PSEUDO-RANDOM TESTING OF ARITHMETIC CIRCUITS

CANTURK ISCI

THIS DISSERTATION IS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE M.SC. DEGREE IN VLSI SYSTEM DESIGN AT THE WESTMINSTER UNIVERSITY.

SUPERVISORS:
IZZET KALE
RICHARD C.S. MORLING

NOVEMBER 2001

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Dr. R.C.S. Morling and Prof. I. Kale for their invaluable help and guidance during the project. I would also like to thank Mr. Alan Wood, for his prompt support with UNIX system and installations. On a more personal note, I would like to thank my family and friends for helping me keep up my spirit.

ABSTRACT

Multipliers are often the critical functional blocks of datapath architectures. Due to their deeply embedded configurations in datapath architectures and two dimensional iterative array structure, they attain very low controllability and observability, which entails Built in Self Test (BIST) for multiplier testing. In this project, a parameterized multiplier is designed and BIST techniques for efficient multiplier testing are investigated. Deterministic fault simulation for single stuck at model is used to determine the fault coverage characteristics of the investigated methods, which are also compared to cell fault model (CFM), which is defended by [2] and [3] to be more comprehensive. Both board level and hierarchical faulting are applied in fault simulation. For hierarchical faulting, an over detailed method is observed to be less optimistic than CFM. Various well-known and original input pattern generation techniques are investigated, with emphasis on PRBS generation using Linear Feedback Shift Registers (LFSRs) and Cellular Automata (CA). Effects of different seed are discussed with a practical method for seed determination. The use of repeated patterns as input is observed to be extremely efficient. In both exhaustive and repetitive pattern testing, pseudorandom sequences overperform regular deterministic sequences. For output compression, Multiple Input Signature Registers (MISRs) are seen to be very effective, with a high weighted characteristic polynomial. A complete parameterized top level system with BIST circuit is designed and a constant size test set with repetitive pattern generation using LFSR with seed x7B is tested for larger size multipliers and is seen to be equally effective, with less hardware cost than of general pseudorandom testing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Below is a list of abbreviations used in this report. The abbreviation, the description of the abbreviation and first mention page in the report are listed in the respective columns:

ASIC	Application Specific Integrated Circuit	1
ATE	Automatic Test Equipment	9
ATPG	Automatic Test Pattern Generation	10
BIST	Built in Self Test	1
CA	Cellular Automata	2
CAD	Computer Aided Design	4
CPA	Carry-Propagate Array	2
CSA	Carry-Save Array	2
DfT	Design for Testability	4
DSP	Digital Signal Processor	1
DUT	Device Under Test	7
EDA	Electronic Design Automation	1
EDDM	Electronic Design Data Model	40
FA	Full Adder	47
FAN	FANout oriented test generation algorithm	14
GF(2)	Galois Field of 2	26
HA	Half Adder	50
HDL	Hardware Description Language	39
IC	Integrated Circuit	1
LASAR	Logic Automated Stimulus And Response	13
LFSR	Linear Feedback Shift Registers	2
LSI	Large Scale Integration	4
LSSD	Level-Sensitive Scan Design	13
LTG	LAMP2 Test Generator	14
MAC	Multiply-Accumulate	1
MD	Multiplicand	44
MISR	Multiple Input Signature Register	2
MR	Multiplier	44
MSB	Most Significant Bit	44
MSI	Medium Scale Integration	5
OEM	Original Equipment Manufacturer	6
PCB	Printed Circuit Board	5
PODEM	Path Oriented DEcision Making	13
PRBS	Pseudorandom Binary Sequence	2
RTL	Register Transfer Level	39
SB	Sign Bit	44
SMART	Sensitizing Method for Algorithmic Random Testing	14
SRTG	Shift Register Test Generator	13
SSI	Small Scale Integration	5
VHDL	VHSIC Hardware Description Language	2
VHSIC	Very High Speed Integrated Circuit	2
VLSI	Very Large Scale Integration	1

I-INTRODUCTION

I.1 – INTRODUCTION TO THE PROJECT :

With the immense present and ongoing technological developments in communications and computerization, digital signal processors (DSPs) and general purpose processors are two of the most active fields in Very Large Scale Integration (VLSI) and Application Specific Integrated Circuit (ASIC) design. Datapath architectures, which constitute the operational backbone of these two systems are of particular importance, as even though the significant advances in computation speed, electronic design automation (EDA) tools and integrated circuit (IC) technology instigate faster, smaller and less power dissipating circuits with less production time overhead, the requirements for efficient testing methodologies get stubbornly more demanding as such architectures are deeply embedded in overall system structure, with low controllability and observability, and increased gate to pin ratios. [1],[2],[3],[4]

Multipliers are the most critical functional blocks of datapath architectures in terms of speed and area, and due to their common use in datapath architectures, they are very well integrated in the design automation processes with several alternatives favoring regularity in structure, power consumption, speed and/or area. However, due to their deeply embedded configuration in the datapath architectures, the low observability and controllability, which are even deteriorated by the general regular 2 dimensional iterative array structure[3], make multiplier testing a significant bottleneck in the design process. Therefore, Built in Self Test (BIST) architectures are the preferred solution in multiplier testing for most applications ([3],[5]), as efficient BIST methods provide testing at the operation speed of the overall system – *at speed testing* –, very high fault coverage with moderate amount of test vectors – constant or linearly dependent on multiplier size – and reduced test time, which in turn reduce the test cost.

In this project, we investigate several BIST techniques for testing parallel multipliers, and then the scrutinized BIST techniques will be extended to Multiply-Accumulate (MAC), structures with also the addition of the inherent nonlinearities of MAC architectures, namely rounders for precision loss in fixed point multipliers and limiters/clippers to prevent wrapping due to overflow in accumulators. The project can be vaguely separated into 4 phases:

(0- Gaining background information on testing, BIST, Pseudorandom Binary Sequence (PRBS) Generation and output compression techniques)

1- Design of a parameterizable signed – 2s complement multiplier:

In this phase, which might be considered as the design entry, we design a parameterizable multiplier using VHDL(VHSIC[1] Hardware Description Language) in Mentor Graphics' Renoir®. There is a significant amount of literature on well-known types of array and tree multipliers, and during the design entry, we refer to [6],[7],[8] for signed Carry-Propagate Array (CPA) multipliers; to [9] for signed Carry-Save Array (CSA) multipliers; to [8],[5] for modified booths array multipliers; to [7],[10],[3] for tree structures and to [4] for CSA for MAC architectures. As the initial design, we've chosen a simple signed CPA structure for the parameterizable multiplier.

(1➔2- Design data management and undertaking necessary steps to convert the design data into the fault simulation format using Mentor's Design Architect® and AutologicII®)

2- Investigation of several BIST alternatives for input pattern generation:

In this phase, we investigate several well-defined and original BIST schemes for an efficient set of input test vectors in terms of fault coverage using Mentor's QuickfaultII®. Several techniques described in [13],[1],[14],[11],[12],[15] and [3] are observed and particular emphasis is built on exhaustive and nonexhaustive PRBS generation with Linear Feedback Shift Registers (LFSRs) ([1],[13],[14],[11]) and Cellular Automata (CA) ([13],[15]). Finally, the application of repetitive patterns, described in [3], is investigated as well as evaluation of the fault models described in [3],[6] and [23].

3- Application of output compression:

In this phase, we investigate the outcome of applying output compression for the multiplier outputs, mainly by application of signature analysis as described in [13],[14],[1] and [11]. Techniques for improving fault coverage might be applied depending on the outcomes of the designed 16 bit Multiple Input Signature Register (MISR). Other output compression techniques, described generally in [1],[3] and [13] and techniques specifically used multipliers as in [3].[5] might be investigated for comparison depending on the time bounds

4- Extension to MAC Structure:

[1] VHSIC : VERY-HIGH-SPEED INTEGRATED CIRCUITS

In this phase, we planned to extend the multiplier structure to a MAC structure, which is the backbone of most DSP processor architectures[8]. An efficient BIST scheme for the MAC architecture with rounders and limiters was to be investigated.

I.2 – INTRODUCTION TO PROJECT REPORT :

In the project report, we try to demonstrate the project progress, in a systematical way, which combines the chronological progress with the formerly described project phases. The first section herein provides a brief introduction to the project and report. The second section aims to give notional and theoretical information about testing and BIST. In the third section we provide an in depth description of PRBS generation using LFSRs and CA and signature analysis, specifically multiple input signature analysis. The fourth section describes the used tools during the project as well as providing a comprehensive explanation of the design data management and design flow. In the fifth section, the first phase of project progress, the design entry is described and designed blocks are introduced, with their conformance to the specifications. In the sixth section, investigation of various pattern generation techniques is discussed with the application of several alternative pattern generation methods. The results are either included within the section or in the appendices or merely referred from the attached CD, depending on the respective volume of data to be presented. Sixth section, which describes mainly the core of the project progress, concludes investigation of BIST techniques with the determination of output compression technique and the final simulation results. The remainder of the section, which is basically an extension to verify the effectiveness of the proposed BIST scheme demonstrates the fault coverage for larger size multipliers with the applied constant volume test set, and points to an interesting outcome and encountered computational problems with increasing circuit size. The seventh section demonstrates the project workplan. The eighth section states the conclusions reached after the evaluation of the results, and the ninth section suggests any further work that could be undertaken. Finally, the tenth section describes the cited references and used bibliography.

∨ To abide by the set page limitation on theoretical information, section two should be considered as supplementary material about testing, and can be skipped for only complete theoretical information about PRBS generation and signature analysis, in section three.

II-TESTING AND BUILT IN SELF TEST

Since the electronics technology accomplished higher levels of integration into a single silicon chip that led to Large Scale Integration (LSI), which preceded VLSI, the applications of electronic systems have experienced an almost unlimited expansion. However, despite the many advantages provided by VLSI, the inherent high integration level started to necessitate very sophisticated testing strategies in order to verify the correct device operation. As the electronics market stimulated the use of VLSI in a variety of tasks from critical military applications to consumer products, the reliability of the products' functioning gained an escalating importance. The expanding demand for ASIC applications led to development of more sophisticated Computer Aided Design (CAD) tools; which have shown most significant progress in layout and simulation, with yet more inferior improvement in testing. This consequently leads to designs with superior complexity, but which are in contrast extremely difficult to test effectively. Moreover, due to the low volume attribute of ASICs, the high test costs cannot be retaliated with large amounts of mass production. Thereupon, despite the traditional design point of view, that design and test can be considered as two different aspects of development, current design processes consider testing as an integral part of design rather than design and test being two mutually exclusive processes. At present, as the number of gates per chip exceeds millions, not only the design process is resolutely bound to the designed circuits' being testable, but also some percentage of auxiliary device circuitry is intentionally included on devices, in order to assure the functionality of the 'actual' circuit is verified to an adequate level. The two concepts mentioned in the last argument, the former leading to Design for Testability (DfT), while the latter to BIST, are extensively scrutinized in this section, with the exception of fault models and fault simulation techniques which are discussed in section VI for the sake of clarity.

II.1 – EVOLUTION OF TESTING

In the early times of electronics engineering, when systems were constituted from discrete components, testing of digital systems comprised three distinct phases:

1- Each discrete component was tested for concordance to its specifications
2- The components were assembled into more complex digital elements (i.e. flip flops etc.), and these were tested for correct functionality

3- The higher level system was built up and was tested for functionality

As the systems acquired higher complexity, the 3rd phase began to become increasingly difficult to accomplish and other means of system verification were begun to be sought. In [16], R.D. Eldred suggested another way, which is well-known and used as structural test at present, in order to test the hardware of a system instead of the burdensome functional test. The first applications of the proposed structural test was to discrete components on Printed Circuit Boards (PCBs), which then began to be applied to ICs as the electronics technology developed into higher levels of integration([14]). Though the problems of IC testing was not very much different from that of the PCBs, the objective of testing had then changed to discard the faulty units rather than locating the defective components and replace them. In the case of Small Scale Integration (SSI) and Medium Scale Integration (MSI), the problems were relatively as simple as PCB testing, since ([1]):

1- Internal nodes of the devices were easily controlled and observed from the primary inputs and outputs of the devices.
2- The simplicity of circuit functions permitted the use of exhaustive testing
3- More complex systems were constructed from basic, thoroughly tested components.

As LSI and VLSI advanced as the prominent technologies, the gate/pin ratios increased rapidly, thus reducing the controllability and observability of internal nodes drastically. Consequently, the testing problems exacerbated by VLSI circuits can be stated as ([1]):

1- Increased testing costs, which depend on test time and therefore circuit complexity
2- Increased time overhead for efficient test pattern generation and verification
3- Increase in the volume of test data

Recently, rapid changes in VLSI technology and more extensive use of ASICs have even worsened these problems. Certain features of ASIC applications make VLSI testing even more demanding ([1]):

1- ASIC designs generally require a short design time overhead due to the competitive nature of the market, and hence they have a short product lifetime. Therefore, immense efforts for efficient testing strategies may yet diminish the market value due to time overhead.
2- As ASICs are by definition application specific, they generally have a low production volume which cannot compensate for significant test costs.
3- The uniqueness of ASICs requires unique testing strategies for each specific application.

At present the objectives of VLSI testing can be described as follows ([13],[1]):

1- To ensure the circuit is functionally correct, free of design errors before fabrication
2- To ensure the device is free of fabrication errors after fabrication (fault detection)
   (i) To locate the source of a fault within an IC (Physical fault location)
   (ii) To locate a faulty component or connection within the complete system (Component Fault Location)

In order to comply with the above objectives, VLSI testing consists of several phases, which require the involvement of both the IC manufacturer (vendor), and the Original Equipment Manufacturer (OEM)([13]):

1- <u>IC Fabrication Checks</u>: Tests by vendor to ensure all fabrication steps have been performed correctly during wafer production
2- <u>IC Design Checks:</u> Tests to ensure prototype ICs perform correctly.
3- <u>IC Production Checks:</u> Tests to ensure produced ICs are defect free
4- <u>Acceptance Tests:</u> Tests by OEM to ensure the incoming ICs are functionally correct
5- <u>Product Tests:</u> Tests by OEM for final manufactured products.

The 1st and 3rd phases is the sole responsibility of vendor, while the 2nd step is either verified by the vendor in the case of standard ICs or by the OEM in the case ASICs. The last 2 phases are only relevant to OEM, who should verify their concordance[i].

II.1.1 – Functional and Structural Testing:

Before structural testing was proposed, digital systems were tested to verify their compliance with their intended functionality, i.e. in this philosophy, a multiplier would be tested whether it would multiply and so forth. This testing philosophy is termed as *functional testing*, which can be defined as, applying a series of determined meaningful inputs to check for the correct output responses in terms of the device functionality ([13]). Although this methodology imparts a good notion of circuit functionality, under the presence of a definitive fault model, it is very difficult to isolate certain faults in the circuits in order to verify their detection ([14]). With the proposal of *structural testing*, which might be defined as, consideration of possible faults that may occur in a digital circuit and applying a set of inputs tailored for detecting these specified faults. In [16], the suggested technique was to define the digital system as a combination of primitive/common basic blocks such as AND, OR, XOR, etc. and inject faults to each gate of the circuit consecutively and generate input test patterns to propagate these faults to observable outputs. As obvious structural testing relies on the fault models described for the Device Under Test (DUT), and any result obtained in this manner is unworthy without a proper description of used fault models. Fault models and fault simulation techniques developed as a result of the above described technique will be elaborated in section VI.

II.1.2 – Controllability and Observability:

These terms, which are mentioned in section II.1, were introduced in 1970s in order to describe the ease – or difficulty – of testing the nodes of a digital circuit. *Controllability* is a measure of how easily – or hardly – can a node of a digital circuit can be driven from the accessible (primary) inputs. Observability, is a measure of how easily the logical value of a given node can be propagated to the observable (primary) outputs. In general, controllability decreases as the distance between the to be controlled node and primary inputs – i.e. the number of internal gates between primary inputs and the node – increases, and observability decreases as the distance from primary outputs increases([13]). A representative plot for observability and controllability can be demonstrated as in figure II-1.

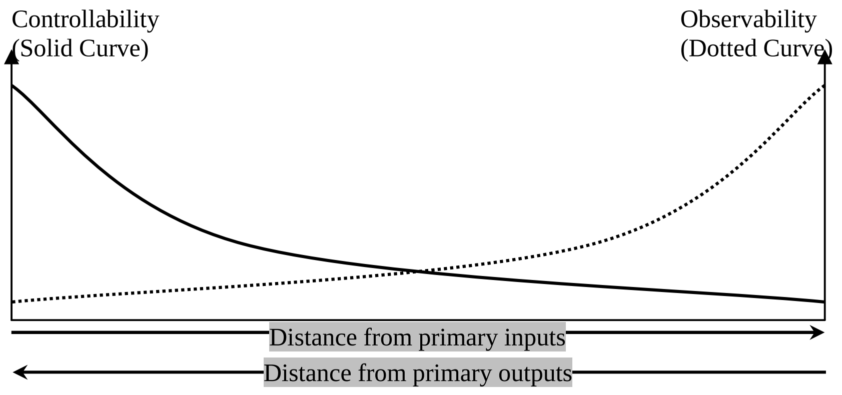


Figure II-1, General Characteristics of Controllability and Observability([13])

There have been various proposed schemes in order to quantify controllability and observability, to identify circuits with low controllability and/or observability during the design phase and modify these accordingly. Some of these include A Testability Measurement Program (TMEAS), TESTSCREEN, Sandia Controllability Observability Analysis Program (SCOAP), Computer-Aided MEasure for LOgic Testability (CAMELOT), VLSI testability analysis program (VICTOR) and COMET[ii].

II.2 – ONLINE VS. OFFLINE TEST

The formerly described VLSI testing phases are usually performed to guarantee a designed digital system conform to their design specifications during the prototype and final production and post-production checks. During these tests, the system is not actually yet operated to perform its normal intended operation, i.e. for a microprocessor, is not yet placed on a motherboard to perform as the CPU of a computer. Regarding the description in [14, p.6], the system is termed *online* in the latter described situation, while is termed *offline*, during the initial production check phases as described in the former case. Obviously, the tests described previously all refer to offline tests, and as a matter of fact, most of the testing effort in digital testing is geared toward offline testing. Nevertheless, occasionally the digital device is required to be tested during the online mode to assure correct operation and state of the device before the initiation of a critical task as well as for error recovery purposes. If the testing of system is performed while the device continues its normal operation, the performed test is named *online test*. However, if the system is needed to end its normal operation to reset the system into a testing mode, the performed testing methodology is termed *offline test.* The testing strategies described in this report and applied in the project are all examples of offline test strategies.

II.3 – COST OF TESTING IN VLSI CIRCUITS

As the VLSI circuits grew in terms of number of components per silicon die, the problem of finding a restricted number deterministic input test patterns to fully or to an acceptable extent test the digital circuit has dramatically intensified. Despite the fact that there is no principal disparity in the determination of each functional test, the complexity of the digital devices yields unmanageable volumes of input test patterns. For instance, a test attempt to test a microprocessor through all the possible states it may encounter during normal operation is irrational. Moreover, for a fully combinational digital circuit, to verify correct functional operation, one should apply all the possible input combinations in the circuit truth table. For instance for an n input, m output device; the total volume of input data to be applied is 2^n words of each n bits long, and the total volume of output data to be observed is 2^n words of each m bits long. For a numerical example, for a 16x16 multiplier, n=32 and m=32 ➔

Total input data volume: 2^{32}.32bits = 2^7.1Gb = <u>128 Gb</u>

Total output data volume: 2^{32}.32bits = $\underline{\underline{128\ Gb}}$

Also considering the time overhead, if each test vector could be applied at a rate of 100MHz, the total time to test a single device would be:

Total test Time: $2^{32}.10^{-8}s \cong \underline{\underline{42.9s}}$

The presence of a storage element – which is inherent in sequential circuits – even worsens this problem. For a circuit with s storage elements, all possible input combinations should be tested for each 2^s internal states, demanding a total number of $2^n.2^s$ input and output patterns [13].

Therefore, a structured subset of all possible inputs to such complex systems must be determined for a reasonable volume of test data. [19] suggests that for circuits that cannot be partitioned into smaller mutually exclusive subcircuits, the number of deterministic tests required to fully test the circuit is linearly proportional to the number of gates in the circuit([13]). However according to the prediction of [19] and results cited in [1, pp.15-18], test application time is squarely proportional to the number of gates.

Literally, testing costs in terms of monetary aspects also involve the Automatic Test Equipment (ATE), as well as development of test strategy ([1, pp.17-18]), which include tester operation, ownership and maintenance costs. ATE costs depend on the data storage volume, output sampling frequency and test time. Therefore, DfT techniques, most prominently BIST alternatives, reduce test costs significantly due to both reduced test data volume and provision of at-speed testing while output sampling rate is kept much lower.

II.4 – TESTING TERMINOLOGY

Before proceeding to the details of testing, it is imperative to describe certain terms related to digital circuit testing. The three terms, which are used to define the test data are([13]):

(i) Input test vector (input vector/test vector): Applied parallel binary signals to the circuit under test via the available primary inputs, at one instance. For example, for a circuit with 8 primary inputs, 10001010 might be one of the applied input test vectors.

(ii) Test Pattern: Applied test vector plus the fault free outputs observed from the available parallel primary outputs. For example, for the above hypothesized circuit with 4 primary outputs, if the fault free outputs are 1111 for primary inputs 10001010, the test pattern is 10001010 1111.

(iii) Test Set: The complete set of all test patterns applied to the circuit under test to determine its non-faulty operation. The test set comprises all the sequence of applied test vectors and to be observed non-fault outputs. An exemplary test set with the test patterns and vectors demonstrated is as in table 1.

Test Patterns		
	Test vectors	**Non-faulty outputs**
Pattern1	10001010	1111
Pattern2	11001001	1010
Pattern3	00100100	0001
.	.	.
.	.	.

Table1, Exemplary Test Set

Although in general literature, these terms are not strictly followed, we will stick to these definitions throughout the project report.

II.5 – TEST PATTERN GENERATION

Test pattern generation is the process of defining an effective test set which will drive the circuit under test so that the faults in the circuit will cause a different response at the primary outputs from the non-faulty outputs. The algorithms used in test pattern generation are usually directed to non-functional testing, which concentrate on propagating any available faults on the circuit nodes to primary outputs. This type of testing is termed *fault oriented* testing ([1]). Test pattern generation is strongly related to fault modeling. Therefore, the applied fault model as well as the faulting hierarchy must be precisely elaborated before test pattern generation. Eldred ([16]) was the first to present a method for test pattern generation for combinational circuits and this introduced concept led to one dimensional path sensitization techniques, which were later developed into a multiple path sensitization technique by J. P.

Roth in [27] in 1966. Roth's D-Algorithm is later optimized for various Automatic Test Pattern Generation (ATPG) techniques.

Test pattern generation methods can be loosely classified into the three major branches ([13]]):

(i) Manual generation
(ii) Automatic (Algorithmic) generation
(iii) Pseudorandom generation

We describe test pattern generation in the above three categories. However, a more comprehensive classification is as shown in figure II-2.

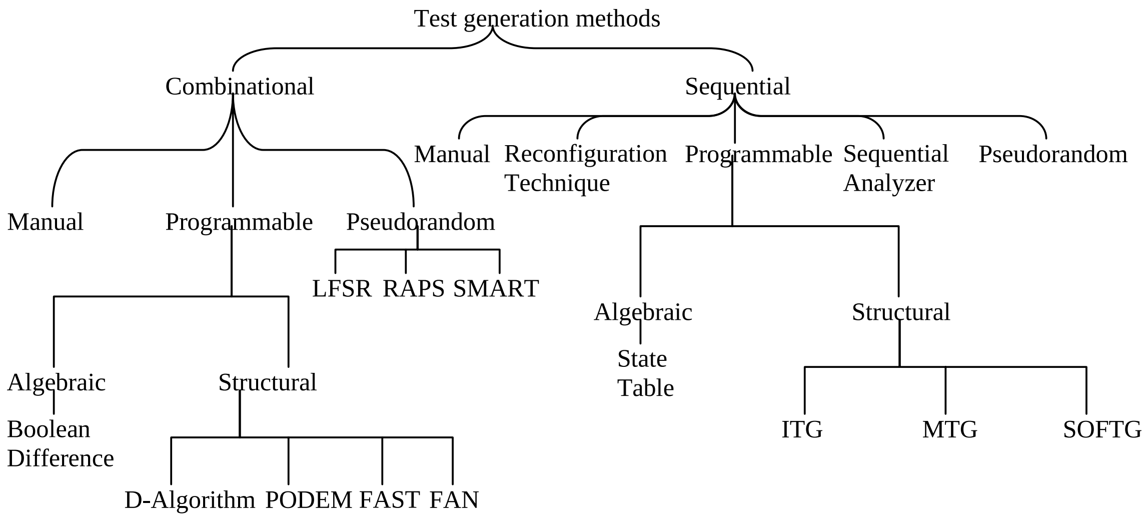


Figure II-2, Test pattern generation techniques ([1])

II.5.1 – Manual Test Pattern Generation ([13]):

Manual test pattern generation might be used by original circuit or system designer depending on the extensive detailed knowledge of the system. The deterministic test set consists of test patterns for specific functional conditions and/or that will provide the propagation of certain node faults. The advantage of this technique lies in the efficient determination of an effective smaller deterministic set, but might require extensive analysis time.

II.5.2 – Automatic Test Pattern Generation ([1],[13],[14]):

Automatic (algorithmic) test pattern generation is the widely applied technique for test pattern generation, as the gate count in the VLSI systems increases rapidly. Mostly, dedicated ATPG programs are utilized for this task, which work on predetermined fault models – usually single stuck at 1. Most ATPG programs choose a faulty node at circuit, propagate this fault to an observable output and backtrace to primary inputs in order to specify the required test vectors for all faults within the circuit. In the forthcoming descriptions of ATPG techniques, the first described one, Boolean difference method does not depend on the described backtracking scheme, while all the others do.

i- Boolean Difference Method[iii]:

Boolean Difference Method uses Boolean algebraic relations for test vector determination. The formal definition of Boolean difference is:

$$\frac{d}{dx_i} f(x) = f(x_1,..,x_i,..,x_n) \oplus f(x_1,..,\bar{x}_i,..,x_n) \qquad (1)$$

where,

f(x): a function of n independent variables

d/dx: difference operator

If the above Boolean difference is 1, then the fault on input line x_i can be detected. If Boolean difference is 0, f(x) is independent of x_i and fault cannot be detected. Boolean difference method covers both stuck at 1 and stuck at 0 faults, but Boolean algebra involved requires extensive computation time and memory and therefore is not the used method in practice.

ii- Single Path Sensitization:

Single path sensitization traces a signal path from faulty node to the primary outputs by setting the other inputs of any logic gate such that the output sensitive to changes in the faulty node. Then, by backward simulation, the required input test vectors are defined. Single path sensitization technique sensitizes only a single path from the faulty node to primary outputs, which deteriorates faults detection probability for reconvergent fanouts. To overcome this shortcoming, D algorithm is proposed.

iii- Roth’s D Algorithm:

D algorithm is a more formal description of path sensitization method and is the foundation point of many ATPG programs used in practice. D algorithm is based on the ‘calculus of D cubes’. It sensitizes all the paths from the faulty node to an observable primary output and therefore is robust against reconvergent fanout. The application of D algorithm requests the knowledge of all gates in the circuit and all interconnection. In D algorithm terminology, D represents a fault sensitive node that is fault free when D=1 and D’ represents a fault sensitive node that is fault free when D=0. The algorithm defines a fault, generates a *D cube of failure*, generates a fanout list for the faulty gate and propagates the effect of faulty gate through all gates in the fanout list via path sensitization, which is termed as *D Drive* process. Finally the algorithm performs a backward simulation for consistency of assigned logic values to primary inputs[iv].

iv- Improvements over D Algorithm[v]:

Although D algorithm is computationally more efficient than Boolean Difference Method, the computation time for test pattern generation is still a significant concern and several modifications over D algorithm are proposed in order to reduce computational cost.

A modification of D-algorithm was Logic Automated Stimulus And Response (LASAR), which worked backwards from the primary outputs by assigning logical values to outputs and working backwards for gate logic values.

In order to reduce test pattern generation cost, an alternative form of fault simulation, called TEST-DETECT is also integrated to D algorithm, which determined what other faults could be detected by each defined test vector. TEST-DETECT starts at primary outputs and backtraces the whole circuit to define *D-chains.*

Following the above two improvements, Path Oriented DEcision Making (PODEM) algorithm is utilized in an ATPG system, named PODEM-X, which comprised a fault simulator, three test pattern generation programs and a test pattern compaction program. PODEM-X is involved in IBM’s DfT methodology, Level-Sensitive Scan Design (LSSD). In PODEM, the path from faulty node to a primary input is backtraced, with branching decisions done heuristically at each step. Once a primary input is reached, the simulator is invoked to verify whether the target fault is sensitized or not. PODEM-X uses a test generation program, named Shift Register Test Generator (SRTG) to test the shift registers. The test generation strategy applied in PODEM-X comprises a general test with RAndom Path Sensitization test generator (RAPS) and fault oriented cleanup tests with PODEM to generate test patterns for remaining uncovered fault conditions.

FUTURE is a test pattern generation system produced by NEC Corp. Similar to PODEM-X, it also has a global and a fault oriented test generator. Fault coverage is determined by a concurrent fault simulator. The fault oriented test pattern generator is named FANout oriented test generation algorithm (FAN), which is verified to be more efficient than PODEM because of the applied heuristics.

LAMP2 Test Generator (LTG) was developed in AT&T labs in order to improve the test pattern generation efficiency for large circuits. LTG also uses two test pattern generation schemes, global test patterns and cleanup test patterns. The global pattern generation is performed by the program Sensitizing Method for Algorithmic Random Testing (SMART) and the fault oriented pattern generation is performed by a procedure named FAST. As a comparison of techniques, test generation for a circuit of 75000 gates took 1.6 CPU hours with LTG, and a circuit of 45000 gates took 7.2 hours and 14 hours for the previous described techniques ([1, pp.123-124]). However, no information is disclosed on the effectiveness and volume of generated test sets!

Another test generation system, HITEST utilizes artificial intelligence concept on expert systems.

II.5.3 – Pseudorandom Test Pattern Generation ([1],[13],[15]):

Manual and algorithmic pattern generation techniques described in the above two sections can be grouped as deterministic techniques as they are based on specifically defining input test vectors that enable the detection of certain faults within the circuit under test. The advantage of deterministic tests is, they provide a compact test set that are targeted to the detection of the defined fault list; and the obvious disadvantage is the extensive computation cost and complexity. At the other extreme, fully exhaustive testing, i.e. applying all the possible $2^{\#\text{ of primary inputs}}$ input combinations, involves almost no complexity and there is no computation cost for the determination of the test set. Obviously, this second approach bears the disadvantage of unrealistic test data volumes for reasonably large circuits. As the general engineering practice, an intermediate scheme that negotiates between complexity and volume is well made use of, named pseudorandom test pattern generation. This third technique to be described in this section relies upon probabilistic measures of random test patterns and the plausible fact that, any random pattern applied to the circuit under test is very likely to detect several faults in the circuit and thus, can be a candidate for a deterministic test vector – though it might not be the most efficient. The fault coverage relation for truly random input test vectors is well investigated in the literature, and tests made in [28] for different combinational circuits, which are tabulated in table 2, revealed the following fault coverage relation approximation for a combinational circuit with N applied random test vectors[13, p.69].

$$FC = \left[1 - e^{(-\lambda \log_{10} N)}\right] \times 100\% \qquad (2)$$

where;

FC	: Fault Coverage
N	: Number of applied random test vectors
λ	: A constant reflecting certain properties of the combinational circuit

Circuit	**# of Primary inputs**	**# of Gates**	**% Fault coverage with respect to # of applied random input test vectors, N**		
			N=100	N=1000	N=10000
(1)	63	926	86.1	94.1	96.3
(2)	54	1103	75.2	92.3	95.9

Table 2, Fault coverage for two combinational circuits with random test vectors [13, p.69]

As a means of quantification of the above expression, the Matlab script in Appendix E-1 is run for N=1000 and λ=1 and the resulting fault coverage plot is as shown in figure II-3. As can be observed, application of random test patterns reveals the same effect as deterministic patterns with a high coverage at the start and a decelerating follow-up. As a result, a small subset of the whole random input state space can be applied to a circuit under test for a fairly comprehensive fault coverage, and as in ATPG, the remaining faults can be targeted with a cleanup testing strategy. Beyond this, as a matter of practice, it is not rational to apply truly random input test vectors as they are very hard to generate algorithmically and it is redundant to apply the same vectors twice for combinational faults, which is a probable case of complete randomness. Therefore, pseudorandom pattern generation, rather than truly random sequences, is the widely applied pattern generation technique. The advantages of PRBS generation is twofold: they are a pseudorandom sequence of all possible states of inputs, except 000..00, without any repetition of states, and they are very easy to generate in hardware. The second statement is also one of the reasons why PRBS generators are very attractive for BIST.

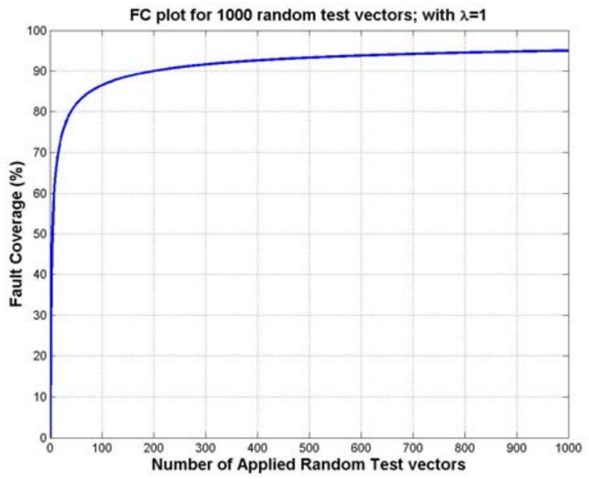


Figure II-3, Fault coverage plot for 1000 random test vectors

On chip PRBS generation is accomplished by utilization of LFSRs or CA, which will constitute the majority of the subject matter for section III. Therefore, we will not here delve into the details of LFSR or CA principles. However, as a requisite to the flow of the context, we here define the general LFSR and CA structures.

LFSR is a serially connected flip-flop configuration – shift register configuration – with feedbacks from certain flip-flop outputs – taps – that are XORed together –added in modulo 2 – and connect back to first flip-flop’s input. The number and position of taps determine the length and sequence of generated PRBS pattern. An exemplary 8 stage LFSR with tap connections that provide maximum possible sequence length (2^n-1 patterns) is as shown in figure II-4.

