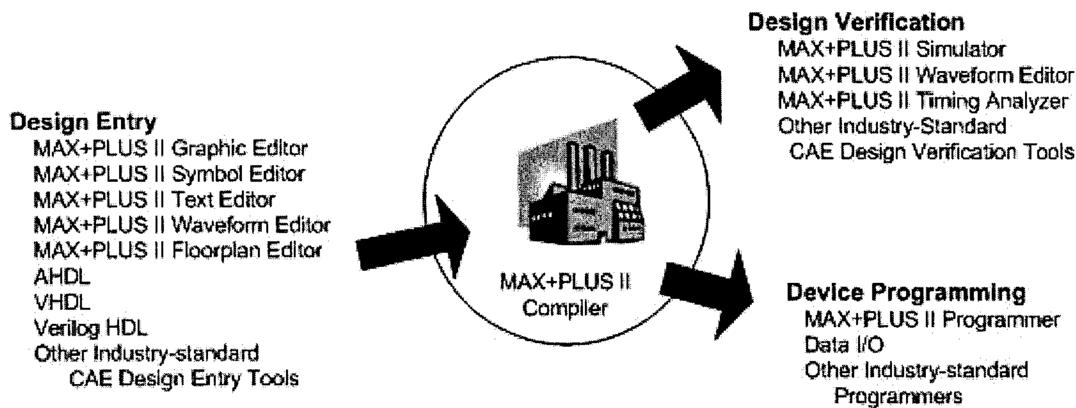


Figure 5.4: *MAX PLUS II Design Environment*

MAX PLUS II development software, shown in ***Figure 5.5***, is a fully integrated package for creating logic designs for Altera programmable logic devices, including the Classic, MAX5000, MAX7000, MAX9000, FLEX6000, FLEX8000, and FLEX10K families of devices.

MAX PLUS II offers a full spectrum of logic design capabilities: a variety of design entry methods for hierarchical designs, powerful logic synthesis, timing-driven compilation, partitioning, functional and timing simulation, linked multi-device simulation, timing analysis, automatic error location, and device programming and verification. MAX PLUS II both reads and writes Altera hardware description language (AHDL) files and standard EDIF netlist files, Verilog HDL files, VHDL files, and ORCAD schematic files. In addition, MAX PLUS II reads Xilinx netlist files and writes standard delay format (SDF) files for a convenient interface to other industry-standard CAE software.

MAX PLUS II offers a rich graphical user interface complemented with an illustrated, easy-to-use on-line help system. The complete MAX PLUS II system includes 11 fully integrated applications that take one through every step of creating a design (a logic design, including all sub-designs, is called a "project" in MAX PLUS II).

Many features and commands, such as opening files, entering device, pin and logic cell assignments and compiling the current project are shared by many or all MAX PLUS II applications, so that learning to use one application gives one a head start on learning to use the others. The design editors (the graphic, text, and waveform editors) and auxiliary editors (the floorplan and symbol editors) also share numerous features. Each design editor allows one to perform similar tasks, such as finding a signal or symbol, in the same way. We can easily combine different types of design files in a hierarchical project, choosing the design entry format that works best for each functional block. A large library of Altera-supplied megafunctions and macrofunctions including functions from the library of parameterized modules (LPM), provide a wide range of design entry options. Designer can work with different MAX PLUS II applications simultaneously. For example, we can open multiple design files and transfer information among them while compiling or simulating another project; or, we can view an entire project hierarchy and move smoothly from one hierarchical level to another, while MAX PLUS II automatically starts the appropriate design editor for each file.
The MAX PLUS II compiler lies at the heart of the MAX PLUS II system, providing powerful project processing that one can customize to achieve the best possible silicon implementation of the project. Automatic error location and extensive documentation on error and warning messages make design modifications quick and easy. We can create output files in a variety of formats for functional, timing, and linked multi-device simulation; timing analysis; and device programming. At every step of the design process, MAX PLUS II makes it easy for us to focus on our project—not on how to use the software. The superb integration of the MAX PLUS II software helps us maximize our efficiency and productivity, putting us in control of our logic design environment.

5.3.2 DESIGN FLOW UNDER ALTERA MAX PLUS

The process of taking a new project from conception to completion can be simplified as follows:

1. Create a new design file or a hierarchy of multiple design files in any combination of the MAX PLUS II design editors, i.e., the **graphic, text, and waveform editors**.
2. Specify the top-level design file name as the project name.
3. Assign a device family for the project. One can either allow the compiler to select a device for one or assign a specific device.
4. Open the MAX PLUS II compiler window and choose the start button to compile the project. If we wish, we can turn on the **Timing SNF Extractor** module to create a netlist file for timing simulation and timing analysis. However, we did our compilation with the **Functional SNF Extractor** option.
5. If the project compiles successfully, one can optionally perform a simulation and timing analysis:
   - To run a timing analysis, open the MAX PLUS II **timing analyzer window**, select an analysis mode, and choose the **start button.**
   - To run a simulation, we must first create vector inputs in a **simulator channel file (.scf)** in the **waveform editor or in a vector file (.vec)** in the text editor. Then, we open the MAX PLUS II simulator window and select the Start button.
6. Open the MAX PLUS II programmer window and either insert a device into a programming adapter on the master programming unit (MPU) or connect the BitBlaster, ByteBlaster, or FLEX download cable to a device that is mounted in-system.
7. Choose the program button to program an EPROM- or EEPROM-based device, or choose the Configure button to configure an SRAM-based device.

5.3.3. MAX PLUS II APPLICATIONS

MAX PLUS II software consists of 11 application programs and the MAX PLUS II manager. Different design entry applications can be active simultaneously, allowing designers to switch between them with a click of the mouse or a menu command. At the same time, we can run one of the background applications, viz. the compiler, simulator, timing analyzer, or programmer. Commands shared by the various applications function in the same way, making your logic design task easier.

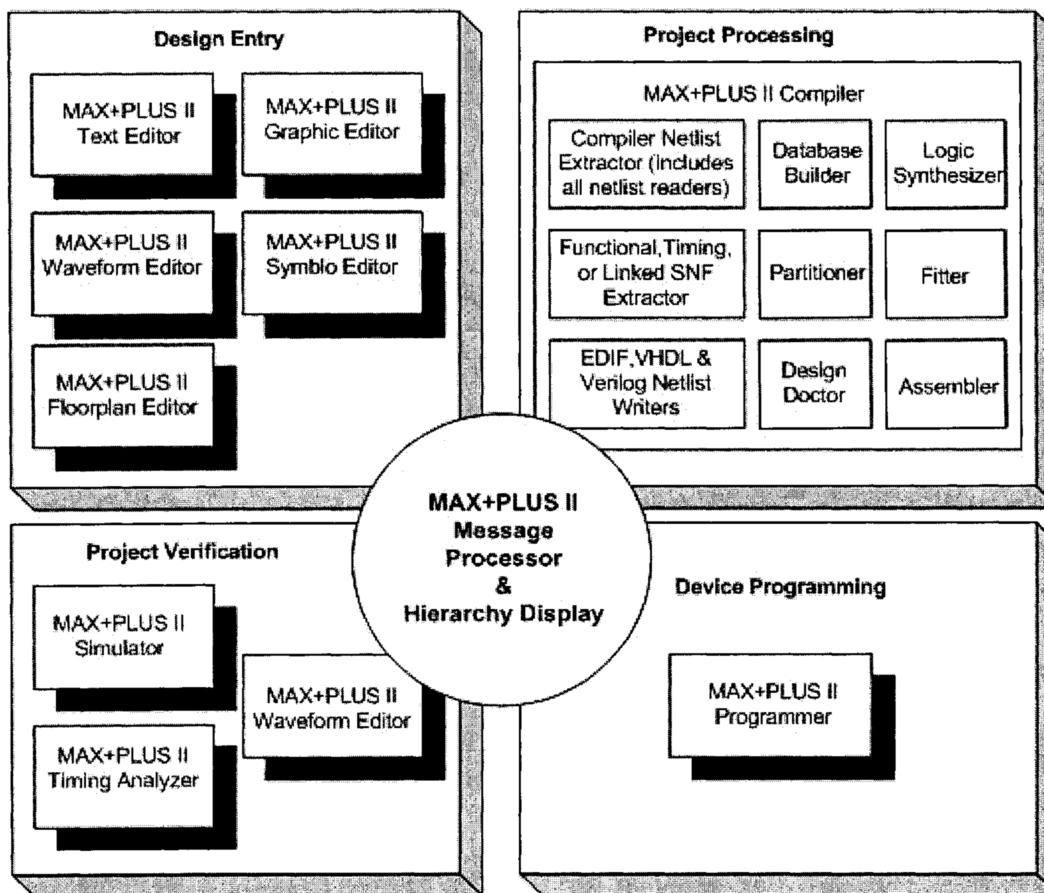


***Figure 5.5:** MAX PLUS II applications.*

We describe the MAX PLUS II applications as follows (***Figure 5.5***):

Hierarchy display: Displays the current hierarchy of files as a hierarchy tree with branches that represent subdesigns. User can tell at a glance whether a design file is a schematic, text, or waveform design; which files are currently open; and which user-editable ancillary files are available for the project. User can also directly open or close one or more files in a hierarchy tree and enter resource assignments for them.

Graphic editor: Lets designer enter a schematic logic design in a true what–you-see-is-what-you-get (WYSIWYG) environment. While the Altera-provided primitives, megafunctions, and macrofunctions serve as designer's basic building blocks, designer can also use custom symbols. Graphic editor was extensively used in implementing the wrapper around the core. It became very easy to add the multiplexers and the demultiplexers in the graphic editor.

Symbol Editor: Allows user to edit existing symbols and create new ones. The *Figure 3.21* in Chapter 3 is the symbol of the SOC which was designed in the symbol editor.

Text Editor: The text editor lets user create and edit text-based logic design files written in AHDL, VHDL, and Verilog HDL. With the text editor, user can also create, view, and edit other ASCII files used with MAX PLUS II applications. Although user can create HDL files with other text editors, the MAX PLUS II text editor allows user to take advantage of context-sensitive help, syntax coloring, and AHDL, VHDL, and Verilog HDL templates.

Waveform Editor: Serves a dual role, as a design entry tool and as a tool for entering test vectors and viewing simulation results. Some of the snap shots of the waveform editor for our simulations are shown in the next chapter.

Floorplan Editor: Lets designer assign logic to physical device pin and logic cell resources in a graphical environment. Designer can edit pin placements in a device package view and assign signals to individual logic cells in a more detailed logic array block (LAB) view. Designer can also view the results of the last compilation.

Compiler: Processes logic projects targeted for Altera Classic, MAX5000, MAX7000, MAX9000, FLEX6000, FLEX8000, and FLEX 10K device families. It performs most tasks automatically. However, user can customize all or part of the compilation process. We select the Functional SNF Extractor processing mode during the compilation of our CUTs.

Simulator: Enables user to test the logical operation and internal timing of user's logic circuit. Functional simulation, timing simulation, and linked multi-device simulation are available.

Timing analyzer: Analyzes the performance of designer's logic circuit after it has been synthesized and optimized by the compiler.
Programmer: Lets designer program, configure, verify, examine, and test Altera devices.

Message processor: Displays error, warning, and information messages on the status of user's project and allows user to locate the source of a message automatically in the original design file(s), ancillary file(s), and assignments floorplan. It helped us debug our Verilog design files while compiling.

5.4 INPUT VECTOR FILE THROUGH MAX PLUS II

In Altera MAX PLUS II, vector file is used as the source of input vectors for simulation. A vector specifies the logic levels for an individual node within a project. The simulator uses vectors to simulate the behavior of the CUT. The programmer and simulator use vectors for functional testing. Vectors for simulation and functional testing can be defined in vector files (.vec). Functional testing vectors can also be stored in programming files. The vector file provides an ASCII text format for specifying simulation input conditions and the nodes to be simulated. A vector file can also be used to create a waveform design file for waveform design entry. Vectors that have been saved in the programming file can also be used for functional testing. A vector file that is used for functional testing must contain all input vector logic levels, and expected output logic levels that are typically derived from a simulation or from a previously tested device. In addition, the file must conform to the following rules:

INPUTS:

- Input logic levels include low, high, and high impedance (0, 1, and Z respectively).
- All inputs in the file must also be listed in the fit file for the CUT.
- All input pins that are in the fit file but not in the vector file, SCF (simulator channel file) or programming file are assigned a low (0) logic level.
- All input pins on the device that are not used in the implementation are assigned a low (0) logic level.
- Use pin names rather than probe or buried node names for all nodes in the file.

OUTPUTS:

- All I/O pins that are in the fit file but not in the vector file, SCF (simulator channel file), or programming file are assigned a high impedance (Z) logic level.
- All I/O pins on the device that are not used in the implementation are assigned a high impedance (Z) logic level.
- All buried nodes and outputs in the vector file, SCF (simulator channel file), or programming file that are not listed in the fit file are assigned an undefined (X) logic level as soon as testing begins.
- Use node names rather than probe names for all nodes in the file.

During functional testing, outputs are compared according to functional comparison rules. The simulator looks for a vector file or a simulator channel file, when we execute the SIMULATE command. It automatically loads the newest vector file or SCF in the project directory with the same filename as the project. When we enter VECTOR, the simulator automatically generates an SCF for the file. Expected output logic levels are optional. They may be of any type: user-defined expected outputs, outputs from other simulations, or actual functional device outputs (e.g., in a table file created during a previous simulation). We used the MINTEST vector [19] for our deterministic test pattern generation. We made minor changes to its format in order to make it compatible to run in Max Plus II. The following example shows a sample vector file, including a separate pattern section for optional expected output values.

```
% units default to ns %
START 0 ;
STOP 1000 ;
INTERVAL 100 ;
INPUTS CLOCK ;
PATTERN
0 1 ;          % relative vector values %
            % CLOCK ticks every 100 ns %
INPUTS DATAINX DATAINY ;
PATTERN        % test every combination of %
            % DATAINX and DATAINY %
0>   0 0
220>  1 0
320>  1 1       % absolute time vector values %
570>  0 1
720>  1 1
OUTPUTS  DATAOUTX, DATAOUTY;
```

The ***Figure 5.6*** below shows an example Mintest vector file for combinational circuit c432.

```
INTERVAL 1 ;
START 0 ;
STOP 1ns ;
INPUTS Gin_dat0  Gin_dat1  Gin_dat2  Gin_dat3  Gin_dat4  Gin_dat5  Gin_dat6
Gin_dat7  Gin_dat8  Gin_dat9  Gin_dat10  Gin_dat11  Gin_dat12  Gin_dat13
Gin_dat14  Gin_dat15  Gin_dat16  Gin_dat17  Gin_dat18  Gin_dat19  Gin_dat20
Gin_dat21  Gin_dat22  Gin_dat23  Gin_dat24  Gin_dat25  Gin_dat26  Gin_dat27
Gin_dat28  Gin_dat29  Gin_dat30  Gin_dat31  Gin_dat32  Gin_dat33  Gin_dat34
Gin_dat35  ;
PATTERN
 0>1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 1 1
 1>1 0 0 0 1 0 0 0 1 0 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0
 2>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 3>1 1 1 0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0
 4>1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 0 0 1 0 1 0 1
 5>1 1 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 1 0 1 0 0 0 0 1
 6>0 0 1 1 0 0 0 0 1 0 0 1 0 0 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0
 7>1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 0 0 0 0
 8>0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1 1
 9>1 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 1 0 1 1 1 0 0 0 0 1 0 0 1 1
10>0 0 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0
11>0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 1 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 0
12>1 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 0 0 0 1 1 0 1
13>0 1 1 1 0 1 0 0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 1 0 0 1 1 1 0
14>0 0 1 1 1 0 1 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 1 0 1 1
15>1 0 0 0 1 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 1 1 0 1 1 0 0 1 1
16>1 0 0 0 1 0 0 1 1 0 1 0 0 0 1 1 1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0
17>0 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 1 1
18>0 0 0 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1
19>1 0 1 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1
20>1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 1 1 1 0 0 1 0 0 0 1 0 0 1 1 1 0 1 1
21>0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 1 0 0 1
22>0 1 0 0 1 1 1 0 0 1 1 0 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 1 0 1
23>1 1 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1
24>0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
25>1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0 1 1 1 1 1 0 0 0 1 1 1
26>0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0
;
OUTPUTS Gout_dat0  ;

OUTPUTS Gout_dat1  ;

OUTPUTS Gout_dat2  ;

OUTPUTS Gout_dat3  ;

OUTPUTS Gout_dat4  ;

OUTPUTS Gout_dat5  ;
```

Figure 5.6: *Mintest Vector File for Combinational Circuit c432 (c432.vec)*

5.5 GRAPHIC DESIGN FILE FOR COMBINATIONAL SOC (SOC_COMB)

Theses deign files were developed in Altera Max Plus II software. They represent the graphical view of the architecture of our SOCs.

The ***Figure*** *5.7* shows the graphic design file of our first SOC i.e. SOC_COMB. It consists of all the combinational circuits.

5.6 GRAPHIC DESIGN FILE FOR SEQUENTIAL SOC (SOC_SEQ)

The ***Figure 5.8*** shows the graphic design file of our second SOC i.e. SOC_SEQ. It consists of all the sequential circuits.

5.7 GRAPHIC DESIGN FILE FOR MIXED SOC (SOC_MIXED)

The ***Figure 5.9*** shows the graphic design file of our third SOC i.e. SOC_MIXED. It consists of both combinational and sequential circuits.

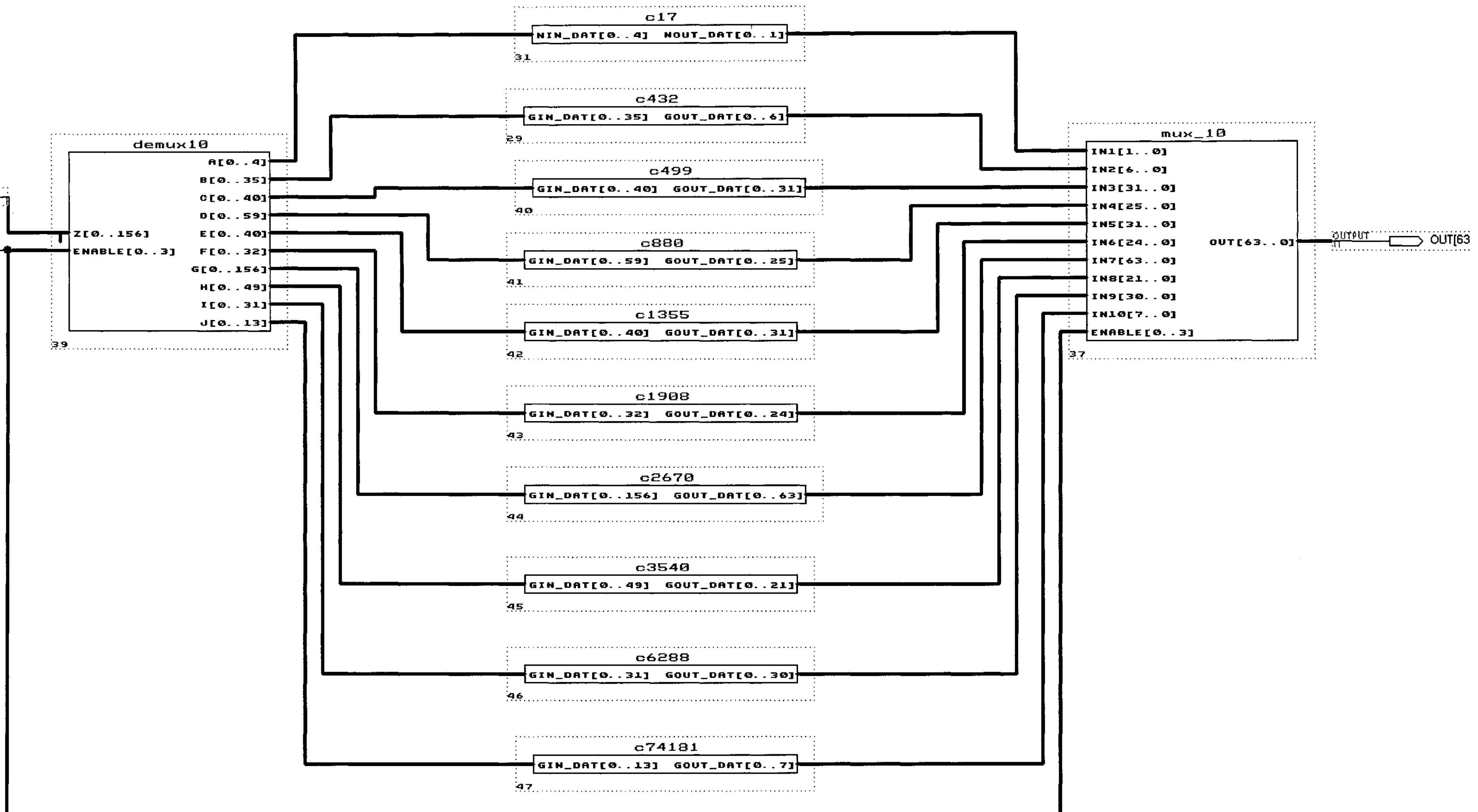


Figure 5.7: *Graphic Design File for SOC_COMB (SOC_comb.gdf)*

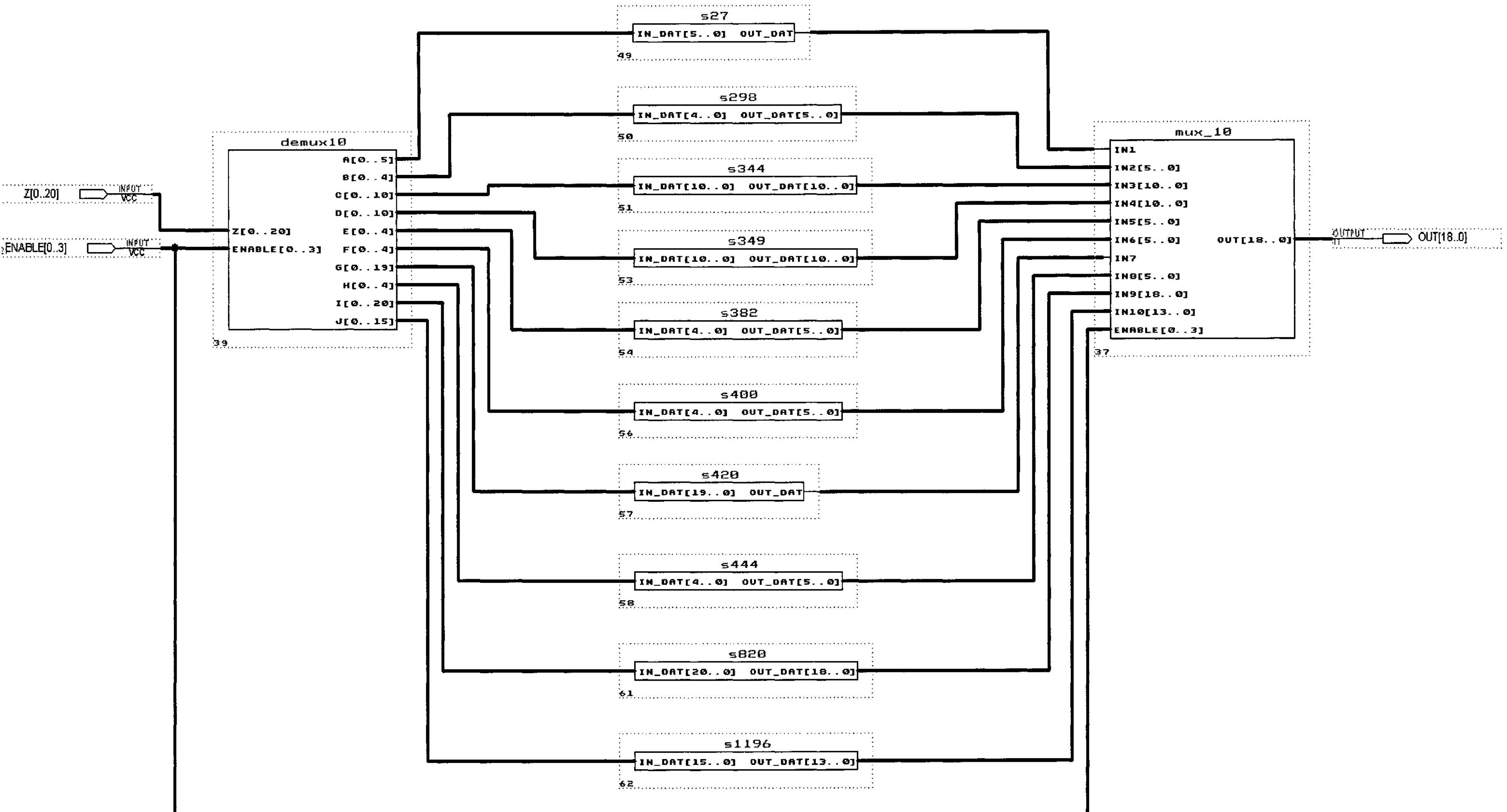


Figure 5.8: *Graphic Design File for SOC_SEQ (SOC_seq.gdf)*

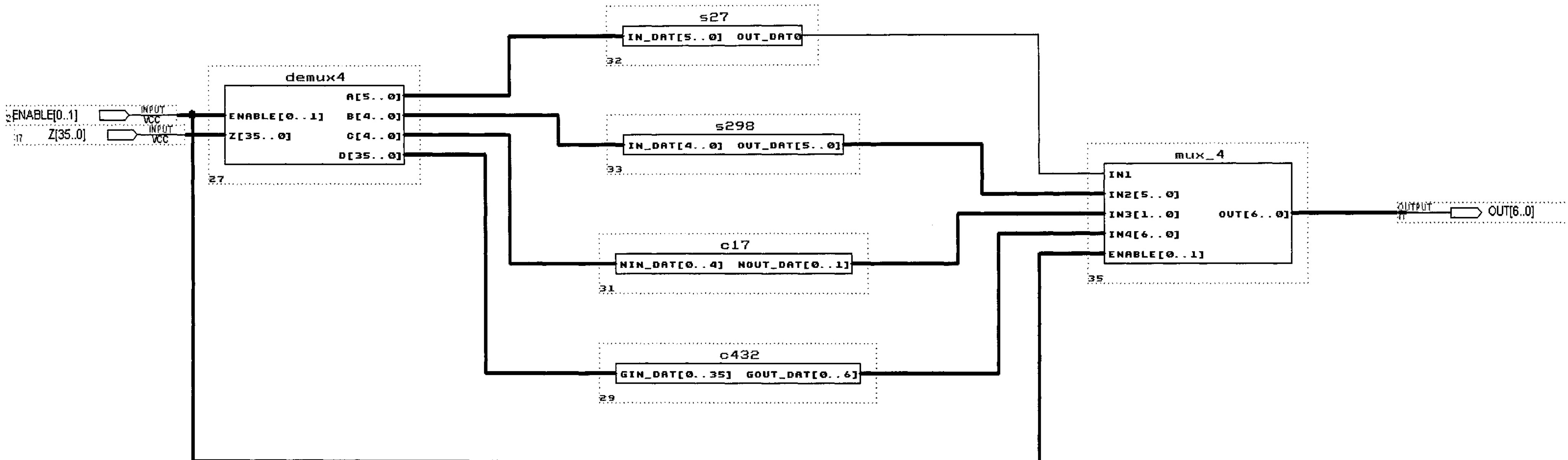


Figure 5.9: *Graphic Design File for SOC_MIXED (SOC_mixed.gdf)*

5.8 SYSTEM CONFIGURATION

We implemented our two programs using C programming language on an IBM-compliant personal computer with AMD Athlon (TM) processor which is AT/AT compatible having 256 MB of RAM . In our implementation, we used system function of C to call MAX PLUS II for compilation and simulation for each CUT in the SOCs. The CUT is represented in Verilog HDL gate-level description.

5.9 RUNNING THE PROGRAMS

To run the programs, first all the executables (.exe) are placed in the same directory then we include the pathname of this directory in our search path. We used the "system" commands to search and run the executables. We placed all the combinational circuits, sequential circuits and mixed circuits in separate folders but in the same directory. The Max Plus II application (maxplus.exe) is also installed in the same directory.

We run the whole test generation process without any intervention of the user. In the beginning, the user is asked to select a core from the SOC. For example, in the SOC_COMB the user is asked to enter an "ENABLE" value. This selects one of the circuits in the SOC_COMB as shown in the ***Figure 5.10*** below. This ENABLE value is set to the Enable of the demultiplexer of ***Figure 5.7*** and the core is selected for test. Accordingly the TAM bus is connected to the CUT. Once a selection is made, the second program is invoked which is the fault simulator. In the example shown in the figure, we selected the circuit c499 (ENABLE = 0010). The ***Figure 5.11*** shows that the CUT is c499 and is loaded. The user is asked to enter the test pattern number for the CUT. Once we enter a value for it, the fault simulation starts. Max Plus II starts running in the background. Once all the faults are injected, the fault simulator stops and the result file is generated which is saved in the destination directory. The first program deletes all the intermediate files generated by the Max Plus II. This test generation was random test generation. If we want to perform deterministic test generation, the user is asked to enter

the test vector file (which is also stored in the same directory). The ***Figure 5.12*** shows the screen shot where the user enters the test vector file. The rest of the test generation process is carried out in similar fashion.

We perform the test generation for the other two SOCs similarly. In the second SOC, SOC_SEQ, the user is not asked for the number of test patterns. The fault simulator generates random test patterns for sequential circuits (discussed in Chapter 4). ***Figure 5.13*** show the screen shot of the core selection for SOC_SEQ and ***Figure 5.14*** shows the screen shot of the core selection of the SOC_MIXED.

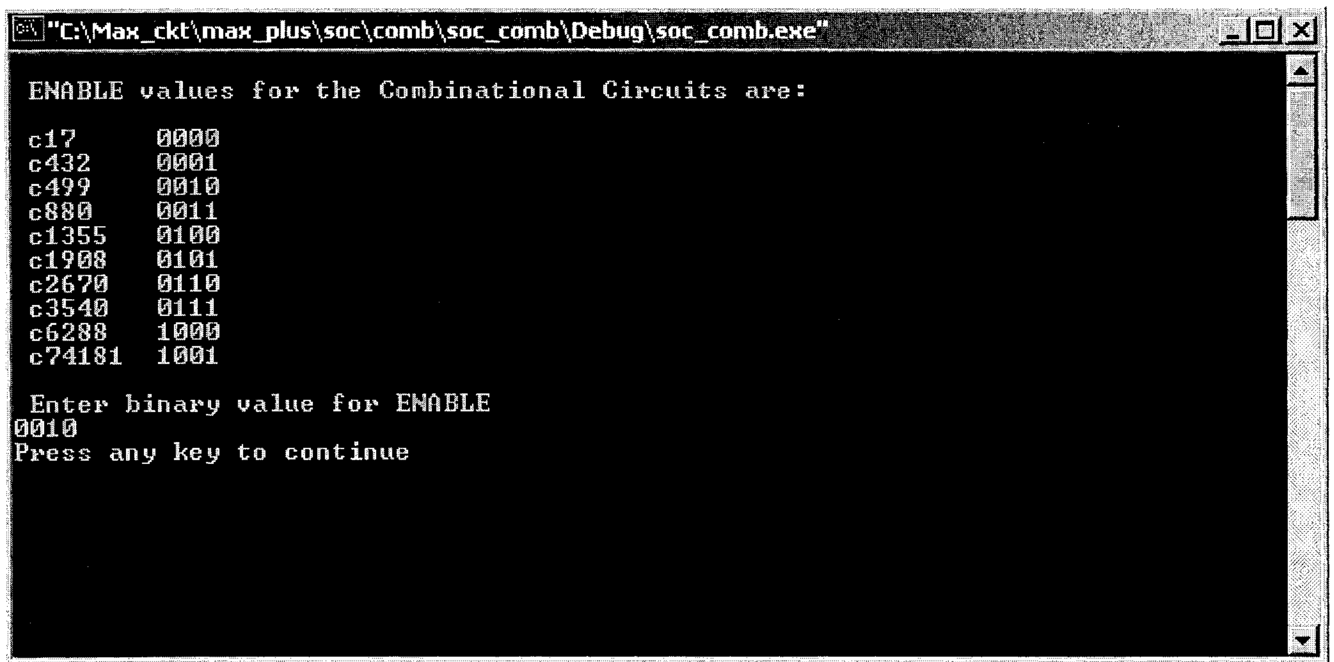


Figure 5.10: *Screen shot of SOC_comb.exe*

```
C:\Max_ckt\max_plus\soc\comb\soc_comb\comb_c499.exe

The File Name is c499
Opening file     :        "c499.v"
Input numbers    :        41
Output numbers   :        32
Wire numbers     :        0

Input is Gin_dat0 Gin_dat1 Gin_dat2 Gin_dat3 Gin_dat4 Gin_dat5 Gin_dat6 Gin_dat7
 Gin_dat8 Gin_dat9 Gin_dat10 Gin_dat11 Gin_dat12 Gin_dat13 Gin_dat14 Gin_dat15 G
in_dat16 Gin_dat17 Gin_dat18 Gin_dat19 Gin_dat20 Gin_dat21 Gin_dat22 Gin_dat23 G
in_dat24 Gin_dat25 Gin_dat26 Gin_dat27 Gin_dat28 Gin_dat29 Gin_dat30 Gin_dat31 G
in_dat32 Gin_dat33 Gin_dat34 Gin_dat35 Gin_dat36 Gin_dat37 Gin_dat38 Gin_dat39 G
in_dat40

Output is Gout_dat0 Gout_dat1 Gout_dat2 Gout_dat3 Gout_dat4 Gout_dat5 Gout_dat6
Gout_dat7 Gout_dat8 Gout_dat9 Gout_dat10 Gout_dat11 Gout_dat12 Gout_dat13 Gout_d
at14 Gout_dat15 Gout_dat16 Gout_dat17 Gout_dat18 Gout_dat19 Gout_dat20 Gout_dat2
1 Gout_dat22 Gout_dat23 Gout_dat24 Gout_dat25 Gout_dat26 Gout_dat27 Gout_dat28 G
out_dat29 Gout_dat30 Gout_dat31

Enter test pattern number        :
```

Figure 5.11: *Screen shot of comb_c499.exe*

```
C:\Max_ckt\max_plus\soc\comb\comb_c499_pattern.exe

The File Name is c499
Opening file     :        "c499.v"
Input numbers    :        41
Output numbers   :        32
Wire numbers     :        0

Input is Gin_dat0 Gin_dat1 Gin_dat2 Gin_dat3 Gin_dat4 Gin_dat5 Gin_dat6 Gin_dat7
 Gin_dat8 Gin_dat9 Gin_dat10 Gin_dat11 Gin_dat12 Gin_dat13 Gin_dat14 Gin_dat15 G
in_dat16 Gin_dat17 Gin_dat18 Gin_dat19 Gin_dat20 Gin_dat21 Gin_dat22 Gin_dat23 G
in_dat24 Gin_dat25 Gin_dat26 Gin_dat27 Gin_dat28 Gin_dat29 Gin_dat30 Gin_dat31 G
in_dat32 Gin_dat33 Gin_dat34 Gin_dat35 Gin_dat36 Gin_dat37 Gin_dat38 Gin_dat39 G
in_dat40

Output is Gout_dat0 Gout_dat1 Gout_dat2 Gout_dat3 Gout_dat4 Gout_dat5 Gout_dat6
Gout_dat7 Gout_dat8 Gout_dat9 Gout_dat10 Gout_dat11 Gout_dat12 Gout_dat13 Gout_d
at14 Gout_dat15 Gout_dat16 Gout_dat17 Gout_dat18 Gout_dat19 Gout_dat20 Gout_dat2
1 Gout_dat22 Gout_dat23 Gout_dat24 Gout_dat25 Gout_dat26 Gout_dat27 Gout_dat28 G
out_dat29 Gout_dat30 Gout_dat31

 Enter vector file name
c499

Enter test pattern number, default value is 51   :
```

Figure 5.12: *Screen shot of comb_c3540.exe with test vector c499.vec*

```
"C:\Max_ckt\max_plus\soc\seq\soc_seq\Debug\soc_seq.exe"

 ENABLE values for the Combinational Circuits are:

 s27      0000
 s298     0001
 s344     0010
 s349     0011
 s382     0100
 s400     0101
 s420     0110
 s444     0111
 s820     1000
 s1196    1001

 Enter binary value for ENABLE
0001
Press any key to continue
```

Figure 5.13 Screen shot of SOC_seq.exe

```
"C:\Max_ckt\max_plus\soc\Mixed_soc\Debug\soc_mixed.exe"

 ENABLE values for the Sequential and Combinational Circuits are:

 s27      00
 s298     01
 c17      10

 Enter binary value for ENABLE
10
Press any key to continue
```

Figure 5.14 Screen shot of SOC_mixed.exe

5.10 SUMMARY

We start this chapter by giving the system overview. The method how our program does the parsing, is then described. We then talk about the three SOCs that we developed by presenting their Verilog code. We also discuss about Altera design environment that we used for compilation and simulation. We also present the graphic design files for the SOCs. Finally, we describe how the programs run and the connectivity between the programs, supported by some snap shots of the actual program.