CA structure is quite similar to that of LFSR, with the inherent shift register configuration. The basic difference from the LFSR is, the interconnections of individual flip-flops now always include an XOR operation and there is no global feedback. CA consist of 2 types of primary cells, namely 90 and 150 cells, and certain combination of these cells reveal maximum length sequences. The only difference between 90 and 150 cells is, 150 cells have an additional self feedback from the flip-flop output to back to its input. An exemplary 4 stage CA, with appropriate 90 and 150 cell configuration for maximum length PRBS is as shown in

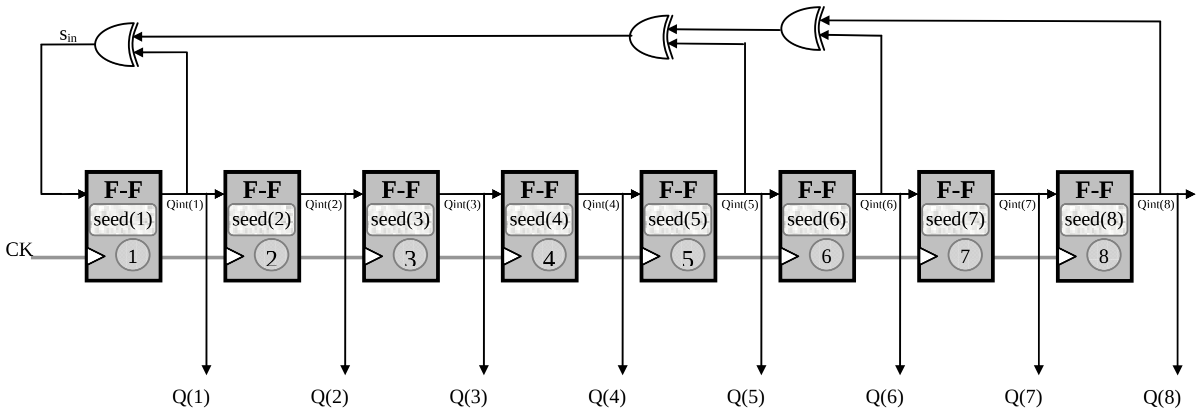


figure II-5.

Figure II-4, An 8 stage maximum length LFSR

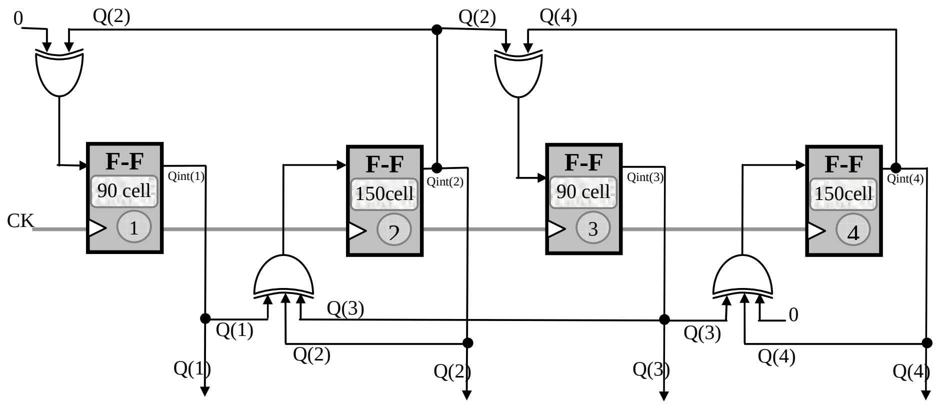


Figure II-5, A 4 stage maximum length CA

As an oversimplified comparison between CA and LFSRs, CA reveal a more random pattern sequence, while LFSRs incur less hardware cost and complexity.

II.6 – TEST DATA COMPRESSION

As mentioned so forth throughout the text, one of the major concerns in testing is the large volumes of test data for reasonably large circuits. This problem must be realized as having two corollaries, the volume of test pattern generation data and the volume of needed to be observed output data, which can be imprecisely named as input data and output data. In this section we discuss techniques to reduce the input and/or output data volume.

II.6.1 – Input Test Data Compression ([13]):

Input compression, also known as *input compaction*, relies on the possibility that, for a multiple output circuit, some of the outputs might be only dependent on some of the inputs and not all. Thus, considering a full exhaustive test to be applied, the amount of required test vectors reduces to half per such independent input. As a hypothetical example, for a 7 input circuit, if 3 of the inputs controlled only a subset of the outputs, 4 of the inputs controlled only another subset and 5 controlled the rest of outputs, the total amount of required exhaustive test patterns would be, in the worst case: $2^3+2^4+2^5 = 56 < 2^7=128$. However, there is an inherent assumption in this application, any possible unexpected interference between the disjoint inputs are discarded, and therefore, the two tests cannot be said to have exactly same effectiveness. Hence, this process is similar to functional decomposition in Switching and Automata Theory, and can be done by: functional partitioning the circuit, minimizing the Boolean relation of each output to see which inputs they are dependent to or more sophisticated techniques like the determination of spectrum of each output function[vi].

II.6.2 – Output Test Data Compression ([1,13,14,15]):

Output compression, also known as *output compaction* or *space compression* is the predominant technique in data compression. The concept of output compression relies on compressing the acquired output data into a much less data volume, which carries all but most fault detection properties that can be inferred from the uncompressed raw output. Inevitably, any such compression technique involves some amount of 'miss rate' that a faulty raw output will be compressed into a signature equivalent to that of a non-faulty output, which is termed '*aliasing*'. Several well-studied techniques for output data compression are described in the following subsections.

i- Syndrome (1s count) testing:

The simplest of the output compression techniques is 1s counting, which is basically the count of 1s (or 0s) at the output of the circuit under test, when a full exhaustive test is applied. Therefore, for an n input circuit, 1s count might range from 0 to 2^n. There is a slight terminology difference between 1s count and syndrome. Syndrome is the normalized 1s count value to the all possible 2^n inputs. Which can be related to 1s count as a 1-1 correspondence as shown below:

$$syndrome = \frac{ones - count}{2^n} \tag{3}$$

In general, syndrome count is not a satisfactory output compression technique, due to its high fault masking. A crude estimation of fault masking for syndrome testing can be done as follows.

If we consider a test with R test vectors, the raw output will be an R bit long sequence, which can have 2^R possible combinations. Out of these 2^R combinations, 1 will be the fault free output while the rest 2^R-1 will be corresponding the faulty output sequences. If the fault free output sequence has s 1s, the total number of sequences having s 1s in the 2^R possibilities is([13]): $\binom{R}{s}$, where 1 of these combinations correspond to the fault free signature and the rest to the faulty signatures. Therefore, out of the 2^R-1 possible faulty sequences, $\binom{R}{s}$-1 will have the same signature as the fault free response, which reveals a fault masking probability of;

$$P_{fm} = \frac{\binom{R}{s} - 1}{2^R - 1} \tag{4}$$

Hence, the above relations are based on the assumption that all the output combinations are equally probable, which is rarely true for combinational circuits. Nevertheless, an interesting observation is, if the fault free output has very low or very large amount of 1s, the fault masking probability is very low.

ii- <u>Accumulator-Syndrome testing:</u>

Accumulator-syndrome testing is a modification of the syndrome testing of previous subsection, where the integral of the syndrome is considered as the final signature. In discrete point of view, the integral corresponds to the accumulation of the syndrome count values – hence the name accumulator. An exemplary case is depicted in table 3.

Output sequence	0	0	1	0	1	0	0	1	1	1	0	…
Syndrome count	0	0	1	1	2	2	2	3	4	5	5	…
Accumulator-syndrome values	0	0	1	2	4	6	8	11	15	20	25	…

Table 3, Syndrome count vs. Accumulator-Syndrome Testing

Obviously, the accumulator-syndrome testing depends on the order of the applied test vectors.

iii- <u>Transition Count testing:</u>

Transition count relies on counting the number of transitions from 1→ 0 and 0→ 1, rather than counting the number of 1s or 0s at the output. This test is also dependent on the order of applies test vectors. As in syndrome testing, this technique also has a margin of error for missed faulty sequences, but this can be taken care of with ordering the test vector sequence such that, the output sequence generated is all 0s followed by all 1s or vice versa. However, this approach has two major drawbacks, 1) the input test set should be in fixed order, which reduces BIST generation possibility 2) each output requires a separate set of test vectors. Consequently this technique is also not very much utilized in applications.

The fault masking probability for transition counting, computed in a similar fashion as in subsection (i) is([14]):

$$P_{fm} = \frac{2\binom{R-1}{s} - 1}{2^R - 1} \tag{5}$$

where;

s: transition count value for the fault free output

iv- <u>Alternative Mathematical Coefficients[vii]:</u>

Despite the generally accepted use of Boolean algebra in definition of digital circuits, other mathematical modeling methods coexist which might be utilized in circuit testing. Some of these alternatives are *Arithmetic, Reed-Muller* and *Spectral Coefficients.*

In arithmetic coefficients, the functional expression for a circuit is defined in arithmetic addition rather than Boolean addition. The arithmetic coefficients are coefficients of the minterms in an expression, which can take various integer values rather than the general binary values used in Boolean Algebra’s sum of products form.

In Reed-Muller coefficients, the canonic expansion of the functional expression of a circuit is defined in terms of XOR relations. The Reed-Muller coefficients represent the coefficients for the minterms in the XOR relations.

In spectral coefficients, the coefficients of the minterms in Boolean Algebraic form are transformed into Spectral domain using *‘Hadamard’* orthogonal transformation. The resulting coefficients are generally interpreted as correlation coefficients.

In terms of testing, arithmetic and Reed-Muller coefficients are not very well utilized as test signatures in comparison with spectral coefficients. Spectral coefficients are researched vastly for testing purposes, they are seen to provide very useful information for determining the correlation between different inputs of a circuit for both input and output compression. However, the major drawback is stated as the computational cost of computation of the coefficients due to the required transformation.

v- Parity check testing:

Parity checking simply checks whether the resulting output sequence has even number of 1s or vice versa. Directly following from this definition, parity checking will detect all single bit errors and any multiple errors that will change the even-odd parity of the output sequence. In terms of BIST prospects, parity checking can simply be applied with an XOR and a shift register. The computer fault masking probability in a similar fashion with the previous cases reveals:

$$P_{fm} = \frac{(2^R/2)-1}{2^R-1} \tag{6}$$

Moreover, parity checking of multiple output sequences can be done via the application of a big pre-XOR to all the outputs.

vi- Signature Analysis:

Amongst the techniques described so far, signature analysis is the widespread used technique in practice for BIST output compression technique. The signature analysis process is not fundamentally different from the LFSR, but the input XOR of the LFSR is connected to the output data sequence rather than being kept at 0. As all the test vectors are applied, the output is fed into the XOR and this causes the transitions in the LFSR states, then after the application of the whole test set, the remaining signature is read as the output signature. Hence, in this methodology, the LFSR does not necessarily traverse all the possible 2^n-1 states, as the next states are dependent on the serial output value. Based on the previously stated assumption, the fault masking probability of signature analysis can be determined as follows. Once again for R input test vectors, there will be R output values generated as the output sequence. Considering the number of stages in the LFSR as n, where n<R, only 2^n sequences can be produced as the signature from the n bit LFSR. Assuming all these signatures are equally probable, the number of sequences having the same signature would be: $2^R/2^n=2^{R-n}$, out of which 1 will be the fault free output and the rest faulty sequences, sharing the same signature with the fault free one. As a result the fault masking probability can be summarized as:

$$P_{fm} = \frac{2^{R-n} - 1}{2^R - 1} \tag{7}$$

which can be approximated to $P_{fm} = \frac{1}{2^n}$, for large R.

One further enhancement to signature analysis, in today's testing circuits is the application of multiple outputs to the same LFSR, which is termed as Multiple Input Signature Analysis.

[i] For more detailed information on IC fabrication, testing and OEM-Vendor relation refer to [13, pp. 1-7]
[ii] For more detailed information on testability measures, refer to [13, pp. 20-25],[17],[18, pp. 4-11]
[iii] For more information on Boolean difference method, the reader may refer to [13, pp. 51-55].
[iv] Comprehensive information on single path sensitization and D algorithm is available in [1, pp. 108-115] and [13, pp.48-63].
[v] For more information on described techniques, please refer to [1, pp. 115-144], [13, 63-67],[14, 10-11]
[vi] For more information on input compression methods, please refer to [13, pp. 120-124]
[vii] For more information on the described coefficients, refer to [13, pp. 127-141],[14, pp.103-108]

III-PRBS GENERATION AND SIGNATURE ANALYSIS

In this project, we extensively investigate PRBS generation and signature analysis as BIST for multipliers. In the previous section, we have provided a general introduction to testing and various techniques. In this section, we specialize ourselves into PRBS generation using LFSRs and CA and into signature analysis, specifically multiple input signature analysis.

III.1 – PRBS GENERATION USING LFSRS

As described in section II, LFSR structure, basically, is a shift register configuration that propagates the stored patterns from left to right. The modification that provides the PRBS generation is due to the XOR feedback of the selected flip-flop outputs, named *taps*. When the taps are chosen properly, the LFSR will traverse through all possible states except for the all 0s state and will produce a maximum length PRBS sequence named *M-sequence*. In order for the desired operation, the LFSR should be first initialized to a well-known stage, which is usually referred to as *seed*. For an n stage LFSR, there are 2^n-1 states, and the M-sequence is 2^n-1 bits long. Hence, the M-sequence is periodic, and after the 2^n-1 distinct values, it repeats itself in the next samples. The forbidden state, which the LFSR never traverses is usually referred to be 00..00, but if one of the feedback values are inverted, or XNOR instead of XORs are used, the forbidden state may be altered. The reason why the all 0s state is considered the forbidden state is, when all the flip-flop values are 0, the XOR of $p<n$ outputs will reveal a 0 regardless of the location of the taps as: $(0\oplus 0) \oplus 0 \oplus 0 \ldots \oplus 0 = 0 \oplus 0 \ldots = 0 = 0$. Therefore, the fed back value is always 0, and the LFSR always stays in the 00….0 state.

Regarding the above described output generation scheme for the LFSR, the following listed properties can be deduced as the general characteristics of an n stage LFSR:

(1) A max-length LFSR generates a length 2^n-1, periodic m-sequence, obviously, with period 2^n-1. Consequently, the LFSR, traces all the corresponding 2^n-1 states of n bit length periodically.

(2) In one period of the m-sequence, the number of 1s exceeds the number of zeros by exactly 1. Therefore, the total number of 1s in an m-sequence is 2^{n-1} and the total number of 0s is 2^{n-1}-1. The reason for this is, as the LFSR traverses all the states

except for all 0s for once within one period. All the m-sequence bits, which can be considered as, in practice, the LSBs or MSBs, more generally any single bit of the states, correspond to a single state and all the bits for possible state combinations, except for the all 0s state exist in the m-sequence, which would imply a final 0 in the m-sequence. Therefore, the missing 0 results in the number of 1s being 1 more than the number of 0s.

(3) (The Run Property) In the m-sequence, there will be a total of 2^{n-1} runs – bursts of the same bit – of each bit. Among these runs, ½ of the runs will have length 1, ¼ of the runs will have length 2, 1/8 of the runs will have length 3 and so on. In addition to these, there will be one additional run of 1s with length n.

(4) As a corollary to (3), the number of transitions in the m-sequence will be 2^{n-1}.

(5) (Shift and Add Property) Every m-sequence has a cyclic shift and add property such that, if the original m-sequence is rotated and added to itself in mod_2, the resulting sequence is also a rotated version of the original m-sequence.

(6) The autocorrelation of the m-sequence is constant for every shift value and the number of matching bits for any shift is ~½ of the total amount of bits = 2^n-1. More specifically, if we define the autocorrelation function as ([13],[14]):

$$C(\tau) = \frac{1}{p}\sum_{1}^{p} a_{\tau} \tag{8}$$

where;

p:2^n-1

τ: shift between the 2 sequences ($1 \le \tau \le 2^n-2$)

a_τ: 1 if the two entries after the shift are the same

-1 if the two compared entries are different

for all τ, $C(\tau)=-1/p$. Which reveals, for the n bit LFSR, 2^{n-1} –1 bits of the m-sequence always match and 2^{n-1} bits don't match. Although this property does not have any significance in terms of testing, it indicates the randomness feature of the generated PRBS.

(7) (Window Property) If a sliding window of length n is moved along the m-sequence, the 2^n-1consequtive n-tuples observed are seen exactly once in a period, which actually represent the 2^n-1 unique states.

(8) (Decimation Property) Every proper decimation of an m-sequence is also an m-sequence ([14, p. 80]).

Having described the properties and PRBS generation outline for the LFSRs, one important concept which needs to be elucidated is how to choose the taps for max-length PRBS generation. The theory of how to choose the tap positions for max-length PRBS generation is discussed in the following subsection.

III.1.1 – Theory of LFSR Taps:

It might be easily verified that, not all combinations of the tap choices will provide an m-sequence. Therefore, the theory to determine the tap locations which can provide an m-sequence must be established and the requirements for the LFSR to generate the all possible 2^n-1 states must be stated.

After the initial transition period, the values within the shift register can be considered as previous values of the feedback input at the first flip-flop. As shown in figure III-1, assigning the feedback input y_i, the following flip-flop outputs can be named as y_{i-1},y_{i-2} and so on. As a general description, the taps can be described via switches c_i, where c_i=0 represents an open – nonexistent – connection and a 1 representing an existent connection. As XOR operation is a modulo 2 summation operation, all the multiple XORs to be performed can be combined as a modulo 2 adder.

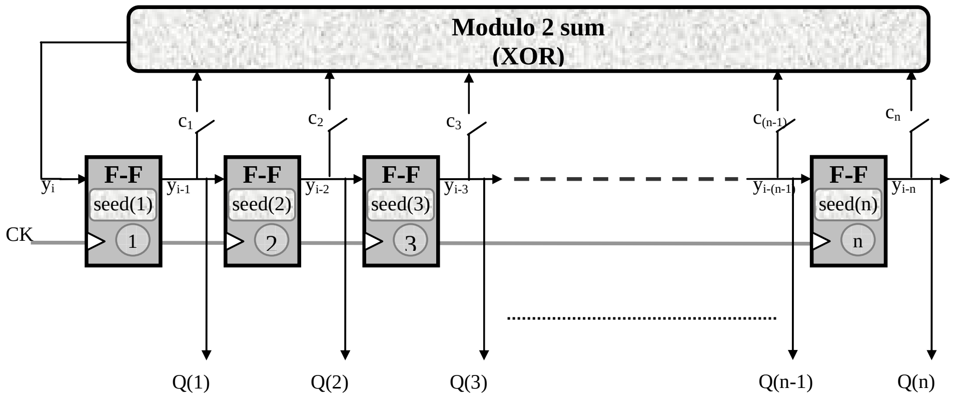


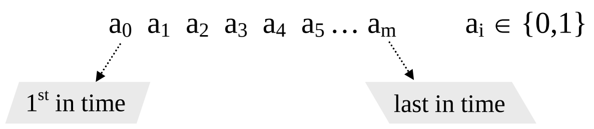
Figure III-1, General Description of the LFSR Structure

With this description, the feedback value y_i can be described with the following expression:

$$y_i = \sum_{j=1}^{\infty} c_j y_{i-j} \tag{9}$$

where the sigma operator represents modulo 2 addition rather than base 10 addition. Although the summation is shown to be bounded by infinity, it will be truncated to the length of the LFSR, as all the following c_j are intuitively zero afterwards.

As a preliminary to the rest of the discussion, a well-applied practice in coding theory is to represent a binary sequence as a polynomial, where the binary values represent the polynomial coefficients and the powers polynomial variables represent the timing or sequencing information. More formally, for the sequence described below:



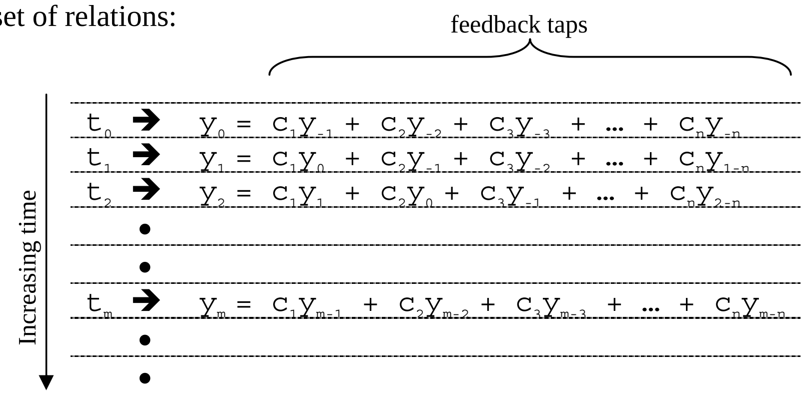
$$a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ldots a_m \qquad a_i \in \{0,1\}$$

the corresponding polynomial, G(x) is:

$$G(x) = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_mx^m$$

Hence, all the algebra defined for the polynomials are in modulo 2 arithmetic, which are referred to as operations in *Galois Field of 2* (GF(2)). In GF(2), addition and subtraction are equivalent as 0+1 = 1+0 = 0-1 = 1-0 = 1 and 1+1 = 0+0 = 1-1 = 0-0 = 0. Multiplication is performed as normal bitproduct and add fashion where the summation of partial products is done in GF(2). Division is also performed in the usual fashion, but the subtractions are performed in GF(2).

Now returning back to the discussion of taps, when we consider the polynomial representation for y_i input with respect to time, for time, t, 0 to time, m, t_m, we reach the following set of relations:



$$\begin{array}{lll}
t_0 & \Rightarrow & y_0 = c_1y_{-1} + c_2y_{-2} + c_3y_{-3} + \ldots + c_ny_{-n} \\
t_1 & \Rightarrow & y_1 = c_1y_0 + c_2y_{-1} + c_3y_{-2} + \ldots + c_ny_{1-n} \\
t_2 & \Rightarrow & y_2 = c_1y_1 + c_2y_0 + c_3y_{-1} + \ldots + c_ny_{2-n} \\
 & \bullet & \\
 & \bullet & \\
t_m & \Rightarrow & y_m = c_1y_{m-1} + c_2y_{m-2} + c_3y_{m-3} + \ldots + c_ny_{m-n} \\
 & \bullet & \\
 & \bullet &
\end{array}$$

Now, if we write the corresponding polynomial, $G_y(x)$ for the y_i sequence,

$$G_y(x) = y_0x^0 + y_1x^1 + y_2x^2 + \ldots + y_mx^m + ..$$

which can be written in the compact form:

$$G_y(x) = \sum_{m=0}^{\infty} y_m x^m \qquad (10)$$

and substituting y_m with the feedback expression, $y_m = \sum_{j=1}^{\infty} c_j y_{m-j}$,

$$G_y(x) = \sum_{m=0}^{\infty}\left(\sum_{j=1}^{n} c_j y_{m-j}\right)x^m \tag{11}$$

further modification of this relation reveals:

$$\boxed{G_y(x) = \frac{\sum_{j=1}^{n} c_j x^j\left(y_{-j}x^{-j} + y_{-(j-1)}x^{-(j-1)} + \cdots + y_{-1}x^{-1}\right)}{1+\sum_{j=1}^{n} c_j x^j}} \tag{12}$$

Hence, the above equation describes how the initial conditions and the tap positions are related to the generated PRBS sequence y_i. The parenthesized term in the numerator is equivalent to the initial condition sequence, and the c_j terms in the denominator are equivalent to the feedback taps. Two immediate observations to be deduced from the above expression are:

(i) If all the initial conditions, $y_{-1},\ldots,y_{-n}$ are 0, the $G_y(x)$ polynomial is zero, independent of the denominator. This explains the all 0s forbidden state as the initial condition as the output sequence will be all 0s.

(ii) If all initial conditions are zero except for y_{-n} – the seed for the last flip-flop -, then the numerator is:

$$c_n x^n.(y_{-n}x^{-n}) = c_n$$

as a result, for an n stage LFSR, there should always be a feedback from the last flip-flop, in order to be able to produce all LFSR states that are expected to include 000…001. For $c_n = 1$, the $G_y(x)$ polynomial equals 1/(denominator polynomial).

The denominator polynomial, $1+\sum_{j=1}^{n} c_j x^j$, is referred to as the *characteristic polynomial, P(x),* of the LFSR, and together with the initial conditions, which are referred to as the *seed*, defines the generated PRBS sequence by the LFSR. As long as the LFSR is known to generate an m-sequence, the highest order of the characteristic polynomial defines the length of the LFSR. As described above, the PRBS sequence can be analytically derived for the LFSR, given the initial seed and the characteristic polynomial, however, this is usually not the practical approach as the notion of PRBS generation is also to reduce the time overhead for pattern generation. As an example ([13, pp.78-86]), for the 4 stage LFSR shown in figure III-2, the characteristic polynomial is:

$$P(x) = 1 + c_1x^1 + c_2x^2 + c_3x^3 + c_4x^4 = 1 + 1.x^1 + 0.x^2 + 0.x^3 + 1.x^4 = 1 + x^1 + x^4$$

and, as described in (ii), the polynomial representing the y_i becomes $G_y(x) = \dfrac{1}{P(x)}$ for the given seed = 0001.

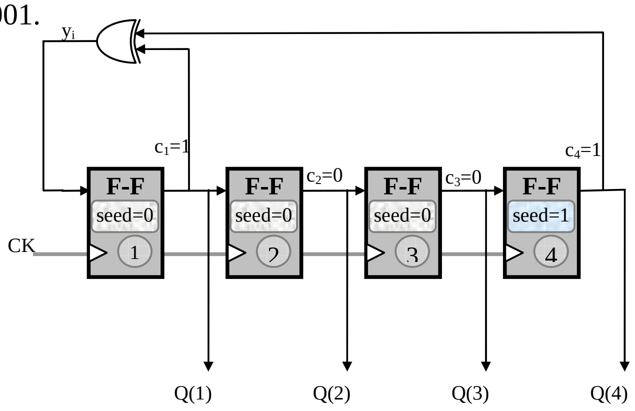


Figure III-2, 4 stage Max-length LFSR

The resulting y_i m-sequence can be computed by doing the above polynomial division in GF(2):

$$
\begin{array}{l|l}
1 & 1 + x + x^4 \\ \cline{2-2}
\underline{1 + x + x^4} & 1 + x + x^2 + x^3 + x^5 + x^7 + \ldots = G_y(x) \\
\quad x + x^4 & \quad \searrow y_0 \quad \searrow y_1 \quad \searrow y_2 \quad \cdots\cdots \\
\quad \underline{x + x^2 + x^5} & \\
\qquad x^2 + x^4 + x^5 & \\
\qquad \underline{x^2 + x^3 + x^6} & \\
\qquad\quad x^3 + x^4 + x^5 + x^6 & \\
\qquad\quad \underline{x^3 + x^4 + x^7} & \\
\qquad\qquad x^5 + x^6 + x^7 & \\
\qquad\qquad \underline{x^5 + x^6 + x^9} & \\
\qquad\qquad\quad x^7 + x^9 & \\
\qquad\qquad\quad \underline{x^7 + x^8 + x^{11}} & \\
\qquad\qquad\qquad x^8 + x^9 + x^{11} & \\
\qquad\qquad\qquad\quad \bullet & \\
\qquad\qquad\qquad\quad \bullet &
\end{array}
$$

Hence, as a result,

$$G_y(x) = 1.x^0 + 1.x^1 + 1.x^2 + 1.x^3 + 0.x^4 + 1.x^5 + 0.x^6 + 1.x^7 + \ldots$$

and the generated PRBS is, from 1st output to last is,

PRBS Sequence: 1 1 1 1 0 1 0 1 …

Hence, the used characteristic polynomial is known to produce an m-sequence, therefore, the resulting sequence is expected to be periodic with 2^4-1=15, and if the above division process is continued up to x^{15}, which is the 16th output sample, the sequence will be seen to repeat itself.

Regarding the above division process, it is proven ([1], [14]) that for $G_y(x)$ to have maximum period, the characteristic polynomial must be not factorizable. Moreover, as $G_y(x)$ will still be periodic with 2^n-1, the characteristic polynomial must be a factor of $1+x^{2^n-1}$. The polynomials that satisfy above conditions are *primitive polynomials*, which are a special case of *irreducible polynomials,* and are used as the characteristic polynomial for maximum length LFSRs. Primitive polynomials have very interesting properties some of which also relate to PRBS generation ([13],[14]). In the next subsection, we discuss some of these properties briefly. The complete descriptions and proofs might be referred from the cited references.

III.1.2 – Primitive Polynomials:

As described in the above section, the principal feature of the primitive polynomials is, they cannot be factorized into 2 or more smaller order polynomials. Other properties are described within this subsection.

As n – can be regarded as the length of LFSR or order of polynomial – increases, the number of possible primitive polynomials increase rapidly, most of the textbooks include one of the minimum term polynomials for each n, but there are also alternative polynomials with minimum number of terms for $n \geq 3$ ([13]).

For any primitive polynomial, P(x), the reciprocal of the polynomial, which is defined as $P^*(x) = x^n P\left(\frac{1}{x}\right)$ is also a primitive polynomial, and the m-sequence generated by the reciprocal primitive polynomial is exactly the reverse of the m-sequence generated by the original polynomial.

III.1.3 – Alternative LFSR Configurations:

Although we have focused on a single type of LFSR configuration so far, other LFSR configurations also exist. Referring the so far described configuration as the TDL (Tapped Delay Line) configuration – with an analogy to DSP, another configuration is the TDA (Time Delay and Accumulate) structure, which is shown in figure III-3. This structure is also termed as *true polynomial divider* due to its algebraic properties.

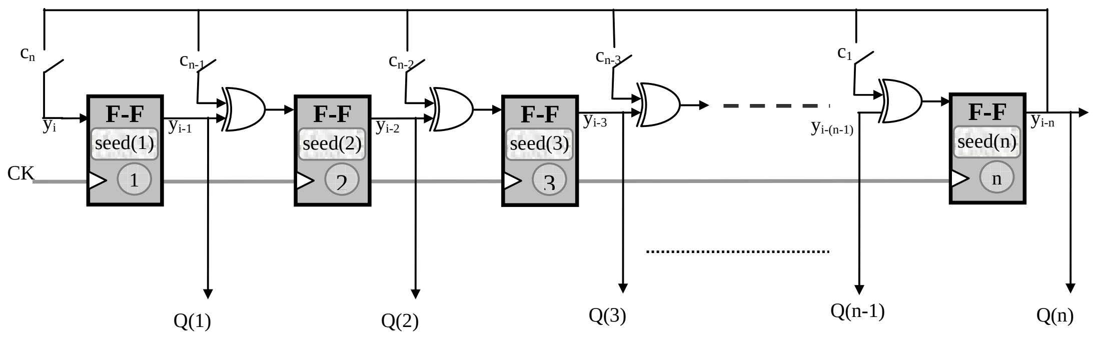


Figure III-3, TDA – LFSR structure

This structure, when the same characteristic polynomial is used in the shown reversed order, produces the same output sequence as the TDL structure. However, the internal states in the 2 configurations are not always the same([13]). This structure is not the preferred structure as it incurs additional delay in the forward datapath and for manufacturing purposes ([13],[1]). However, the TDA structure also has an advantage in data compression as it performs true polynomial division and the remainder in the LFSR is the correct remainder only for TDA structure.

Another structure, aiming more control over generated test patterns is *Nonlinear Feedback Shift Registers (Non-LFSR)*, which is described in figure III-4. This structure tries to achieve a more effective test set, without reverting to storing the test vectors in ROM, thus minimizing the test cost. Instead of the XOR functions, the feedback function is realized with NAND, NOR, etc. combinational functions. The major problem with this structure is the design overhead to define the combinational function, which might not even be realizable and to abide by predefined sequencing rules ([1, pp. 168-169]).

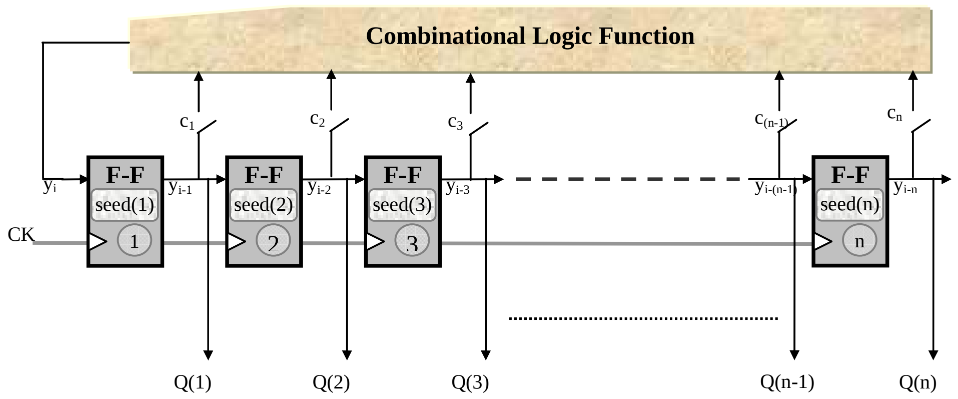


Figure III-4, Nonlinear Feedback shift Register

Further improvements on the TDL and TDA structures have also been investigated and hybrid structures to minimize the logic cost are proposed as cited in [13, p.88].

III.2 – PRBS GENERATION USING CA

A second less studied alternative for PRBS generation is cellular automata. As introduced in section II, the basic shift register structure also exists in CA, with the ultimate difference that all the cell interconnections have some XOR operation and that no global feedback is required. Therefore, the regular shifting of data within the shift register is not existent in CA ([13],[15]). The cells for CA are defined in terms of flip-flops and XOR combinations of neighbor cells. The input to any cell depends only on its adjacent neighbors and maybe itself depending on whether the cell is a 90 or 150 cell, as shown in figure III-5. Considering the possible inputs from the three stated cells, the input functions for the 90 and 150 cells are described in table 4.

Q_{k-1}	Q_k	Q_{k+1}	90 Cell: $Q_{k-1}\oplus Q_{k+1}$	150 Cell: $Q_{k-1}\oplus Q_k\oplus Q_{k+1}$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	0	1

Table 4, 90 and 150 functions of CA

The terminology for 90 and 150 cells comes from the decimal value of the binary outputs for the two cells. As can be calculated, they sum up to 90 and 150 respectively.

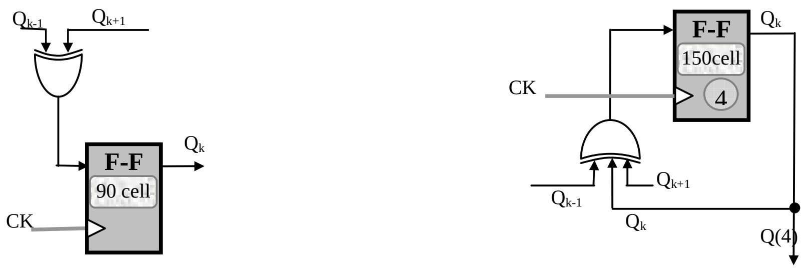
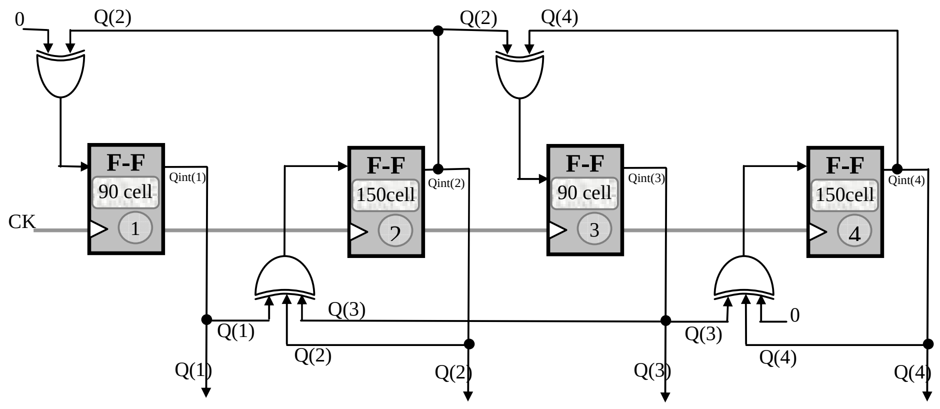


Figure III-5, 90 and 150 CA cells

Other functions of the three outputs are also investigated and it is formally proven that, only 90 and 150 cells produce m-sequences for PRBS generation. Moreover, not all combinations of 90 and 150 cells can produce m-sequences. It is theoretically proven that, for n≤150, at most 2 150 cells are sufficient to produce a configuration of 90 and 150 cells that produce m-sequences. An exemplary 4-stage max-length CA, copied from section II, is redisplayed here in figure III-6 and the corresponding m-sequence is also displayed and compared to the response of a 4 stage LFSR. As can be seen in the m-sequence, unlike the LFSRs, the CA do not exhibit the cyclic shift behavior. As demonstrated in the figure, the LFSR output follows a cyclic shift of 1s (and 0s) from left to right, while no such pattern is distinguishable in CA. As expected, the forbidden all 0s state also exists in CA.