CHAPTER 6
EXPERIMENTAL RESULTS

In this chapter, we present in detail the various simulation and compilation results we performed on the cores. We have divided the chapter in three parts. In the first part, we present tabular results and figures about the SOC comprised of all the sequential circuits. In the second part, we talk about the SOC comprised of all the combinational circuits. And finally we talk about the SOC comprised of both sequential and combinational circuits. We developed four mixed SOCs, which we discussed in the last part. We generated the results using C programming language on an IBM-compliant personal computer with AMD Athlon (TM) processor which is AT/AT compatible having 256 MB of RAM. In our implementation, we used system function of C to call ALTERA's MAX PLUS II for compilation and simulation for each CUT in the SOCs. The CUT is represented in Verilog HDL gate-level description.

6.3 SEQUENTIAL CIRCUIT RESULTS (SOC_SEQ)

In this section we list the results in the tabular form for the SOC comprising of all the sequential circuits. We named this SOC as SOC_SEQ. Once the CUT is isolated and has been assigned dedicated input and output line through the TAM, the CUT is tested independently. ***Table 6.1.1*** shows the basic specifications of 10 ISCAS 89 Sequential Benchmark Circuits. The graphical results are further shown in the ***Figure 6.1***. ***Table 6.1.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.1.4*** shows the number of injected faults, test vectors, detected faults and fault coverage for the SOC. ***Figure 6.3*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of Wires in the circuit
S27	3	10	6	1	12
S298	14	119	5	6	127
S344	15	160	11	11	164
S349	15	161	11	11	91
S382	21	158	5	6	173
S400	21	164	5	6	53
S420	16	218	20	1	70
S444	21	181	5	6	196
S820	5	289	20	19	275
S1196	18	529	16	14	63

Table 6.1.1: *Basic Specifications of 10 ISCAS 89 Sequential Benchmark Circuits*

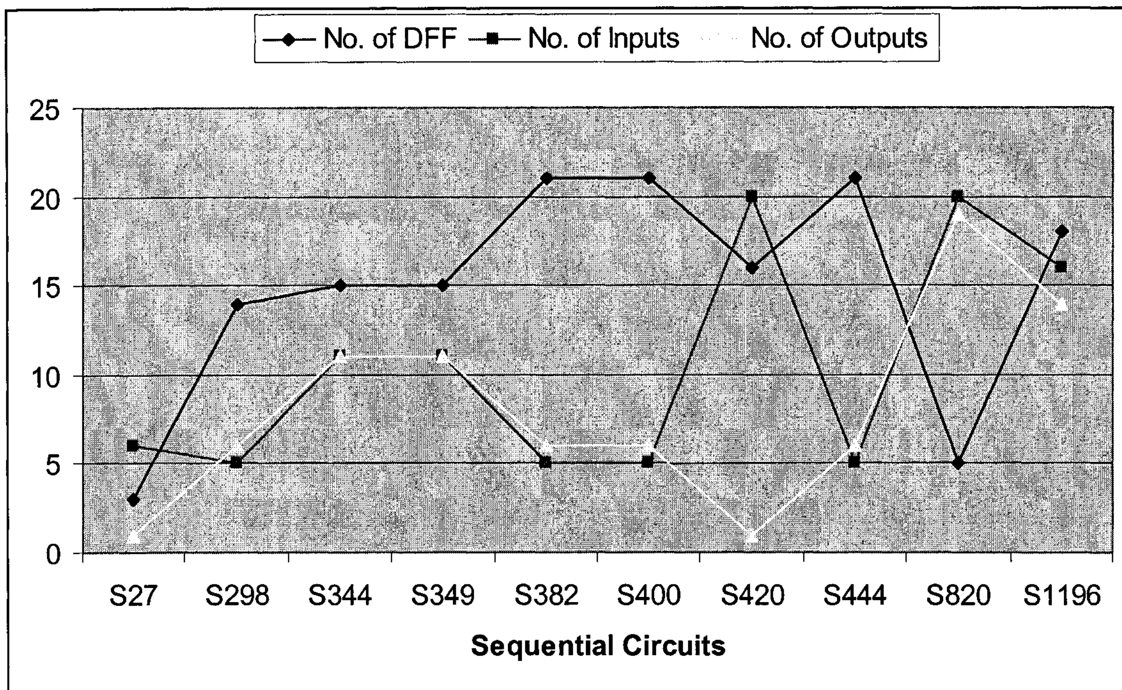


Figure 6.1: *No. Of DFFs, Inputs and Outputs of Sequential Circuits*

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of wires in the circuit	No. of test vectors used for simulation	Total Testing Time (in seconds)
S27	3	10	12	14,999	221
S298	14	119	127	14,999	1196
S344	15	160	164	14,999	1106
S349	15	161	91	14,999	1269
S382	21	158	173	14,999	692
S400	21	164	53	19,999	703
S420	16	218	70	14,999	685
S444	21	181	196	14,999	202
S820	5	289	275	19,999	1795
S1196	18	529	63	14,999	1376

Table 6.1.2: *Testing Time and Number of DFFs, Gates, Inverters, and Wires of Benchmark Sequential Circuits*

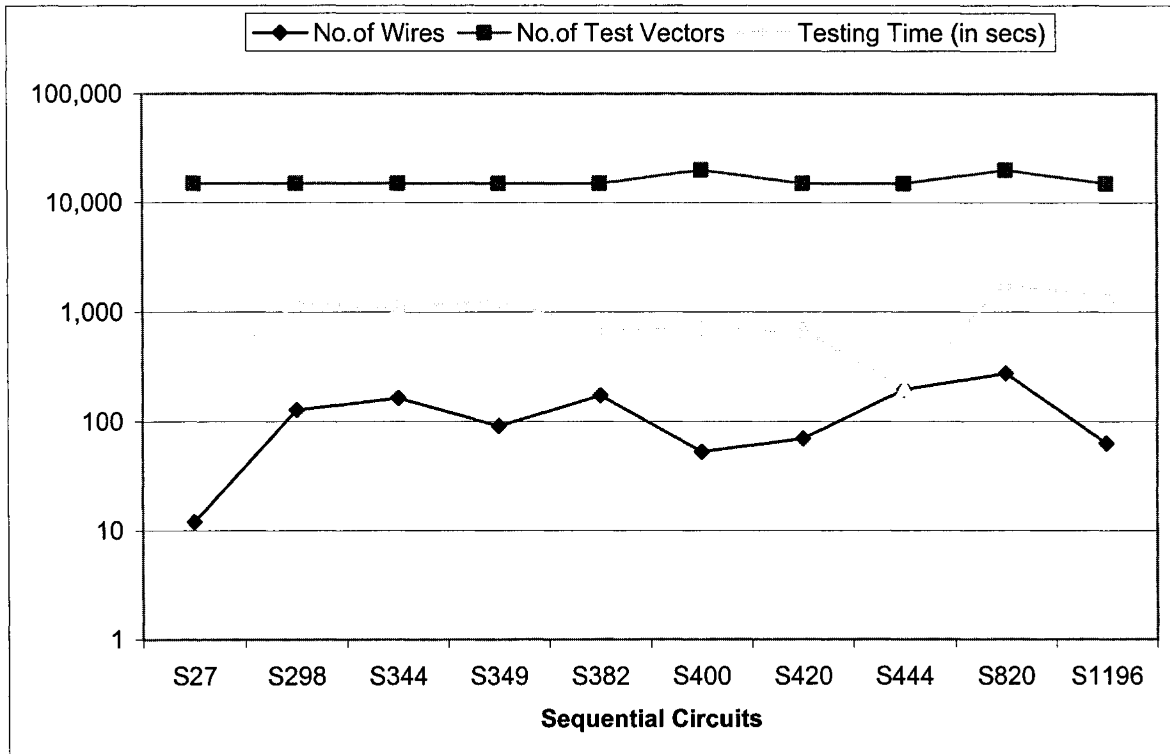


Figure 6.2: *No. Of Wires, Vectors and Testing Time of Sequential Circuits*

Circuit Name	No. Of DFFs in the circuit	No. of wires in the circuit	No. of detected faults
S27	3	12	30
S298	14	127	244
S344	15	164	189
S349	15	91	217
S382	21	173	77
S400	21	53	24
S420	16	70	45
S444	21	196	36
S820	5	275	132
S1196	18	63	171

Table 6.1.3: *Number of DFFs, Wires and Detected Faults*

Circuit Name	No. of test vectors used for simulation	No. of faults injected	No. of faults detected	Fault coverage (%)
S27	14,999	34	30	88.23
S298	14,999	272	244	89.7
S344	14,999	234	189	80.77
S349	14,999	222	217	97.75
S382	14,999	174	77	76.55
S400	19,999	124	24	19.35
S420	14,999	178	45	25.28
S444	14,999	54	36	66.67
S820	19,999	194	132	68.04
S1196	14,999	182	171	93.96

Table 6.1.4: *Fault Coverage, Number of Injected Faults, Test Vectors, and Detected Faults*

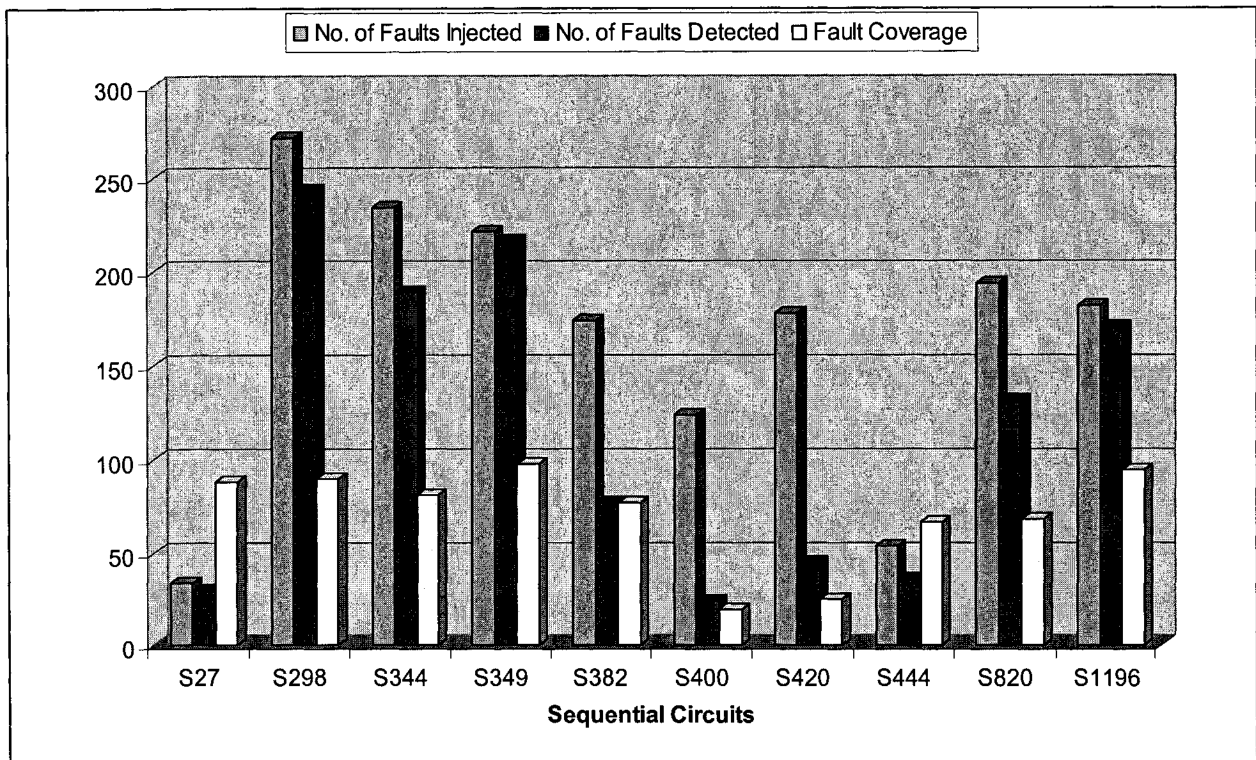


Figure 6.3: *Number of Injected Faults, Detected Faults and Fault Coverage of Sequential Circuits*

6.2 COMBINATIONAL CIRCUIT RESULTS (SOC_COMB)

In this section we list the results in the tabular form for the SOC comprising of all the combinational circuits. We named this SOC as SOC_COMB. Once the CUT is isolated and has been assigned dedicated input and output line through the TAM, the CUT is tested independently. We used ***random test generation technique*** for simulation of the each CUT. ***Table 6.2.1*** shows the basic specifications of 10 ISCAS 85 Combinational Benchmark Circuits. The graphical results are further shown in the ***Figure 6.4***. ***Table 6.2.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.2.4*** shows the number of collapsed faults, detected faults, undetected faults and fault coverage for the SOC. ***Figure 6.6*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of gates in the circuit	No. Of inputs in the circuit	No. of outputs in the circuit	No. of Wires in the circuit
C17	6	5	2	4
C432	160	36	7	153
C499	202	41	32	170
C880	433	60	26	357
C1355	546	41	32	516
C1908	880	33	25	855
C2670	1193	157	64	1129
C3540	1669	50	22	1638
C6288	2416	32	31	2386
C74181	58	14	8	50

Table 6.2.1: *Basic Specifications of 10 ISCAS 85 Combinational Benchmark Circuits*

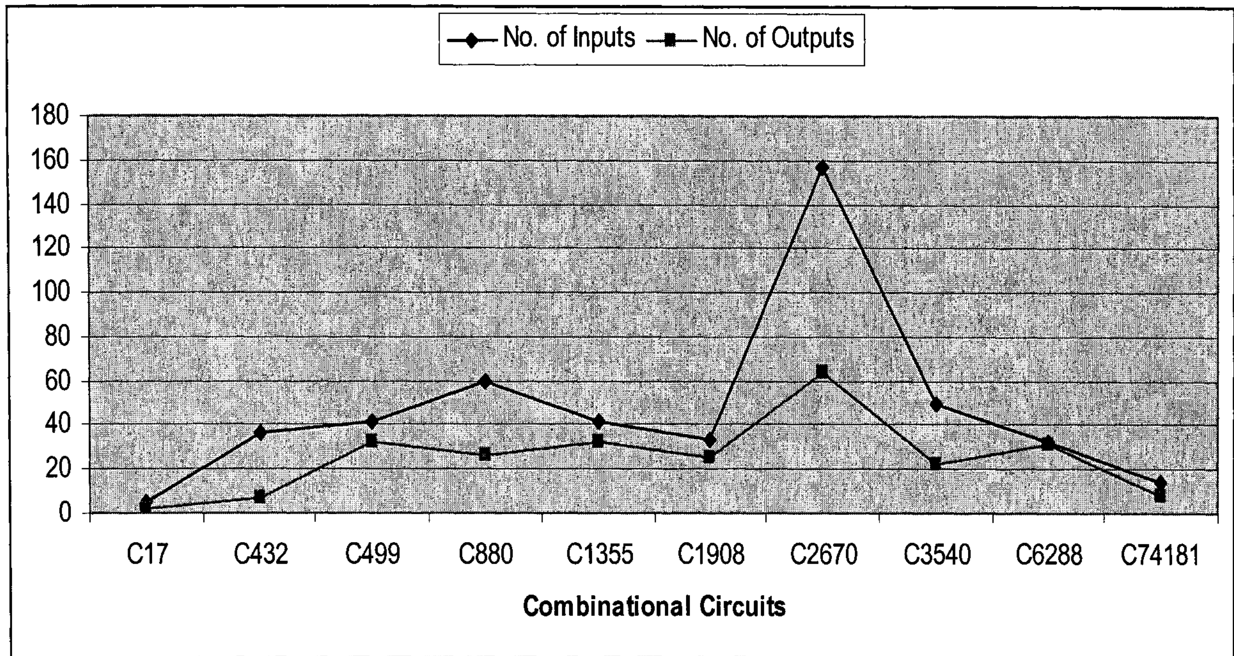


Figure 6.4: No. Of Inputs and Outputs of Combinational Circuits

Random Test Generation for Combinational Circuits

Circuit Name	No. of gates in the circuit	No. of Wires in the circuit	No. of test patterns applied for simulation	Total Testing Time (in seconds)
C17	6	5	23	21
C432	160	36	30	188
C499	202	41	20	106
C880	433	60	30	271
C1355	546	41	20	110
C1908	880	33	20	126
C2670	1193	157	30	1355
C3540	1669	50	25	372
C6288	2416	32	20	58
C74181	58	14	20	21

Table 6.2.2: *Testing Time and Number of Gates, Inverters, and Wires of Benchmark Combinational Circuits*

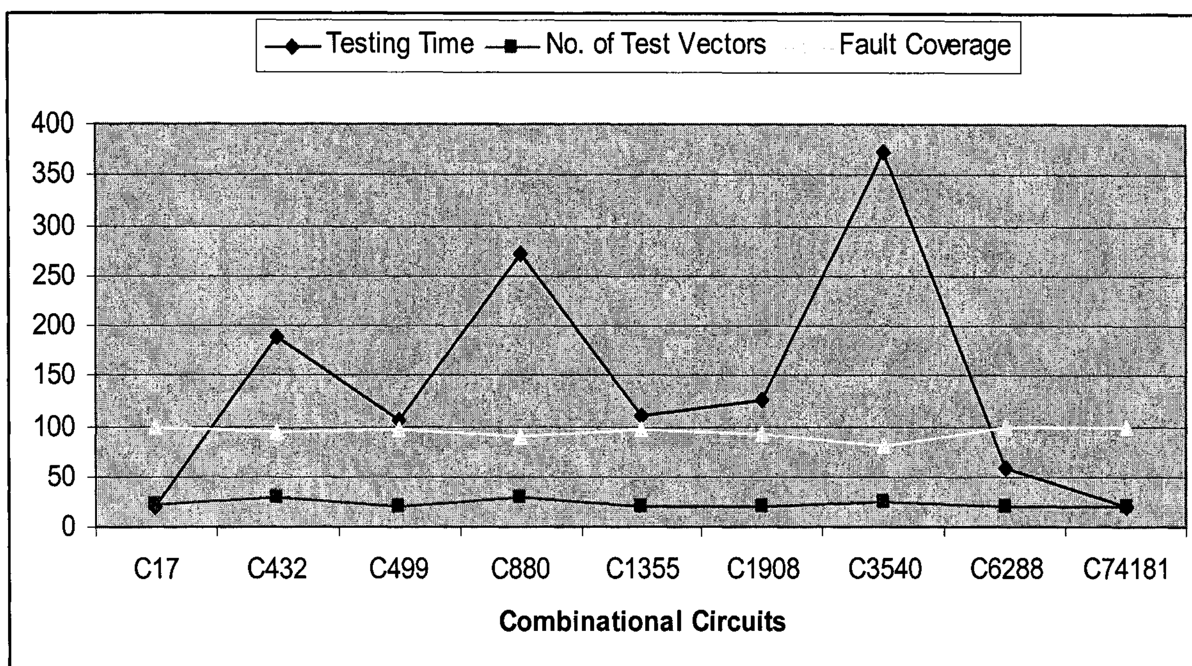


Figure 6.5: *No. Of Test Vectors, Testing Time and Fault Coverage of Combinational Circuits*

Circuit Name	No. of test patterns applied for simulation	No. of collapsed faults	No. of faults detected
C17	23	22	22
C432	30	86	81
C499	20	146	142
C880	30	172	156
C1355	20	146	142
C1908	20	119	107
C2670	30	442	376
C3540	25	144	117
C6288	20	126	126
C74181	20	44	44

Table 6.2.3: *Number of Injected Faults, Test Vectors, and Detected Faults*

Circuit Name	No. of collapsed faults	No. of faults detected	No. of faults Undetected	Fault Coverage (in %)
C17	22	22	0	100
C432	86	81	5	94.19
C499	146	142	4	97.26
C880	172	156	16	90.7
C1355	146	142	4	97.26
C1908	119	107	9	92.241
C2670	442	376	66	85.07
C3540	144	119	25	81.25
C6288	126	126	0	100
C74181	44	44	0	100

Table 6.2.4: *Fault Coverage, Number of Collapsed Faults, Detected Faults, and Undetected Faults*

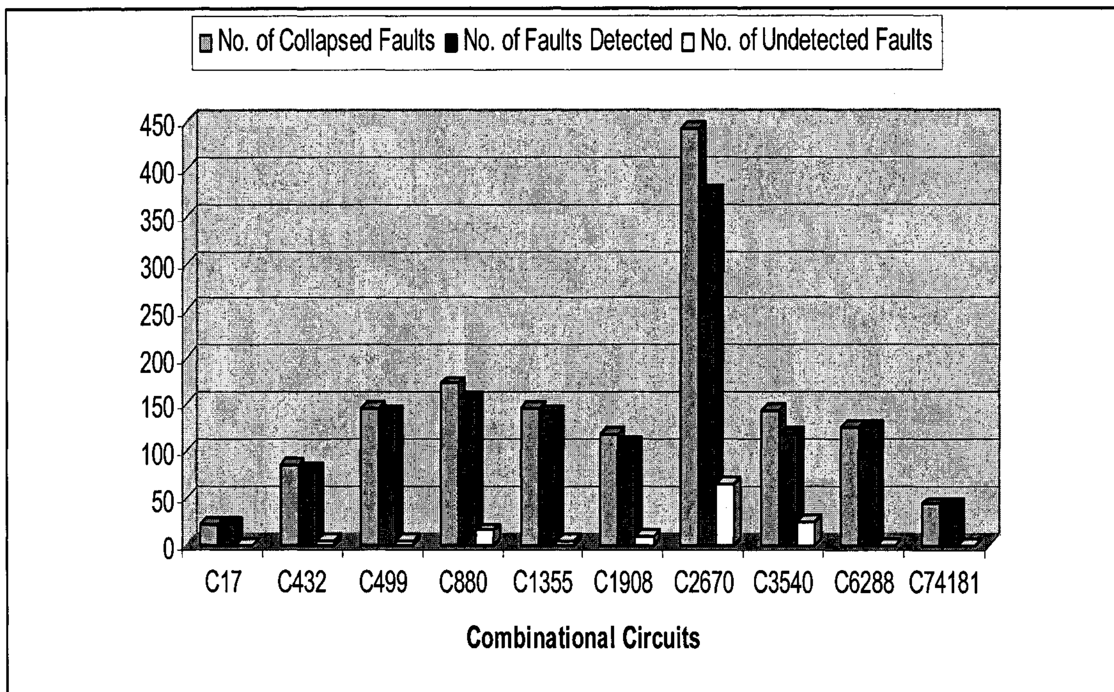


Figure 6.6: *No. Of Collapsed, Detected and Undetected Faults of Combinational Circuits*

6.3 Deterministic Test Generation for Combinational Circuits (MINTEST Vectors Used)

In this section, we used ***deterministic test generation*** for the SOC. We applied the MINTEST vectors as input vectors to each of our CUT. The results are then generated as follows. ***Table 6.3.1*** emphasizes on the CPU testing time and the number of test vectors applied. The graphical results are further shown in the ***Figure 6.7***. ***Table 6.3.3*** shows the number of collapsed faults, detected faults, undetected faults and fault coverage for the SOC. ***Figure 6.8*** shows the bar chart of the number of collapsed, injected and detected faults.

Circuit Name	No. of gates in the circuit	No. of Wires in the circuit	No. of test patterns applied for simulation (MINTEST VECTORS)	Total Testing Time (in seconds)
C17	6	5	-	-
C432	160	36	26	167
C499	202	41	51	52
C880	433	60	15	106
C1355	546	41	83	81
C1908	880	33	105	194
C2670	1193	157	-	-
C3540	1669	50	-	-
C6288	2416	32	11	55
C74181	58	14	-	-

Table 6.3.1: *Testing Time and Number of Gates, Inverters, and Wires of Benchmark Combinational Circuits*

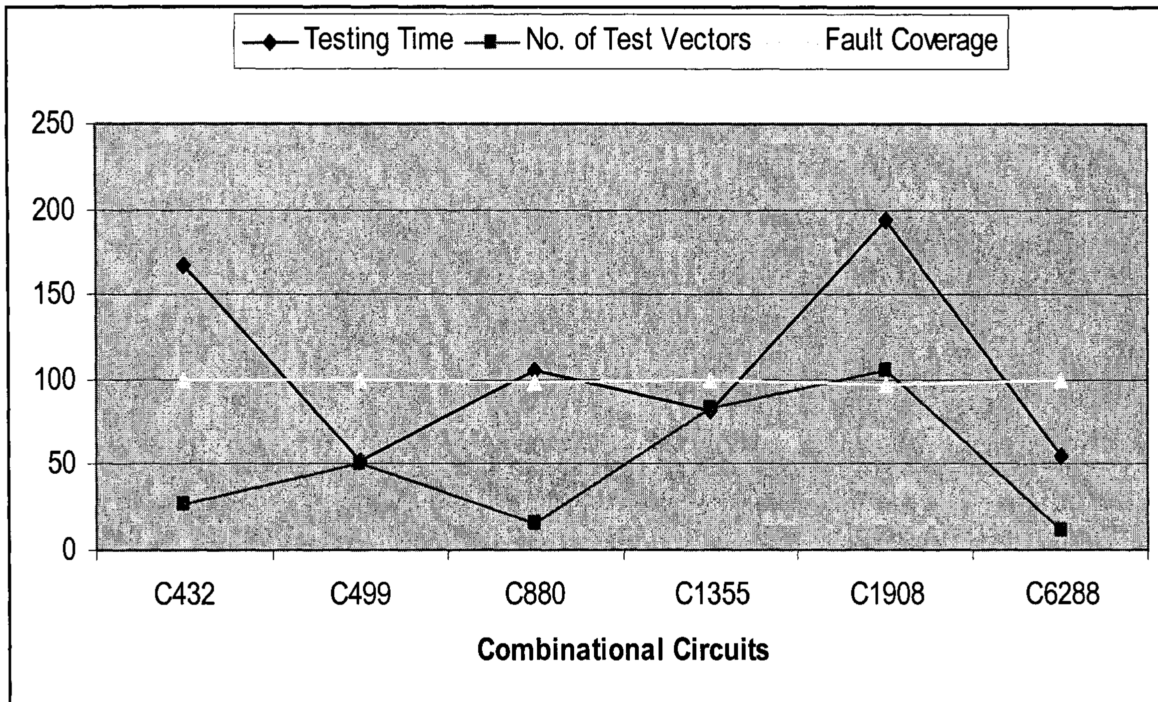


Figure 6.7: *No. Of Test Vectors, Testing Time and Fault Coverage of Combinational Circuits (Deterministic Test)*

Circuit Name	No. of test patterns applied for simulation	No. of collapsed faults	No. of faults detected
C17	-	-	-
C432	26	86	86
C499	51	146	146
C880	15	172	170
C1355	83	146	146
C1908	105	116	113
C2670	-	-	-
C3540	-	-	-
C6288	11	126	126
C74181	-	-	-

Table 6.3.2: *Number of Injected Faults, Test Vectors, and Detected Faults*

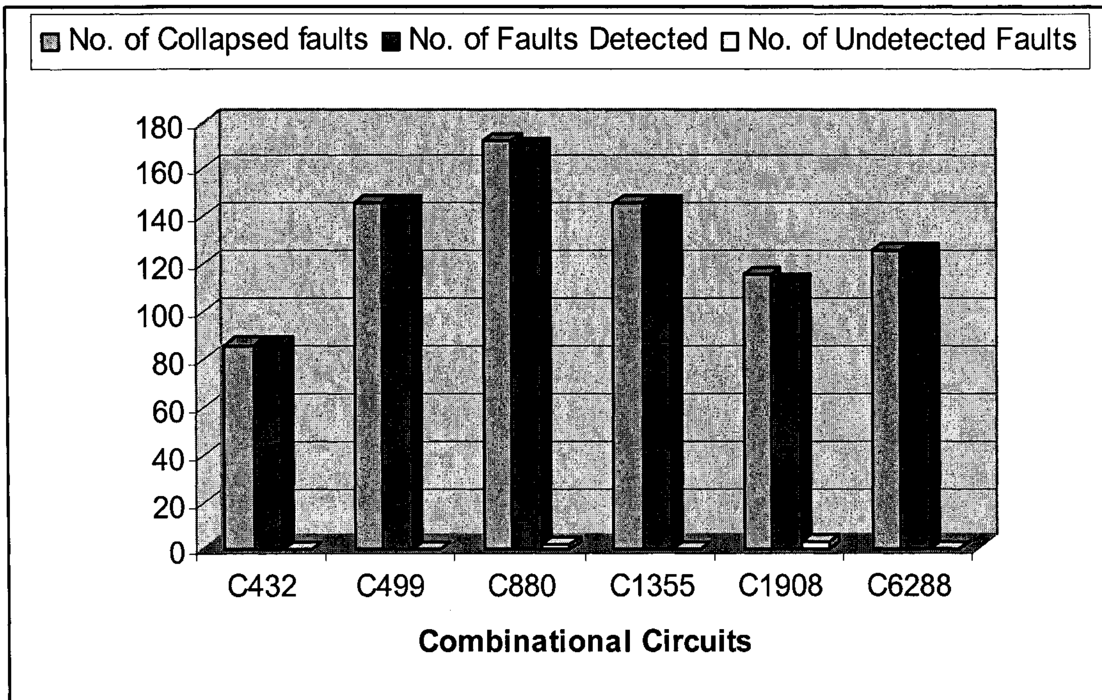


Figure 6.8: *No. Of Collapsed, Detected and Undetected Faults of Combinational Circuits (Deterministic Test)*

Circuit Name	No. of faults detected	No. of Collapsed Faults	No. of faults Undetected	Fault Coverage (in %)
C17	-	-	-	-
C432	86	86	0	100
C499	146	146	0	100
C880	170	172	2	98.84
C1355	146	146	0	100
C1908	113	116	3	97.41
C2670	-	-	-	-
C3540	-	-	-	-
C6288	126	126	0	100
C74181	-	-	-	-

Table 6.3.3 *Fault Coverage, Number of Collapsed Faults, Detected Faults, and Undetected Faults*

6.4 COMBINATIONAL & SEQUENTIAL CIRCUITS RESULTS (MIXED_SOC)

In this section we list the results in the tabular form for the SOC comprising of both sequential and combinational circuits. We named this SOC as MIXED_SOC. We designed four mixed SOCs and named then as MIXED_SOC_1, MIXED_SOC_2, MIXED_SOC_3 and MIXED_SOC_4.