(a). 4 stage max-length CA

CA sequence				*LFSR sequence*			
Q(1)	*Q(2)*	*Q(3)*	*Q(4)*	*Q(1)*	*Q(2)*	*Q(3)*	*Q(4)*
1	0	0	0	**1**	0	0	0
0	1	0	0	0	**1**	0	0
1	1	1	0	0	0	**1**	0
1	1	1	1	**1**	0	0	**1**
1	1	0	0	**1**	**1**	0	0
1	0	1	0	0	**1**	**1**	0
0	0	0	1	**1**	0	**1**	**1**
0	0	1	1	0	**1**	0	**1**
0	1	1	0	**1**	0	**1**	0
1	0	1	1	1	**1**	0	**1**
0	0	1	0	**1**	1	**1**	0
0	1	0	1	1	**1**	1	**1**
1	1	0	1	0	1	**1**	1
1	0	0	1	0	0	1	**1**
0	1	1	1	0	0	0	1

(b). Output sequences of LFSR and CA

Figure III-6, A 4 stage maximum length CA and its output sequence compared with an LFSR output

The generated m-sequence for CA can be derived in a similar manner as LFSR, but the GF(2) operations are demonstrated rather using matrix relations. If we define a transition matrix T, which defines the next state from the current state such that:

$$[Q_{next}]=[Q_{present}]*[T] \tag{13}$$

where, [Q] represents a length n – number of stages – array row vector with the above described Q outputs of CA and [T] represents a nxn state transition matrix[1]. As each output depends only itself – if 150 cell – and its adjacent neighbors, the transition matrix is a tridiagonal matrix with the 1st diagonals completely 1s and the main diagonal only 1 for the corresponding 150 cells. As an example, the next state for the CA in figure III-6 when the state is 1111 – line 4 – can be found as:

$$\begin{bmatrix}1 & 1 & 0 & 0\end{bmatrix} = \begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix} \times \begin{bmatrix}0 & 1 & 0 & 0\\ 1 & 1 & 1 & 0\\ 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 1\end{bmatrix} \tag{14}$$

which is in accordance with line 5. It is worth reemphasizing that all the arithmetic is performed in GF(2), due to mod2 operation of XOR gates. As a direct consequence, as the 2nd next output is $[Q_{next}]*[T]$, the 2nd output is related to the current one as $[Q_{present}]*[T]^2$; and the argument can be easily furthered to the nth next output as:

$$[Q_k]=[Q_0]*[T]^k \tag{15}$$

where Q_0 represents the current state and Q_k represents the output state after k clocks.

It is also proven that, CA and LFSRs are isomorphic, so that for every LFSR configuration there is a corresponding CA configuration and vice versa[2].

In the previous section, a mathematical condition for the LFSR configuration was derived to achieve max-length PRBS, which had revealed the characteristic polynomial should be a primitive polynomial. A similar argument can be hold for the CA by considering the T matrix. For an n-stage CA, the output states must be different for 2^n-1 clocks. Therefore, the state transition should never map the initial state back to itself before the $(2^n)^{th}$ clock, which can be stated in GF(2) algebra as:

$$\text{For } k=1,\ldots,2^n-1, \qquad [T]^k \neq [I]$$

$$\text{and}$$

$$\text{For } k=2^n, \qquad [T]^k = [I]$$

where, [I] represents the identity matrix.

[1] I use a slightly different notation than the described notations from [13] and [15], which is more convenient to associate with the drawn hardware structure. However, the reader should be aware that, the given relation holds as the T matrix is symmetric matrix

As demonstrated, the CA can be equally well applied as LFSRs, for PRBS generation, and their comparison therefore is important for the choice of the more appropriate one, which is the context of the following subsection.

III.2.1 – LFSR vs. CA:

As presented, either LFSRs or CA can be used as on chip PRBS pattern generators. In terms of area penalty, CA are inferior to LFSRs due to the requirement of at least 1 2 input XOR gate per flip-flop. The hardware difference becomes extremely significant as the number of stages increases as the total number of required XORs is at most 4 for max-length PRBS with LFSRs. Hence, regarding the MVSD2 ILP, the circuit area of a flip-flop is not predominantly larger than a 2 input XOR and it cannot be confidently stated that the XOR areas are insignificant compared to flip-flops. However, on the other hand, the LFSR configuration requires a global feedback that runs through the length of the LFSR, which is very undesirable in terms of interconnect delays and capacitances. Therefore, this feedback might be a serious critical path bottleneck in long LFSRs. Conversely, the interconnect length in CA is just between the adjacent cells and it is independent of the length of the CA. Moreover, the output of the CA compared to the output of LFSR – as shown in figure 6 – is seen to be more naturally random, with no cyclic shift behavior. Analytically, CA output does not possess the high cross correlation between individual output bits that LFSR output bears, which is inherent in any shift register action. Consequently, it can be suggested that the CA output will provide a better approximation for the fault coverage relation $FC = \left[1 - e^{(-\lambda \log_{10} N)}\right] \times 100\%$ described in section II. However, although intuitively it would be expected that CA serves better to the notion of PRBS generation and therefore it must be more effective than LFSR, it is not a concept that can be verified quantitatively in general.

III.3 – SIGNATURE ANALYSIS

As described in section II, signature analysis was first developed by HP® to test PCBs using a probe, which was connected to the nodes within the circuit and applied to the input of the LFSR through an XOR gate. Therefore, the signature analyzer works like an LFSR, following the LFSR output sequence as long as the input is zero and inverses the feedback bit if the input bit is 1. As the input 1 will produce a 1 input from 0 feedback, there is no

[2] As stated in [13] this isomorphism is a rather loosely used term, as it does not guarantee the same output sequence, but the same states with a different order

deadlock for all 0s and the signature analyzer can be reset to all 0s state initially. The error coverage probability of the signature analyzer had been derived in section II.

Due to the fundamentals being the same as in LFSRs, the polynomial relation in the signature analyzer can be developed in a similar manner. Recalling the discussion in section III.1 about the true polynomial divider, for the signature analyzer, a *Divisor Polynomial* is defined instead of the characteristic polynomial of LFSRs. Divisor polynomial is the same as characteristic polynomial defined for the TDA LFSR. In relation to the characteristic polynomial of TDL type LFSR, the divisor polynomial is reciprocal of the characteristic polynomial, P(x), or simply, the same bitstream read in reverse order, first bit representing the highest order of magnitude. An example is demonstrated in table 5.

Characteristic Polynomial	1	$1.X^1$	$0.X^2$	$0.X^3$	$0.X^4$	$1.X^5$	$1.X^6$	$0.X^7$	$1.X^8$
Bitstream Representation	1	1	0	0	0	1	1	0	1
Divisor Polynomial	$1.X^8$	$1.X^7$	$0.X^6$	$0.X^5$	$0.X^4$	$1.X^3$	$1.X^2$	$0.X^1$	1

Table 5, P(x) and D(x) relation

Hence, the divisor polynomial is defined by the same bit sequence, with the sequence of orders reversed. This is also achieved by taking the reciprocal of P(x), but used in reverse order for polynomial division in signature analysis. Upon this distinction between D(x) and P(x), the polynomial relation for the signature analyzer can be expressed as:

$$\frac{I(x)}{D(x)} = G(x) + \frac{R(x)}{D(x)} \tag{16}$$

where,

I(x): data input bit stream expressed as a polynomial
R(x): The residue – signature – remaining in the signature analyzer
D(x): divisor polynomial, G(x): output sequence

As an example, considering the same LFSR configuration in figure III-2, with an additional XOR input to the first flip-flop, where the serial input is incorporated, the divisor polynomial will be $D(x) = x^4 + x^3 + 1$, as easily deduced from table 5. Assuming a bit stream of 1010011[3], the I(x) polynomial will be $x^6 + x^4 + x^1 + 1$. Doing the polynomial division in GF(2) as described in page 7 will reveal:

$$\begin{array}{l|l}
x^6 + 0 + x^4 + 0 + 0 + x^1 + 1 & x^4 + x^3 + 1 \\ \cline{2-2}
x^6 + x^5 + x^2 & \underset{\searrow\, y_0}{x^2} + \underset{\searrow\, y_1}{x^1} + \underset{\searrow\, y_2}{0} = G(x) \\ \cline{1-1}
\quad x^5 + x^4 + 0 + x^2 + x^1 + 1 & \\
\quad x^5 + x^4 + 0 + 0 + x^1 & \\ \cline{1-1}
\qquad 0 + 0 + x^2 + 0 + 1 = R(x) &
\end{array}$$

Compared to the LFSR case, one subtle difference is, the G(x) polynomial generated in the signature analysis case is not continuing indefinitely, as the applied input stream is finite length. Quantitatively, a length 'L' input stream produces a signature of length '$\underline{\underline{n}}$' – obviously as the length of LFSR is n – and length $\underline{\underline{\text{L-n}}}$ output sequence. This conclusion applied to the above example will compel the output stream to be '110' and the residue to be '0101'. Recalling subsection II.1.3, these results must be noted to be valid for only the TDA LFSR, as stated, being the *true polynomial divider*.

However, originally being developed for PCB testing, where the probe can access internal circuit nodes, the application of signature analysis to embedded VLSI systems is not realistic as each output requires a separate signature analyzer. In today's practical BIST circuits, multiple input signature analyzers (MISRs) are used to serve as a solution for multiple outputs.

III.3.1 – Multiple Input Signature Registers:

For BIST of multiple output circuits, MISR are extensively utilized in practice due to their easy and low cost implementation and efficient fault coverage. MISR, in principle is not different from the single input signature analyzer, but instead of taking one serial input from the first flip-flop, every flip-flop in the MISR has one input coming from the primary outputs of the to be tested circuit as shown in figure III-7. Therefore, for an n output circuit, at least an n stage MISR is used. In figure III-7, the c_n switches define the divisor polynomial D(x) and the Q outputs show the internal stages. The residue is usually clocked out serially through one of the flip-flop outputs.

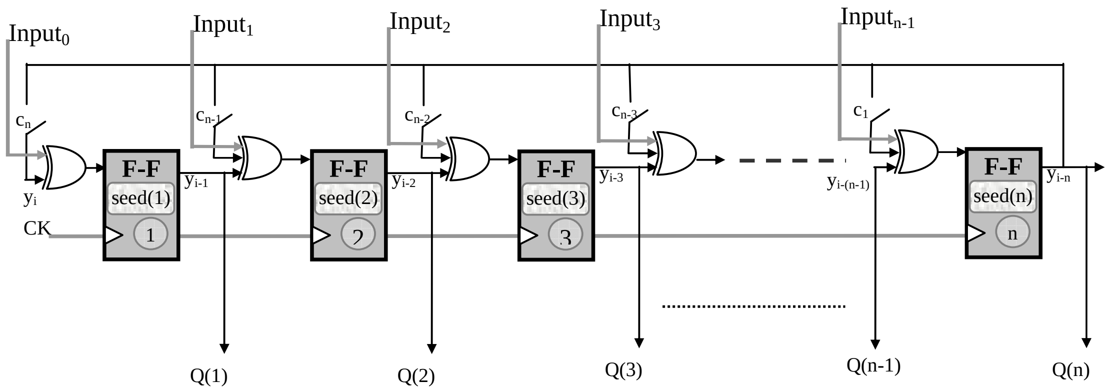


Figure III-7, Multiple Input Signature analyzer

[3] In polynomial division, the first received bit is on the left. Thus, the first bit in time has the highest order in the polynomial

Considering $I_i(x)$ the current set of primary outputs to be compressed, and current state of the MISR as $S_i(x)$, the next state output of the MISR can be represented as:

$$S_{i+1}(x) = [I_i(x) + x.S_i(x)] \bmod D(x) \tag{17}$$

If the initial state of the MISR is known, for instance to be $S_0(x)$, then, the k^{th} state can be computed as below, from repetitive application of above relation.

$$S_k(x) = [x^{k-1}I_0(x) + x^{k-2}I_1(x) + \ldots + x^1 I_{k-2}(x) + x^0 I_{k-1}(x)] \bmod D(x) \tag{18}$$

where, the $I_i(x)$ polynomials represent the I^{th} set of inputs to the signature analyzer.

Consequently, the expected signature can be analytically derived from the expected outputs of the circuit, prior to application of signature analysis. In terms of fault masking properties, MISR is scrutinized similarly as the single input signature analyzer, and again under the same assumptions – uniform distribution of error within bits, the fault masking probability is found to be ([14]):

$$P_{fm} = \frac{2^{L-1} - 1}{2^{n+L-1} - 1} \tag{19}$$

where,

L: length of the input test set
n: number of signature analyzer stages

A more comprehensive description of fault coverage stated in [1], where the length of signature analyzer, 'n', might be larger than the width of the test vectors, 'm' is:

$$P_{fm} = \frac{2^{mL-n} - 1}{2^{mL-1} - 1} \tag{20}$$

which reduces to $P_{fm} = \frac{1}{2^n}$ for large L. Hence, this expression is the same as the one found for single input signature analyzer in section II. Signature analyzers own some important error masking properties. These are not derived in this text due to space limitations, but the outcomes will be shortly stated. Further reference could be found in ([13] and [14]).

- <u>Single Cycle Errors:</u> All errors that occur within a single cycle are always detected.
- <u>Single Bit Errors:</u> As a direct consequence of the first statement, all single bit errors are detected
- <u>Single Output Errors:</u> If the error occurs at only one circuit primary output, the fault masking probability is the same as that of a single input signature analyzer.
- <u>Error Cancellation:</u> Different from the case of single input signature analyzers, error cancellation is a probable incident in MISRs. Error cancellation

occurs when an error occurs at one input of signature analyzer and while this error is propagated to the first feedback on the shift register path, another error on another input ‘collides’ with the propagated error and they are XORed together. As a result, in GF(2), the difference between the original and error-prone output drops back to null. One solution to this is, using characteristic polynomials with high number of coefficients – low hamming distance -, and feeding the error as soon as possible before it can be cancelled. However, as stated in [14], the probability of error cancellation in a MISR is $\underline{\underline{2^{1-n-L}}}$, which is quite low for large set of test vectors.

As a result, since error correction is very unlikely, the MISR has almost equivalent fault coverage as a single input signature analyzer, under the same condition stated in [1]’s derivation; large number of applied test vectors, L.

IV-DESIGN/SIMULATION TOOLS AND DATA MANAGEMENT

Although chronologically the importance of design data management and variety of used tools is emphasized after the end of design entry phase, it is more appropriate to describe design data and tool flow prior to any design phase in order to provide a better comprehension of the whole design process. As described in the introduction section, the primary goal of the project is the design of a parameterizable multiplier in VHDL and to investigate input pattern generation and output compression techniques using a fault simulator. However, the nature of fault simulation and the scarce compatibility of design entry tool and fault simulator require meticulous data management and design-test flow.

IV.1 – GENERIC DESIGN FLOW

A general description to Design → Fault Simulation flow can be stated as follows. The design entry is done either with a Hardware Description Language (HDL) or manual component placement and routing. For HDL entry, the design must then be synthesized regarding a vendor specific component library. After then, in both methods, the rawest description of the design data can be obtained in terms of a netlist. Which can then be applied to the fault simulator. With a plausible modification to our specific process, the overall design flow can be outlined as shown in figure IV-1.

In this project, we have chosen the track through HDL design entry. Nevertheless, both methodologies have their own pros and cons. With the HDL design entry, the process is quicker and has the advantage of automating several designs with parametric VHDL definitions, yet it lacks a desired amount of control on the multiplier structure. Though the hierarchy and multiplier strategy is well preserved with Register Transfer Level (RTL) level descriptions, the laying out process is still dependent upon the mercy of the synthesizer used. On the other hand, manual placing and routing benefits the advantage of complete control on the multiplier, yet this process is much slower, burdensome and errorprone.

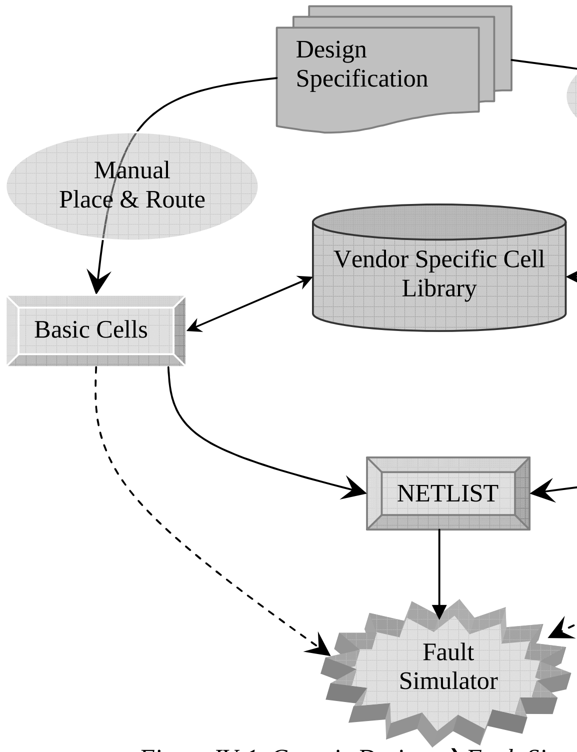


Figure IV-1, Generic Design → Fault Simulation Flow Chart

IV.2 – PROJECT TOOLS AND DATA MANAGEMENT

In this project, a key factor for progress has been data management, which includes managing VHDL design data, source codes, libraries, compilation for simulation and synthesis, source compile dependencies and documentation.

The used input design entry tool is Mentor Graphics' Renoir®, which generates pure HDL code from a semi-schematic design entry format. The VHDL level simulator used is Mentor Graphics' Modelsim®, which acquires the simulation data from Renoir via compilation for simulation. The output acquired from Renoir® is pure VHDL, and as stated in [20, p. 1-20], the used fault simulator, Mentor's QuickFaultII®, does not support VHDL as input data type. Therefore the acquired VHDL sources are compiled for synthesis using Mentor's QuickHDL® compiler[1] with synthesis tag and Mentor's AutologicII® is used to synthesize the design and save in Electronic Design Data Model[2] (EDDM) format, which can then be input to

[1] Autologic II has a synthesis compiler ALCOM, but QuickHDL compiler QVHCOM with –synth tag reveals much elucidating error messages
[2] In all the compilation and EDDM write steps, logical library management should be taken extensive care of in order to make each data format able to identify the physical mappings of its dependencies

QuickFaultII. EDDM format is the storage format used by many mentor graphics and a typical EDDM object is a schematic([21, p. 1-6]). Before applying the generated EDDMs to QuickFault, an intermediate step is to invoke Mentor's Design Architect®, in order to convert the hierarchical symbols generated by Autologic to meaningful symbols to improve design readability, i.e. to make an XOR look like an XOR and a full adder like a full adder. Hence, though this looks rather redundant at the early considerations, as design complexity increases, this turns out to be the only rational way to understand the generated schematic structure. Finally, the synthesized and modified EDDM designs are then opened in Quickfault and fault simulation results are documented from Quickfault. The overall design and tool flow can be summarized as shown in figure IV-2:

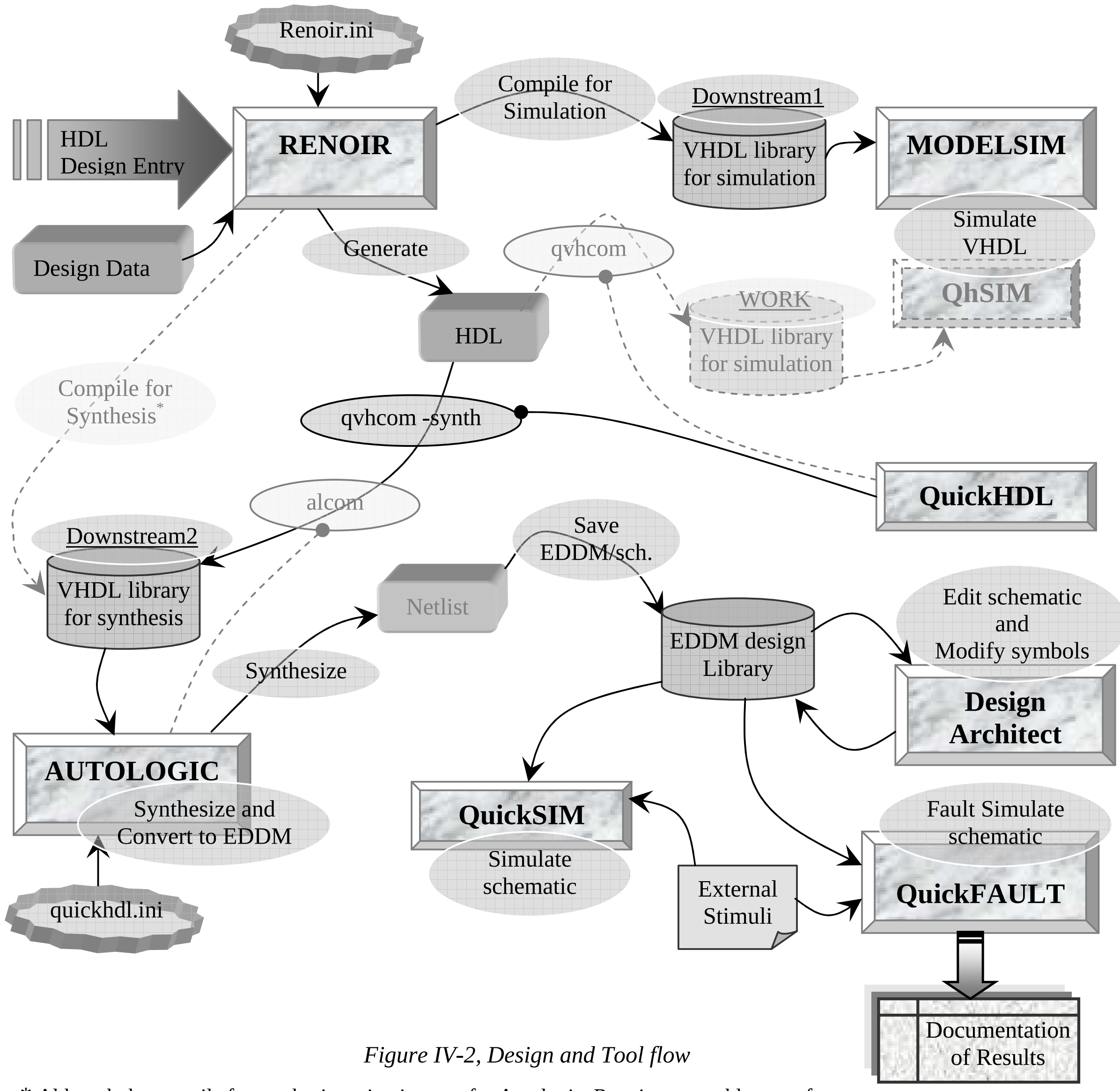
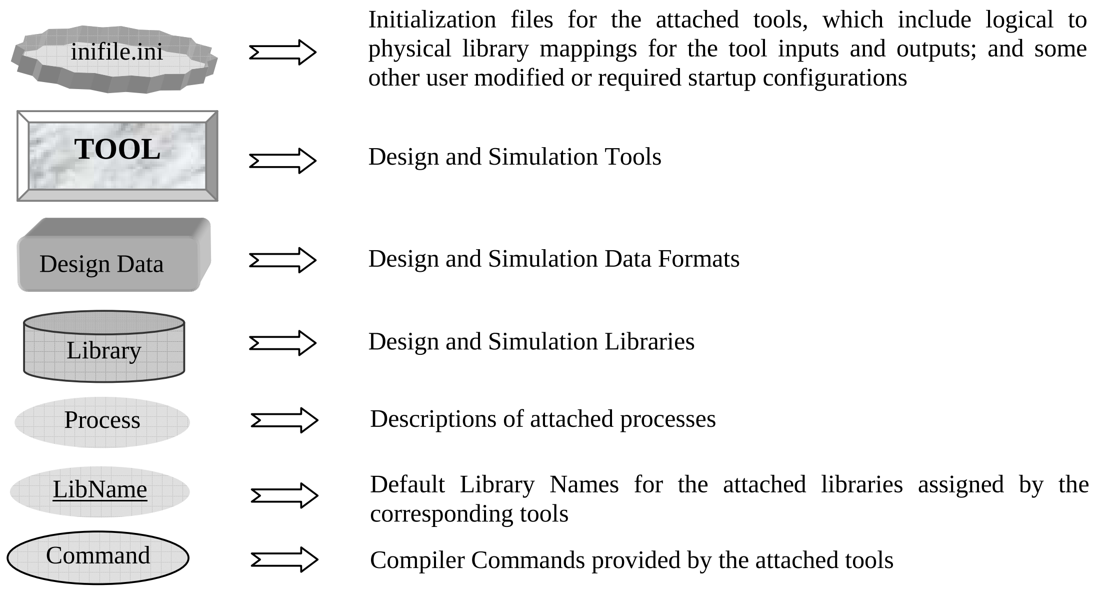


Figure IV-2, Design and Tool flow

* Although the compile for synthesis option is setup for Autologic, Renoir was unable to perform the synthesis, yet it could display the downstream synthesized by qvhcom -synth

Hence in figure IV-2, we used an almost compatible format with [21, pp. 2-1 – 2-2] and [20, pp.1-6 – 1-11], which comprise several other design pathways for synthesis and fault simulation. The design units described in figure IV-2 are defined as shown in figure IV-3:



inifile.ini	Initialization files for the attached tools, which include logical to physical library mappings for the tool inputs and outputs; and some other user modified or required startup configurations
TOOL	Design and Simulation Tools
Design Data	Design and Simulation Data Formats
Library	Design and Simulation Libraries
Process	Descriptions of attached processes
LibName	Default Library Names for the attached libraries assigned by the corresponding tools
Command	Compiler Commands provided by the attached tools

Figure IV-3, Key to Design and Tool flow units

The faded tools and processes, and dashed flowlines are not performed during the design flow, yet are included for the sake of completeness.

As a reference to included CD, the physical paths to the available design data and libraries are as tabulated in table 6:

RENOIR	
Design Data	CD>MSc/DesignFiles/Renoir/DesignData/
Generated HDL	CD>MSc/DesignFiles/Renoir/HDL/
Downstream1	CD>MSc/DesignFiles/Renoir/CompiledData/
Inifile	CD>MSc/DesignFiles/Renoir/CopyofRenoirINIfile
AUTOLOGIC	
VHDL lib. For synthesis	CD>MSc/DesignFiles/Renoir/SynthCompiledData/
EDDM design lib.	CD>MSc/DesignFiles/Renoir/Eddm_sch/
Inifile	CD>MSc/DesignFiles/Renoir/quickhdl.ini

Table 6, Physical Paths to Design Data

IV.2.1 – Auxiliary Project Tools:

Besides the design flow described in IV.2, there are a few other tools used for auxiliary purposes during the design progress. These tools are not directly related to design flow, but are rather utilized for simple tasks such as writing automatically generated data into files, converting file formats and automated comparison of resultant data.

These tools are listed in table 7 with their intended tasks.

Tools	**Function**
HPUX-XV	Capture pictorial data and edit colors for a printer friendly format
Matlab	Produce stimuli for simulation
HPUX shell scripts	Convert ASCII file formats and compare ASCII files

Table 7, Auxiliary tools and functions

The locations of written Matlab scripts are: "CD>MSc/DesignFiles/matlab/", and the generated output stimuli are referenced in several locations generally under "CD>MSc/results/". The XV outputs constitute most of the imported pictures in the report as well as the .gif files under "CD>MSc/results/". Unix scripts are referred usually from "CD>MSc/results/".

V- MULTIPLIER AND BIST CIRCUIT DESIGN

As described in the introduction, in this section we describe the design of a parameterized, signed, parallel multiplier, using VHDL via Renoir. We describe signed binary multiplication and briefly discuss various multiplier architectures and then expatiate upon the design of a parameterized CPA multiplier. Chronologically, the mentioned steps are the first in design progress. Then, we explain the design of the signature analyzer used in the design and finally the LFSR design, which are however, chronologically after the BIST investigation phase during design progress.

V.1 – MULTIPLIER ARCHITECTURE

In general, the multiplier function is divided into two major parts: 1) Bit product generation 2) Addition of bit products to form the final product. Different multiplier architectures emerge from how these two steps are performed. In unsigned multiplication, the second step is mere addition, while in signed multiplication, the second step includes an addition or subtraction in the final level of partial product accumulation depending on the most significant bit (MSB) of multiplier. In unsigned multiplication, final product for an NxN multiplication is 2N bits. However, for signed multiplication, 2N-1 bits are sufficient as long as both multiplier and multiplicand do not take their most negative value simultaneously. Two distinct properties of signed multiplication are: 1) Multiplier and multiplicand are not completely symmetric in hardware architecture 2) In order to keep track of sign, sign extension must be handled during partial product additions.

V.1.1 – Twos Complement Multiplication:

Twos complement multiplication is conceptually not different from mundane multiplication, in terms of binary domain, the only difference is, instead of the ‘-’ sign, the MSB holds the information about the number being positive of negative – therefore contrary to unsigned numbers, adding 0s to left might have a significance –. As in decimal domain, multiplicand (MD) is multiplied by the ‘value’ of each bit of multiplier (MR), and as in decimal, the values of higher significant bits of multiplier are assessed by a left shifting operation performed during partial product addition. As the value of MSB in multiplier, named Sign Bit (SB), can be either negative or zero in twos complement arithmetic, the final step is either a subtraction or 0 addition. In general twos complement multiplication can be demonstrated as in Figure V-1:

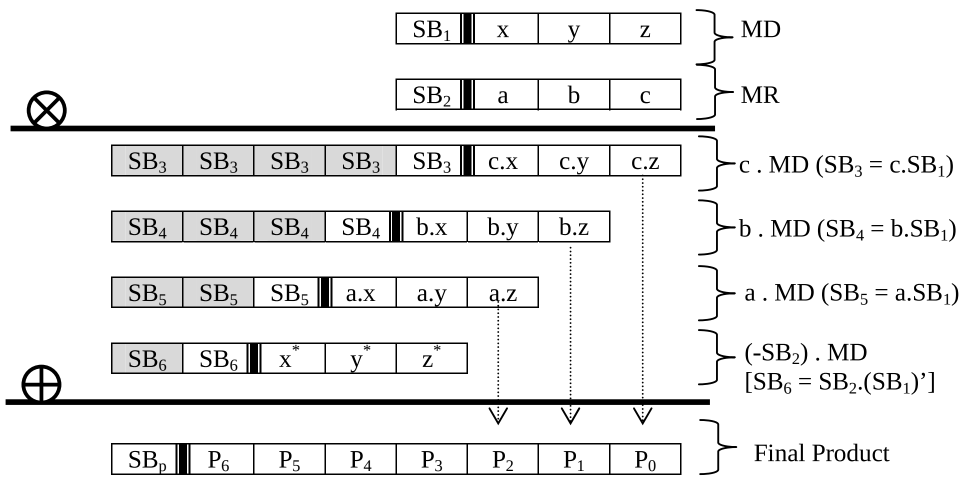


Figure V-1, twos complement multiplication

In the above demonstration, the shaded bits are extended signs for correct signed addition. (.)' denotes the complement of the binary value. As can be deduced, the last set of partial products is either all 0s if SB_2 is 0, or twos complement negation of multiplicand if SB_2 is 1; which is stated as "$(-SB_2)$. MD". Hence, in the above description, it is described such that, for an NxN multiplication, each partial product row must be sign extended to 2N bits, which is to emphasize the correct operation if all the rows are added simultaneously. However if we consider adding first 2 rows at once and then adding the result to the consecutive rows one by one with a cumulative fashion, the sign extension scheme can be adjusted as described in figure V-2. Hence, except for the first row, all rows now require only one bit sign extension, which in turn reduces the required number of bitwise additions by $\underline{(N-1)! - (N-1)}$.

As observed, unlike unsigned multiplication, the partial summations require N+1 bitwise additions for each accumulation step. In unsigned addition, the MSB of sum is gathered from the carry out output of the MSB adder however, in signed multiplication, the adder terms are sign extended and the MSB is the final adder's sum output, where the carry out is discarded.

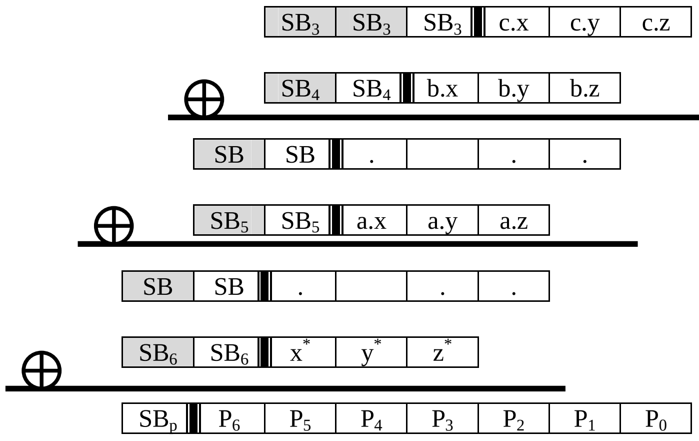


Figure V-2, sign extension after ordering summation

However, as described in [8], a slight modification in the MSB adder can alleviate this redundancy. Regarding the above scheme, signed addition of two N bit numbers can be demonstrated as in figure V-3:

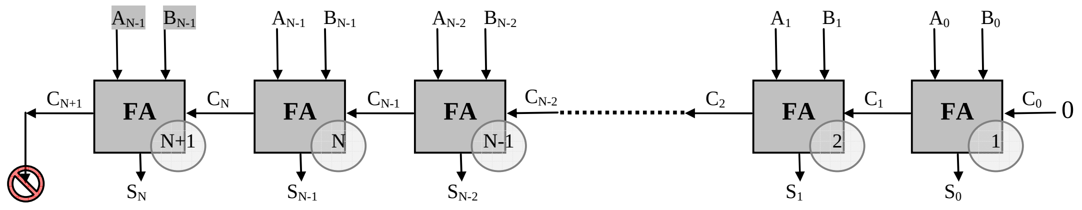


Figure V-3, sign extended addition of signed numbers

The shaded sign extensions for A and B inputs produce the MSB of the sum. When we consider the logical functions of sum and carry out circuits, the sum logic is true when either one or all 3 of the inputs are true and carry out logic is true when at least 2 of the inputs are true. These can be expressed in Boolean domain as:

$$\mathrm{Sum} = A(B)'(C_{in})' + (A)'B(C_{in})' + (A)'(B)'C_{in} + ABC_{in} = A \oplus B \oplus C_{in} \qquad (21)$$

And

$$\mathrm{Cout} = AB + AC_{in} + BC_{in} \qquad (22)$$

Now describing C_N in figure V-3, in terms of the above Cout function:

$$C_N = A_{N-1}B_{N-1} + A_{N-1}C_{N-1} + B_{N-1}C_{N-1} \tag{23}$$

and expressing S_N in terms of above Sum function:

$$S_N = A_{N-1} \oplus B_{N-1} \oplus C_N \tag{24}$$

We then substitute the C_N expression in S_N expression, which reveals:

$$S_N = A_{N-1} \oplus B_{N-1} \oplus (A_{N-1}B_{N-1} + A_{N-1}C_{N-1} + B_{N-1}C_{N-1})$$

and upon simplification of this expression we reach:

$$\boxed{S_N = A_{N-1}B_{N-1} + A_{N-1}(C_{N-1})' + B_{N-1}(C_{N-1})'} \tag{25}$$

Hence, this expression is almost the same as the carry out function for Nth adder, with just the carry in inverted. The revelation of this expression is, if we modify the carry out of the Nth adder as above, we might then use that carry out as the S_N bit, without the expenditure of the N+1th adder. Calling this adder with modified carry out, MSB Full Adder (FA), the adder circuit described in figure V-4 then turns out to be almost at equivalent cost to an unsigned adder circuit.

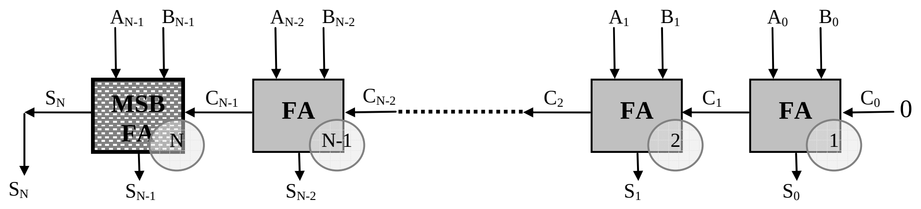


Figure V-4, Modified signed adder

With this modification, each partial sum in multiplication is reduced to N bitwise additions rather than N+1 case described in figure V-2. However, we should be aware that, sign extension requirement is not fully overcome with this method, as for two unequal length operands, the shorter one must still be sign extended up to the length of the longer one.

V.1.2 – Multiplier Architectures:

As discussed at the beginning of the section, different multiplier structures emerge from how the bit product generation and partial summations are performed. Each of the to be described structures have their own advantages and shortcomings compromising speed, power, hardware cost and regularity in structure. Although in general signed multiplication is described as a sign extended unsigned multiplication in most of the cited references, some of these can be modified to the above described scheme.

Of the most well known architecture, CPA multiplier ripples the carries through each row of additions and thus, it actually performs exactly the operation described in figure V-2. the general hardware structure of a CPA multiplier is a matrix of adders with each row rippling the carries to the MSB, which is the modified MSB FA. The structure of an unsigned CPA multiplier is described in [3, p.938], [7, pp. 95 – 98] and [9, pp. 2 – 6]. Structures for signed CPA multiplier are described in [9, p.16], using a regular sign extension scheme and in [8, pp. 39 – 40], which utilizes the modified MSB FA.

CSA multiplier, which has the exact same hardware construct with CPA multiplier for unsigned multiplication, makes use of the fact that, the internal summation results are not required for the output, and instead of rippling the carries through each row, it transfers individual carries to the following row. In carry save structures, a final stage for adding the sums and carries is required, which is named, vector merging addition. CSA multipliers whose final stage is performed with carry ripple addition are named “Braun Array Multipliers”([7]). The architectures for unsigned CSA multipliers are described in [3, p. 938], [6, pp. 344 – 348], [7, pp. 99 – 101] and [9, pp. 10 –11]. One significant advantage of CSA multiplier is, the critical path is single, contrary to 2 critical paths in CPA multiplier ([9, pp. 9 – 10]. Moreover, in the general structure, CPA multiplier requires 3(N-1) adder delays – see MVSD1 ILP, Q.3 for derivation –, [6, p.344] asserts CSA multiplier requires 2N+1 adder delays, for all the outputs to stabilize. Signed CSA architectures are described in [9, p.17], which uses the regular sign extension scheme and in [4], which also uses sign extension and a modified adder array without a final vector merging adder.

Tree or Wallace tree multipliers are a further modification of CSA multipliers, which compromise a regular structure for performance improvements. In Wallace tree multipliers, all partial products are first computed in parallel and then they are added in an adder tree in concordance with the carry save addition principles. A final vector merging adder is again required to combine the final carry and sum vectors. Unsigned tree structures are described in [3, p.939], [7, pp.103 – 105] and [10, pp. 160 – 165]. A signed Wallace tree architecture is described in [22, pp. 77 – 80], which applies a specific algorithm for twos complement multiplication.

Modified Booth's Array Multiplier, which is the most commonly used multiplier in datapath structures as all but most synthesizers generate Modified Booth's Multipliers during synthesis ([2]), recodes the multiplier in 2 bits at a time. Thus, unlike to the previously described multipliers, Booth multiplier modifies the partial product generation structure and reduces the number of generated partial products. The algorithm takes 2 bits of the multiplier at a time and adds 0/±1/±2 . multiplicand, according to the recoding, which also considers the previous bit of multiplier. Recoding is done for 2 bits only in practice, as implementation of negation – invert and inject hot one – and multiplication by two – left shift – is straightforward. The algorithm for 2 bit recoding is demonstrated in table 8.

Current bit pair			Previous bit		Overall Action
bit_{i+1}	bit_i	Action1	bit_{i-1}	Action2	
0	0	Add 0	0	Add 0	Add Nothing
0	1	Add 1	0	Add 0	Add 1x MD
1	0	Add –2	0	Add 0	Add –2x MD
1	1	Add –1	0	Add 0	Add –1x MD
0	0	Add 0	1	Add 1	Add 1x MD
0	1	Add 1	1	Add 1	Add 2x MD
1	0	Add –2	1	Add 1	Add –1x MD
1	1	Add –1	1	Add 1	Add Nothing

Table 8,Modified Booth Algorithm

Modified Booth Multipliers are described extensively in [2], [5] and [8, pp. 29 – 36 & pp. 41 –43].

As the initial design, we have chosen a CPA multiplier due to its regularity and ease of extension for twos complement arithmetic. In the rest of the report, multiplier refers to CPA multiplier unless otherwise stated.

V.2 – INITIATION PART-I: DESIGN OF 3X3 CPA MULTIPLIER

In order to present the experimental route of design, we first design a simple 3x3 CPA multiplier and measure its fault coverage. In this section, we describe the design of the 3x3 multiplier, while in section VI, in 'Initiation Part-II', we'll describe the fault coverage measurement for the multiplier.

The regular structure of a 3x3 signed multiplier, regarding the cited references is as shown in figure V-5. The multiplication operation is: $B_2\ B_1\ B_0 \times A_2\ A_1\ A_0 = P_5\ P_4\ P_3\ P_2\ P_1\ P_0$. As discussed previously, the final stage of multiplication is either a subtraction – if A_2 is 1 – or 0 addition – if A_2 is 0 –, which equals $A_2 . (B(2{:}0))'$ and hot 1 as carry in to the last stage if A_2 is 1. Therefore, the final stage is implemented as addition of $A_2 . (B(2{:}0))'$ and A_2 itself as the hot 1.

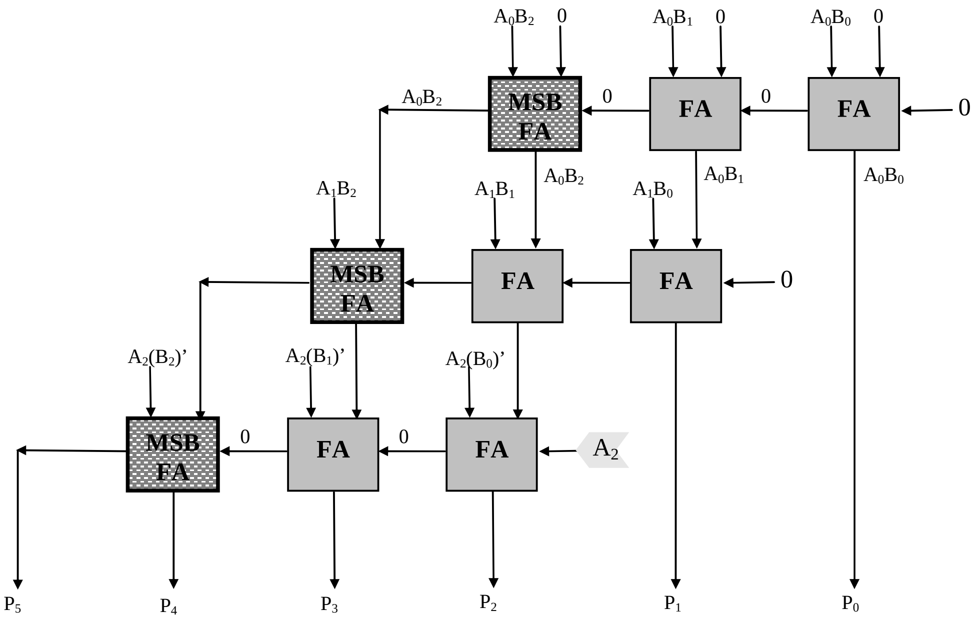


Figure V-5, Regular 3x3 CPA Multiplier Structure

In figure V-5, the adder inputs are the produced bit products by the AND gate matrix. As seen the first column propagates all 0s and the A_0B_i terms are only propagated to the next row. Nonetheless, the MSB FA's modified carry out is nonzero, as the modified carry out function states:

$$S_N = A_{N-1}B_{N-1} + A_{N-1}(C_{N-1})' + B_{N-1}(C_{N-1})' \Rightarrow$$

$$S_3 = (A_0B_2).0 + (A_0B_2).(0)' + 0.(0)' \Rightarrow$$

$$\boxed{S_3 = A_0B_2}$$

As seen in the above structure, the first set of adders is redundant and therefore, a typical structure as demonstrated in figure V-6 is used in almost all designs in preference to the above structure, which removes all the redundant blocks and reduces the FA logic to Half Adder (HA) logic for adders with one of the inputs constantly wired false.

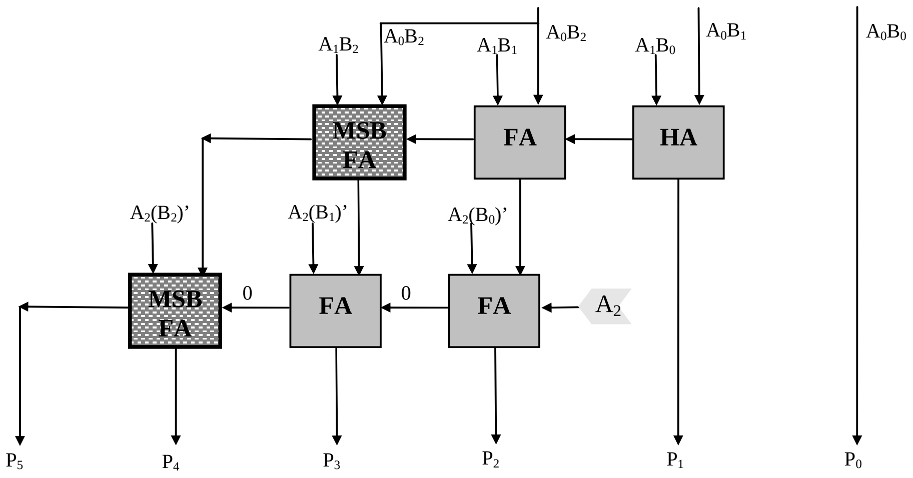


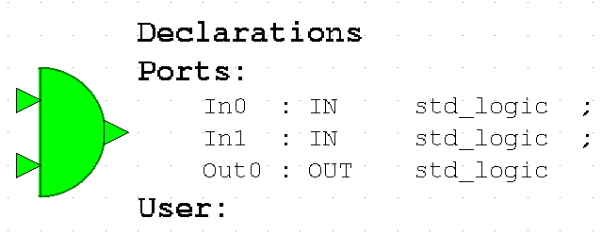
Figure V-6, Used 3x3 CPA Multiplier Structure

Hence, this structure performs precisely the accumulation scheme described in figure V-2, with the final sign extended additions taken care of by the MSB FA.

In order to build the above architecture, we built the lowest hierarchy blocks – leaf cells (the pit product generators and adder blocks), using dataflow (RTL[1]) level VHDL in Renoir. The blocks that are built as the leaf cells are described in V.2.1 to V.2.5. Finally, in V.2.6, designed 3x3 CPA multiplier is demonstrated.

V.2.1 – HHandH:

Firstly for bit product generation, the required AND gates are generated. HHandH represents that both inputs to the gate are active high and output is active high. The symbol for HHandH is as shown in figure V-7. The generated VHDL code is attached in Appendix A-1.



```
Declarations
Ports:
    In0  : IN    std_logic  ;
    In1  : IN    std_logic  ;
    Out0 : OUT   std_logic
User:
```

Figure V-7, HHandH symbol

[1] RTL level VHDL is used in almost all design codes in order to have direct control on synthesis process

HHandH is used to generate all bit products except for the last row, as the B inputs to the last row are inverted.

VHDL level simulation (Modelsim Simulation) results of HHandH are as shown in figure V-8. The signal traces display correct operation of HHandH.

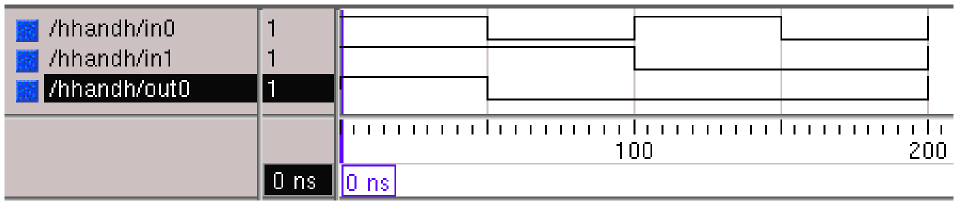


Figure V-8, HHandH simulation result

For each of the described components in the report, the Renoir symbols and schematic structural VHDL representations are in:
"CD>MSc/DesignFiles/Renoir/DesignData/*componentname*". If component names are different from the subsection header, they will be written in parentheses in the title of subsection.

The generated VHDL codes are in:
"CD>MSc/DesignFiles/Renoir/HDL/*componentname_vhdlarchitecturename*".

The compiled files for VHDL simulation with Modelsim are in:
"CD>MSc/DesignFiles/Renoir/CompiledData/*componentname/*"

The Modelsim simulation results are in:
"CD>MSc/DesignFiles/Renoir/ModelSimResults/*componentname*"

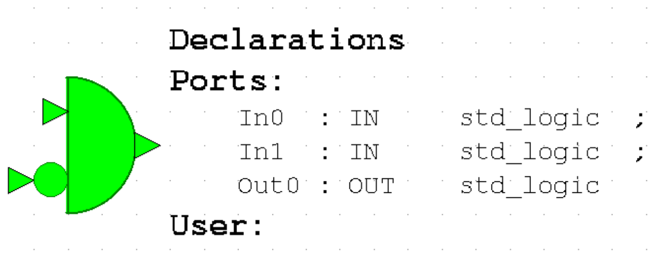
Included Renoir figures, either captured from Renoir graphical interface – in case of symbols – or printed as postscript files – in the case of structural design blocks – can also be referenced from:

"CD>MSc/DesignFiles/report/renoirfigs/*componentnamesym.gif*" for printed symbols
and
"CD>MSc/DesignFiles/report/renoirfigs/*componentnamestruct.ps*" for attached structural designs.

For the rest of the report, all the described components and higher level blocks have the above correspondence with the attached CD, therefore they will not be referred individually for the sake of brevity.

V.2.2 – HLandH:

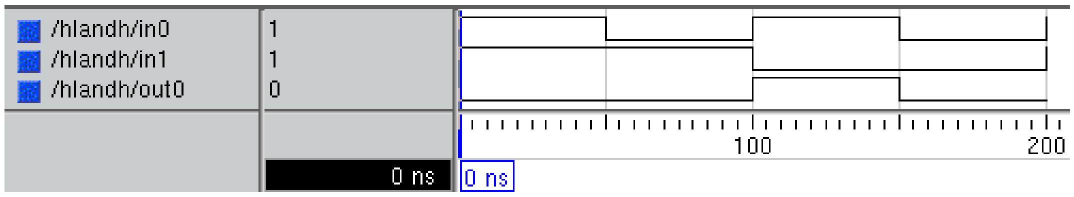
As the last row of partial summations requires inverted B inputs, a separate semi-inverted AND gate is generated. HLandH represents that input in0 of the AND gate is active high while input in1 is active low and output is active high. The symbol for HLandH is as shown in figure V-9. The generated VHDL code is attached in Appendix A-2.



```
Declarations
Ports:
    In0  : IN   std_logic  ;
    In1  : IN   std_logic  ;
    Out0 : OUT  std_logic
User:
```

Figure V-9, HLandH symbol

VHDL level simulation results of HLandH are as shown in figure V-10. The signal traces display correct operation of HLandH.

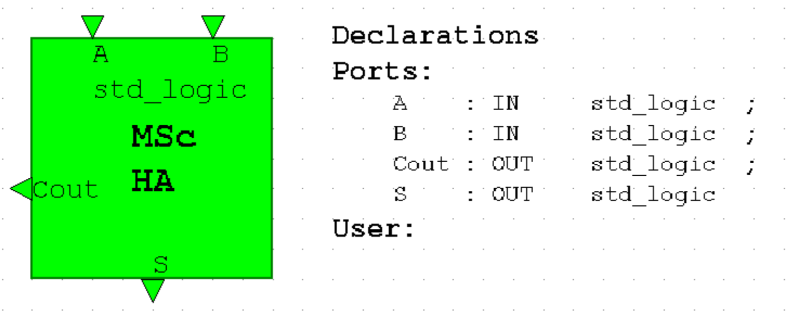


```
/hlandh/in0    1
/hlandh/in1    1
/hlandh/out0   0
               0 ns
```

Figure V-10, HLandH simulation result

V.2.3 – Half Adder (HA):

The symbol for HA is as shown in figure V-11. The generated VHDL code is attached in Appendix A-3.



```
Declarations
Ports:
    A     : IN     std_logic  ;
    B     : IN     std_logic  ;
    Cout  : OUT    std_logic  ;
    S     : OUT    std_logic
User:
```

Figure V-11, HA symbol

VHDL level simulation results of HA are tabulated in table 9, which is acquired from the list window of Modelsim.

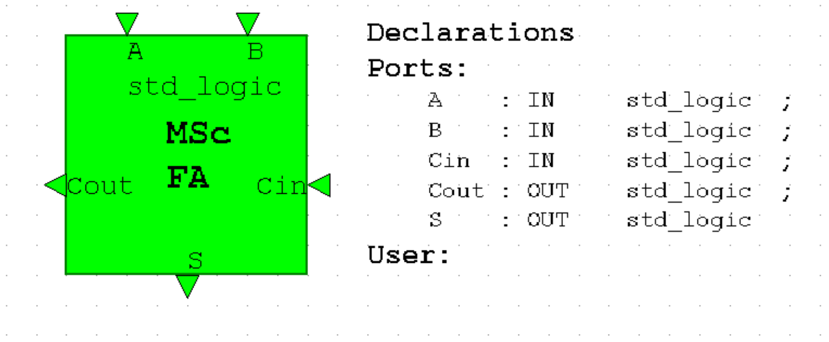
ns	a	b	cout	s
0	1	1	1	0
50	0	1	0	1
100	1	0	0	1
150	0	0	0	0
200	1	1	1	0

Table 9,HA Simulation List

As seen in the listed simulation results, HA functions in accordance with the specification.

V.2.4 – Full Adder (FA):

The symbol for FA is as shown in figure V-12. The generated VHDL code is attached in Appendix A-4.



```
Declarations
Ports:
    A     : IN     std_logic  ;
    B     : IN     std_logic  ;
    Cin   : IN     std_logic  ;
    Cout  : OUT    std_logic  ;
    S     : OUT    std_logic
User:
```

Figure V-12, FA symbol

VHDL level simulation results of FA are tabulated in table 10:

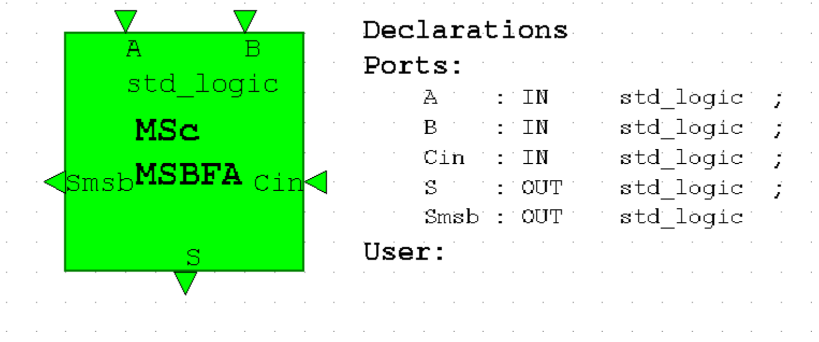
ns	a	b	cin	cout	s
0	0	0	1	0	1
50	0	0	0	0	0
100	0	1	1	1	0
150	0	1	0	0	1
200	1	0	1	1	0
250	1	0	0	0	1
300	1	1	1	1	1
350	1	1	0	1	0
400	0	0	1	0	1

Table 10,FA Simulation List

As seen in the listed simulation results, FA functions as desired.

V.2.5 – Modified Full Adder for Signed Addition (MSBFA):

The symbol for MSBFA is as shown in figure V-13. The generated VHDL code is attached in Appendix A-5.



```
Declarations
Ports:
    A    : IN   std_logic ;
    B    : IN   std_logic ;
    Cin  : IN   std_logic ;
    S    : OUT  std_logic ;
    Smsb : OUT  std_logic
User:
```

Figure V-13, MSBFA symbol

VHDL level simulation results of MSBFA are tabulated in table 11:

ns	a	b	cin	s	smsb
0	0	0	0	0	0
50	0	0	1	1	0
100	0	1	0	1	1
150	0	1	1	0	0
200	1	0	0	1	1
250	1	0	1	0	0
300	1	1	0	0	1
350	1	1	1	1	1
400	0	0	0	0	0

Table 11,MSBFA Simulation List

As seen in the listed simulation results, MSBFA functions as expected.

V.2.6 – Signed Parallel 3x3 CPA Multiplier (CPAmult3x3):

With all the required leaf cells designed and assured for correct functionality, the 3x3 CPA multiplier is designed structurally in Renoir. The symbol associated with the 3x3 multiplier is as shown in figure V-14. The structural design, which is done in a semi-schematic manner in Renoir is attached in Appendix B-1 and the generated VHDL code is in Appendix A-6.

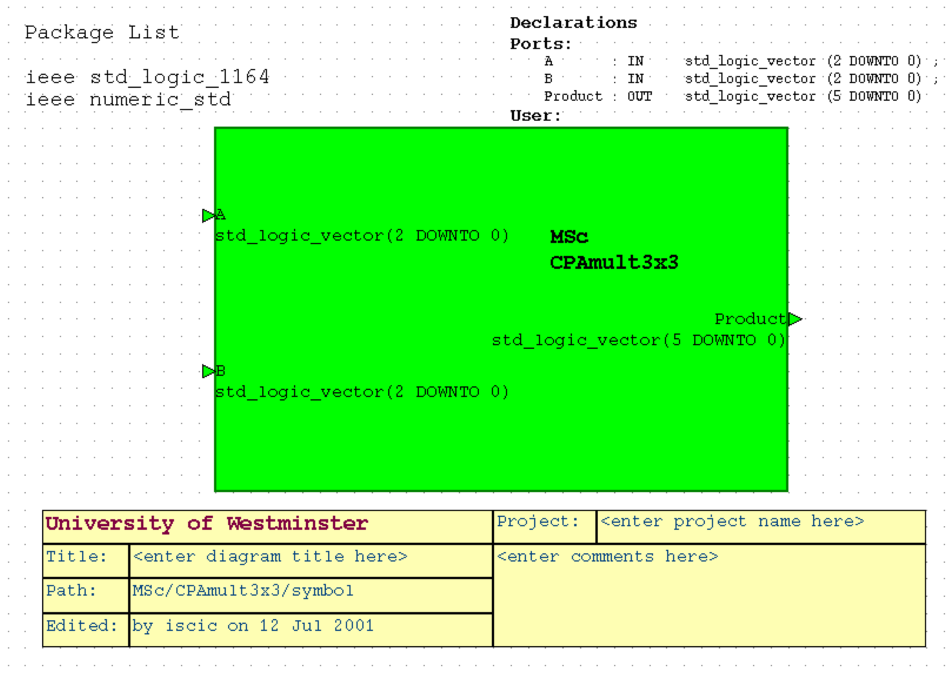

Package List
ieee std_logic_1164
ieee numeric_std
Declarations
Ports:
A : IN std_logic_vector (2 DOWNTO 0) ;
B : IN std_logic_vector (2 DOWNTO 0) ;
Product : OUT std_logic_vector (5 DOWNTO 0)
User:
A
std_logic_vector(2 DOWNTO 0)
MSc
CPAmult3x3
Product
std_logic_vector(5 DOWNTO 0)
B
std_logic_vector(2 DOWNTO 0)
University of Westminster
Project: <enter project name here>
Title: <enter diagram title here>
<enter comments here>
Path: MSc/CPAmult3x3/symbol
Edited: by iscic on 12 Jul 2001

Figure V-14, CPAmult3x3 symbol

As seen in Appendix B-1, the designed multiplier is exactly the same structure described in figure V-6, with the bit product generation also included. VHDL level simulation results of CPAmult3x3 are tabulated in Appendix C-1, where the A, B inputs and P output are displayed in signed decimal format to provide easy examination of multiplier operation. As can be seen in the appendix, the correct functionality of 3x3 multiplier is verified in VHDL level simulation.

V.3 –DESIGN OF PARAMETERIZED CPA MULTIPLIER (CPAMULTNXN)

Having ascertained the correct functionality of the signed parallel multiplier, we design the parameterized signed, parallel CPA multiplier. The generic structure of an NxN CPA multiplier, with a matrix shape rather than the demonstrated ladder like structure is shown in figure V-15.

Regarding the structure below, several for and if frames are used in Renoir in order to achieve the generic structure shown in Appendix B-2. For the bit products, an NxN matrix, ‘bpmatrix’ is defined, which holds the generated bit products by the matrix of AND gates. The generated matrix has the form:

$$\begin{array}{cc}
\left[\begin{array}{c|cc|c}
A_{n-1}\overline{B}_{n-1} & A_{n-1}\overline{B}_{n-2} & \cdot\ \ \cdot & A_{n-1}\overline{B}_{0} \\
\hline
A_{n-2}B_{n-1} & \cdot & \cdot\ \ \cdot & \cdot \\
\cdot & \cdot & \cdot\ \ \cdot & \cdot \\
\cdot & \cdot & \cdot\ \ \cdot & A_{1}B_{0} \\
\hline
A_{0}B_{n-1} & A_{0}B_{n-2} & \cdot\ \ \cdot & A_{0}B_{0}
\end{array}\right]_{NxN} &
\begin{array}{l}
\longleftarrow \text{Row N-1} \\
\Big\downarrow \text{N-1 downto 1} \\
\longleftarrow \text{Row 1}
\end{array} \\
\begin{array}{ccc}
\uparrow & & \uparrow \\
\text{Column N-1} & \xrightarrow{\text{N-1 downto 0}} & \text{Column 0}
\end{array} &
\end{array}$$

Hence, the described rows and columns refer to the rows and columns of generated matrix of HHandH and HLandH gates in the CPAmultNxN structural design.

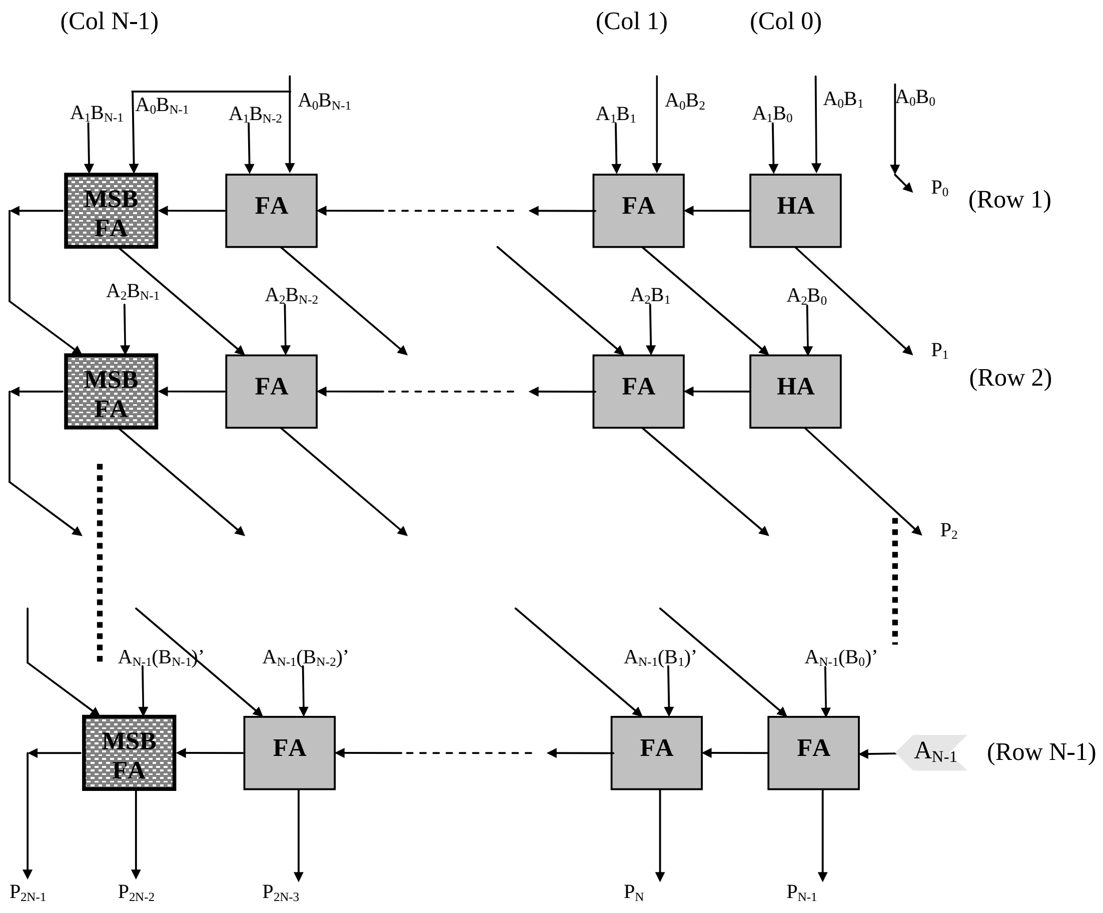


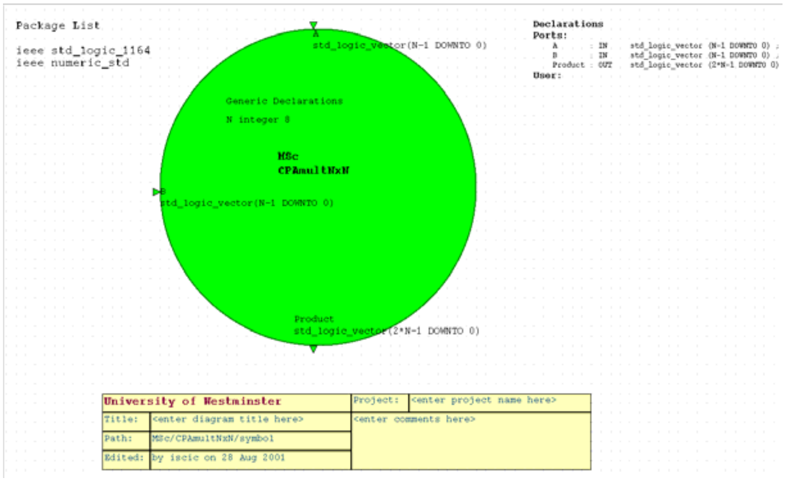
Figure V-15, NxN CPA Multiplier Structure

As can be observed in figure V-15, the 1st and last rows are different from the intermediate rows, as 1st one has both inputs from bit products, and last one having FA in the LSB. Therefore, during generation, these two rows are isolated with 2 if frames in Renoir. The carries of the 1st row are propagated by ‘carryripple1’, an array of (N-1:0), with carripple1(i) corresponding the ripple between columns i and i-1. Therefore, carry in to the ith column adder in the first row is carryripple1(i). The partial sums being transferred to the next rows are stored in a 2D array – not a matrix – named ‘sums’. The format of sums is

$$\begin{array}{cc}
\begin{bmatrix} [Partialsums\_from\_row\_1] \\ [.....] \\ [.....] \\ [Partialsums\_from\_row\_N-2] \end{bmatrix} & \begin{array}{l} \leftarrow \text{Row 1} \\ \quad \big\downarrow \text{1to N-2} \\ \\ \leftarrow \text{Row N-2} \end{array} \\
\uparrow \text{Column N-1} \xrightarrow{\text{N-1 downto 0}} \uparrow \text{Column 0} &
\end{array}$$

The partials sums are represented as an array of (N-1:0)elements representing the outputs of each partial sum row. i.e. sums(i)(j) represents the output from ith partial sum row, and this partial sum's jth bit. The carries inside the intermediate generated rows are in a matrix of (2:N-2) rows, representing the rows of multiplier and (N-1:1) columns, representing the individual carries within a row. Similar to carryripple1, the 'row'th row and ith column of this matrix, 'carryripples', represents the carry in to ith column adder in row 'row'. Finally, for the last row, the carries are stored in an array named 'carryripple2' functioning the same way as carryripple1 and the outputs are directly added to the MSBs of the product bus.

The corresponding symbol in Renoir, for the generated CPAmultNxN structural design is as shown in figure V-16 and the generated VHDL code from the structural design is in Appendix A-7.