6.4.1 (MIXED_SOC_1)

Table 6.4.1.1 shows the basic specifications of ISCAS 89 Sequential and ISCAS 85 Combinational Benchmark Circuits. The graphical results are further shown in the ***Figure 6.9***. ***Table 6.4.1.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.4.1.3*** shows the number of injected faults, test vectors, detected faults and fault coverage for the SOC. ***Figure 6.11*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of Wires of the circuit
S27	3	10	6	1	12
S298	14	119	5	6	127
S344	15	160	11	11	164
S349	15	161	11	11	91
C17	-	6	5	2	4
C1355	-	546	41	32	516
C1908	-	880	33	25	855

Table 6.4.1.1: *Basic Specifications of 2 ISCAS 89 Sequential & Combinational Benchmark Circuits*

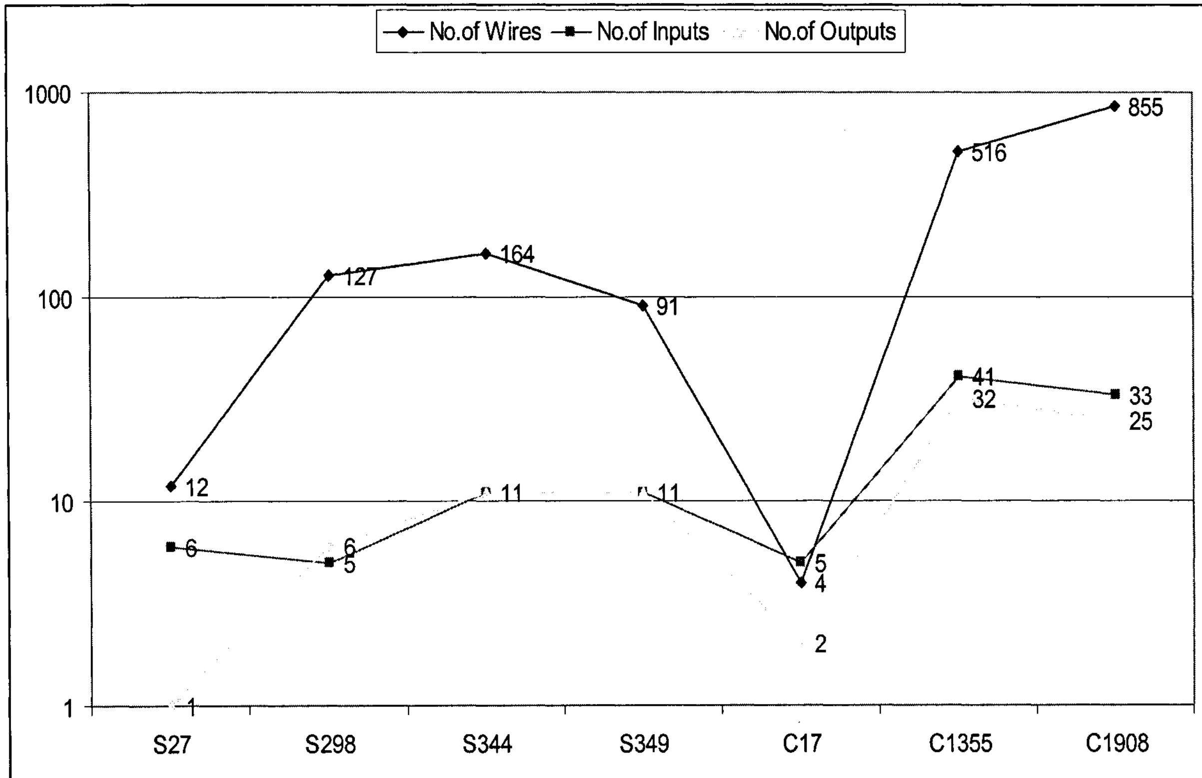


Figure 6.9: *No. Of Wires, Inputs and Outputs of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of wires in the circuit	Total Testing Time (in seconds)
S27	14,999	12	221
S298	14,999	127	1196
S344	14,999	164	1106
S349	14,999	91	1269
C17	23	4	21
C1355	20	516	110
C1908	20	855	126

Table 6.4.1.2: *Number of Wires, Test Vectors and Testing Time*

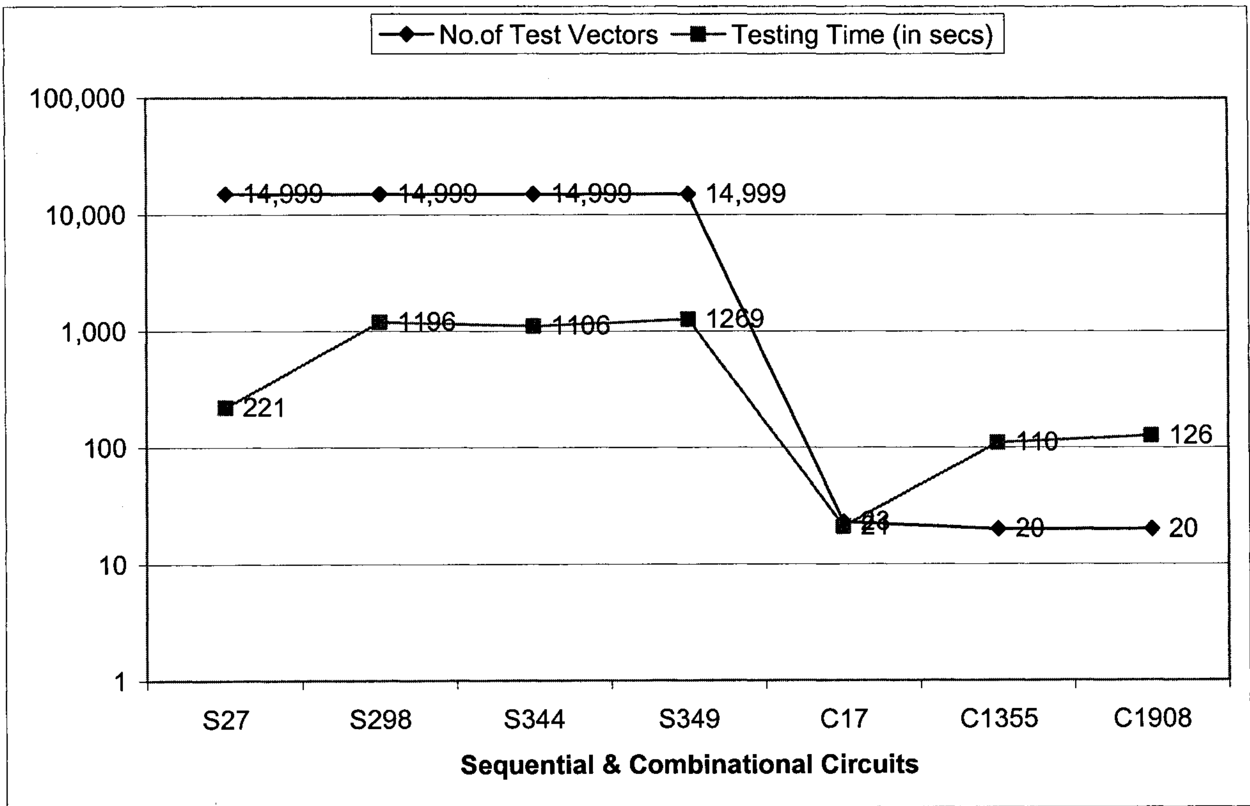


Figure 6.10: *No. Of Test Vectors and Testing Time of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of faults injected	No. of faults detected	**Fault Coverage (in %)**
S27	14,999	34	30	88.23
S298	14,999	272	244	89.7
S344	14,999	234	189	80.77
S349	14,999	222	217	97.75
C17	23	22	22	100
C1355	20	146	142	97.26
C1908	20	119	107	92.241

Table 6.4.1.3: *Fault Coverage, Number of Injected Faults, Test Vectors, and Detected Faults*

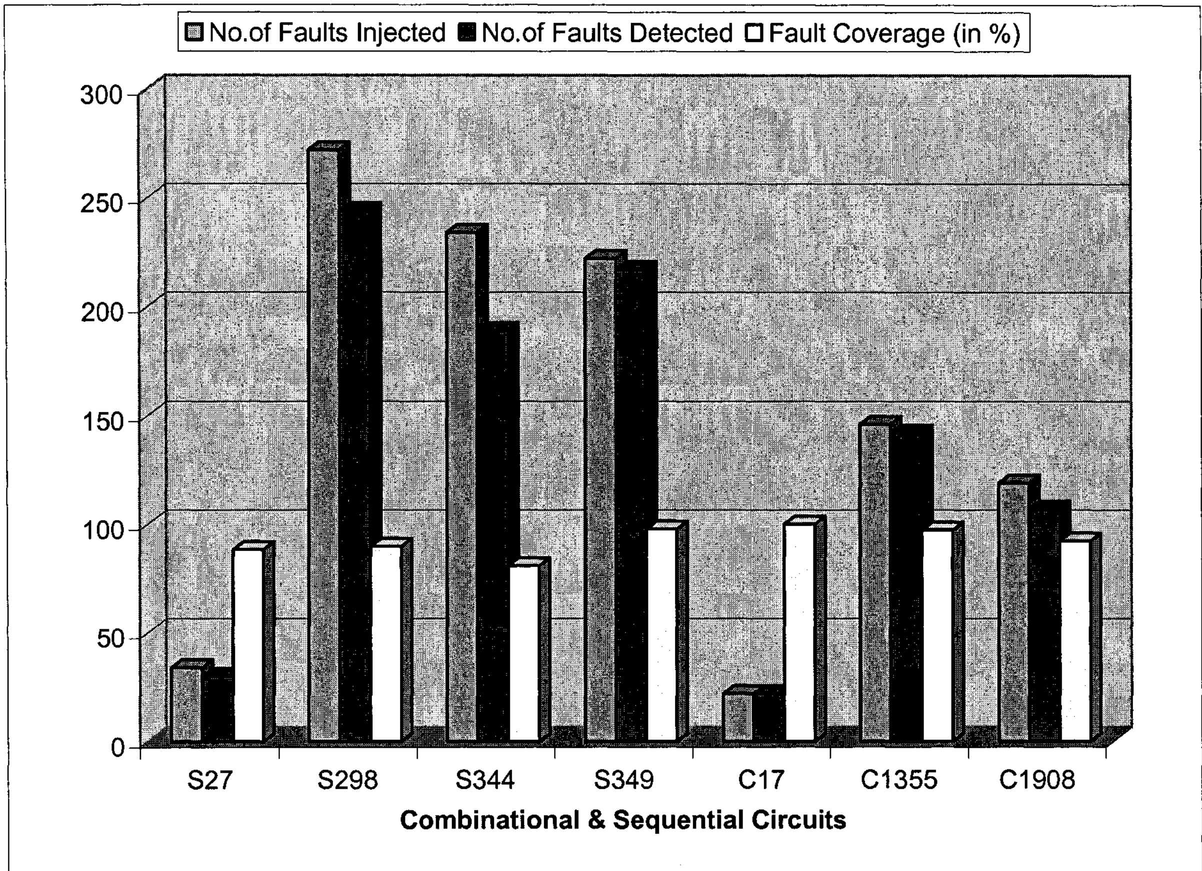


Figure 6.11: *No. Of Injected Faults, Detected Faults and Fault Coverage of Sequential & Combinational Circuits*

6.4.2 (MIXED_SOC_2)

Table 6.4.2.1 shows the basic specifications of ISCAS 89 Sequential and ISCAS 85 Combinational Benchmark Circuits. The graphical results are further shown in the ***Figure 6.12***. ***Table 6.4.2.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.4.2.3*** shows the number of injected faults, test vectors, detected faults and fault coverage for the SOC. ***Figure 6.14*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of Wires of the circuit
S444	21	181	5	6	196
S820	5	289	20	19	275
C880	-	433	60	26	60
C3540	-	1669	50	22	41

Table 6.4.2.1: *Basic Specifications of 2 ISCAS 89 Sequential & Combinational Benchmark Circuits*

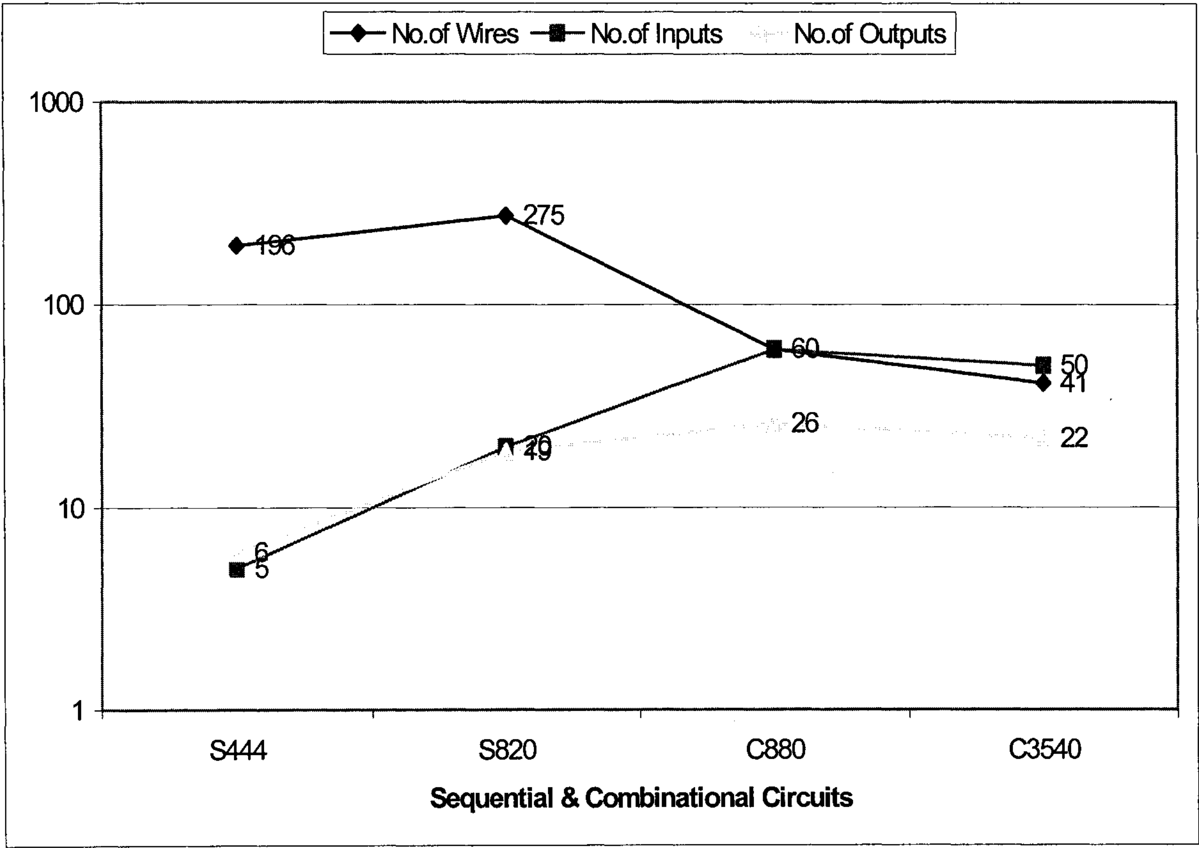


Figure 6.12: *No. Of Wires, Inputs and Outputs of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of wires in the circuit	Total Testing Time (in seconds)
S444	14,999	196	202
S820	14,999	275	1795
C880	30	60	271
C3540	20	41	110

Table 6.4.2.2: *Number of Wires, Test vectors and Testing time*

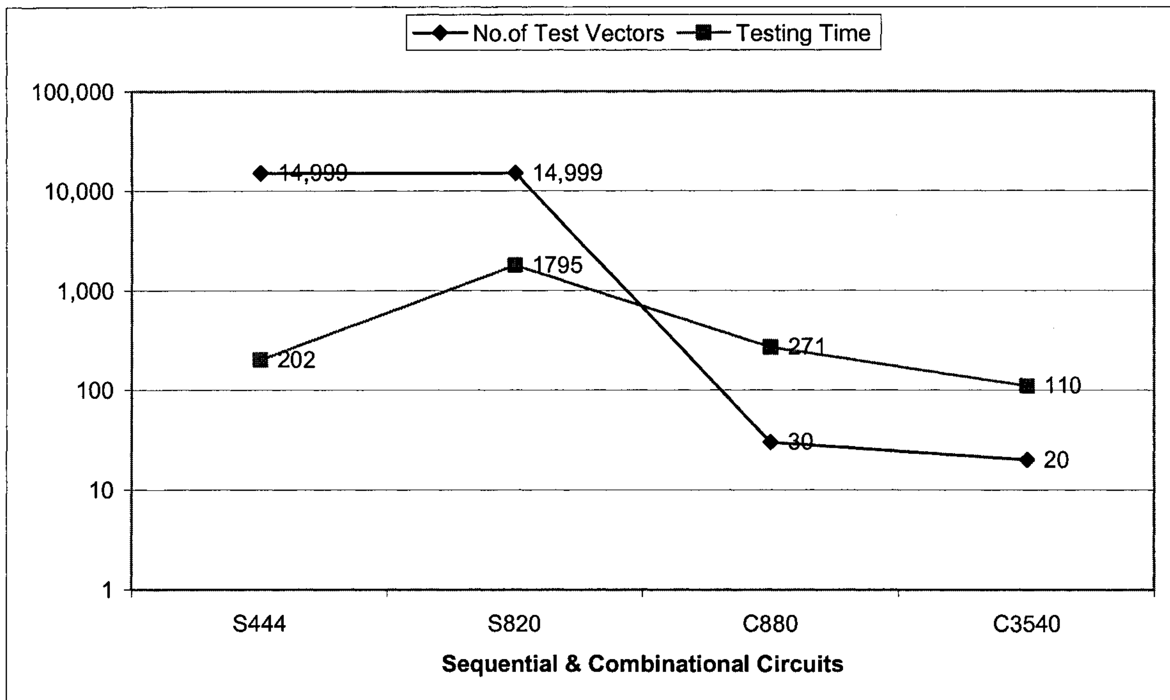


Figure 6.13: *No. Of Test Vectors and Testing Time of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of faults injected	No. of faults detected	**Fault Coverage (in %)**
S444	14,999	194	132	68.04
S820	14,999	182	171	93.96
C880	30	172	156	90.7
C3540	20	144	119	81.25

Table 6.4.2.3: *Fault Coverage, Number of Injected Faults, Test Vectors, and Detected Faults*