A
std_logic_vector(N-1 DOWNTO 0)
Generic Declarations
N integer 8
MSc
CPAmultNxN
B
std_logic_vector(N-1 DOWNTO 0)
Product
std_logic_vector(2*N-1 DOWNTO 0)

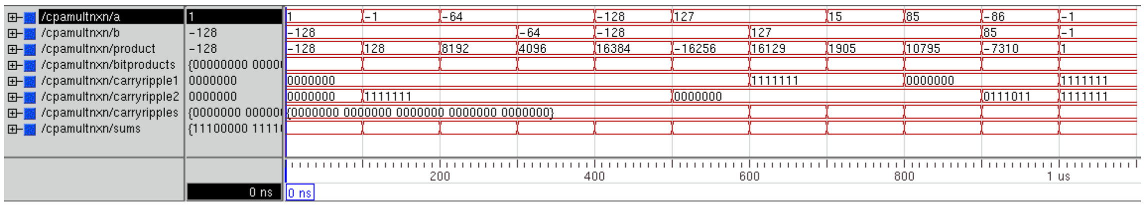
```
Package List

ieee std_logic_1164
ieee numeric_std
```

```
Declarations
Ports:
    A       : IN     std_logic_vector (N-1 DOWNTO 0) ;
    B       : IN     std_logic_vector (N-1 DOWNTO 0) ;
    Product : OUT    std_logic_vector (2*N-1 DOWNTO 0)
User:
```

University of Westminster		Project:	<enter project name here>
Title:	<enter diagram title here>	<enter comments here>	
Path:	MSc/CPAmultNxN/symbol		
Edited:	by iscic on 28 Aug 2001		

Figure V-16, CPAmultNxN symbol

The VHDL level simulation results for the parameterized CPA multiplier reveals correct operation of the design for various tried multiplier widths. An exemplary simulation result for an 8x8 multiplier (N=8) is as shown in figure V-17.



```
/cpamultnxn/a             1
/cpamultnxn/b             -128
/cpamultnxn/product       -128
/cpamultnxn/bitproducts   {00000000 0000(
/cpamultnxn/carryripple1  0000000
/cpamultnxn/carryripple2  0000000
/cpamultnxn/carryripples  {0000000 00000(
/cpamultnxn/sums          {11100000 1111(
                          0 ns
```

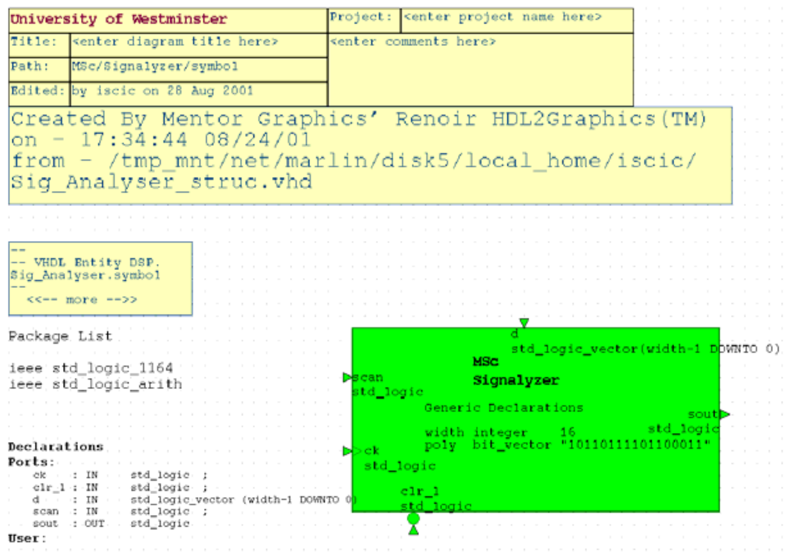
Figure V-17, Exemplary Simulation Trace for CPAmultNxN

After the completion of the design of the parameterized multiplier, the first step of the design entry has actually been finished and the next step in project progress is investigation of input pattern generation and output compression techniques. In terms of chronology, after the confirmed design of CPAmultNxN, we moved to reading the Renoir designs in QuickFault for fault simulation and the described design data management notions in section IV are scrutinized at this point. As described in section IV, the design pathway has been from Renoir → (QuickHDL) → Autologic → Design Architect → QuickFault and the operations done after these intermediate steps are the discussion of section VI. Therefore, chronologically, section VI is the next step performed after this point and it is recommended that the following subsections within this section are referred back after the examination of section VI.

V.4 –DESIGN OF PARAMETERIZED SIGNATURE ANALYZER (SIGNALYZER)

As described in section VI, the output compression technique chosen is signature analysis. Although the first signature analyzer is defined for 16 bits for an at most 255 vector input, here we introduce the design of a parameterized signature analyzer which can be configured for required number of bits and characteristic polynomials.

The associated symbol for the signature analyzer is as shown in figure V-18 and the VHDL code is attached in Appendix A-8.



University of Westminster		Project:	<enter project name here>
Title:	<enter diagram title here>	<enter comments here>	
Path:	MSc/Signalyzer/symbol		
Edited:	by iscic on 28 Aug 2001		

```
Created By Mentor Graphics' Renoir HDL2Graphics(TM)
on - 17:34:44 08/24/01
from - /tmp_mnt/net/marlin/disk5/local_home/iscic/
Sig_Analyser_struc.vhd
```

```
--
-- VHDL Entity DSP.
Sig_Analyser.symbol
--
  <<-- more -->>
```

```
Package List

ieee std_logic_1164
ieee std_logic_arith
```

```
Declarations
Ports:
    ck    : IN    std_logic  ;
    clr_l : IN    std_logic  ;
    d     : IN    std_logic_vector (width-1 DOWNTO 0)
    scan  : IN    std_logic  ;
    sout  : OUT   std_logic
User:
```

```
Generic Declarations

width integer    16
poly  bit_vector "1011011110110011"
```

Figure V-18, Signalyzer symbol

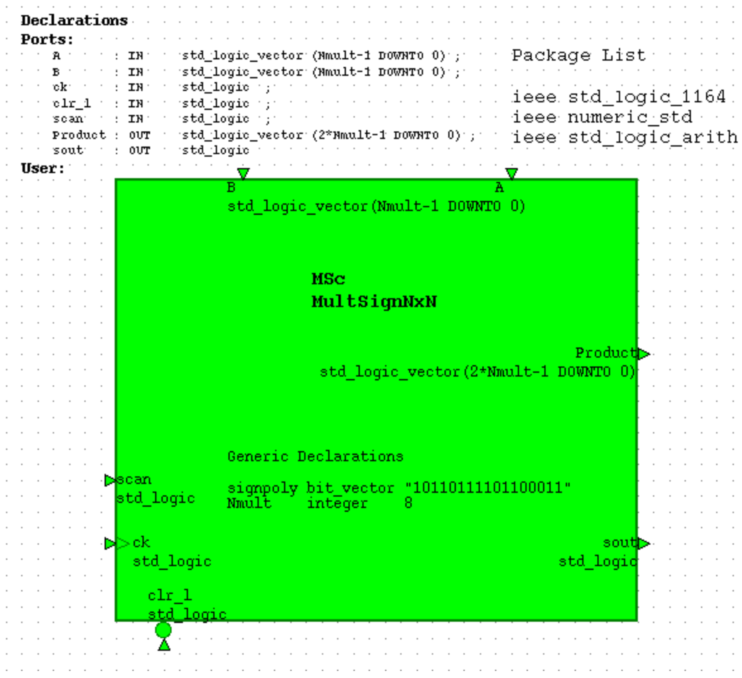
The width generic defines the width of the desired signature analyzer and the poly represents the characteristic polynomial for the signature analyzer, which defines the tap locations.

An informative VHDL level simulation for the signature analyzer is shown in Appendix C-2, which uses the characteristic polynomial defined for the specific 8x8 multiplier. Therefore, the signature analyzer is run for 256 clock cycles and is seen to have a period of 255 as expected due to the polynomial used. The simulation is done so that, D inputs to signature analyzer are kept at all 0s to prevent any interference from input and to make the signature analyzer work like an LFSR, except for the first cycle where the LSB is kept 1 to initiate the feedback after clear. As seen in the simulation list, the signature analyzer works exactly as expected in VHDL level.

V.5 –DESIGN OF 8X8 MULTIPLIER + SIGNATURE ANALYZER (MULTSIGN8X8)

After the signature analyzer is assured for correct operation, multiplier and signature analyzer are connected in a higher hierarchy to initiate the design of final circuit with BIST circuitry also included. The Renoir symbol for the designed system is as shown in figure V-19. The Renoir structural design for the MultSign8x8 is attached in Appendix B-3 and the generated VHDL code for the multiplier + signature analyzer circuit is attached in Appendix A-9.

The generics ‘signpoly’ and ‘Nmult’ describe the characteristic polynomial of the signature analyzer and width of the NxN multiplier respectively. As seen in the structural design in Appendix B-3, the product bits are connected to signature analyzer ‘d’ input and the serial signature analyzer residue is clocked out via ‘sout’.



```
Declarations
Ports:
    A       : IN    std_logic_vector (Nmult-1 DOWNTO 0) ;
    B       : IN    std_logic_vector (Nmult-1 DOWNTO 0) ;
    ck      : IN    std_logic  ;
    clr_l   : IN    std_logic  ;
    scan    : IN    std_logic  ;
    Product : OUT   std_logic_vector (2*Nmult-1 DOWNTO 0) ;
    sout    : OUT   std_logic
User:

Package List

ieee std_logic_1164
ieee numeric_std
ieee std_logic_arith

Generic Declarations

signpoly bit_vector "10110111101100011"
Nmult    integer    8
```

Figure V-19, MultSign8x8 symbol

A demonstrative VHDL level simulation of MultSign8x8 is as tabulated in table 12. As seen, scan input, which disables the parallel ‘d’ inputs to the MISR is always kept false, as this feature is not used in general in the way signature analysis is done during the project. As Modelsim permits different types of signal radix in the same list, in most Modelsim lists and traces, the ‘a’, ’b’ and ‘product’ buses are displayed in signed decimal format for quick analysis of signal values’ concordance with the expected multiplier behavior. Yet, on the other hand, most of the LFSR type output signals are displayed in hex format and ‘a’ and ‘b’ might be therefore displayed also in hex for PRBS input pattern generation analysis cases.

As can be deduced from the simulation list, the multiplier + signature analyzer circuit works in the expected way, where ‘sout’ will be the primary output observed by the tester during testing.

ns	Scan	clr_l	Ck	b	a	product	sout
0	0	0	1	-1	1	-1	0
50	0	1	0	-1	1	-1	0
100	0	1	1	-1	1	-1	1
150	0	1	0	-1	0	0	1
200	0	1	1	-1	0	0	0
250	0	1	0	-1	-128	128	0
300	0	1	1	-1	-128	128	0
350	0	1	0	127	-128	-16256	0
400	0	1	1	127	-128	-16256	0
450	0	1	0	127	127	16129	0
500	0	1	1	127	127	16129	0
550	0	1	0	-128	-128	16384	0
600	0	1	1	-128	-128	16384	0
650	0	1	0	-86	85	-7310	0
700	0	1	1	-86	85	-7310	1
750	0	1	0	-86	85	-7310	1

Table 12,MultSign8x8 Simulation List

V.6 –DESIGN OF PARAMETERIZED LFSR (LFSR)

In order to finalize the BIST circuitry, the LFSR is designed as the last hierarchical block in the design process. After investigation of several BIST input pattern generation techniques in section VI, the technique decided upon is PRBS generation using an LFSR with seed = x7B (0111 1011), where also repetitive patterns are used to have a constant volume of test vectors with an insignificant downgrade on fault detection efficiency. As described in section VI, a repetition length 4 sequence is used, meaning 2^4 = 16 different inputs repeated for the length of each ‘a’ and ‘b’ multiplier inputs. In order to apply all possible combinations these two 16 repeated patterns, 16.16 = 256 vectors are needed, yet with the LFSR, 255 of these 256 patterns are provided, with all 0s, which is already observed to have zero effect after the 255 vectors, unprovided.

To provide a more flexible overall system in VHDL description, the designed LFSR is constructed as a parameterized LFSR with the tap locations, seed and length explicitly defined by the designer. The used LFSR structure, is the TDL type (Type A) LFSR, due to its better path delays as the XOR gates are separate from the flip-flop chain. The associated Renoir symbol for the generated LFSR is as shown in figure V-20 and the generated VHDL code is in Appendix A-10.

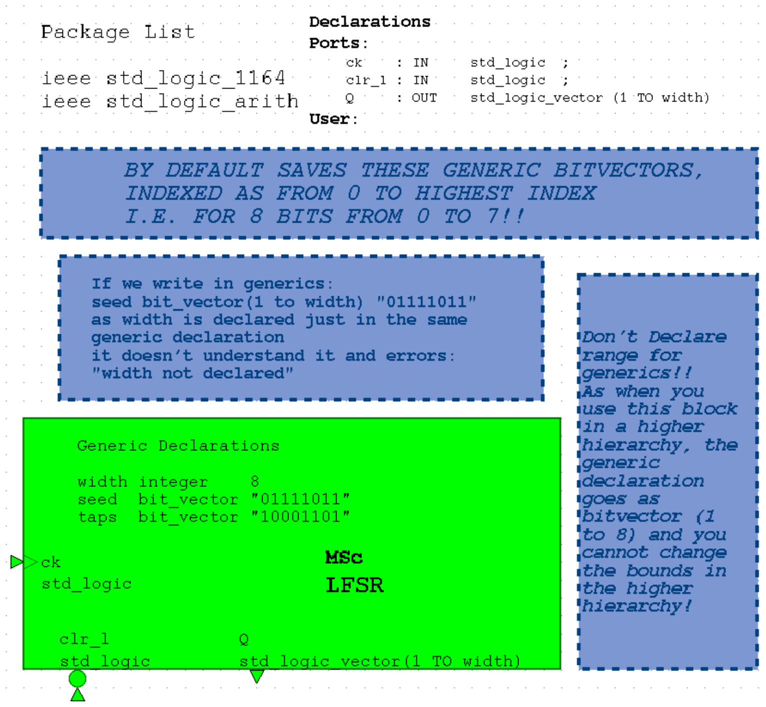

Package List
ieee std_logic_1164
ieee std_logic_arith
Declarations
Ports:
ck : IN std_logic ;
clr_l : IN std_logic ;
Q : OUT std_logic_vector (1 TO width)
User:
BY DEFAULT SAVES THESE GENERIC BITVECTORS, INDEXED AS FROM 0 TO HIGHEST INDEX I.E. FOR 8 BITS FROM 0 TO 7!!
If we write in generics: seed bit_vector(1 to width) "01111011" as width is declared just in the same generic declaration it doesn't understand it and errors: "width not declared"
Don't Declare range for generics!! As when you use this block in a higher hierarchy, the generic declaration goes as bitvector (1 to 8) and you cannot change the bounds in the higher hierarchy!
Generic Declarations
width integer 8
seed bit_vector "01111011"
taps bit_vector "10001101"
ck
std_logic
MSc
LFSR
clr_l
std_logic
Q
std_logic_vector(1 TO width)

Figure V-20,LFSR symbol

The generic, width, defines the length of the LFSR, seed defines the initial seed to be used, which the LFSR is set to when clr_L is active and taps define the taps of the LFSR. For the seed, the leftmost bit represents the seed into the leftmost flip-flop, and the rightmost bit represents the seed into the rightmost cell. For the taps, a ‘1’ represents an existing tap while a ‘0’ represents a nonexistent tap, leftmost bit referring to the leftmost flip-flop and rightmost bit to rightmost flip-flop. As an example, the hardware to be realized for the generic values given in figure V-20, which is actually the LFSR used in the final BIST circuitry is shown in figure V-21. Seed = “01111011” describes the 1st and 6th flip-flops are reset, while the rest are set. Taps=”10001101” describes that there are taps from the outputs of 1st, 5th, 6th and 8th flip-flops, which also reveals the characteristic polynomial of the used 8 bit LFSR is:

$$1 + x^1 + x^5 + x^6 + x^8$$

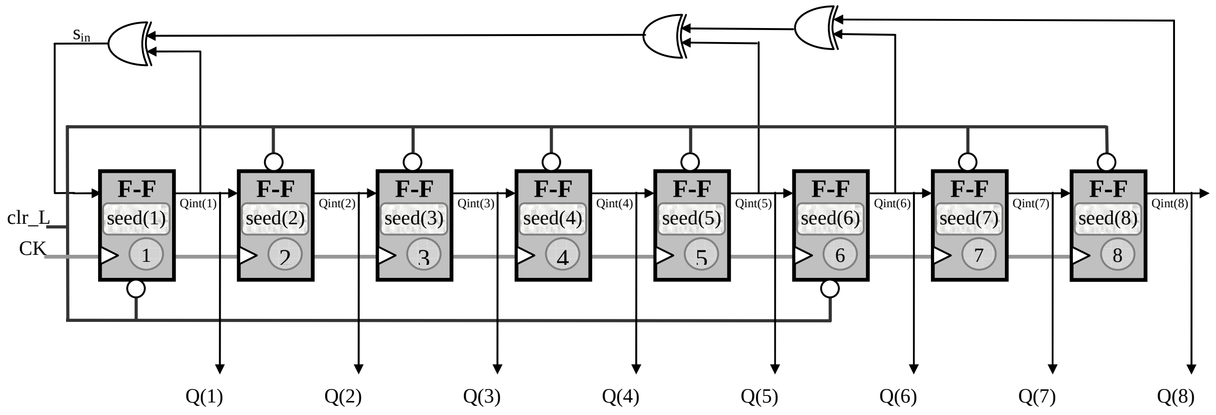


Figure V-21, Structure of LFSR

The VHDL level simulation, for the LFSR with the above parameters used is shown in Appendix C-3. The simulation list is strobed at every 90ns, therefore, only one Q output change is displayed per row. The initialization period is the first 50 ns and is not displayed in the simulation list. The input signals are produced as demonstrated in figure V-22:

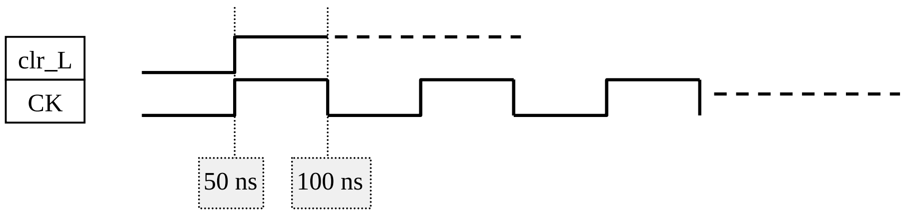


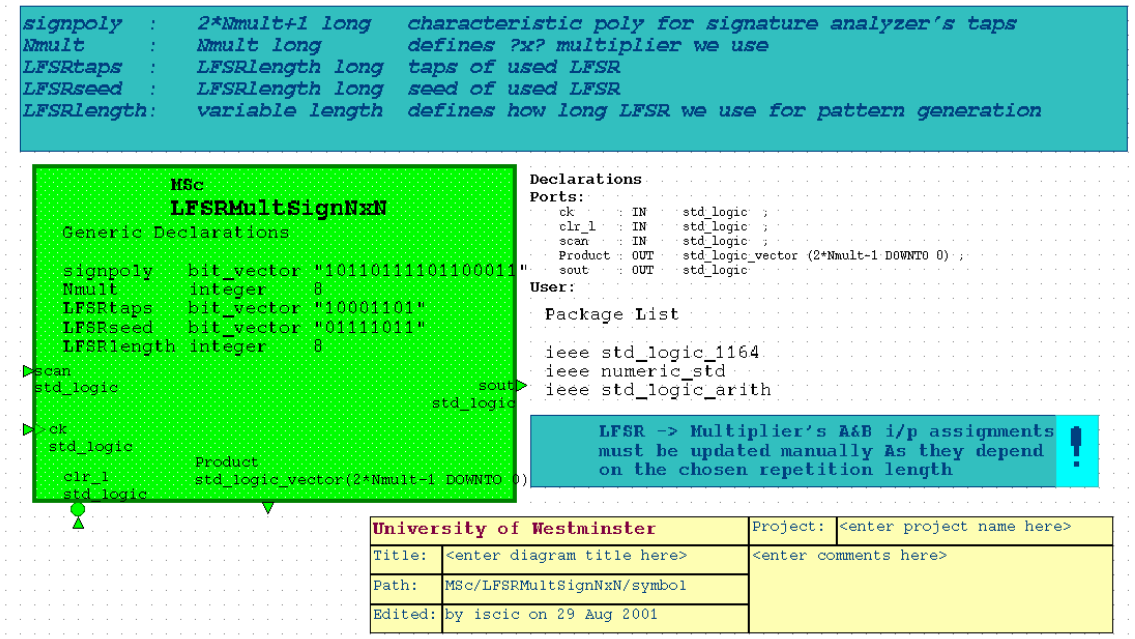
Figure V-22, LFSR Simulation input Signals

As seen in the simulation list, the 8 bit LFSR with the given characteristic polynomial produces a maximum length sequence, with a period of 255. Seed B7 is the next state after the listed last state F6, strobed at 25490 ns.

V.7 –DESIGN OF TOP LEVEL SYSTEM (LFSRMULTSIGNNXN)

After the evident VHDL simulation results for the LFSR, all of the required blocks for the top level design are completed and assured of correct operation. In this section, we build the final system with BIST circuitry for both input pattern generation and output compression included. In the top level structural design, which is attached in Appendix B-4, it is seen that the 4 right bits of LFSR’s ‘Q’ output, Q(5:8) in figure V-21 are assigned to both 4 bit portions of multiplier’s ‘a’ input, a(7:4) and a(3:0); and 4 left bits of ‘Q’ output, Q(1:4) in figure V-21 are assigned to both 4 bit portions of multiplier's 'b' input, b(7:4) and b(3:0). As discussed in section VI, this 4 bit long repeated pattern of input test patterns are seen to be very effective in fault detection, revealing a very high fault coverage with a small set of test vectors. Identical to MultSign8x8 design, the output of multiplier, product, is fed to the 'd' input of signature analyzer and the 'sout' output of signature analyzer reveals the codeword for compressed output data. Although the whole system is designed to be parameterized for variable sizes of multiplier, LFSR and signature analyzer, variable tap locations and seed for LFSR, and variable characteristic polynomial for the signature analyzer, the assignment of Q output bits to multiplier inputs is required to be done manually for each design as stated on the Renoir structural design sheet. This is to provide the designer with sufficient flexibility for the choice of repetition length – 4 in our case – and any desired permutation of outputs.

The generated Renoir symbol for LFSRMultSignNxN is as shown in figure V-23 and the generated VHDL code for the design is in Appendix A-11.



```
signpoly   :   2*Nmult+1 long   characteristic poly for signature analyzer's taps
Nmult      :   Nmult long       defines ?x? multiplier we use
LFSRtaps   :   LFSRlength long  taps of used LFSR
LFSRseed   :   LFSRlength long  seed of used LFSR
LFSRlength:    variable length  defines how long LFSR we use for pattern generation
```

```
MSc
LFSRMultSignNxN
Generic Declarations

signpoly   bit_vector "10110111101100011"
Nmult      integer    8
LFSRtaps   bit_vector "10001101"
LFSRseed   bit_vector "01111011"
LFSRlength integer    8
```

```
Declarations
Ports:
   ck      : IN    std_logic  ;
   clr_l   : IN    std_logic  ;
   scan    : IN    std_logic  ;
   Product : OUT   std_logic_vector (2*Nmult-1 DOWNTO 0) ;
   sout    : OUT   std_logic
User:

 Package List

 ieee std_logic_1164
 ieee numeric_std
 ieee std_logic_arith
```

LFSR -> Multiplier's A&B i/p assignments must be updated manually As they depend on the chosen repetition length

University of Westminster		Project:	<enter project name here>
Title:	<enter diagram title here>	<enter comments here>	
Path:	MSc/LFSRMultSignNxN/symbol		
Edited:	by iscic on 29 Aug 2001		

Figure V-23, LFSRMultSignNxN symbol

As seen in figure V-23, all the generic parameters from lower hierarchy is passed onto the top level and the top level design in general enables the specification of the multiplier as well as desired BIST circuit at the top level without the necessity to involve with the lower levels except for the case explained above and also exclaimed in the symbol.

The system inputs, clear and clock, have the same functionality as in the case of LFSR and scan input is the signature analyzer input which might be used to disable the multiplier output at the end of test run and the signature left in the signature analyzer can be clocked after the test input application, but in all our simulations we disable the scan input and strobe 'sout' during input test vector application.

The VHDL level simulation, with the generic parameters as assigned in the symbol figure, is attached in Appendix C-4. The simulation is again run for 255 clock cycles, which is the determined complete test for the system. The inputs clr_L and ck are driven as described in figure V-22 and scan input is always kept low as discussed formerly. As demonstrated in the simulation list, the signal values comply with the expectations. The LFSR is seen to take x7B seed value at the 266th cycle and traverses all the possible 255 states. The multiplier inputs and output are displayed in signed decimal and their functional operation is confirmed. The 'sout' values of signature analyzer are the primary test outputs that the test equipment observes during the 255 test cycles.

V.8 –16X16 MULTIPLIER WITH BIST (LFSRMULTSIGNNXN_16)

After the completed design of the top level system for the 8x8 multiplier, larger multipliers with BIST circuitry are designed in order to test the effectiveness of the applied constant set of test vectors. First a 16x16 multiplier with the BIST is designed. Due to the flexibility of the VHDL design, only the top level generics that are shown in figure V-23 are updated to the desired values and the internal LFSR output assignments are modified. As the symbol for the larger multiplier designs share the same symbol with LFSRMultSignNxN, the symbols are not printed again. The structural Renoir design which shows the LFSR output $\rightarrow$ multiplier input assignments for the LFSRMultSignNxN_16 is in Appendix B-5. As seen in the appendix, all the four 4 bit portions of both a and b inputs of the multiplier are assigned to Q(5:8) and Q(1:4) outputs of LFSR repetitively, thus, the inputs a and b having repeated patterns every 4 bits. The signature analyzer characteristic polynomial is computed from the high weight, Hamming distance 5 polynomial of the 16 bit signature analyzer by a simple polynomial multiplication, where the characteristic polynomial of 16th degree is multiplied by itself to produce a characteristic polynomial of 32nd degree. Hence, the length of the generated

PRBS sequence is not increased as the newly computed polynomial is already a non-primitive polynomial with two factors of degree 16 – which are also computed from the polynomial multiplication of 2 different 8th degree polynomials -, therefore, the length of the PRBS that could be generated from the signature analyzer characteristic polynomial is the same length as of the 16 bit signature analyzer, which is chosen to have a length of 255. The polynomial multiplication in GF(2) (Galois Field of 2) for the32nd degree characteristic polynomial is as shown below:

$$(1+x^2+ x^3+ x^5+ x^6+ x^7+ x^8+ x^{10}+ x^{11}+ x^{15}+ x^{16}) * (1+x^2+ x^3+ x^5+ x^6+ x^7+ x^8+ x^{10}+ x^{11}+ x^{15}+ x^{16}) =$$

$$= \boxed{1+x^4+ x^6+ x^{10}+ x^{12}+ x^{14}+ x^{16}+ x^{20}+ x^{22}+ x^{30}+ x^{32}}$$

Hence, this polynomial multiplication operation is best done via Matlab as polynomial multiplication is equivalent to convolution of coefficients. Therefore, the operation done in Matlab for characteristic polynomial calculation is:

```
Newpoly = mod(conv(oldpoly1,oldpoly2),2)
```

The mod$_2$ is to realize the arithmetic operation results in GF(2).

Acquired from the above calculation, the generic signpoly is assigned to:
“100010100010101010001010000000101”

and the generic Nmult is assigned to16. Other generics, which refer to the LFSR parameters are kept the same as figure V-23, as the input pattern generation scheme is constant, regardless of multiplier size. The VHDL code generated for the LFSRMultSignNxN_16, is attached in Appendix A-12 and the VHDL level simulation, is as shown in Appendix C-5. As seen in the simulation list, the inputs are driven the same way as in section V.7 and the results are in concordance with expectations.

V.9 –32X32 MULTIPLIER WITH BIST (LFSRMULTSIGNNXN_32)

After the fault simulation of LFSRMultSing16x16, a 32x32 multiplier is generated from the parameterized top level design. The signpoly generic is updated for a 64 bit signature analyzer, and the characteristic polynomial is acquired from polynomial multiplication of the

characteristic polynomial for the 32 bit signature analyzer by itself. Evidently, the PRBS length from this polynomial will be still 255, which is the preferred characteristic for the max 255 pattern input test vectors. The polynomial multiplication is performed as described in section V.8 and the resultant signpoly generic value is:
"10000000100010000000100010001000100000001000100000000000000010001"

The Nmult generic is obviously assigned to 32 and the LFSR related parameters are not changed. The structural Renoir description of the LFSRMultSignNxN_32 is as shown in Appendix B-6. As the repetitive pattern scheme suggests, all the eight 4 bit portions of b inputs are assigned to Q(1:4) and all the eight 4 bit portions of a inputs are assigned to Q(5:8). The VHDL code generated from the structural description is attached in Appendix A-13 and VHDL level simulation of LFSRMultSignNxN_32 is in Appendix C-6, which reveals correct functionality.

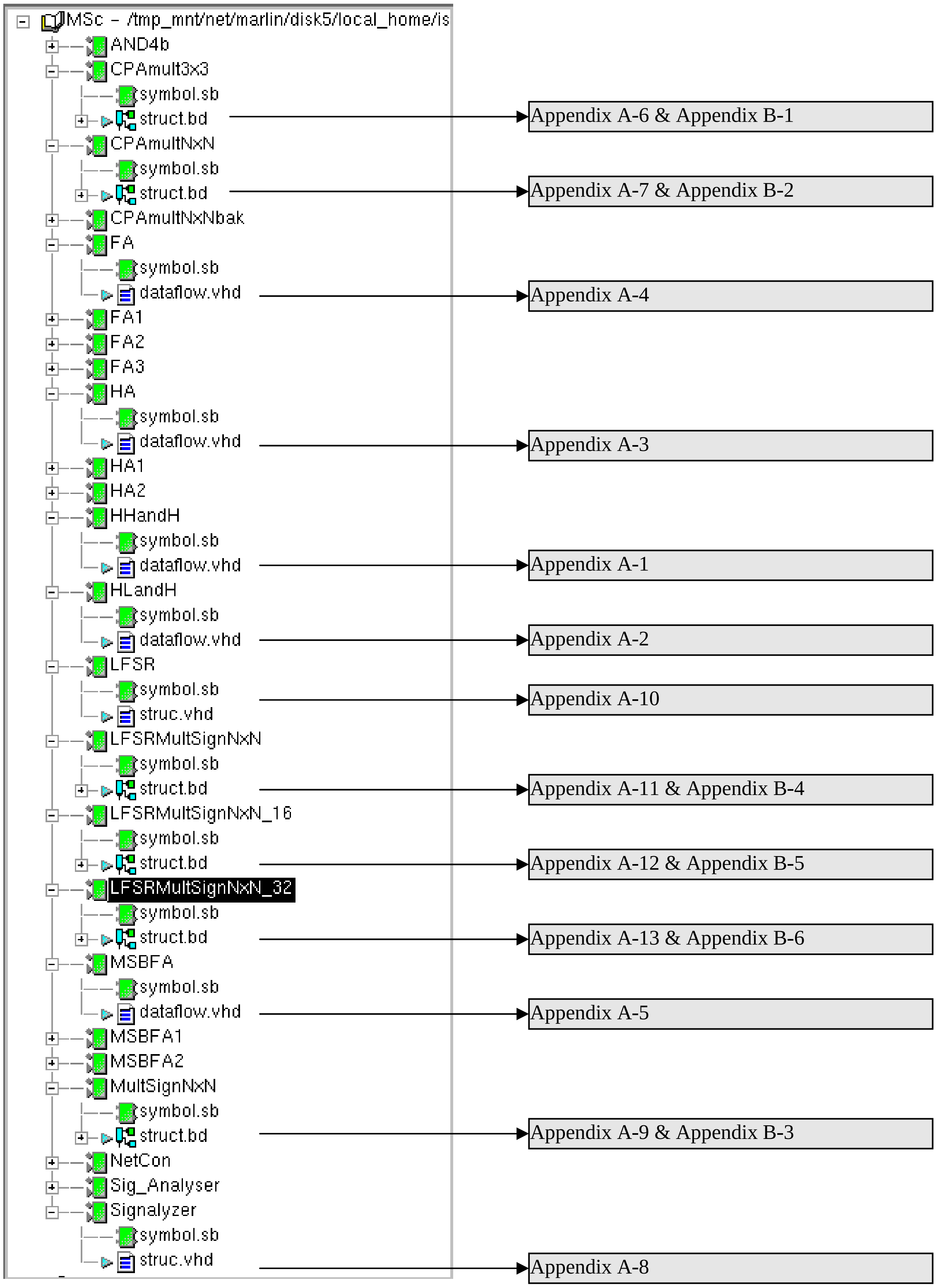
V.10 –24X24 MULTIPLIER WITH BIST (LFSRMULTSIGNNXN_24[2])

As the fault simulation of the 32x32 multiplier failed due to insufficient memory space, a moderately smaller sized 24x24 multiplier is designed after 32x32 multiplier. The characteristic polynomial for the 48 bit signature analyzer is acquired by polynomial multiplication of the characteristic polynomials for the 16 bit signature analyzer and 32 bit signature analyzer, in the way described formerly. The resultant signpoly generic value is:
"1011111000110000101010011010100000110111011101111"
and the Nmult value is 24. The structural Renoir design of LFSRMultSignNxN is in Appendix B-7 and the generated VHDL code is in Appendix A-14. The VHDL level simulation reveals correct operation, but is not included in the appendix for brevity.

V.11 – DESIGN SUMMARY

Finally, having described all design blocks, the MSc design library in Renoir is displayed in figure V-24, to describe the overall design blocks and also each design and its appendix reference is shown for quick reference.

[2] As this design is done after the CD is written, the corresponding data is missing in the submitted CD



```
MSc – /tmp_mnt/net/marlin/disk5/local_home/is
  AND4b
  CPAmult3x3
    symbol.sb
    struct.bd              ——> Appendix A-6 & Appendix B-1
  CPAmultNxN
    symbol.sb
    struct.bd              ——> Appendix A-7 & Appendix B-2
  CPAmultNxNbak
  FA
    symbol.sb
    dataflow.vhd           ——> Appendix A-4
  FA1
  FA2
  FA3
  HA
    symbol.sb
    dataflow.vhd           ——> Appendix A-3
  HA1
  HA2
  HHandH
    symbol.sb
    dataflow.vhd           ——> Appendix A-1
  HLandH
    symbol.sb
    dataflow.vhd           ——> Appendix A-2
  LFSR
    symbol.sb              ——> Appendix A-10
    struc.vhd
  LFSRMultSignNxN
    symbol.sb
    struct.bd              ——> Appendix A-11 & Appendix B-4
  LFSRMultSignNxN_16
    symbol.sb
    struct.bd              ——> Appendix A-12 & Appendix B-5
  LFSRMultSignNxN_32
    symbol.sb
    struct.bd              ——> Appendix A-13 & Appendix B-6
  MSBFA
    symbol.sb
    dataflow.vhd           ——> Appendix A-5
  MSBFA1
  MSBFA2
  MultSignNxN
    symbol.sb
    struct.bd              ——> Appendix A-9 & Appendix B-3
  NetCon
  Sig_Analyser
  Signalyzer
    symbol.sb
    struc.vhd              ——> Appendix A-8
```

Figure V-24, Design Library

VI- INVESTIGATION OF BIST TECHNIQUES

This section, which can be considered as the core of the project includes the phases 2 and 3 described in the Introduction, describing the investigated BIST strategies for both input pattern generation and output compression. We first discuss the terms of fault simulation, deterministic and alternative fault simulation techniques, different fault models, including single stuck at model QuickFault uses, Cell Fault Model (CFM)([23]), defended against single stuck at model in [3] and their comparison. Hierarchical vs. Board level faulting, opportunistic vs. cycle based simulation and pin/net faulting are described and the choices for the project fault simulations are justified [20]. Starting with board level faulting and then expanding to hierarchical faulting up to primitive cells, the multiplier design is simulated with respect to several input pattern generation techniques for efficient fault coverage. The techniques are described in an evolutionary manner in order to demonstrate how each new technique is initiated. The fault coverage result for each case is demonstrated and commented for pros and cons. All the plotted and listed data are also included in "CD>MSc/Results/*designname*". The resultant data are either included in the text, or appended in the appendices or referred from the CD depending on the size of the data. Chronologically, this section starts after the completed design of CPAmultNxN, parameterized, signed, parallel, CPA multiplier up to the end of input test pattern generation techniques, after then, output compression technique and further simulations for larger multipliers are performed in a design → fault simulation manner for each top level design. As the signature analysis reveals extremely effective results, other techniques are not investigated and the output compression is concluded with signature analysis.