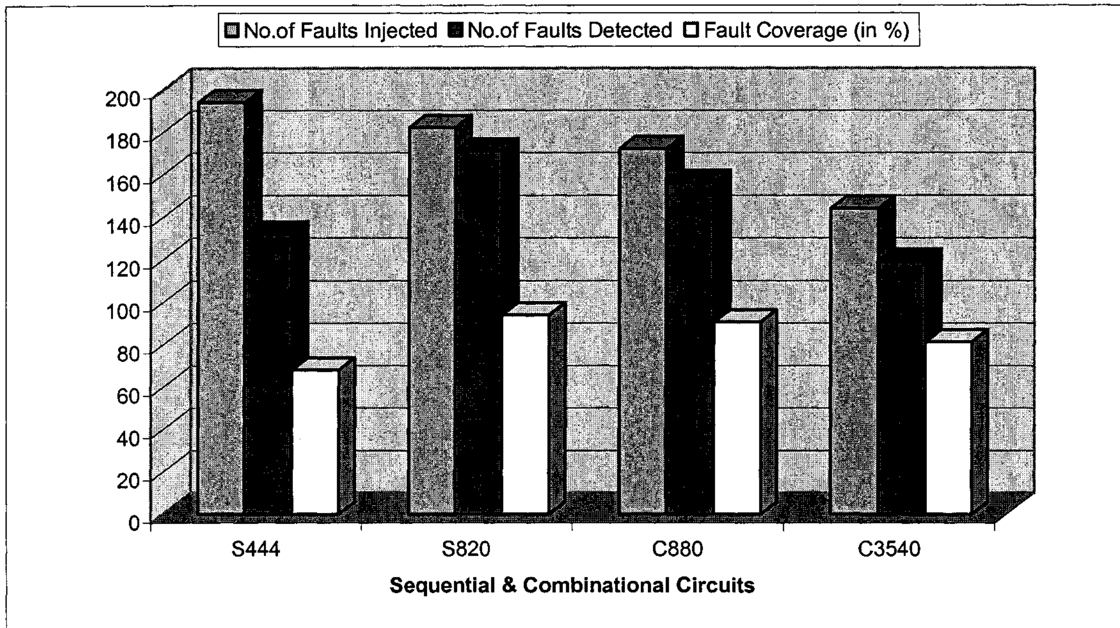


Figure 6.14: *No. Of Injected Faults, Detected Faults and Fault Coverage of Sequential & Combinational Circuits*

6.4.3 (MIXED_SOC_3)

Table 6.4.3.1 shows the basic specifications of ISCAS 89 Sequential and ISCAS 85 Combinational Benchmark Circuits. The graphical results are further shown in the ***Figure 6.15***. ***Table 6.4.3.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.4.3.3*** shows the number of injected faults, test vectors, detected faults and fault coverage for the SOC. ***Figure 6.17*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of Wires in the circuit
S420	21	164	5	6	53
S444	16	218	20	1	70
S820	5	289	20	19	275
C880	-	433	60	26	357
C3540	-	1669	50	22	1638
C6288	-	2416	32	31	2386
C74181	-	58	14	8	50

Table 6.4.3.1: *Basic Specifications of 2 ISCAS 89 Sequential & Combinational Benchmark Circuits*

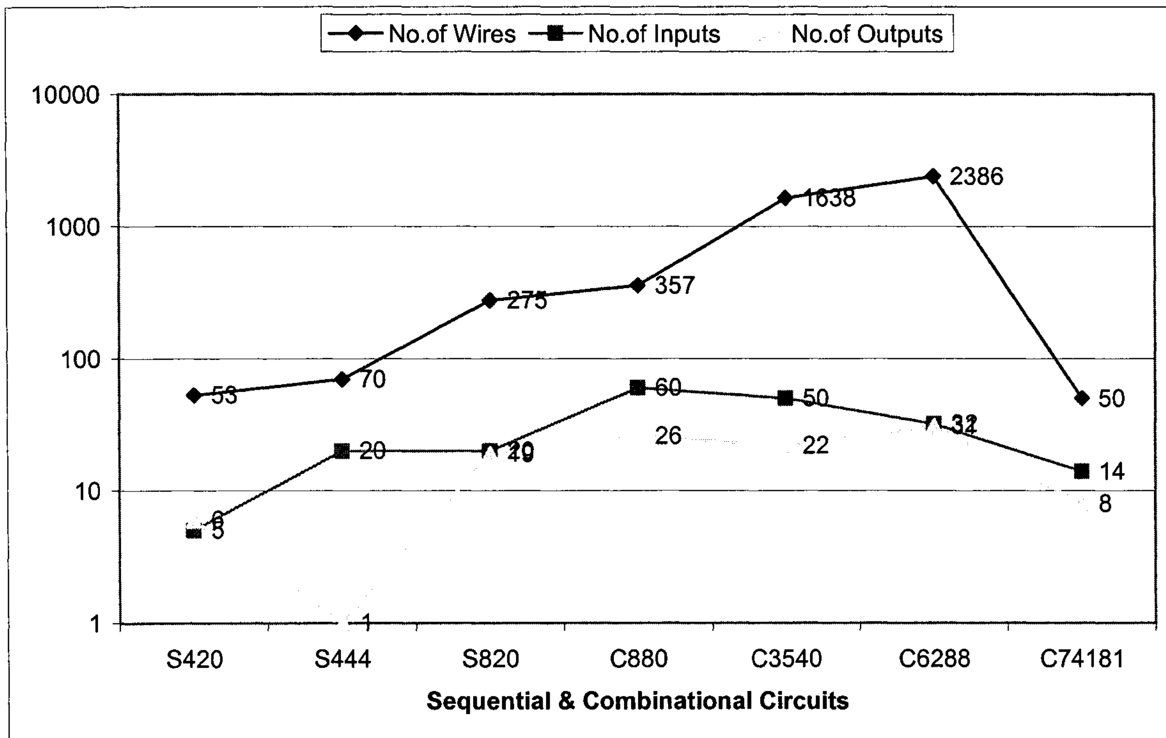


Figure 6.15: *No. Of Wires, Inputs and Outputs of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of wires in the circuit	Total Testing Time (in seconds)
S420	14,999	53	685
S444	14,999	70	202
S820	14,999	275	1795
C880	30	357	271
C3540	25	1638	372
C6288	20	2386	58
C74181	20	50	21

Table 6.4.3.2: *Number of DFFs, Wires and Detected Faults*

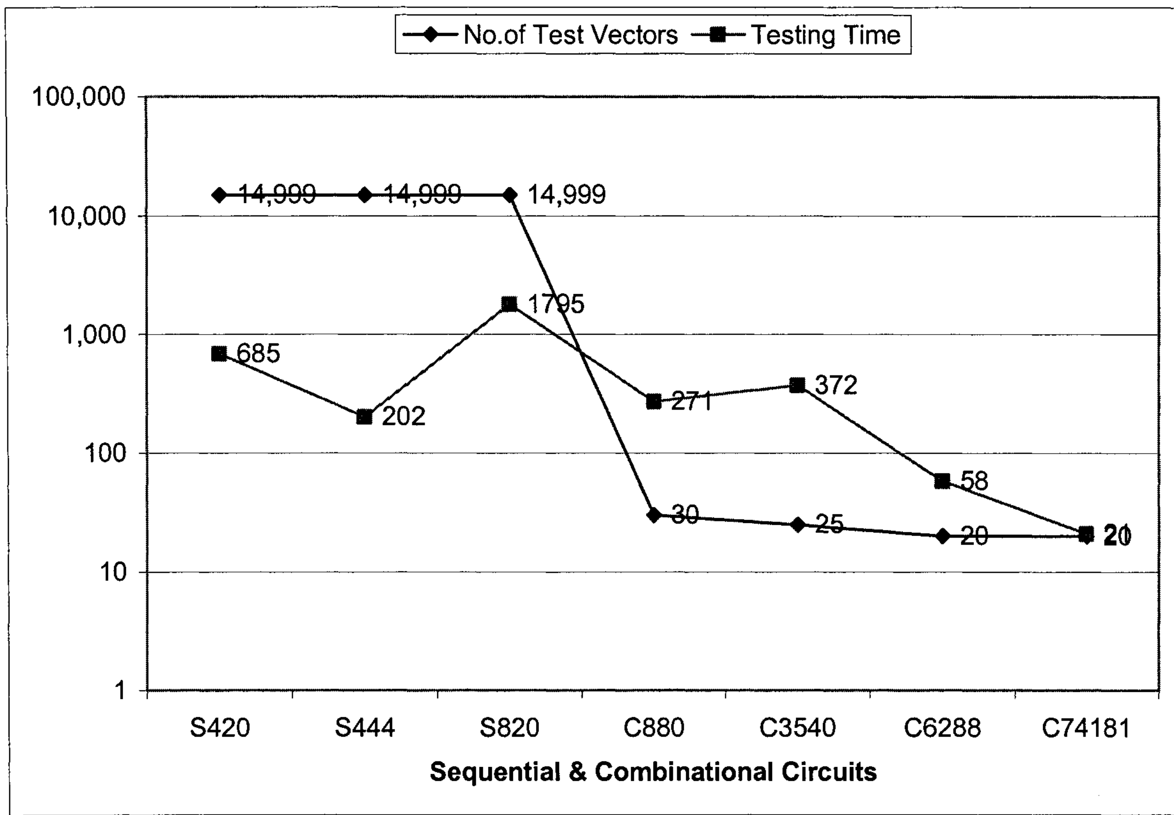


Figure 6.16: *No. Of Test Vectors and Testing Time of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of faults injected	No. of faults detected	**Fault Coverage (in %)**
S420	14,999	178	38	25.28
S444	14,999	54	36	66.67
S820	19,999	194	132	68.04
C880	30	172	156	90.7
C3540	25	144	119	81.25
C6288	20	126	126	100
C74181	20	44	44	100

Table 6.4.3.3: *Fault Coverage, Number of Injected Faults, Test Vectors, and Detected Faults*

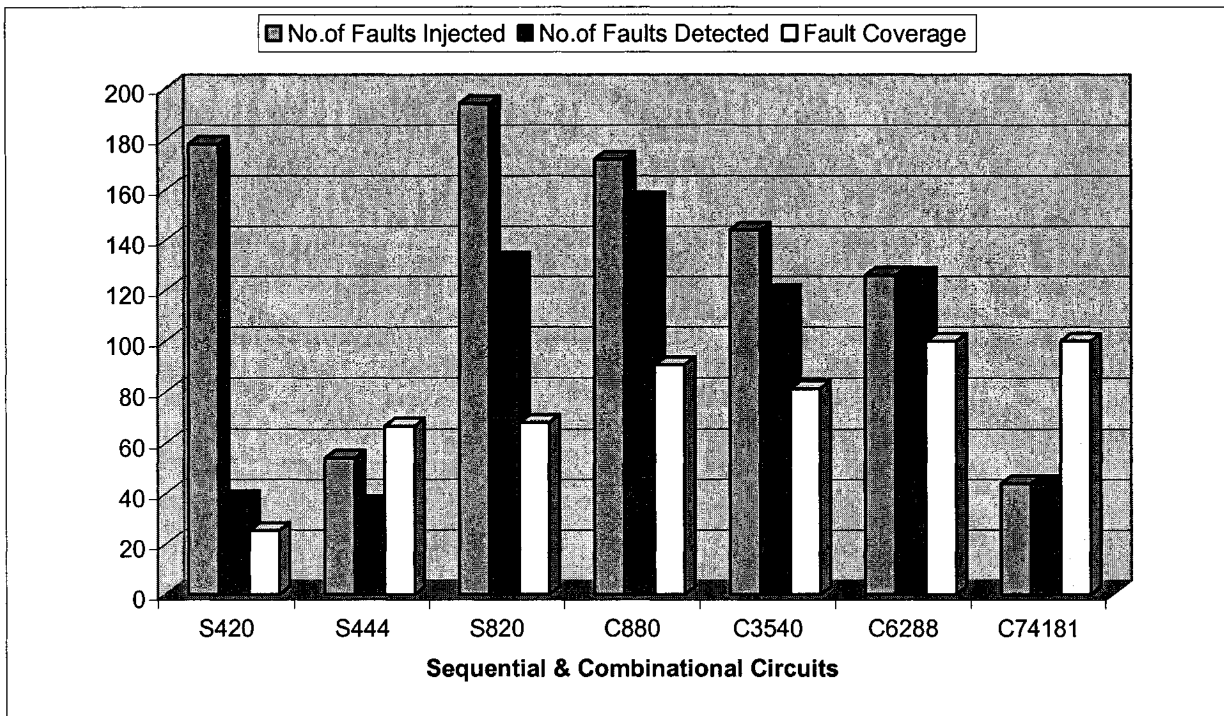


Figure 6.17: *No. Of Injected Faults, Detected Faults and Fault Coverage of Sequential & Combinational Circuits*

6.4.4 (MIXED_SOC_4)

Table 6.4.4.1 shows the basic specifications of ISCAS 89 Sequential and ISCAS 85 Combinational Benchmark Circuits. The graphical results are further shown in the ***Figure 6.18***. ***Table 6.4.4.2*** emphasizes on the CPU testing time and the number of test vectors applied. The CPU testing time is the time taken for the compilation and the simulation of the CUT including the time to get connected to the TAM. ***Table 6.4.4.3*** shows the number of injected faults, test vectors, detected faults and fault coverage for the SOC. ***Figure 6.19*** shows the bar chart of the number of injected faults, detected faults and fault coverage.

Circuit Name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of Wires of the circuit
S349	15	161	11	11	91
S382	21	158	5	6	173
C432	-	160	36	7	153
C499	-	202	41	32	170
C74181	-	58	14	8	50

Table 6.4.4.1: *Basic Specifications of 2 ISCAS 89 Sequential & Combinational Benchmark Circuits*

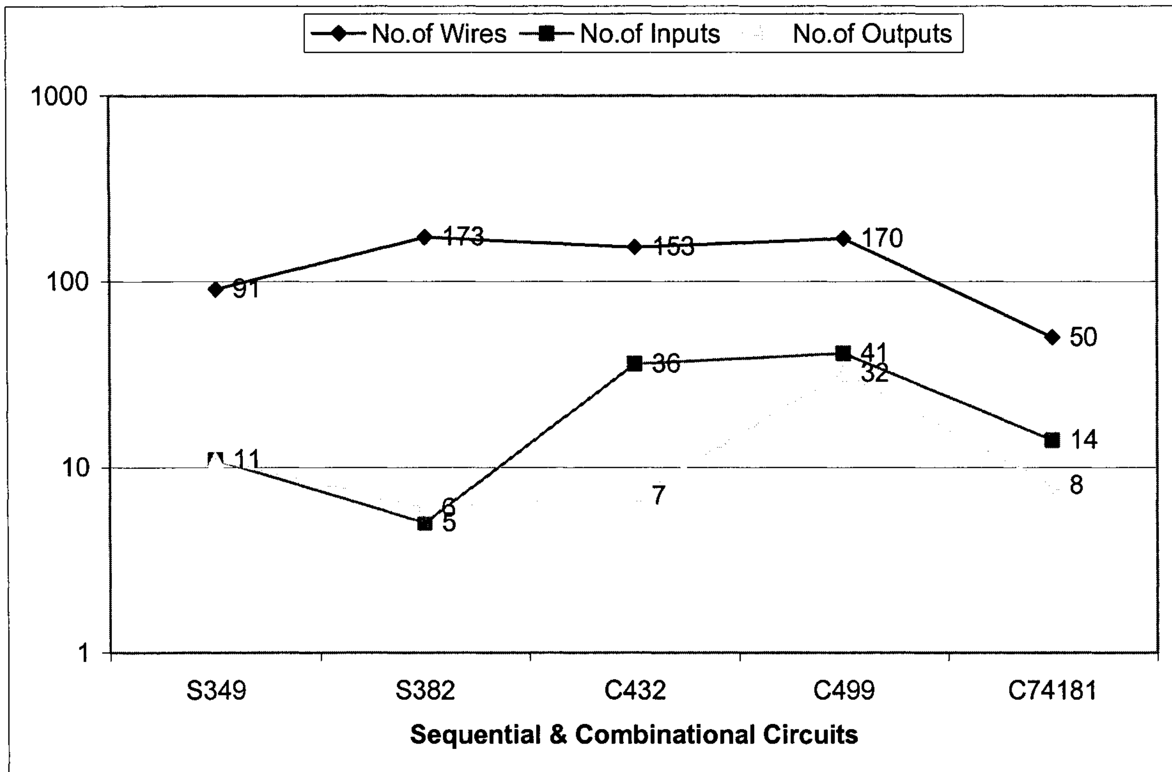


Figure 6.18: *No. Of Wires, Inputs and Outputs of Sequential & Combinational Circuits*

Circuit Name	No. of test vectors used for simulation	No. of wires in the circuit	Total Testing Time (in seconds)
S349	14,999	91	1269
S382	14,999	173	464
C432	30	153	188
C499	20	170	106
C74181	20	50	21

Table 6.4.4.2: *Number Test Vectors, Wires and Testing Time*

Circuit Name	No. of test vectors used for simulation	No. of faults injected	No. of faults detected	Fault Coverage (in %)
S349	14,999	222	217	97.75
S382	14,999	174	77	76.55
C432	30	86	81	94.19
C499	20	146	142	97.26
C74181	20	44	44	100

Table 6.4.4.3: *Fault Coverage, Number of Injected Faults, Test Vectors, and Detected Faults*

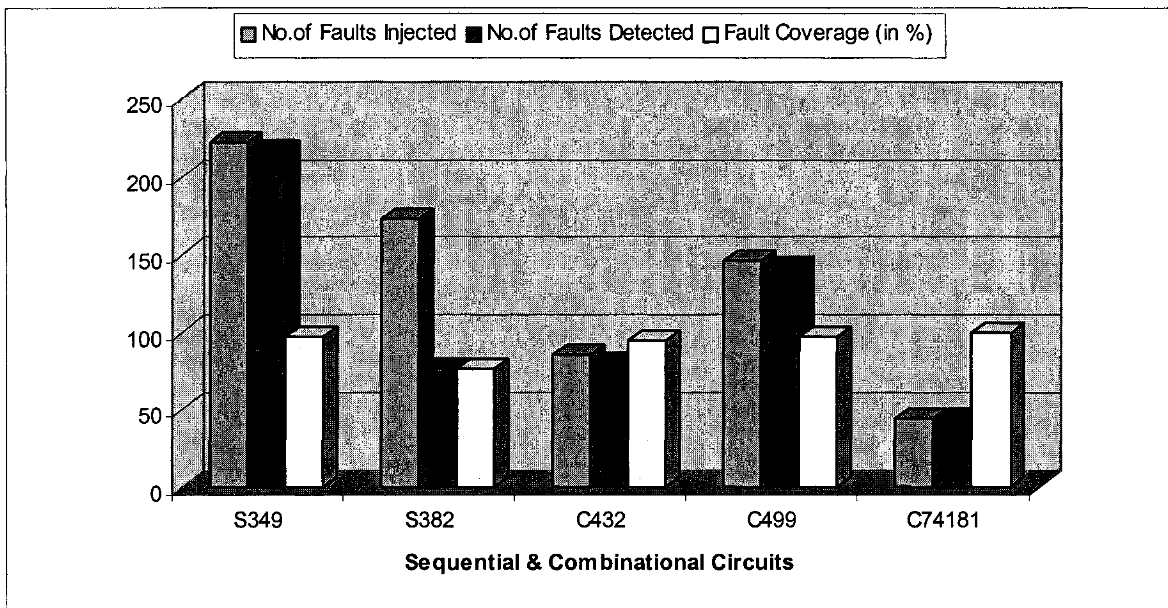


Figure 6.19: *No. Of Injected Faults, Detected Faults and Fault Coverage of Sequential & Combinational Circuits*

Chapter 7
CONCLUSION

7.1 SUMMARY OF THE THESIS

In this thesis, we have presented a system-on-a chip testing methodology. The system consists of a wrapper and test access mechanism. The cores include the ISCAS sequential and combinational circuits. At the gate level, stuck at fault model is used to detect faults. The wrapper separates the circuit under test from other cores, which is assumed to be IEEE P1500 compliant. The test access mechanism plays an important role of transporting the test patterns or test vectors to the desired circuit under test and then transporting the responses to the output pin of the SOC. The faults are then injected using the fault simulator that generates test for the circuit under test. We implemented the test access mechanism (TAM) as a plain signal transport medium, which is shared by all the cores in the system-on-chip. Once the compilation of the cores was successful the fault simulation was carried out, where the test patterns are fed into the cores through the TAM. The selection of the cores is taken care by the program running in the background. The end result gave us the fault coverage of all the cores been tested. The whole process follows a turn-key approach without any intervention from the designer.