Before starting with the fault simulation of the phase, it is worth outlining the intermediate steps taken to convert the VHDL design into the simulated schematic format. The generated VHDL descriptions for the design in "CD>MSc/DesignFiles/Renoir/HDL" are first compiled into Autologic format using 'qvhcom' of QuickHDL and are shown in "CD:\MScCD\MSc\DesignFiles\Renoir\SynthCompiledData\component_name\".
The compiled components are then synthesized and converted to eddm-schematic format using Autologic, and are shown in "CD:\MScCD\MSc\DesignFiles\Renoir\Eddm_sch\". Then, the schematic files are opened in QuickFault and fault simulation is performed.

VI.1 – FAULT SIMULATION AND FAULT MODELS

Fault simulation is the process of exercising the components of a given circuit under a predefined set of faults artificially inserted to the circuit, which depend on the postulated fault modeling. The objective of fault simulation is to verify the effectiveness of a devised test set as well as to improve a given test set for higher fault coverage. The most common types of faults in digital circuits are:

(i) stuck-at faults
(ii) bridging faults
(iii) stuck-open faults
(iv) pattern sensitive faults

However, for computation cost issues, it is not rational to consider all these faults in fault simulation and usually more simplified fault models are used in most fault simulators. As described in [1] and [13], the commonly used technique is the single stuck at model, which is proven to also detect all multiple stuck at faults in a two level combinational circuit. Moreover, it is proven that a set of patterns, which detect all single stuck at faults will also detect bridging faults([1]). Although until recently this model was observed to be fairly adequate in fault representation in real circuits, particularly in bipolar technologies, the advanced CMOS technologies begin to produce faults, which cannot be modeled with the single stuck at model. A more comprehensive technique named “Cell Fault Model”, which is proposed in [23], is defended in [3] as a more realistic technique, but it lacks the required simulation tools for general acceptance. Nevertheless, [24] provides indirect techniques to utilize this model with the current single stuck at fault simulators.

There are several software techniques used for fault simulation, which can be listed as:

- commonly used techniques:
  1. Parallel fault simulation
  2. Deductive fault simulation
  3. Concurrent fault simulation
- more recent techniques
  4. Parallel Valued Lists (PVLs)
  5. Parallel pattern single fault propagation (PPSFP)

More detailed information about these can be obtained from [1] and [20], but is omitted in the context of this text. Another classification of fault simulation is based on the fault sampling technique used. A “deterministic” fault simulator, exactly wires to each net/pin stuck at 0 and stuck at 1 values and exhaustively exercises each node in the specified faulting hierarchy, a “statistical” fault simulator on the other hand uses probabilistic measures for the detectability of each node and uses the fault detection probabilities to compute the final estimated fault coverage. Obviously, the first technique favors precision while the second computation cost.

In all our fault simulations, we use Mentor Graphics’ QuickFault, which is a single stuck at, deterministic fault simulator.

VI.2 – INITIATION PART-II: FAULT SIMULATION OF 3X3 CPA MULTIPLIER

After the design entry in Renoir and VHDL level simulation verification of the CPA multiplier, the leaf level design blocks and higher hierarchy blocks are synthesized into Eddm_schematic format. In order to describe the second phase of experimental route, firstly the 3x3 CPA multiplier is synthesized and simulated for faults. The generated schematic for the 3x3 multiplier is as shown in figure VI-1. The light rectangles connected to the pins of the components represent inserted stuck at 1 faults, while the dark ones represent inserted stuck at 0 faults.

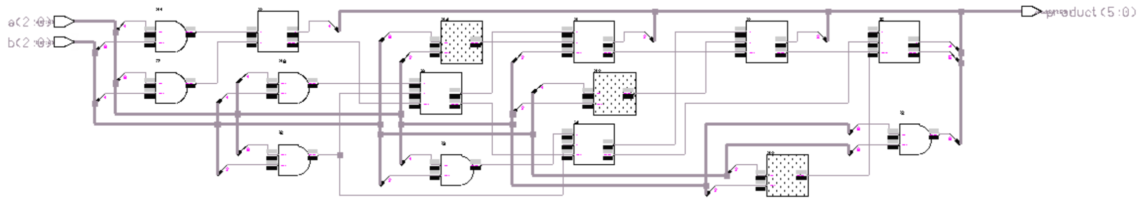


Figure VI-1, Generated CPAmult3x3schematic with inserted faults

As seen in the figure, the faults are inserted at board level, therefore, the internal pins and nets of these blocks are assumed to be non-faulty. This approach is prone to the criticism that the blocks like FA and MSBFA are quite complex, and assuming their internal nodes are fault free is an overoptimistic assumption. However, for hierarchical level simulation, the internal primitives, which are supplied by the vendor must be known and these vary from application to application diminishing the possibility of a generally acceptable model. Another approach, named Cell Fault Model, which is introduced by [23] suggests a different strategy, which is defended to be more comprehensive in [3]. With this board level simulation, there are 112 faults injected in the 3x3 multiplier. The fault list for the inserted faults are appended in Appendix D-1, to provide an example to the fault list format QuickFault produces. The pins are described as: instance/pin, and as seen in the appendix, there is one stuck at 0 and one stuck at 1 fault per pin. DT represents the status of the fault being detected. All the QuickFault fault classifications are defined in [20, pp. 1-14 – 1-15].

In order to determine the fault coverage for the 3x3 multiplier, we devised a simple exhaustive test pattern generation scheme, an upcounter which generates all the possible 2^6 = 64 inputs to the 3x3 multiplier. The 6 bit upcounter as the input pattern generator provides the patterns as:

Bits	**decimal**
000000	0
000001	1
.	.
.	.
.	.
111111	2^6-1=63

The forcefile, which produces the stimulus is listed in Appendix D-2, to provide an example to the forcefile format used in QuickFault – the general simulation environment being QuickSim. Hence, all other fault lists and stimulus files are referred from the CD and are not appended in the report due to unmanageable hardcopy volumes. For the fault simulation, cycle based testing is used which represents an Automatic Test Equipment (ATE), much realistically, than the opportunistic test. The stimulus provides a new test pattern every 50ns and therefore, the primary output, product, is strobed every 40 ns after each new pattern is provided and as the QuickFault allows only compare window testing rather than single sampling point ([25, pp. 10 - 12]) the strobing window is chosen as 5 ns. With 40 ns after the test vector application, all the outputs are well stabilized and therefore unrealistic high fault detection results due to the comparison of transient values are prevented. The defined test cycle information is as shown below:

```
NAME: cyc1  Period: 50.0
```

```
DETECTORS:
After      Window
------      ------
40.0         5.0

EVENT STREAM:
Event   Supersede   Time
-----   ---------   ----
EN          no    0.0ns
```

With the above described setup for primary output tests and input stimuli, we have run the fault simulation, and the total of 64 patterns which test the circuit for all possible input combinations are expected to have a 100% fault coverage, as long as there is no inherent redundancy in the circuit. As QuickFault also extracts information for schematic level simulation, the schematic level simulation results are also observed from QuickFault during fault simulation. The simulation result for an exemplary period of simulation is plotted in figure VI-2.

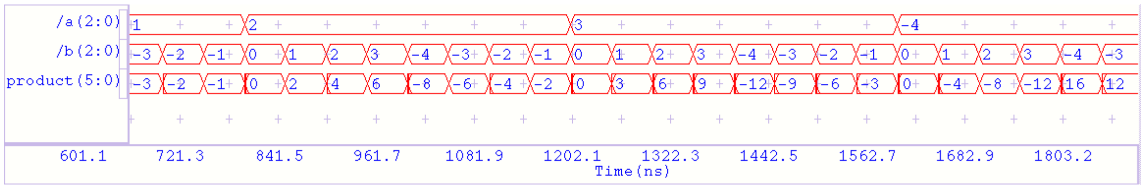


Figure VI-2, Schematic Level Simulation Trace for 3x3 Multiplier

As can also be observed in the fault list, all the faults are detected in 37 cycles out of 64. which reveals, when starting from 000000, around 40% of the exhaustive test is redundant in multiplier testing. As shown in figure VI-3, 100100 is the last effective test pattern in the input test set.

bits	**decimal**	
000000	0	
000001	1	
.	.	Required
.	.	
100100	36	
.	.	
.	.	Redundant
111111	2^6-1=63	

Figure VI-3,Effectiveness of applied input patterns

The overall grade of fault simulation for the 3x3 multiplier, as shown below, shows, as expected, 100% fault coverage is achieved with the exhaustive test.

```
CURRENT STATUS
--------------
Total Faults       :  112
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                    :  3300.0ns
Total Faults                :  112
   Untestable Faults        :  0
   Testable Faults          :  112
      Undetected Faults     :  0         (0.00%)
      Detected Faults       :  112       (100.00%)
      Possible Faults       :  0         (0.00%)
      Hyperactive Faults    :  0         (0.00%)
      Hypertrophic Faults   :  0         (0.00%)
      Oscillatory Faults    :  0         (0.00%)
      HM Dropped Faults     :  0         (0.00%)
```

Hence, fault grade information is useful to observe the overall effect of applied test patterns, but does not reveal any information about how the percentage of detection builds up. In order to see the accumulation of fault detection QuickFault provides a plot of fault coverage percentage vs. input test cycles and a histogram of number of detected faults at each cycle. As seen in the fault coverage plot for the 3x3 multiplier in figure VI-4, the fault coverage ascends up to 100% in cycle 37, with the nature of the gradient highly correlated to the cycles, revealing which input patterns are more effective in detecting the faults.

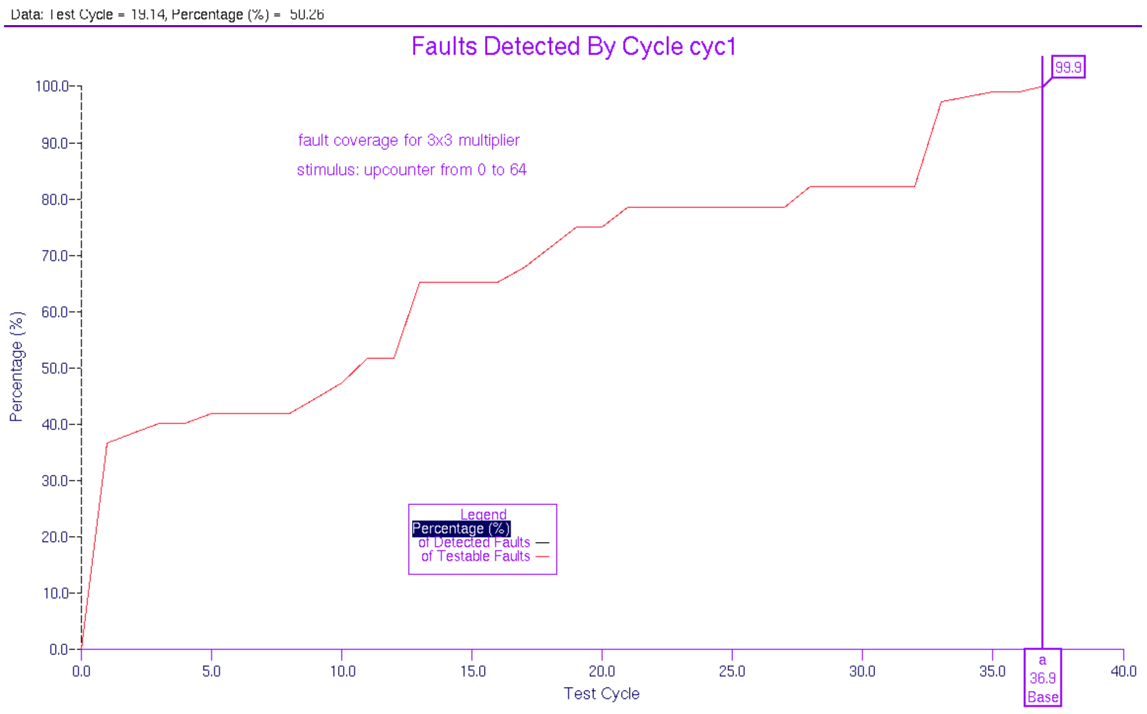


Figure VI-4, Fault coverage (%) vs. test cycle plot for 3x3 multiplier

On the other hand, as seen in the histogram plot in figure VI-5, although you cannot easily observe the total fault coverage up to a certain cycle, the effect of each individual cycle is very clearly displayed, which is extremely hard to observe from the plot in figure VI-4. Moreover, another information easily obtained from the histogram is the redundant cycles within the first 37 cycles and the cycles which are very effective in fault detection, i.e. the cycles that produce strong peaks in histogram, which are also quite obscure in the plot. In our case, there are only 17 out of the 37 cycles which are actually effective in fault detection, and cycles 13 (001101) and 33 (100001) are seen to be very effective in this order of input patterns.

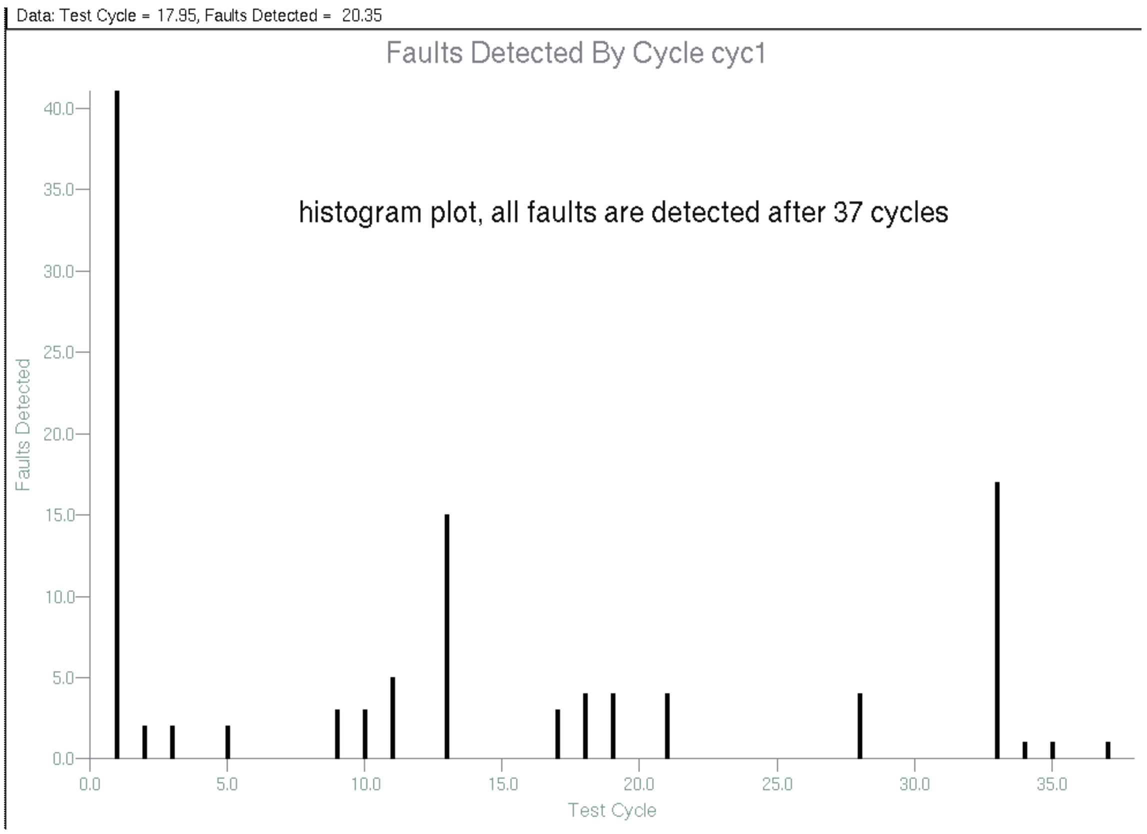


Figure VI-5, Number of Detected Faults vs. test cycle histogram for 3x3 multiplier

Obviously, these two statistics are very elucidating and reveal several guiding information. Fault coverage plot can reveal the amount of required patterns to achieve a certain lower limit on fault coverage, while histogram can be used to determine effective starting points for the BIST circuit – as made use of in seed determination for LFSRs and CA – and to determine an efficient set of deterministic test patterns. Hence, as the cycle based test is performed such that, each strobe is for 1 test pattern, the terms test cycle and test pattern are used loosely interchangeably throughout the text. As can be deduced from Appendix D-1, the histogram and plot data can also be extracted from the fault list data after simulation. Therefore, for batch simulation, or to perform fault simulation from a remote machine without graphics interface, the text data written in the fault list can be used to independently plot the histogram and fault coverage plot.

From now on, the fault lists, input stimuli and cycle info data will not be included in the report, as they are rather auxiliary information. However, all the fault simulation data is also included in the “CD>MSc/Results/”. From here on, until the inclusion of BIST circuit in the synthesized hardware, all the fault simulation results are in “CD>MSc/Results/mult8x8/”. The fault simulations after the inclusion of designed BIST hardware are referenced from the design name, as “CD>MSc/Results/*DesignName*”. The standard naming convention for the fault simulation data is as shown in table 13:

Stimuli	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTstimulus.txt
Schematic Level Simulation Results	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTsim.gif
	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTsimlist.txt
Fault Lists	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTfaultlist.txt
Cycle Infos	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTcycleinfo.txt
Fault Grades	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTfaultsummary.txt
Fault Coverage Plots	CD>MSc/Results/mult8x8/appliedBIST/appliedBISTplot.gif
Histograms	CD>MSc/Results/mult8x8/appliedBIST/appliedBISThisto.gif

Table 13, Fault simulation data locations in CD

Moreover, the plots included in the text are also stored in: “CD>MSc/DesignFiles/report/quickfaultfigs”, but it is recommended that the figures in the Results/ are preferred as the report figures are color edited and have worse resolution.

VI.3 – BIST FOR 8X8 MULTIPLIER

Having described the design flow and general data structure with Initiation parts I and II, we start the investigation of BIST techniques for the 8x8 multiplier, designed in Renoir, using the parameterized multiplier. The synthesized 8x8 multiplier is in: “CD>MSc/DesignFiles/Renoir/Eddm_schematic/ cpamultnxn_8/”.

Initially we use board level faulting for the fault simulation of 8x8 multiplier, and then we revert to the hierarchical fault injection using the generic library ($MGC_Genlib) of Renoir as the hierarchical leaf level, which includes only AND, OR gates, inverters, set-reset flip-flops, etc., which is a very low level hierarchical model compared to the models most vendors provide. Several well-practiced and original BIST techniques are applied as stimuli and their fault coverage performances are observed, with specific emphasis given on PRBS generation schemes. Repeated patterns, which are shown to provide very high fault coverage with a small set of input test vectors([3]) are observed and used within PRBS generation structures.

During the investigation of different BIST schemes, we use automatically generated stimulus files, either created from QuickFault or using the Matlab Scripts in "CD>MSc/DesignFiles/matlab". With the quick turnover of this approach, various techniques are investigated without the necessity of building the BIST hardware. After we conclude the investigation and decide upon the BIST structure based on the observed performances, we move back to design entry phase and generate the blocks required for the BIST circuit. The next step is the investigation of output compression techniques and comparison of the fault coverage results with and without output compression.

VI.3.1 – BIST = 16 bit Upcounter for Board Level Faulting:

As the first probable BIST scheme, we use an upcounter of 16 bits length, which produces all the possible 2^{16} = 64K = 65536 input patterns for the multiplier. With board level faulting, the number of injected faults is 932 and we apply exhaustive testing for all 64K input patterns. From the experience gained by the fault simulation of 3x3 multiplier with the same upcounter scheme, we expect again around 40% of the overall exhaustive test to be redundant.

The applied stimulus is in "CD>MSc/Results/mult8x8/upcountstimulus.txt" and the fault list for the inserted faults is in "CD>MSc/Results/mult8x8/faultlist.txt" and the test cycle information is in "CD>MSc/Results/mult8x8/cycleinfo.txt". The schematic generated for the 8x8 multiplier and the injected faults are as shown in figure VI-6, which also displays an exemplary schematic level simulation for the multiplier with the inputs and output in signed decimal radix. From now on, for all the stimuli, one pattern duration is 100 ns, and strobing time is 90 ns.

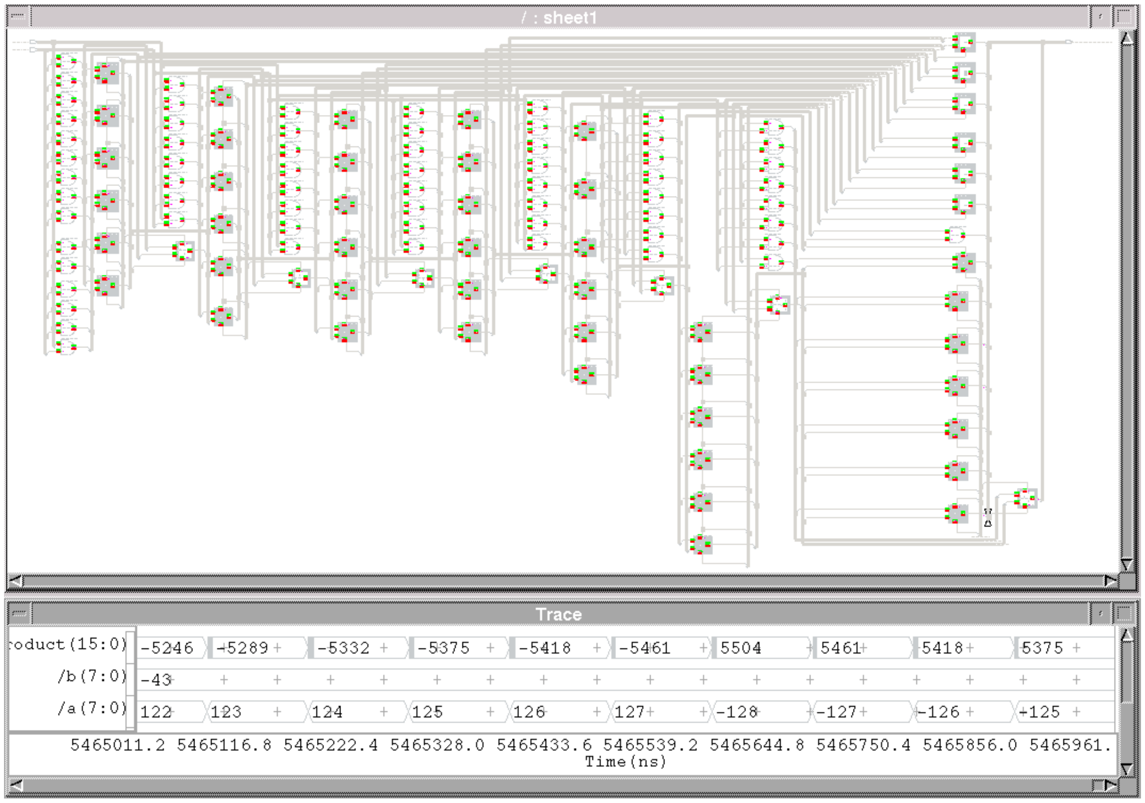


Figure VI-6, Multiplier 8x8 Schematic, Inserted Faults and Simulation trace for the upcounter test

With the applied stimulus, 100% fault coverage is achieved in 32897 cycles out of 64K. Exceeding our expectation, the redundancy is around 50 %, and the last effective input pattern is: $32897\text{-}1_{10} = 32896_{10} = 10000000\ 10000000_2$, which is the most negative value possible multiplied by itself. The grade of the fault simulation is as shown below:

```
--------------
Total Faults       :  932
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                     :  5466000.0ns
Total Faults                 :  932
   Untestable Faults         :  0
   Testable Faults           :  932
      Undetected Faults      :  0          (0.00%)
      Detected Faults        :  932        (100.00%)
      Possible Faults        :  0          (0.00%)
      Hyperactive Faults     :  0          (0.00%)
      Hypertrophic Faults    :  0          (0.00%)
      Oscillatory Faults     :  0          (0.00%)
      HM Dropped Faults      :  0          (0.00%)
```

The histogram and the fault coverage plot are as shown in figures VI-7 and VI-8 respectively.

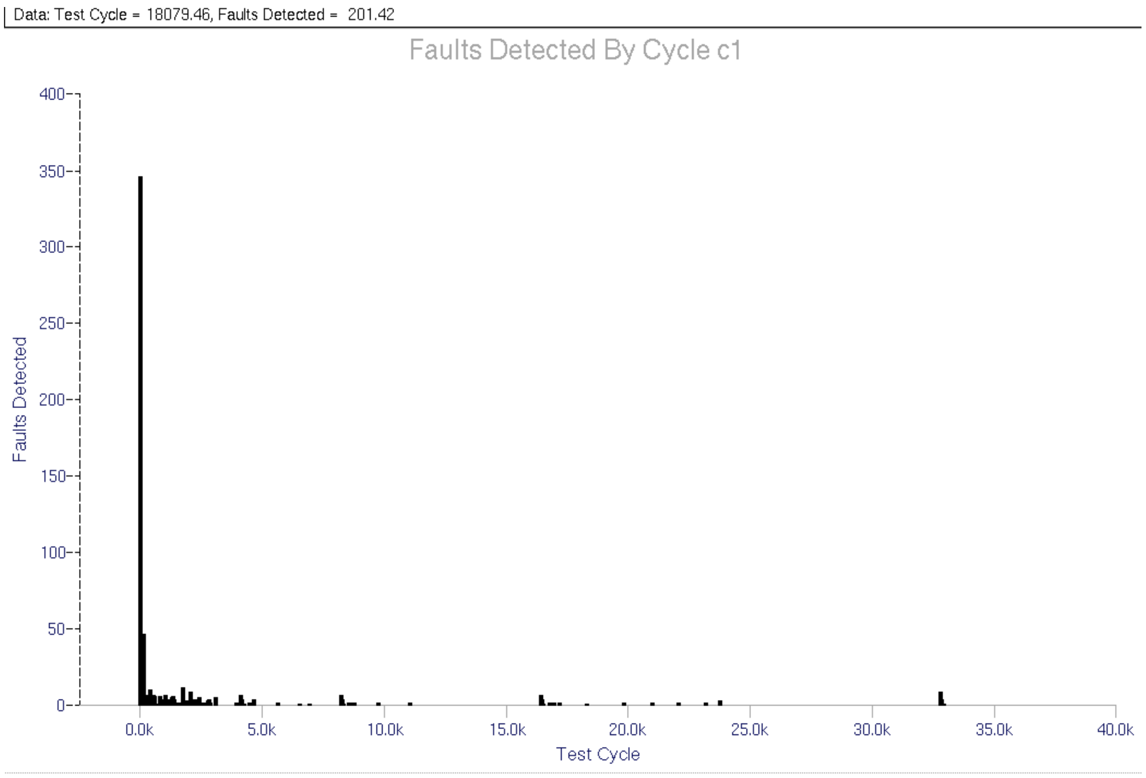


Figure VI-7, Number of Detected Faults vs. test cycle histogram for board level upcount test

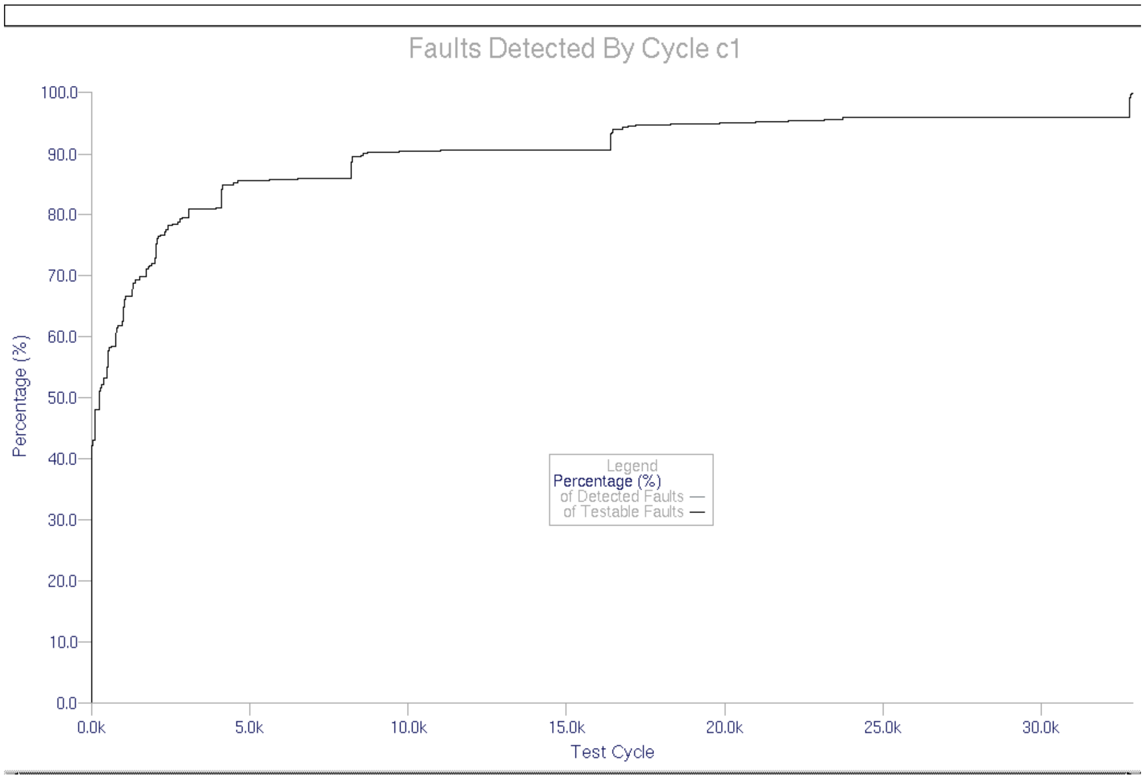


Figure VI-8, Fault coverage (%) vs. test cycle plot for board level upcounter test

As can be seen in histogram, there is a vast redundancy within the 32897 cycles as well. After the initial hyperactive region, the cycles that are effectively detecting faults are distributed within the 32897 cycles. The only seemingly nonredundant region is right at the end of the displayed patterns where a set of consecutive patterns actively detect remaining faults.

The plot, after the initial steep increase in fault coverage, shows an almost flat incline and then, a step like jump at regions around 4K, 8K, 17K and 33K where the effective cycles are seen to concentrate.

VI.3.2 – BIST = 16 bit Downcounter for Board Level Faulting:

Similar to first BIST, we use a downcounter of 16 bits length, which produces all the possible 2^{16} = 64K = 65536 input patterns for the multiplier. We use board level faulting (932 faults) and apply the test for all 64K input patterns in the reverse order of previous section. All the fault simulation data are in “CD\MSc\Results\mult8x8\downcount”. With the same strobing setup, we achieve 100% fault coverage exactly at 32897th cycle again. The fault grade is as shown below:

```
CURRENT STATUS
--------------
Total Faults       :  932
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                     :  4000000.0ns
Total Faults                 :  932
   Untestable Faults         :  0
   Testable Faults           :  932
      Undetected Faults      :  0         (0.00%)
      Detected Faults        :  932       (100.00%)
      Possible Faults        :  0         (0.00%)
      Hyperactive Faults     :  0         (0.00%)
      Hypertrophic Faults    :  0         (0.00%)
      Oscillatory Faults     :  0         (0.00%)
      HM Dropped Faults      :  0         (0.00%)
```

The acquired histogram and fault coverage plot are as in figures VI-9 and VI-10 respectively.

As can be observed from the plots, the fault coverage builds up much rapidly than the upcounter. However, it still requires a huge 32897 patterns to achieve 100% fault coverage. The histogram is also much less spread providing much better concentrated set of patterns for large amount of fault detection.

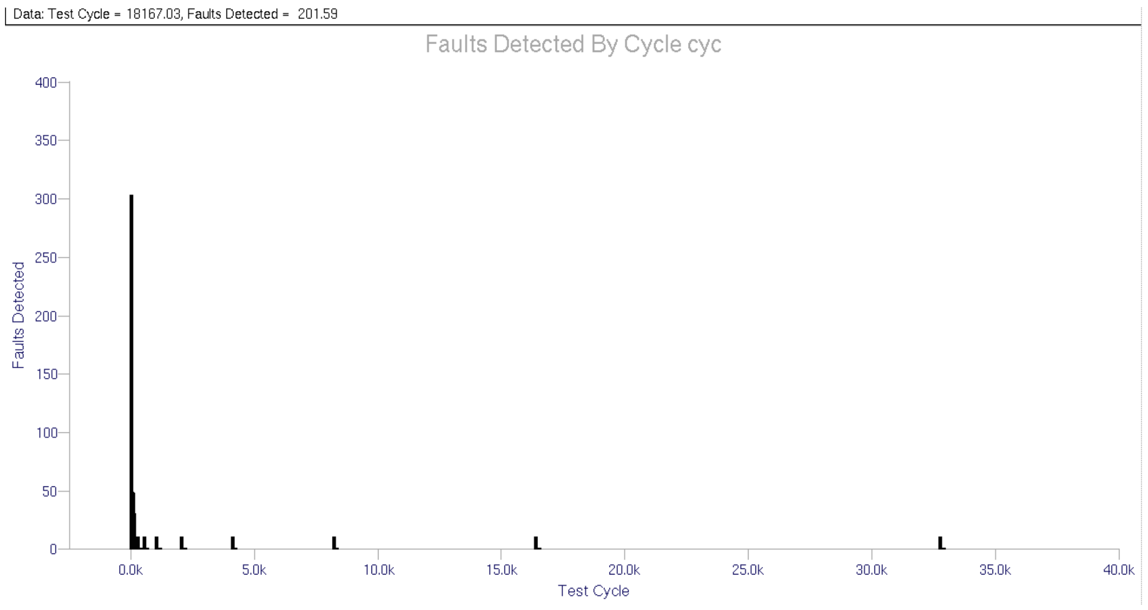


Figure VI-9, Number of Detected Faults vs. test cycle histogram for board level downcount test

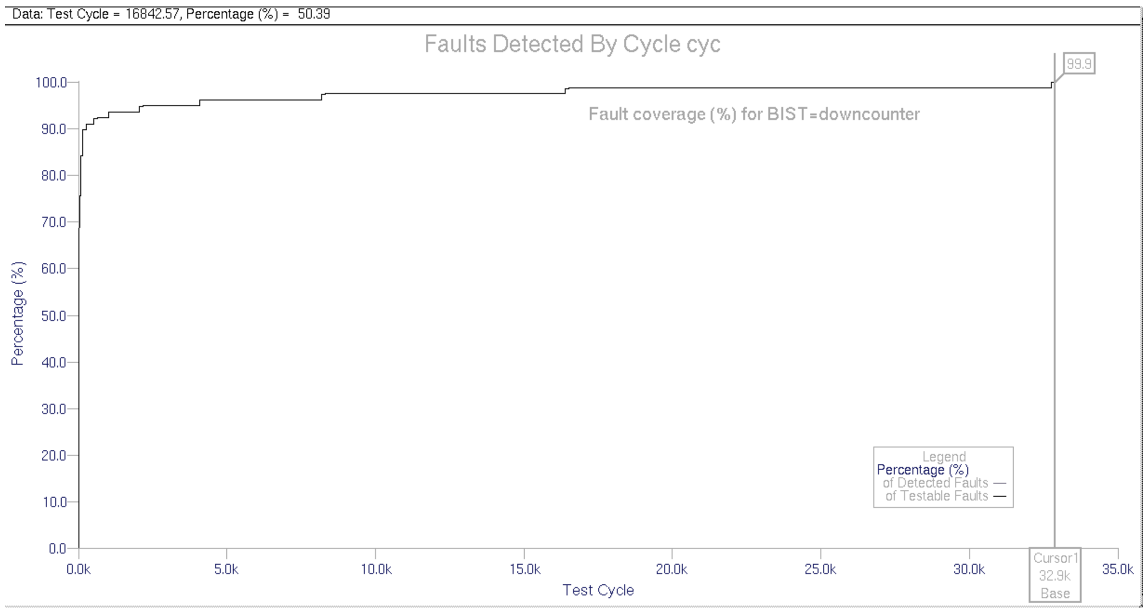


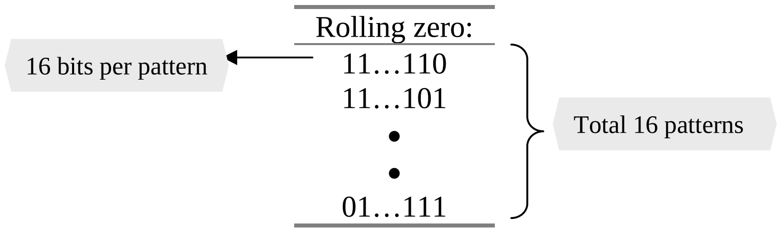
Figure VI-10, Fault coverage (%) vs. test cycle plot for board level downcounter test

VI.3.3 – BIST = 16 bit Rolling 0 for Board Level Faulting:

Although the above two cases demonstrate the case of full exhaustive testing, with the probably simplest BIST to be thought, they also produce very discouraging results as for only 932 faults, which are at the highest level, around 32000 vectors are needed for full coverage. And as seen in the histograms, more than 90% of the vectors are actually redundant and applied only to follow the BIST sequence. As a matter of fact, for 932 faults, much less than

932 vectors must suffice for fault coverage, as number of vectors being more than number of faults inherently signals inevitable redundancy.