TAM Implementation: We implemented the Multiplexer Based Direct Access TAM Architecture. The TAM is used to drive the test vectors from the test source i.e. the fault simulator to the desired core under test (CUT) and to transport to test response from the CUT back to the fault simulator. The selection of the core in the SOC has also been implemented as a part of the TAM. We determined the width of the TAM by the core, which has the maximum number of the input/output pins, within the SOC. This was based on the theory developed in Chapter 3.

Wrapper Implementation: We then implemented our own wrapper for one SOC, through which we were able to test the CUT in two modes: normal and test mode. We used multiplexers and the demultiplexers to design the wrapper. We added a multiplexer at

each input of the core and a demultiplexer at each output of the core. We also reviewed the various Wrapper and TAM design procedures.

Fault Injection and Detection: Next, we implemented the software fault injection technique to every CUT to compute the fault coverage. We used the stuck-at-fault model for testing of the CUTs. The fault compilation and simulation was done in the ALTERA's Max Plus II design envioumment. Finally, we present the results after each CUT was independently fault tested.

7.1 THESIS CONTRIBUTION

Test development is now seen as a major bottleneck in SOC design and major problem arises to test these multiple cores individually and sometimes the logic to separate the cores. In SOC testing, isolation of the cores involves electrically detaching the input and output ports of the core from the chip logic (or other core) connected to theses ports. We not only tried to simulate this approach but also performed fault injection for each CUT in the SOC. We followed a bottom-up approach, right from designing the SOC to testing it. We were eventually able to get satisfactory results, which are reflected, in the Fault Coverage for each CUT.

7.3 SUGGESTIONS FOR FUTURE RESEARCH

There are several ways of testing a SOC. Many efficient Wrapper and TAM schemes are proposed in recent times. The new design approach proposed by many authors improve upon previous approaches by minimizing the core testing time, as well as reducing the TAM width required for the core. Also, the conflict that appears when sharing the test bus for test data transportation is considered. The IEEE Std 1149.1 is a standard which is developed for testing digital integrated circuits. This boundary scan TAP standard is also a successful way of testing and being implemented by many academic and industrial

research groups. The SOC testing being one of the most challenging part in the design and production of an IP core, there has been a lot of theories developed. We experienced, that there is another efficient way of transporting test vectors into the CUT. It is the scan chain-design technique, which enables scanning in and out circuit inputs and outputs. With the use of scan-design and ATPG tools we can use the same design and produce thousand test vectors in very short time and with high fault coverage. It also simplifies the problem of test pattern generation by reducing the design into purely combinational logic. In our thesis, we implemented the gate level fault simulation. For the past two decades, single stuck at fault model has been very popular. However, with the circuit complexity increasing exponentially more advanced fault models are being explored. The behavioural fault model is such an approach. A behavioural test pattern fault simulation detects faults in a structural model with a low abstraction level. Hence for VLSI this approach is becoming very common. We used the Software Fault Injection techniques because they don't require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection. In our thesis we implemented the whole SOC testing methodology through entire software method. However, we experienced that the testing time would have been drastically reduced if the whole fault test were done on hardware. So another possibility of future research would be to download the core designs on a hardware board and then perform the testing. The Altera's NIOS board could be a possible platform to test it. The interconnectivity of the different cores could be done by writing different protocols and using the AMBA bus. Finally, it may be of significant and challenging research interest to pursue an investigation of testing SOC on real time hardware.

REFERENCES

1. S. R. Das, C. Jin, L. Jin, M. H. Assaf, E. M. Petriu, W. –B. Jone, and M. Sahinoglu, "Implementation of a testing environment for digital IP cores", *Proc. IEEE Instrum. Meas. Conf.*, vol. 2, 2004, pp. 1472-1477.

2. Julien Pouget, Erik Larsson, Zebo Peng, Marie-Lise Flottes, Bruno Rouzeyre "An Efficient Approach to SOC Wrapper Design, TAM Configuration and Test Scheduling"

3. Yervant Zorian, Erik Jan Marinissen, Sujit Dey "Testing Embedded-Core Based System Chips".

4. Vikram Iyengary, Krishnendu Chakrabartyy and Erik JanMarinissen "On Using Rectangle Packing for SOC Wrapper/TAM Co-Optimization".

5. Erik Larsson,, Klas Arvidsson, Hideo Fujiwara and Zebo Peng "Efficient Test Solutions for Core-Based Designs ". Vol. 23, no. 5, may 2004
6. R.Chandramouli and Stephen Pateras "Testing System on a Chip". Artech House, Boston 2000.

7. Erik Jan Marinissen, Yervant Zorian, Rohit Kapur, Tony Taylor and Lee Whetsel "Towards a Standard for Embedded Core Test: An Example". ITC International Test Conference, 1999.
8. Erik Jan Marinissen, Rohit Kapur, Yervant Zorian "On Using IEEE P1500 SECT for Test Plug-n-Play". IEEE International Test Conference (ITC) Atlantic City, NJ, October 1-5, 2000
9. Vikram Iyengar and Krishnendu Chakrabarty "Test Wrapper and Test Access Mechanism Co-Optimization for System-on-Chip". JOURNAL OF

ELECTRONIC TESTING: Theory and Applications 18, 213–230, 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

10. Yervant Zorian, Erik Jan Marinissen, Sujit Dey "Testing Embedded-Core Based System Chips".
11. Zahra Sadat Ebadi and Andre Ivanov "Time Domain Multiplexed TAM: Implementation and Comparison". Page:1530-1591, 2003 IEEE
12. I Ghosh, N. Jha, and S. Dey. "A low overhead design for testability and test generation technique for core-based systems". In *Proc. of International Test Conference*, pages 50-59, 1999.
13. Debashis Bhattacharya "Hierarchical Test Access Architecture for Embedded Cores in an Integrated Circuit", Texas Instruments Incorporated.
14. L.Hong, M.Nahvi, R. Fung, A. Ivanov, R. Saleh "Novel Test Methodologies for SOC/IP Design". Dept. of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC.
15. Lee Whetsel "Addressable Test Ports: An Approach to Testing Embedded Cores".
16. Prab Varma and Sandeep Bhatia "A Structured Test Re-Use Methodology for Core Based System Chips".
17. Erik Jan Marinissen, Robert Arendsen, Gerard Bos, Hans Dingemanse, Maurice Lousberg, Clemens Wouters "A Structured and Scalable Mechanism for Test Access to Embedded Reusable Cores".
18. Daniel G. Sabb, Ibrahim N Haji, Joseph T Rahmeh "Parallel-Concurrent Fault Simulation". IEEE, 1989
19. Research in VLSI Circuit Testing, verification and diagnosis, The University of Illinois at Urbana-Champaign: http://www.crhc.uiuc.edu/IGATE/
20. Masters Thesis by Chuan from University of Ottawa.
21. Chien-In Henry Chen "Behavioral Test Generation and Fault Simulation". IEEE February/March 2003
22. Naim Ben-Hamida, Khaled Saab, David Marche and Bozena Kaminska "FaultMaxx: A Perturbation Based Fault Modeling and Simulation for Mixed-Signal Circuits".

23. Mei-Chen Hsueh, Timothy K Tsai, Ravishankar K Iyer "Fault Injection Techniques and Tools".
24. Masters Thesis by Liwu from University of Ottawa.
25. R.Rajsuman, *System-on-Chip: design and Test.* Artech House, Boston 2000.
26. P.H.Rardell, W.H.McAnney and J.Savir. *Built in Test for VLSI: Pseudo randonm Techniques*. John Wiley, New York, 1987.
27. Krishnendu Chakrabarty. *Design and analysis of efficient response compaction schemes. PhD thesis,* University of Michigan, 1995.
28. K. Chakrabarty, V. Iyengar, and A. Chandra, *Test Resource Partitioning for System-on-a-Chip*, Boston, MA: Kluwer, 2002.
29. K. Chakrabarty (Editor), SOC (System-on-a-Chip) Testing for Plug and Play Test Automation, Boston, MA: Kluwer, 2002.
30. P. K. Lala, *Fault Tolerant and Fault Testable Hardware Design.* London, U.K.: Prentice-Hall, 1985.
31. E. J. McCluskey, "Built-in self-test techniques", *IEEE Design and Test of Computers*, vol. 2, pp. 21-28, Apr. 1985.
32. S. Mourad and Y. Zorian, *Principles of Testing Electronic Systems*. New York: Wiley, 2000.
33. T. W. Williams and K. P. Parker, "Testing logic networks and design for testability", *Computer*, vol. 21, pp. 9-21, Oct. 1979.

PAPERS PUBLISHED AND SUBMITTED IN REFERED JOURNALS AND CONFERENCES

The following is the list of papers published by the author, on some of which portion of the present thesis is based:

1) Sunil R. Das, Mansour H.Assaf, Emil M.Petriu, Dhruv Biswas and Mehmet Sahinoglu, Liwu Jin "*Testing Embedded Cores-based System-on-a-chip (SOC) – test architecture and implementation*" IMC Switzerland, Feb 2004.
2) Sunil R. Das, Mansour H.Assaf, Emil M.Petriu, Dhruv Biswas and Mehmet Sahinoglu "*Systems-On-Chip Test - using modified cut-sets of response data outputs to achieve aliasing free compaction*".
3) Mansour H.Assaf, Sunil R. Das, Emil M.Petriu, Dhruv Biswas and Mehmet Sahinoglu, Liwu Jin "*Hardware and Software co-design in Space Compaction of Core based Digital Circuits*" IMTC 2004 Instrumentation and Measurement Technology Conference, Como, Italy, 18-20 May 2004.
4) Sunil R. Das, Dhruv Biswas, Mansour H.Assaf, Emil M.Petriu "*Developing Test Environment for Embedded Cores-Based System-on-a-Chip (SOC) - IM-5118*", IMTC 2005 Instrumentation and Measurement Technology Conference, Ottawa, Canada, 17-19 May 2005.

APPENDIX: C PROGRAM SOURCE CODE FOR FAULT SIMULATOR

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>
#include <dos.h>
#define  MAX  2048
unsigned long int random_number(unsigned long int seed)
{
  const unsigned long int module = 2147483647L;
  const unsigned long int multiplier = 16807L;
  static unsigned long int ran = seed;

  // initial value is used only at the first time

  ran = (ran * multiplier) % module;
  return ran;
}

int main( )
{
  struct  _timeb t;
  char *timeline1;
  char *timeline2;

  //int t1hour,t1minute,t1second;
  //int t2hour,t2minute,t2second;
  unsigned long int seed;
     int n, m, gen;
     int i,mi,wireline;
      char cutname[10] = "s298",cut[10];
      char vec[20],tbl[20],vecbak[20],vectemp[20],tbak[20],rpt[20],cutx[20],vlgbak[20];
      char cpvecbak[40], cptbl[40],cml[80],cmlx[80],sim[80],simx[80],delv[40];
      char cpvt[40],delvt[40],cpvlg[40],cpvec[40],cpv[40],cpx[40],delspeed[40];
      char w0[20],w1[20],w2[20];
     // char *vec;
      //char *tbl,*vbak,*tbak;
      int flg1=0,flg2=0;
      char inc[800],outc[800],wirec[1800];
     // char *inc;
```

```
// char *outc;
// char incbak[40];
 int innum,outnum,wirenum;
     char ch[800][1000] = {"\0"};
FILE *tem1;
FILE *tem2;
FILE *input;
FILE *output;
FILE *vtbak;
FILE *tblbak;
FILE *report;
char aLine[MAX];
char line2[MAX];
char toFindws[]= "//# wires";
char toFindins[]= "//# inputs";
char toFindouts[]= "//# outputs";
char toFind1[] = "//wire ";
char toFind2[] = "input ";
char toFind3[] = "output ";
char toFind4[] = ".0>";
char *ptr,*ptr1,*ptr2;
char a[1024],aa[1024];
char b[1024];
char c[1024];
char ijf[2],pinname[4];
char itostr[6];
int com, testno=0;
int count[100],countfault[1000];
int j,r=1,s=0;
int r1=1,s1=0;
int pin,pat=1,wir=0;
int bz=-1,ccc=0;
int findfault,findfaultnum=0;
int loopnum=0;
 printf ("Enter Name of CUT is s298");
// cin >> cutname;
 strcpy(cut,cutname);
 strcpy(vec,cutname);
 strcpy(tbl,cutname);
 strcpy(vecbak,cutname);
 strcpy(vectemp,cutname);
 strcpy(tbak,cutname);
 strcpy(rpt,cutname);
 strcpy(vlgbak,cutname);
 strcpy(cutx,cutname);
 strcat(cutname,".v");
```

```
strcat(vec,".vec");
strcat(tbl,".tbl");
strcat(vecbak,"vec.bak");
strcat(vectemp,".bak");
strcat(tbak,"tbl.bak");
strcat(rpt,"report.log");
strcat(vlgbak,"v.bak");
strcat(cutx,"x.v");

printf("cut is %s\n",cut);
printf("cutname is %s\n",cutname);
printf("vec is %s\n",vec);
printf("vecbak is %s\n",vecbak);
printf("vectemp is %s\n",vectemp);
printf("tbak is %s\n",tbak);
printf("rpt is %s\n",rpt);
printf("vlgbak is %s\n",vlgbak);
printf("cutx is %s\n",cutx);



//---------------------------------------------------------------------------


        strcpy(cpvecbak,"copy ");
        strcat(cpvecbak,vec);
        strcat(cpvecbak," ");
        strcat(cpvecbak,cut);
        strcat(cpvecbak,"vec.bak");
        printf("cpvecbak is %s\n",cpvecbak);


        strcpy(cpvec,"copy ");
        strcat(cpvec,vecbak);
        strcat(cpvec," ");
        strcat(cpvec,cut);
        strcat(cpvec,".vec");
        printf("cpvec is %s\n",cpvec);


        strcpy(cml,"c:\\maxplus2\\maxplus2 -c ");
        strcat(cml,cut);
        //strcat(cml," -s -vec d:\\a\\");
        //strcat(cml,vec);
        //strcat(cml," -tbl d:\\a\\");
        //strcat(cml,tbl);
```