At this point, a careful observation of the multiplier circuit reveals an important phenomenon. All the 16 inputs of multiplier are first applied to either to HLandH gates (for MSB partial products) or to HHandH gates (for all other partial products), and then, the number of inputs is reduced to half. Intuitively, a test set that fully exercises the multiplier, should start by fully exercising these input gates and then cleanup tests might be applied for the remaining faults. However, the challenge is to provide this scheme in BIST rather than as a standalone deterministic test set. Considering the AND structure of the input gates, a rolling 0 test pattern as demonstrated in table 14 for the 16 bit inputs seems promising, as for an AND gate, to test the input and output pins for stuck at 1 and stuck at 0, a 01→ 10 → 11 pattern is sufficient[1], which is produced for all the gates with the rolling 0 pattern.


16 bits per pattern
Total 16 patterns

Rolling zero:
11…110
11…101
•
•
01…111

Table 14, Rolling 0 pattern

In order to verify this argument, the rolling 0 pattern, which is quite easy to generate as BIST, is applied as the next test set. The simulation data are in “CD\MSc\Results\mult8x8\rolling0”. As seen in the fault grade, there is 98.07% fault coverage, with only these 16 vectors. Out of the 932 board level faults, 914 are detected and only 18 of the injected faults are undetected. The histogram and fault coverage plots, in figures VI-11 and VI-12 reveal the very efficient impact of each vector and the fault coverage curve is almost piecewise linear, without any flat regions that imply redundancy. In order to have an understanding of the undetected faults, we also demonstrate the undetected faults on the circuit after the application of the rolling 0 test pattern in figure VI-13. As can be seen on the figure, undetected faults are on the MSB bits of the partial adders and on the noninverted

[1] Recalling the initial argument, the astute reader will notice this is not completely true for the HLandH gates, and this will constitute the context of next section. However, to present the development of concepts, the two are described separately.

inputs of the HLandH gates. This latter observation as zoomed on figure VI-13 is also displayed on figure VI-14.

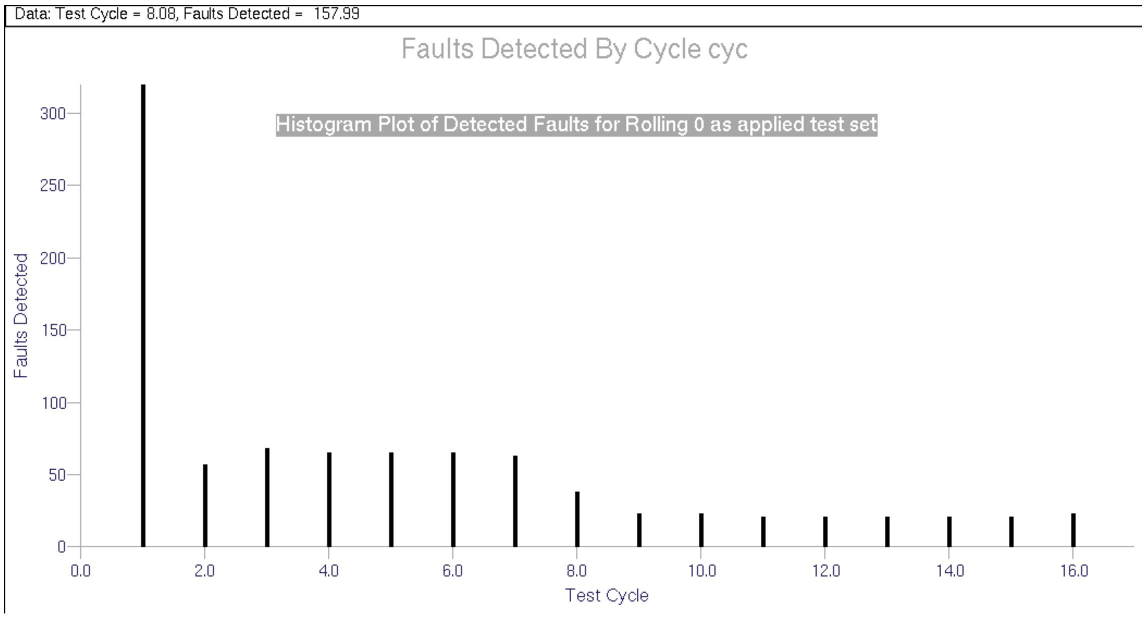


Figure VI-11, Histogram for board level rolling0 test

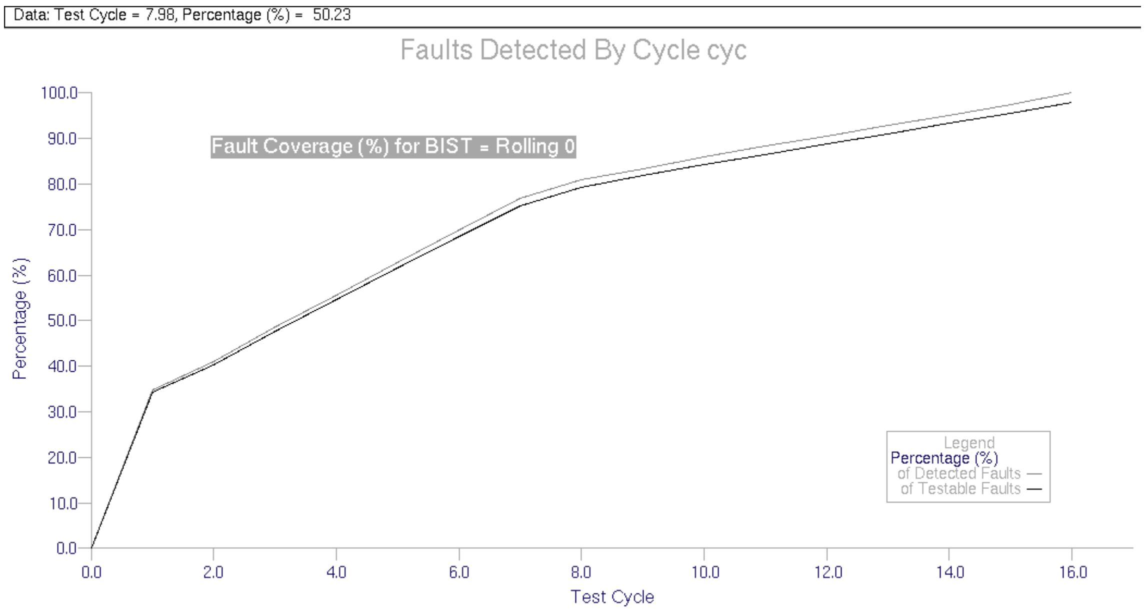


Figure VI-12, Fault coverage (%) for board level rolling0 test

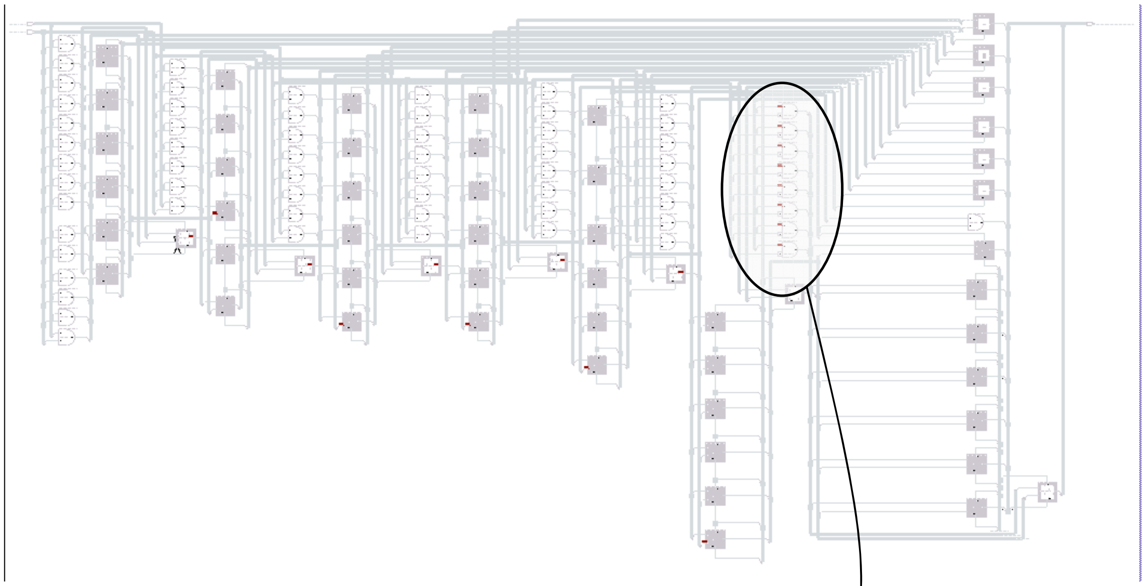


Figure VI-13, Undetected faults

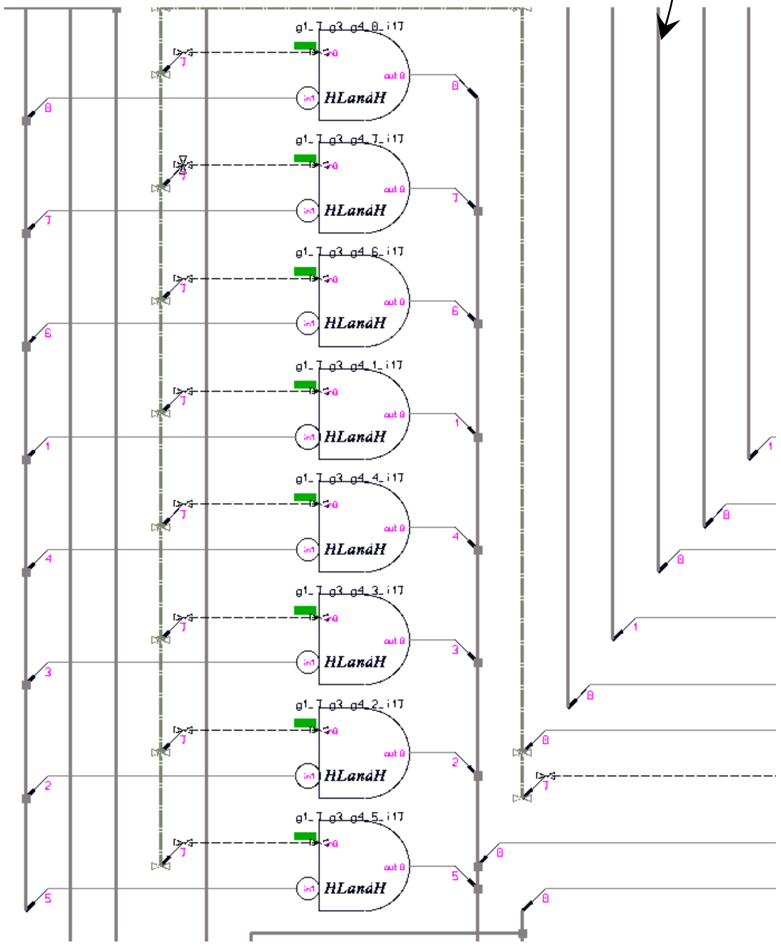


Figure VI-14, Undetected faults on the HLandH gates

Figures VI-13 and VI-14 provide two important observations:

(1) 8 out of 18 undetected faults are on the noninverted inputs of the HLandH gates

(2) all the undetected faults are stuck at 1 faults

These both imply addition of a zero dominated input test vector, which is the discussion of the next section.

VI.3.4 – BIST = 16 bit Rolling 0 + all 0s for Board Level Faulting:

The actually expected reason for the undetected faults on the HLandH gates is evident. For a stuck at 1 fault at the noninverted input terminal of the HLandH gate to propagate to the output, the other input should be true so that the output is sensitized with respect to the noninverted input. To provide a conflict at output, the stuck at 1 test should apply a zero to the tested input. These two facts, which are visualized in figure VI-15 require a 00 pattern at each HLandH gate input, which can be accomplished with an all 0s test pattern in parallel for all the gates. Moreover, there is good chance that this all 0s pattern will also detect the other stuck at 1 faults as only 0s are propagated through the circuit components. Consequently, due to the second observation stated, the circuit has the probable opportunity of being completely tested for the injected faults.

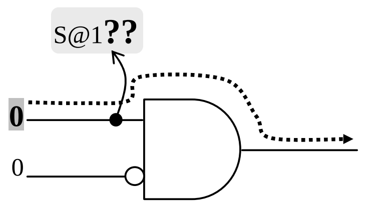


Figure VI-15, Sensitization of noninverting input for stuck at 1 fault

With the inclusion of all 0s, overall test set now becomes as shown in table 15, which can also be generated on chip with simple hardware.

16 bits per pattern

Rolling zero + all 0s:
11…110
11…101
•
•
01…111
00…000

Total 17 patterns

Table 15, Rolling 0 + all 0s pattern

All the fault simulation data and the corresponding results for this BIST are in "CD\MSc\Results\mult8x8\rolling0andall0". A snapshot of the overall fault simulation is as shown in figure VI-16, where the rolling 0 and final all 0s case is demonstrated and the resulting fault grade is shown below:

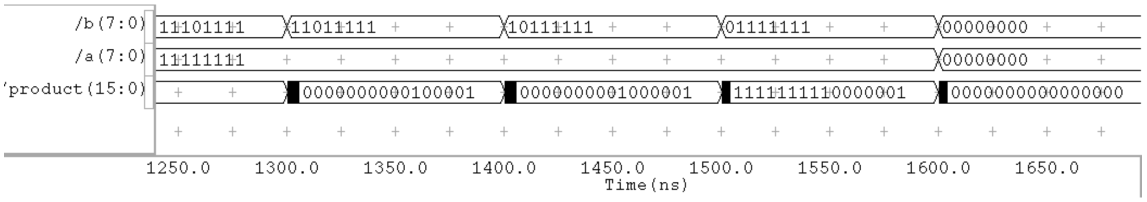


Figure VI-16, Fault simulation for rolling 0 + all 0s

```
CURRENT STATUS
--------------
Total Faults       :  932
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                   :  1700.0ns
Total Faults               :  932
   Untestable Faults       :  0
   Testable Faults         :  932
      Undetected Faults    :  0         (0.00%)
      Detected Faults      :  932       (100.00%)
      Possible Faults      :  0         (0.00%)
      Hyperactive Faults   :  0         (0.00%)
      Hypertrophic Faults  :  0         (0.00%)
      Oscillatory Faults   :  0         (0.00%)
      HM Dropped Faults    :  0         (0.00%)
```

As seen in the grade, all the faults are detected and 100% fault coverage is achieved with just 17 vectors, which are easily produced with a simple BIST scheme. This result is extremely promising compared to the 32897 test patterns of the upcounter and downcounter cases. The achieved histogram and fault coverage plots are also shown in figures VI-17 and VI-18.

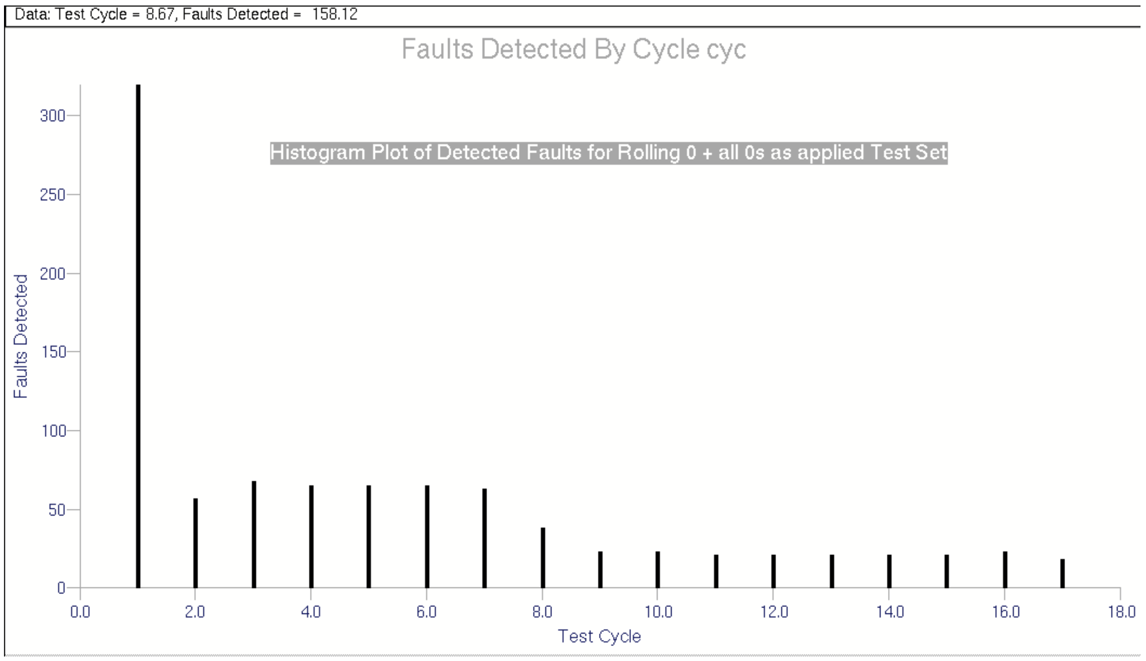


Figure VI-17, Histogram for board level rolling0 + all 0s test

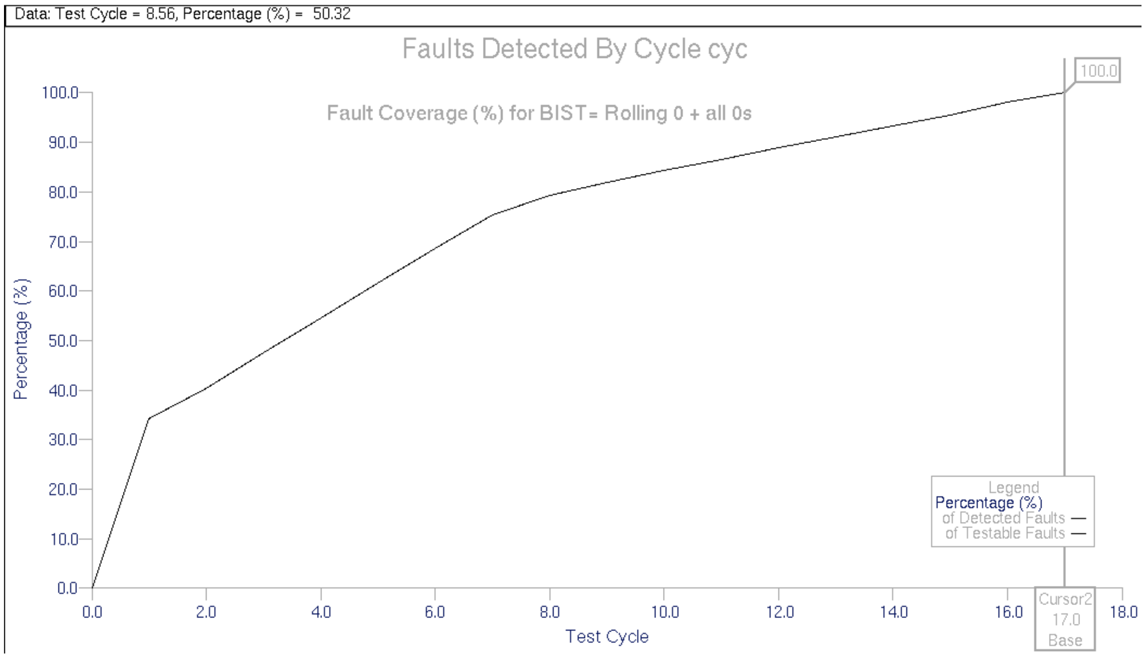


Figure VI-18, Fault coverage (%) for board level rolling0 + all0s test

Although this result seems to have concluded the BIST investigation with an extremely efficient test scheme, one shortcoming discussed previously must be reconciled. The use of board level faulting is very abstract and the results are prone to criticism. Therefore, as a very important milestone in the project, we move forward to hierarchical faulting and verify our results with the new fault models.

VI.4 – BIST FOR 8X8 MULTIPLIER WITH HIERARCHICAL FAULTING

In section VI.3.4, we have found very promising results for the BIST pattern to use for fault testing of the 8x8 multiplier. However, one drawback of the fault simulation technique used was the over-optimistic results of board level faulting as the assumed flawless models are then full adders, half adders etc. some of which are too complex to consider as the primitive blocks. Unfortunately, this is one controversy of fault simulation, as there can be no standard in the selection of primitives as each vendor supplies a different set of primitive blocks, which also makes the compressions between two different fault simulation results by two different parties very cautious to compare. As there is no fixed standard in selection of primitives in fault simulation, the next decision to be made in the project was the selection of the primitives to be used. Considering the board level simulation at one extreme of complexity, we moved to the other possible extreme, the most detailed block level, where the primitives are AND, OR, NOT gates, flip-flops , etc. This was, though over-pessimistic, a thoroughly robust approach as any other model in between of the two extremes would always carry a sign of possible criticism in the fault simulation.

With the described primitive levels, we performed hierarchical fault injection to the 8x8 multiplier circuit, and the total number of injected faults exploded from 932 to 6008, which are displayed on the multiplier schematic in figure VI.19. An immediate difference from the board level faulting as shown in figure VI.6 is the description of faults inside each higher level block which are displayed as up – for stuck at 1 – and down – for stuck at 0 – arrows on the blocks with the number of injected faults written inside the arrows. In order to present the injected errors in the lower hierarchies, each of the higher level blocks are zoomed on figure VI-19 and are printed separately in figures VI-20 to VI-24. As seen in the zoomed figures, the FA and MSBFA circuits have more than 100 faults per instance and therefore have several lower level nodes that can have a stuck at fault.

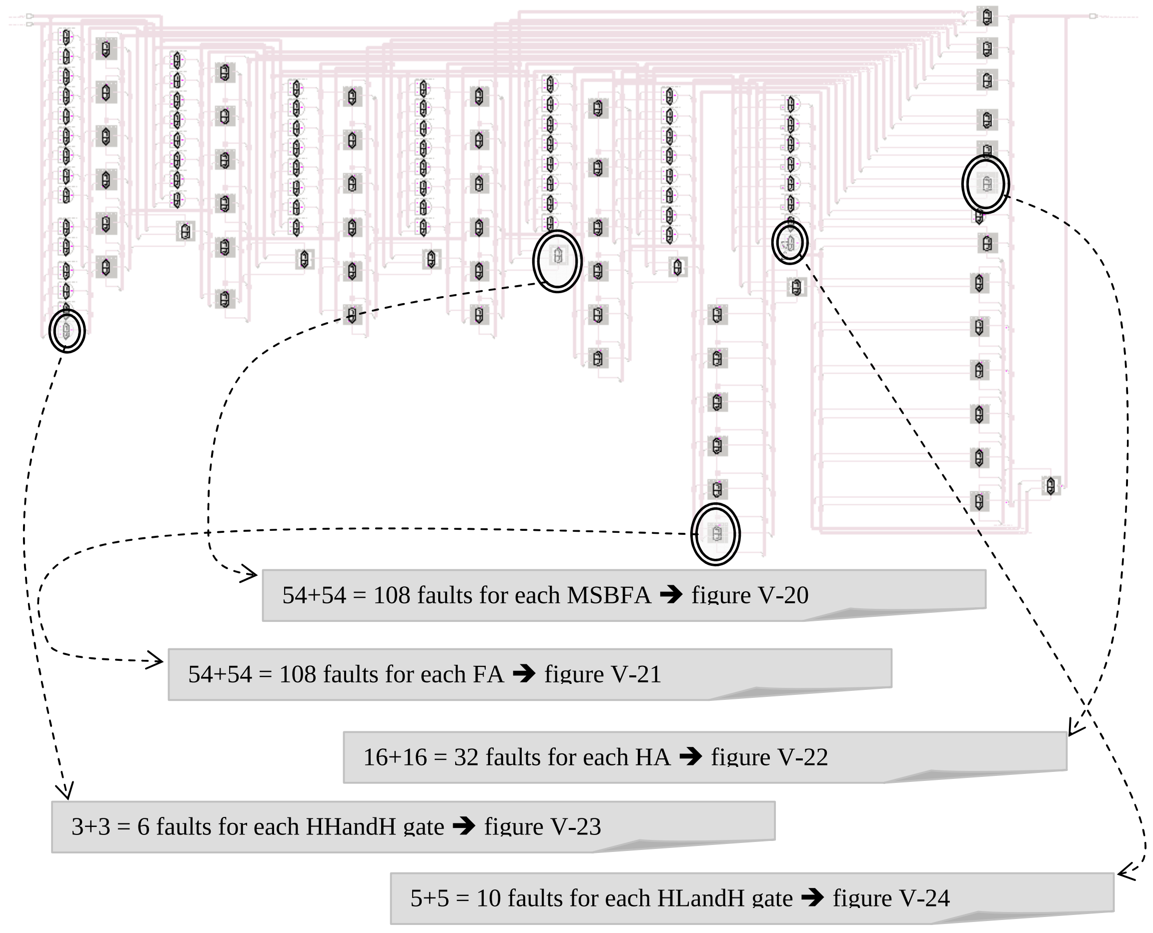

54+54 = 108 faults for each MSBFA ➔ figure V-20
54+54 = 108 faults for each FA ➔ figure V-21
16+16 = 32 faults for each HA ➔ figure V-22
3+3 = 6 faults for each HHandH gate ➔ figure V-23
5+5 = 10 faults for each HLandH gate ➔ figure V-24

Figure VI-19, Hierarchical Fault injection

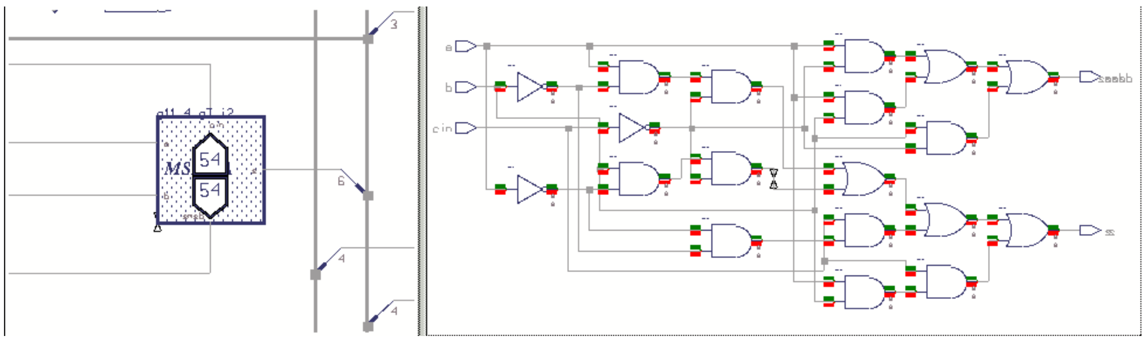


Figure VI-20, Faults inserted to MSBFA

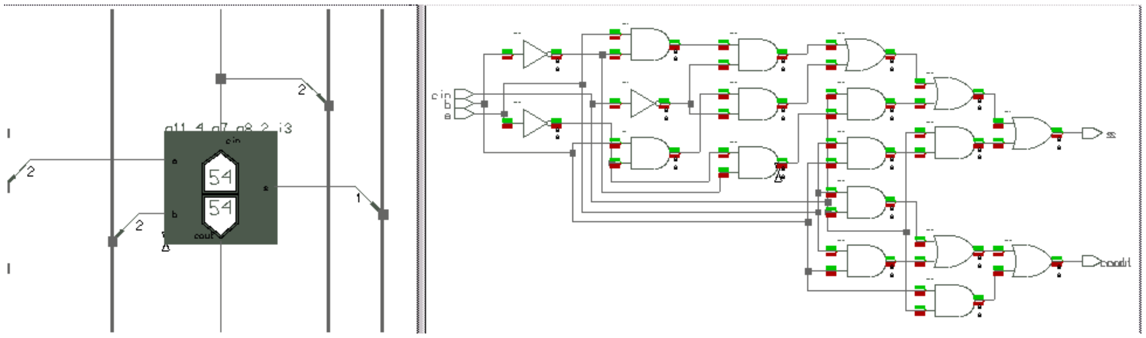


Figure VI-21, Faults inserted to FA

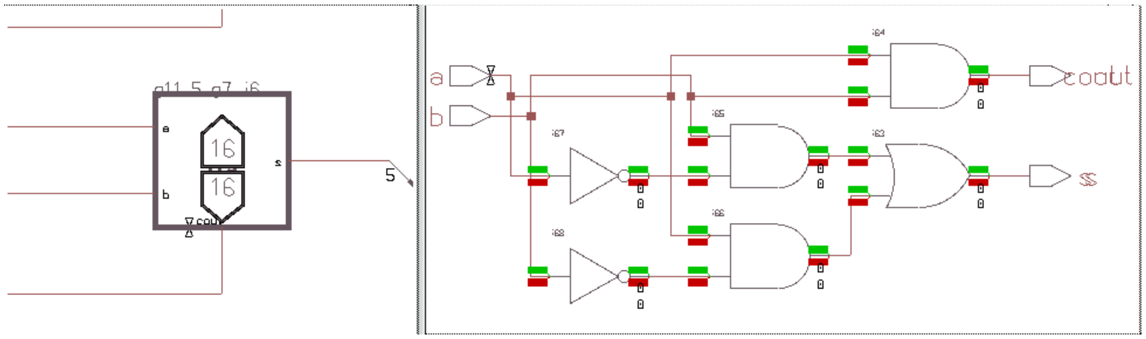


Figure VI-22, Faults inserted to HA

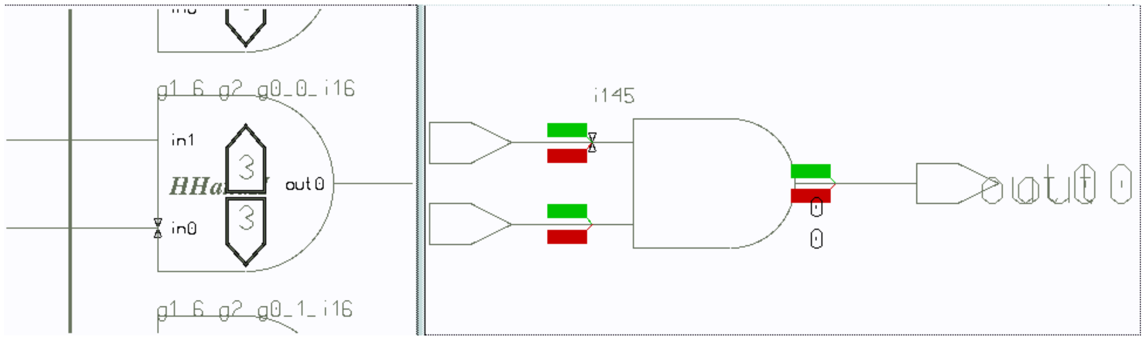


Figure VI-23, Faults inserted to HHandH gate

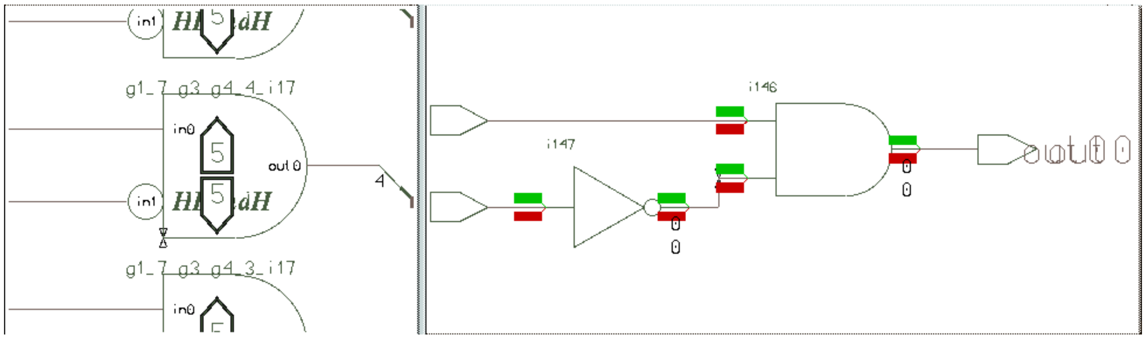


Figure VI-24, Faults inserted to HLandH

Hence, from this point on in the report, all the mentioned BIST techniques are evaluated using hierarchical fault simulation.