```
//strcat(cml," d:\\a\\");
//strcat(cml,cut);
printf("cml is %s\n",cml);


strcpy(cmlx,"c:\\maxplus2\\maxplus2 -c ");
strcat(cmlx,cutx);
//strcat(cml," -s -vec d:\\a\\");
//strcat(cml,vec);
//strcat(cml," -tbl d:\\a\\");
//strcat(cml,tbl);
//strcat(cml," d:\\a\\");
//strcat(cml,cut);
printf("cmlx is %s\n",cmlx);

strcpy(sim,"c:\\maxplus2\\maxplus2 -s");
strcat(sim," -vec ");
strcat(sim,vec);
strcat(sim," -tbl ");
strcat(sim,tbl);
strcat(sim," ");
strcat(sim,cut);
printf("sim is %s\n",sim);

strcpy(simx,"c:\\maxplus2\\maxplus2 -s");
strcat(simx," -vec ");
strcat(simx,vec);
strcat(simx," -tbl ");
strcat(simx,tbl);
strcat(simx," ");
strcat(simx,cutx);
printf("simx is %s\n",simx);

strcpy(cptbl,"copy ");
strcat(cptbl,tbl);
strcat(cptbl," ");
strcat(cptbl,cut);
strcat(cptbl,"tbl.bak");
printf("cptbl is %s\n",cptbl);

strcpy(delv,"del ");
strcat(delv,vec);
printf("delv is %s\n",delv);

strcpy(cpvt,"copy ");
strcat(cpvt,vectemp);
```

```
        strcat(cpvt," ");
        strcat(cpvt,vec);
        printf("cpvt is %s\n",cpvt);

        strcpy(delvt,"del ");
        strcat(delvt,vectemp);
        printf("delvt is %s\n",delv);

        strcpy(cpvlg,"copy ");
        strcat(cpvlg,cutname);
        strcat(cpvlg," ");
        strcat(cpvlg,vlgbak);
        printf("cpvlg is %s\n",cpvlg);

        strcpy(cpx,"copy ");
        strcat(cpx,cutname);
        strcat(cpx," ");
        strcat(cpx,cutx);
        printf("cpx is %s\n",cpx);

        strcpy(cpv,"copy ");
        strcat(cpv,vlgbak);
        strcat(cpv," ");
        strcat(cpv,cutname);
        printf("cpv is %s\n",cpv);

        strcpy(delspeed,"del ");
        strcat(delspeed,cut);
        strcat(delspeed,".CNF");



        system(cpvlg);     //.v to v.bak

//-----------------------------------------------------------------------------------

    fclose(input);
    innum=atoi(inc);
    outnum=atoi(outc);
    wirenum=atoi(wirec);
    printf("wirec is %s\n",wirec);
    printf("wirenum is %d\n",wirenum);
    printf("inc is %s\n",inc);
    printf("outc is %s\n",outc);
    printf("innum is %d\n",innum);
```

```
  printf("outnum is %d\n",outnum);

 //_______________________________________________to get input names

// printf("cutname is %s\n",cutname);
  input = fopen(vlgbak, "r");


       inc[0]='\0';
        for(int z=0; z<= innum -1;z++)

                   {
                          sprintf(inc,"%s%s%d ",inc,"in_dat",z);

                     printf("inc is %s\n",inc);

                   }



//_______ get out name >>

  input = fopen(vlgbak, "r");



       outc[0]='\0';
        for( z=0; z<= outnum -1;z++)

                   {
                          sprintf(outc,"%s%s%d ",outc,"out_dat",z);

                     printf("outc is %s\n",outc);

                   }

  printf("cutname is %s\n",cutname);

 //______________________________________
 //........ get wirename>>

 input = fopen(vlgbak, "r");      //wires to wirec[]
 while ( fgets(aLine, MAX, input) != NULL )
  {
```

```
if ( (strstr(aLine,toFind1)) != NULL )
 {
  //*ptr = 0;
  ptr = strstr(aLine,toFind1);
  ptr=ptr+6;
    for (int i=0;i<1800;++i)     // 1800 chars for in name
     {
      if (*ptr !=';')
       {
        wirec[i]=*ptr;
        ++ptr;
       }
      else
       {
        wirec[i]=',';
        wirec[i+1]='\0';  // end of inchar
        break;
       }
      }
  }
}

 fclose(input);
 printf("wirec is %s\n",wirec);
 printf("OOOOOOOOKKKKKKKK\n");
 printf("outc is %s\n",outc);
 printf("inc is %s\n",inc);

 //____________________________

// create random vectors file...

   srand(time(NULL));
   seed = rand();
   n=innum;
 m=2*(innum+outnum+wirenum);

   printf ("Random numbers are: \n");
input = fopen(vec, "w");
      fprintf (input, "START 0 ;\n");
      fprintf (input, "INTERVAL 1 ;\n");
      fprintf (input, "INPUTS ");
      fprintf (input, inc);
      fprintf (input, ";\n");
     // fprintf (input, "rst CK ;\n");
      fprintf (input, "PATTERN\n");
```

```
int k=0;
      for ( i = 0; i < m; ++i)
{
     for (int count = 0; count < 2*(n-2); ++count)     //n:inputnum, (n-2):rst & CK
   { gen = random_number(seed) % 2;
     if (gen == 0) ch[i][count] = '0';
     else ch[i][count] = '1';
      ++count;
      ch[i][count]=' ';
     }
     printf ("%s\n", ch[i]);

     if (k < m)                  //adding rst & CK
     {
      fprintf (input, "%d> %s",k,ch[i]);

       if (k<1)
        {
         fprintf (input, "1 ");
        }
       else
        {
         fprintf (input, "0 ");
        }

        gen = k % 2;
         if (gen == 0)
         {
           fprintf (input, "1 \n");
         }
         else
         {
          fprintf (input, "0 \n");
         }

          k=k+1;
          gen=k%2;
          if (gen==0)
          {
                fprintf (input, "%d> %s",k,ch[i]);

                 if (k<1)
                  {
                   fprintf (input, "1 ");
                  }
```

```
                else
                 {
                 fprintf (input, "0 ");
                 }

                 gen = k % 2;
                 if (gen == 0)
                 {
                  fprintf (input, "1 \n");
                 }
                 else
                 {
                   fprintf (input, "0 \n");
                 }

                 k=k+1;
            }
          }


  }
        fprintf (input, ";\n");
        fprintf (input, "OUTPUTS ");
        fprintf (input, outc);
        fprintf (input, ";\n");
        fclose(input);


   printf("wirec is %s\n",wirec);
   //printf("ok\n");
   printf("outc is %s\n",outc);
   printf("inc is %s\n",inc);

  system(cpvecbak);      //get a vec.bak
  system(delspeed);
  system(cml);
  system(sim);
  system(cptbl);




                              _ftime( &t );
                               timeline1 = ctime( & ( t.time ) );

//... begin fault injection for inputs
```

```
bz=-1;

for (pin=1;pin<(n-1);pin++)    // ....The first loop,pinnumber
    {

      if (bz<0)          // twice fault injections
       {
        strcpy(ijf,"0");
       }
      else
       {
       strcpy(ijf,"1");
       }

       printf("ijf is %s\n",ijf);

        tem1 = fopen(vecbak, "r");  //change .vec file for input injection
        tem2 = fopen(vec,"w");
    while ( fgets(aLine, MAX, tem1) != NULL )
     {

      if ( (ptr = strstr(aLine,"> ")) != NULL )  // fault injection for inputs
        {
          ptr = ptr + (2*pin);
          strcpy(line2, ptr + 1);
          *ptr = 0;
          strcat(aLine, ijf);
          strcat(aLine, line2);
        }


          fprintf(tem2, "%s", aLine);
     }                              //.end while(input fault injection)


   fclose(tem1);
   rewind(tem2);
   fclose(tem2);

   system(delspeed);
   system(cml);
   system(sim);
   loopnum=loopnum+1;
```

```
//Compare tables result to find fault for inputs-fault-injection

  tem1 = fopen(tbl, "r");  //...compare table and tablebak results
  tem2 = fopen(tbak,"r");
  r=-1;    // for locating the same line for tem1 & tem2
  s=-1;    // for locating the same line for tem1 & tem2
  findfault=0;
  while ( fgets(aLine, MAX, tem1) != NULL )
   {
      if ( (ptr1 = strstr(aLine,"0>")) != NULL )
      {
        r=r+1;
        strcpy(aLine, ptr1 + 3);

         while ( fgets(line2, MAX, tem2) != NULL )
         {
          if ( (ptr2 = strstr(line2,"0>")) != NULL )
           {
            s=s+1;
            if (s==r)
             {
              strcpy(line2, ptr2 + 3);
              if  ((s>0)&&(strcmp(aLine,line2)!=0))    //  s>0:  from  No.1  vector(for
Random vectors)
               {

                countfault[r]=countfault[r]+1;
                //printf("countfault[%d] is %d\n",r,countfault[r]);

                itoa(pin,itostr,10);
                strcpy(cpvt,"copy ");
                strcat(cpvt,tbl);
                strcat(cpvt," ");
                strcat(cpvt,cut);
                strcat(cpvt,itostr);
                strcat(cpvt,ijf);
                strcat(cpvt,"tbl.bak");
                printf("cpvt is %s\n",cpvt);

                system(cpvt);
                findfaultnum=findfaultnum+1;
                findfault=1;

                break;
               }
```

```
              else
              {
               break;
              }
            }
          }
         }

        if (findfault==1)   // found a fault end finding-fault-while-loop
         {break;}

      }
   } //end of wile loop(compare 2 tables)


   fclose(tem1);
   fclose(tem2);


 //above  compare 2 tables to find fault
 //________________________________________


   if (bz<0)        // for inject "0" or "1"
    {
     pin=pin-1;
     bz=bz*(-1);
    }
   else
    {
    bz=bz*(-1);
    }



 }    // End of for() Big loop for inputs-fault-injection


   system(cpvec); //recover the original vector file


   //*************************             end      of      inputs-fault-injection
**************************
```

```
//****************                              start           wire-fault-injection
***********************************

bz=-1;  //for inject "0" or "1"
mi=0;  // locate a wire ( mi-memory the i)

for (wir=0;wir<wirenum;wir++)       // The second Big loop for wire-fault-injection --
wir:wires' number
 {

  if (bz<0)           // twice fault(0 and 1) injections
    {
     strcpy(ijf,"0");

 //_______________________________________________________

     for(i=0;i<30;i++) // the longest number of a wire name
       {
        if (wirec[mi]!=',')
         {
          w0[i]=wirec[mi];
          //i=i+1;
          mi=mi+1;
         }          //get a wire name once, for this wire twice fault injections
        else
         {
          mi=mi+1;
          w0[i]='\0';
          break;
         }
       }
 //___________________________________________


    }
   else
    {
     strcpy(ijf,"1");
    }
```

```
printf("ijf is %s\n",ijf);



  printf("w0 is %s\n",w0);
  w1[0]=',';
  w1[1]='\0';
  w2[0]=',';
  w2[1]='\0';
  strcat(w1, w0);
  strcat(w1, ",");
  strcat(w2, w0);
  strcat(w2,")");


  printf("w1 is %s\n",w1);
  printf("w2 is %s\n",w2);



tem1 = fopen(vlgbak, "r");  //..change the .v file for wire fault injection
tem2 = fopen(cutname,"w");
while ( fgets(aLine, MAX, tem1) != NULL )
{

     if ( (ptr = strstr(aLine,"//wire ")) != NULL )        //find the start of wireline
      {
      wireline=1;
      }


     if ( (ptr = strstr(aLine,w1)) != NULL )
     {

      if(wireline==1)        //if it is the line of wire name, no injection
       {
       wireline=wireline;
       }
      else
       {
       strcpy(line2, ptr + strlen(w1));
       *ptr = 0;
       strcat(aLine, ",");
```

```
      strcat(aLine, ijf);
      strcat(aLine,",");
      strcat(aLine, line2);
     }
    }
   if ( (ptr = strstr(aLine,w2)) != NULL )
    {

      if(wireline==1)         //if it is the line of wire name, no injection
      {
       wireline=wireline;
      }
      else
      {

       strcpy(line2, ptr + strlen(w2));
       *ptr = 0;
       strcat(aLine, ",");
       strcat(aLine, ijf);
       strcat(aLine,")");
       strcat(aLine, line2);
      }
    }

   if ( (ptr = strstr(aLine,";")) != NULL )   //find the end of wireline
    {
     wireline=0;
    }

     fprintf(tem2, "%s", aLine);
 } //end while( end fault injection for a wire)

fclose(tem1);
rewind(tem2);
fclose(tem2);

printf("w0 is %s\n",w0);
printf("w1 is %s\n",w1);
printf("w2 is %s\n",w2);

system(delspeed);
system(cml);
loopnum=loopnum+1;
system(sim);
```

```
// Compare tables result to find fault for wire-fault-injection
//______________________________________________________________

  tem1 = fopen(tbl, "r");  //..compare table and tablebak results..
  tem2 = fopen(tbak,"r");
  r=-1;   // for locating the same line for tem1 & tem2
  s=-1;   // for locating the same line for tem1 & tem2
  findfault=0;
  while ( fgets(aLine, MAX, tem1) != NULL )
   {
      if ( (ptr1 = strstr(aLine,"0>")) != NULL )
      {
        r=r+1;
        strcpy(aLine, ptr1 + 3);
        //*ptr = 0;
        //strcat(aLine, cut);
        //strcat(aLine, "x");
        //strcat(aLine, line2);

         while ( fgets(line2, MAX, tem2) != NULL )
         {
          if ( (ptr2 = strstr(line2,"0>")) != NULL )
           {
            s=s+1;
            if (s==r)
             {
              strcpy(line2, ptr2 + 3);
              if  ((s>0)&&(strcmp(aLine,line2)!=0))    // s>0  from  No.1  vector(for
Random vectors)
               {
                //printf("the line is: %s\n",aLine);
                countfault[r]=countfault[r]+1;
                //printf("countfault[%d] is %d\n",r,countfault[r]);

                strcpy(cpvt,"copy ");
                strcat(cpvt,tbl);
                strcat(cpvt," ");
                strcat(cpvt,cut);
                strcat(cpvt,w0);
                strcat(cpvt,ijf);
                strcat(cpvt,"tbl.bak");
                printf("cpvt is %s\n",cpvt);
```

```
              system(cpvt);
              findfaultnum=findfaultnum+1;
              findfault=1;

              break;
             }
             else
             {
              break;
             }
           }
         }
        }

      if (findfault==1)   // if found a fault, end finding-fault-while-loop
       {break;}

     }
   } //...end of wile loop(compare 2 tables)...


   fclose(tem1);
   fclose(tem2);


//above compare 2 tables to find fault
//_____________________________________


   if (bz<0)      // for inject "0" or "1"
    {
     wir=wir-1;
     bz=bz*(-1);
    }
   else
    {
    bz=bz*(-1);
    }


}   //  End of the second for() Big loop for wire-fault-injection.
```

```
system(cpv);        // recover the initial .v file


//***************  end of wires-fault-injection  *****************

//****************  start output-fault-injection  *****************


bz=-1;  //for inject "0" or "1".
mi=0;  // locate a output line ( mi-memory the i).


for (pin=0;pin<outnum;pin++)      // The second Big loop for output-fault-injection
 {

  if (bz<0)            // twice fault(0 and 1) injections
     {
      strcpy(ijf,"0");
     }
    else
     {
      strcpy(ijf,"1");
     }

     printf("ijf is %s\n",ijf);


    tem1 = fopen(tbak, "r");  //open tab.bak for out fault testing
    r=-1;
    while ( fgets(aLine, MAX, tem1) != NULL )
    {


        if ( (ptr = strstr(aLine,".0>")) != NULL )      //find the start of wireline
          {
           r=r+1;
           if ((r>0)&&(r<m))   //regardless of No.0 and the last vectors(for Random
vectors)
           {
            ptr=ptr+4;
            ptr=ptr+(innum*2);
```

```
            ptr=ptr+2;
            ptr=ptr+(pin*2);     //first output
            a[0]=*ptr;
            printf("a0 is: %c\n",a[0]);
            printf("ijf is:%c\n",ijf[0]);

            if (a[0]!=ijf[0])  // s>5:regardless of 0-5 vectors(s27)
             {
               //printf("the line is: %s\n",aLine);
              countfault[r]=countfault[r]+1;
               //printf("countfault[%d] is %d\n",r,countfault[r]);

               findfaultnum=findfaultnum+1;
               findfault=1;
              break;
             }
            }

           }

    } //end while(end once fault testing for a output)

     fclose(tem1);

     loopnum=loopnum+1;


    if (bz<0)        //for inject "0" or "1"
     {
      pin=pin-1;
      bz=bz*(-1);
     }
    else
     {
     bz=bz*(-1);
     }

  fclose(tem1);


 }    //End of the second for() Big loop for output-fault-injection


    //*************  end of out-fault-injection  *********

//_______________________________________
```

```
// following create a report file........

 tblbak = fopen(tbak, "r");

   while ( (fgets(aLine, MAX, tblbak) != NULL)& (testno<(m+1)) )
   {
    if ( (strstr(aLine,toFind4)) != NULL )
     {
      ptr = strstr(aLine,toFind4);
      ptr=ptr+4;

        for (int i=0;i<2*(n+2);++i)
          {
           if (*ptr != '=')
            {
             a[0]='\0';
             a[i]=*ptr;
             ++ptr;
            }
          }

       report = fopen(rpt, "a");
           fprintf (report, "Test%d %s ",testno,a);
           fprintf (report, " %d faults detected \n",countfault[testno]);
           testno=testno+1;
       fclose(report);
     }
   }

   fclose(tblbak);

     /*     gettime(&t);        // ending time
          t2hour=t.ti_hour;
          t2minute=t.ti_min;
          t2second=t.ti_sec;
                         */

                         _ftime( &t );
                         timeline2 = ctime( & ( t.time ) );
```

```
	report = fopen(rpt, "a");
	fprintf (report, "\n findfaultnum is %d\n",findfaultnum);
	fprintf (report, "totalfaultnum is %d\n",(2*(innum+outnum+wirenum-2)));

	fprintf (report, "start time is %d\n",timeline1);


	fprintf (report, "end time is %d\n",timeline2);

	fclose(report);

printf("loopnum is %d\n",loopnum);
printf("wirenum is %d\n",wirenum);
printf("innum is %d\n",innum);
printf("outnum is %d\n",outnum);
printf("findfaultnum is %d\n",findfaultnum);

return 0;
}
```