VI.4.1 – BIST = 16 bit Rolling 0 + all 0s:

Recalling from section VI.3.4, we had accomplished a 17 vector almost perfect BIST pattern for fully testing the 8x8 multiplier for the 932 board level faults. Now to alleviate any kind of criticism, we perform the same BIST simulation, for the 6008 hierarchical faults. Unfortunately, as the fault grade reveals below, the BIST pattern can detect only 4903 of 6008 patterns with hierarchical faulting.

```
CURRENT STATUS
--------------
Total Faults       :  6008
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                    :  1700.0ns
Total Faults                :  6008
   Untestable Faults        :  0
   Testable Faults          :  6008
      Undetected Faults     :  1105      (18.39%)
      Detected Faults       :  4903      (81.61%)
      Possible Faults       :  0         (0.00%)
      Hyperactive Faults    :  0         (0.00%)
      Hypertrophic Faults   :  0         (0.00%)
      Oscillatory Faults    :  0         (0.00%)
      HM Dropped Faults     :  0         (0.00%)
```

All the fault simulation data and results are in
" CD/MSc/Results/mult8x8/rolling0andall0/hierarsikSim/"

The corresponding fault coverage plot and histogram are shown in figures VI-25 and VI-26, as seen, although the detection is very efficient, the 17 vectors cannot suffice for a high fault coverage for the 6008 faults. Consequently the fault coverage resides at 81.6 %, which is a very unacceptable result, meaning 1 in every 5 of the circuit faults will pass the test undetected. These disappointing results cause the proposed rolling 0 & all 0s to be removed back to shelf and either new methods or effective improvements on this test must be seeked. An observation of the remaining faults (which are in "MScCD/MSc/Results/ /mult8x8/rolling0andall0/hierarsikSim/hierarsikfaultlist.txt" in accordance with the consistent filing) reveals that all the faults in HA, HHandH and HLandH are detected and that, the undetected faults increase toward the significant bit adders.

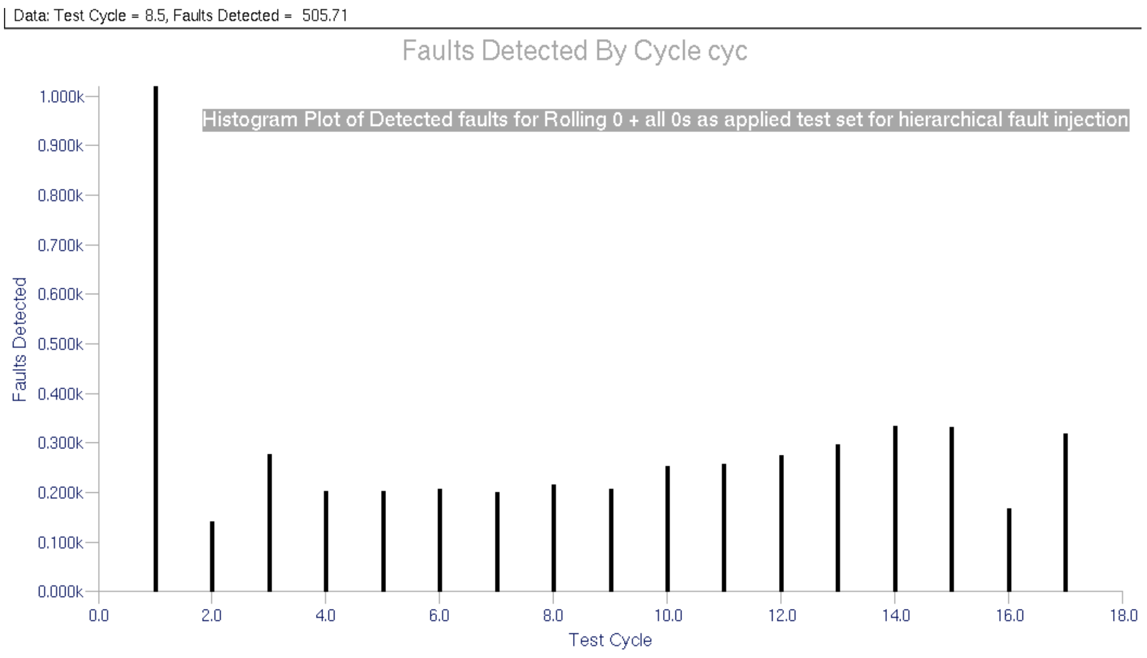


Figure VI-25, Histogram for hierarchical rolling0 + all 0s test

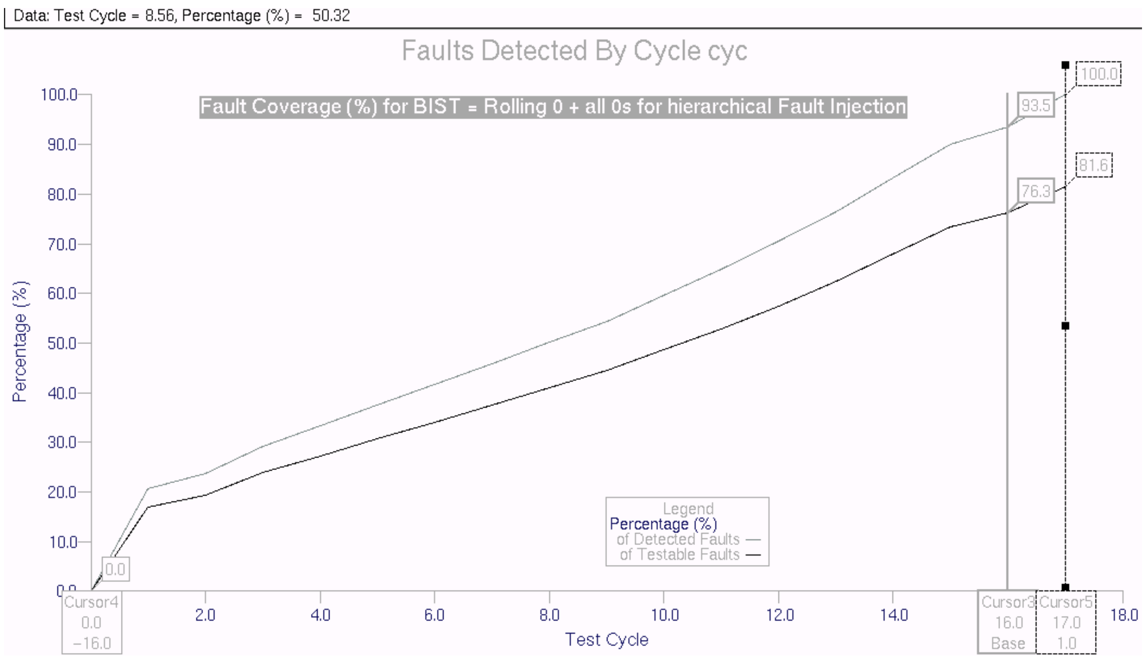


Figure VI-26, Fault coverage (%) for hierarchical rolling0 + all0s test

As the first rescue to the reduced fault coverage, we try to improve the test with several alternatives. Adding a "11…11" sequence is seen to have no effect on fault coverage, while a rolling 1 sequence, described in table 16, provided very low improvement as shown in figure VI-27.

16 bits per pattern

Rolling one:
00…001
00…010
•
•
10…000

Total 16 patterns

Table 16, Rolling 1 pattern

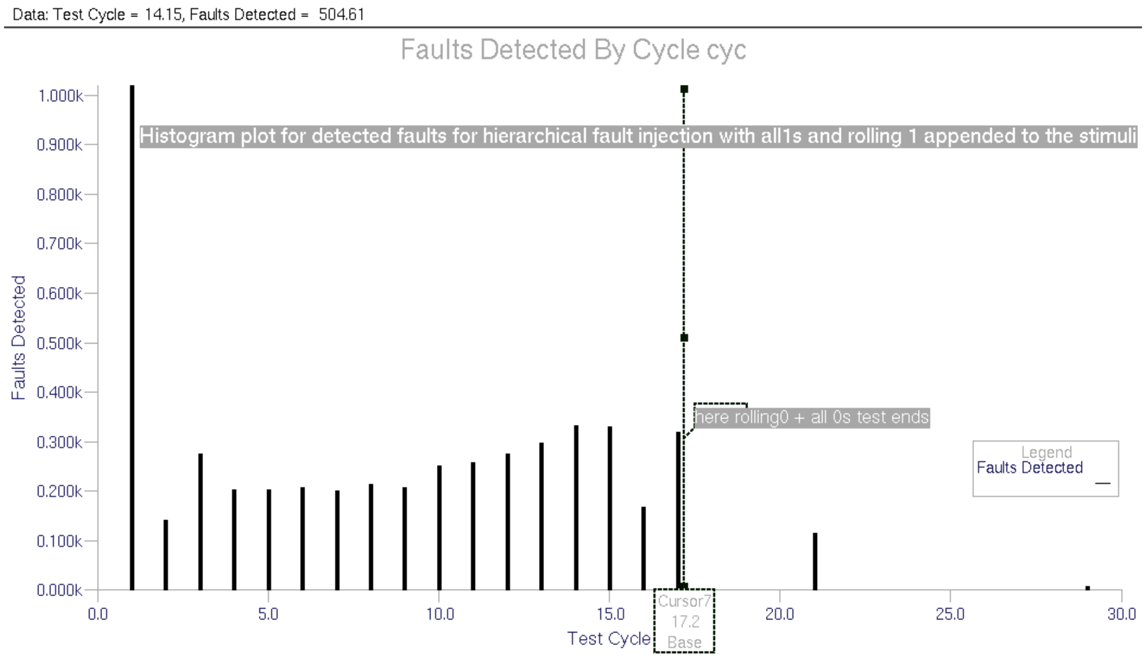

Data: Test Cycle = 14.15, Faults Detected = 504.61

Figure VI-27, Histogram for rolling0 + all 0s test followed by a rolling1 test

Figure VI-27 is a very demonstrative histogram as it displays how ineffective the second test is compared to the first one. These and several other attempts all revealed similar behavior and any kind of postfix on the rolling 0s and all0s test are verified to be ineffective. Then, as a second hope of remedy, other modifications of the test are tried and fault simulations for these tests also revealed slight to no improvements on fault coverage.

For brevity, two of these proposed modifications that are tested are presented below and the corresponding test data are referred from the CD.

<u>Two Rolling 0s:</u> A modification, suggested by Prof. Morling is applying two 8 bit wide rolling 0 sequences in parallel to the a & b inputs of the multiplier. The suggested test set is

described in table 17. The test simulation data and resulting histogram and fault coverage plots as well as simulation trace plot and simulation list are in:
" CD/MSc/Results/mult8x8/tworolling0s/hierarsiksim/"

8 bits per pattern

Rolling zero for a:	Rolling zero for b:
11…110	11…110
11…101	"
•	"
•	"
01…111	"
11…110	11…101
11…101	"
•	"
•	"
01…111	"
●	●
●	●
11…110	01…111
11…101	"
•	"
•	"
01…111	"

8 patterns of a for each b pattern

Total 64 patterns

Table 17,Two Rolling 0s pattern

The resulting fault grade is:

```
CURRENT STATUS
--------------
Total Faults       :  6008
Unsimulated Faults :  0

RESULTS FROM LAST RUN
---------------------
Run Time                     :  6500.0ns
Total Faults                 :  6008
   Untestable Faults         :  0
   Testable Faults           :  6008
      Undetected Faults      :  729       (12.13%)
      Detected Faults        :  5279      (87.87%)
      Possible Faults        :  0         (0.00%)
      Hyperactive Faults     :  0         (0.00%)
      Hypertrophic Faults    :  0         (0.00%)
      Oscillatory Faults     :  0         (0.00%)
      HM Dropped Faults      :  0         (0.00%)
```

Although this result improves fault coverage to 87.9 %, it is still a very low fault coverage. Further additions appended to this set such as a "11…11" sequence at the end seems to improve the performance insignificantly.

Two Rolling 0s + all 1s for each b pattern: This second alternative was considered as the first alternative does not completely cover the rolling 0s and all 0s sequence, as there are two zeros per pattern except for all 0s and all 1s sequences. To cover the single rolling 0s pattern, an all 1s 8 bit pattern is added at the end of each 'a' pattern – counting up to 9 patterns per b pattern now – and a separate all 1s b pattern is also included. This new pattern is described in table 18, and comparing with table 17, the additions are highlighted for easy observation.

8 bits per pattern

Rolling 0 + all 1s for a:	Rolling 0 + all 1s for b:
11…110	11…110
11…101	"
•	"
•	"
01…111	"
11…111	"
11…110	11…101
11…101	"
•	"
•	"
01…111	"
11…111	"
●	●
●	●
11…110	01…111
11…101	"
•	"
•	"
01…111	"
11…111	"
11…110	11…111
11…101	"
•	"
•	"
01…111	"
11…111	"

9 patterns of a for each b pattern

Total 81 patterns

Table 18,Two Rolling 0s + all 1s for each b pattern

Once again, this alternative is also observed to improve fault coverage insignificantly. Moreover, as can be observed from the last 3 tables, the complexity of the generated pattern is also becoming more complex with the new modifications and add-ups. As result of all these, with great regret, we drop the rolling 0 test as the ultimate BIST proposal after the above elaborations for hierarchical faulting.

VI.4.2 – Benchmark BIST = 16 bit Downcounter:

Having observed that faulting level effects the determination of the BIST sequence, we start with a 16 bit fully exhaustive downcounter test, which will serve us as a benchmark when evaluating other strategies. One of the expectations of this fault simulation is to observe that we won't be able to achieve 100% fault coverage with even full-exhaustive testing, as the hierarchical design procedure might inherently involve some untestable nodes at this detailed level. The fault simulation data and results are in:
" CD/MSc/Results/mult8x8/downcount/hierarsiksim/"

The simulation list file:"CD/MSc/Results/mult8x8/downcount/hierarsiksim/ /downcountsimlist.txt" also presents a full simulation of 8x8 multiplier, where the radix are displayed in signed decimal format for easy observation.

As seen in the fault grade ("*faultsummary.txt*"), the fault coverage is 98.49% after all the possible patterns are applied. Therefore, no test can exceed this value of fault coverage. This also raises one counterargument against Cell Fault Model and Cell Fault Coverage (CFC) described in [3]. It is suggested on [3,p. 947] that single stuck at fault simulation fault coverage values are always larger than the CFC values. However, the CFC value of 99.40% for the 8x8 CPA multiplier given on p. 945 is not even achievable with single stuck at model with the primitives we use. As a conclusion, we disagree with [3] that CFC is always more pessimistic than single stuck at model and we assert that this rather depends on the primitive levels used in hierarchical faulting. As another inconsistency, the Verifault values used for comparison in p. 947 are referred without presenting the used primitive level and therefore should be regarded with caution. $\Leftarrow$ This also reemphasizes the difficulty in comparing two different published materials.

The histogram and fault coverage plots for the downcounter are shown in figures VI-28 and VI-29.

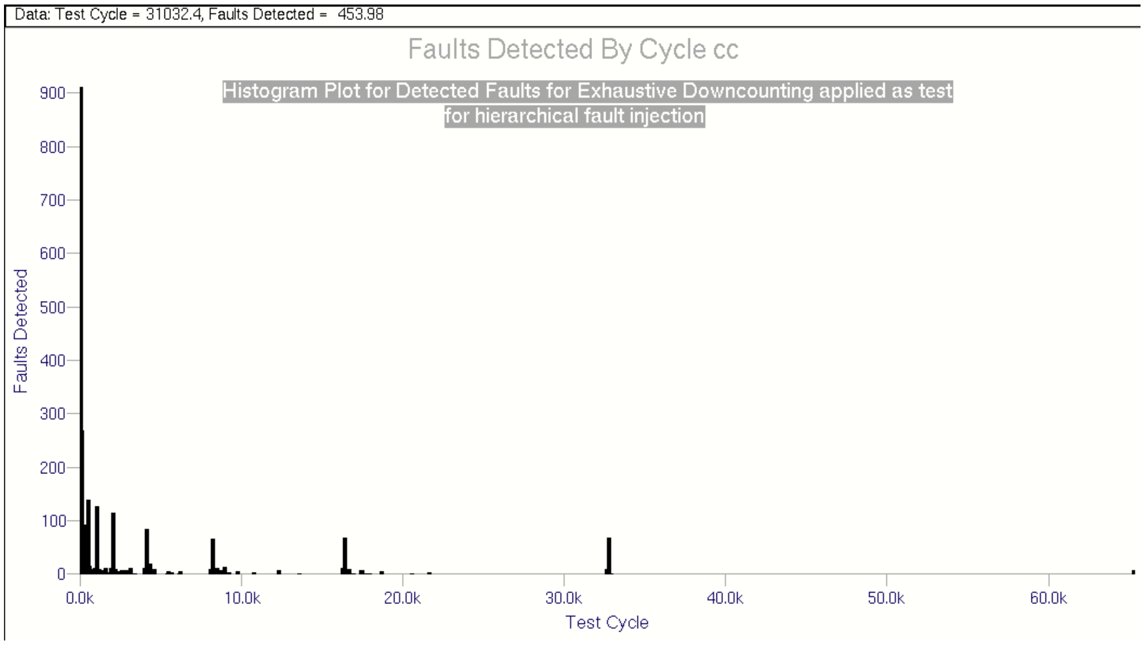


Figure VI-28, Histogram for hierarchical Exhaustive Downcounter test

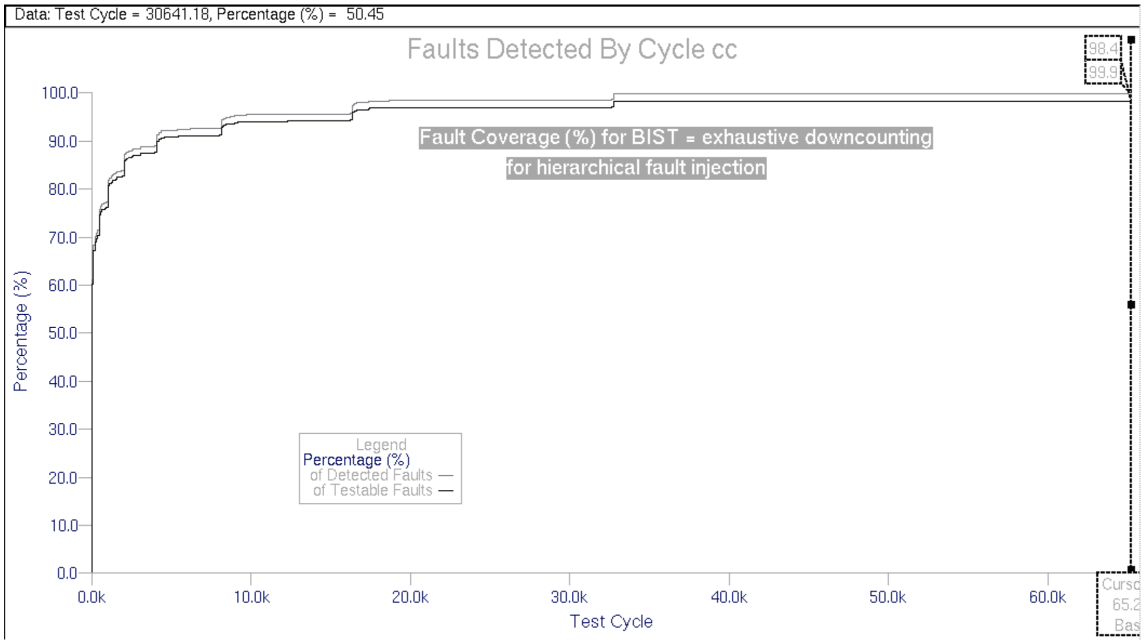


Figure VI-29, Fault coverage (%) for hierarchical Exhaustive Downcounter test

Once again as in previous downcounter test, huge amount of redundancies are observed in both histogram and fault coverage plots. Interestingly, the 32897th sample once again peaks within all other samples. This points to a specific importance of 32897. sample once again. We'll try several new scenarios to reduce these and will try to end up with an efficient, effective, compact test set, which is easy to generate for BIST application. To gain insight to which faults are undetected even with full exhaustive test, we include figure VI-30 which demonstrates the locations of undetected faults.

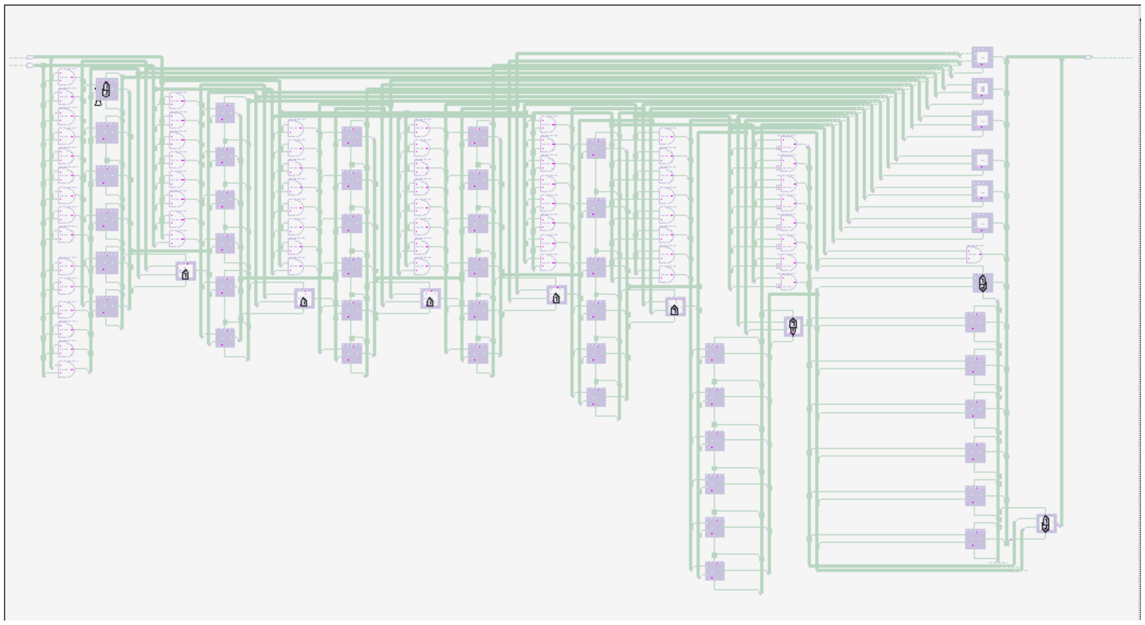
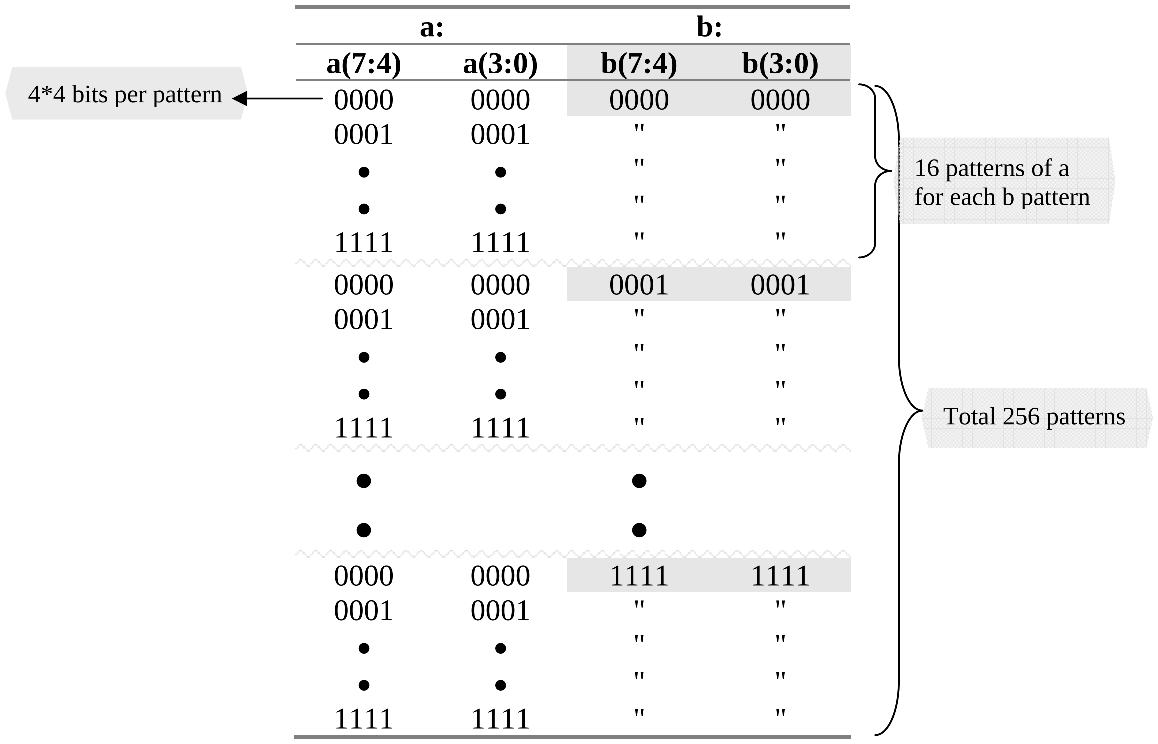


Figure VI-30, Locations of undetected faults for downcounter test in 8x8 multiplier

Interestingly, all the MSBFA adders have invariantly some undetected stuck at 0 faults, which points to a redundancy in the MSBFA design. $\Leftarrow$

VI.4.3 –BIST = Repetitive Patterns - Upcounter:

Repetitive patterns have been widely used in testing of iterative logic arrays and have been shown to provide excellent fault coverage with a small test set ([3],[5],[23]). For the multiplier, the a and b inputs of multiplier are applied the same patterns for every k inputs where k is the defined *"repetition length"*. If we apply the same pattern to both lower and higher 4 bits of one of the 8 bit inputs in our 8x8 multiplier, the repetition length of this pattern is 4. To verify the effectiveness of repetitive patterns we apply a BIST scheme with a repetition length of 4 and an 8 bit upcounter chosen as input pattern generator. The corresponding input test vectors are then as shown in table 19.



a:		b:	
a(7:4)	**a(3:0)**	**b(7:4)**	**b(3:0)**
0000	0000	0000	0000
0001	0001	"	"
•	•	"	"
•	•	"	"
1111	1111	"	"
0000	0000	0001	0001
0001	0001	"	"
•	•	"	"
•	•	"	"
1111	1111	"	"
●		●	
●		●	
0000	0000	1111	1111
0001	0001	"	"
•	•	"	"
•	•	"	"
1111	1111	"	"

4*4 bits per pattern
16 patterns of a for each b pattern
Total 256 patterns

Table 19,Two Repetitive pattern with repetition length(k)=4,
and pattern generator → Upcounter

Hardware consideration for this test set can be finalized with a single 8 bit upcounter as:

b(3:0)&&a(3:0)

and

b(7:4)&&a(7:4)

can be concatenated and the 8 bit counter outputs can be directly connected each of the 8 ports.

With this pattern generation scheme, an excellent fault coverage is achieved with the mere 256 described vectors. As can be referred from: "CD/MSc/Results/mult8x8/greekbist/hierarsiksim/"

The fault grade reveals a 97.02 % fault coverage and as seen in the histogram and fault coverage plots in figures VI-31 and VI-32, 228 cycles/patterns is sufficient for this coverage. Both the histogram and plot show how effective the test set is, most of the patterns contributing to the fault coverage.

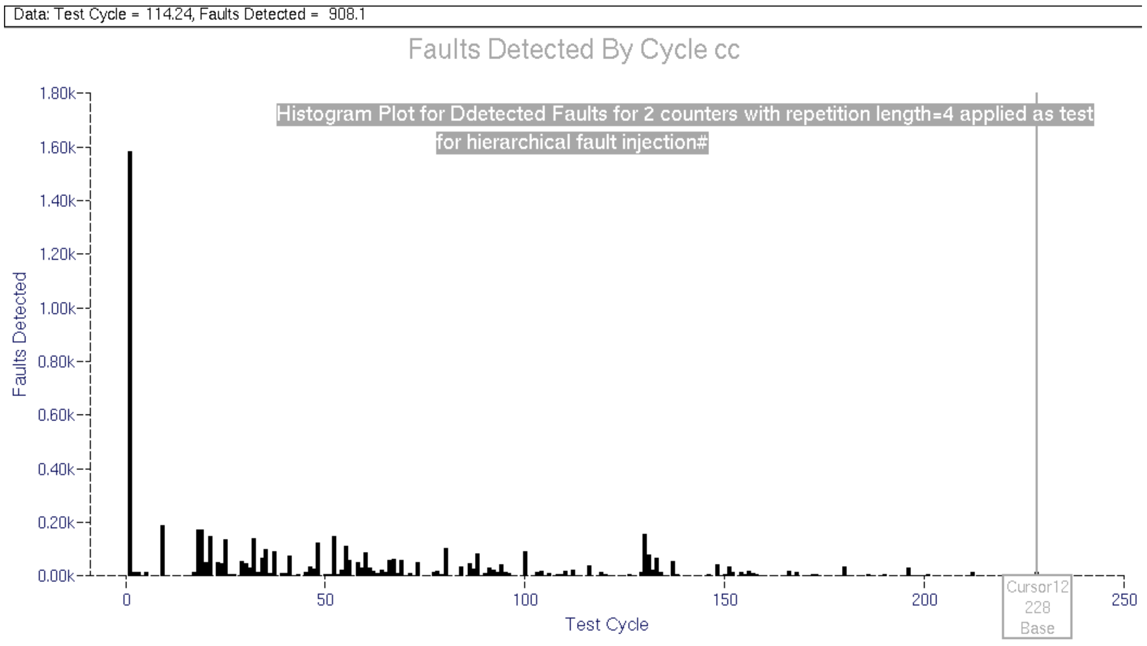


Figure VI-31, Histogram for Repetitive upcounter with k=4

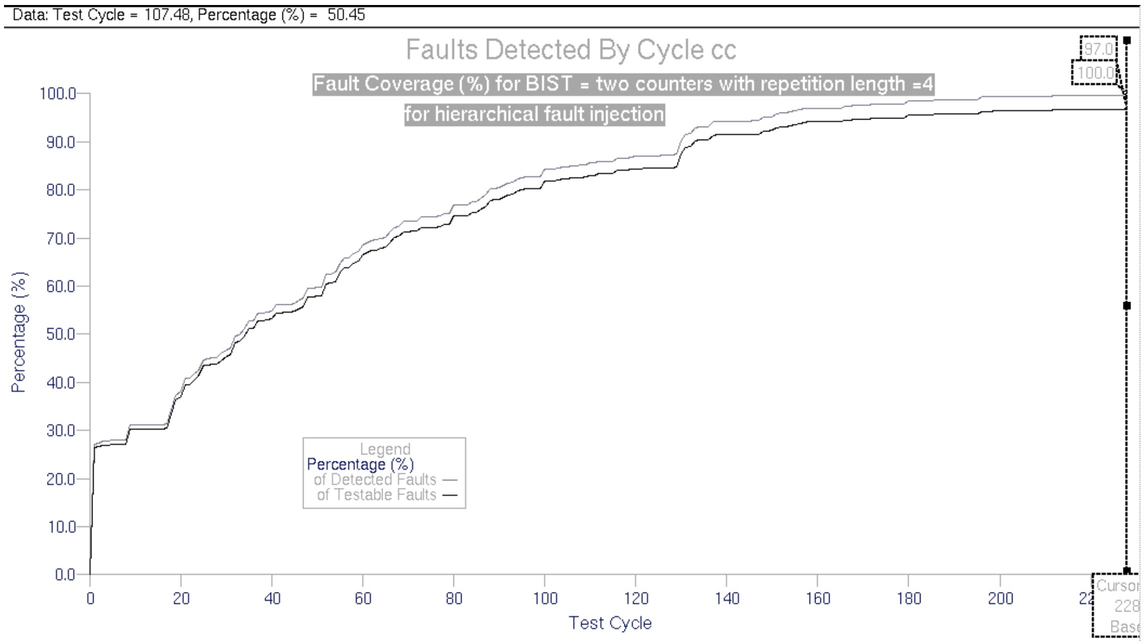


Figure VI-32, Fault coverage (%) for Repetitive upcounter with k=4

Both the histogram and the plot also indicate two other outcomes. There are 2 very ineffective regions right at the start of test, within the first 20 input test vectors and there is a very effective input sequence starting around 127th input test vector. This redundancy observation and peak information, will later be used to even further improve the test set.

This extremely promising result is the best of the achieved so far, and will be considered as the limiting threshold for the following investigations.

VI.4.4 –BIST = 16 bit LFSR:

Moving from deterministic techniques to pseudorandom techniques, we initialize one of the major interests of the project, pseudorandom techniques as BIST for signed parallel multiplier. As the first obvious case to scrutinize we use a 16bit LFSR with seed = x0001, as the hypothetical pattern generation circuit. In order to be able to investigate several LFSR variations, i.e. seed, taps, length, etc., we first wrote a Matlab script that produces the LFSR output and creates an input stimulus file ("*dofile in QuickFault"*) for the multiplier inputs. The first example of the LFSR used with taps at flip-flops 16,5,3,2 for max-length PRBS, is displayed in figure VI-33, with the variable names and indices used in the Matlab script.

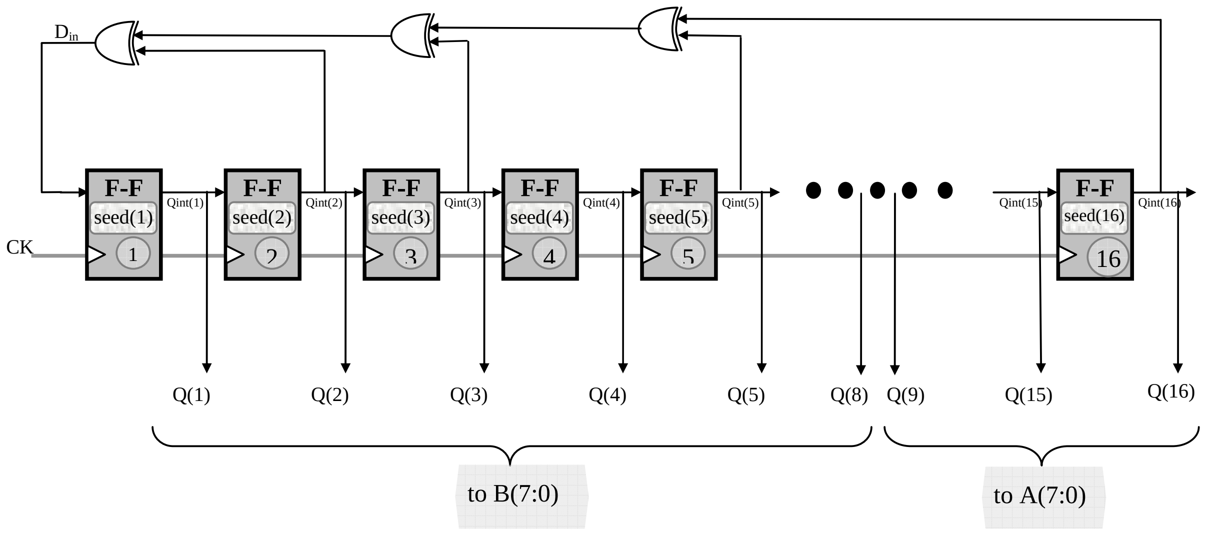


Figure VI-33,The emulated LFSR by the Matlab Script

The generated Matlab script and the written function bin2hex are in appendix E-2 and also can be referred from “CD>MSc/DesignFiles/matlab”. The generated output stimuli for different seeds are in "CD/MSc/Results/mult8x8/LFSR/LFSR16bit/seedxxxx" directories.

With the first seed = x0001, we performed fault simulation for the 8x8 multiplier. The simulation data and results are in: “CD/MSc/Results/mult8x8/LFSR/LFSR16bit/seed0001/”. With the full exhaustive simulation, the fault coverage reached 98.98%. Unexpectedly, the fault coverage settled above 98.49 %, the value achieved by full exhaustive downcounter test. There is no rational explanation to this as the same vectors are applied with only a different sequence. There are no sequential faults in the combinational circuit and this discrepancy is attributed to the software. The histogram and plot are as in figures VI-34 and VI-35.

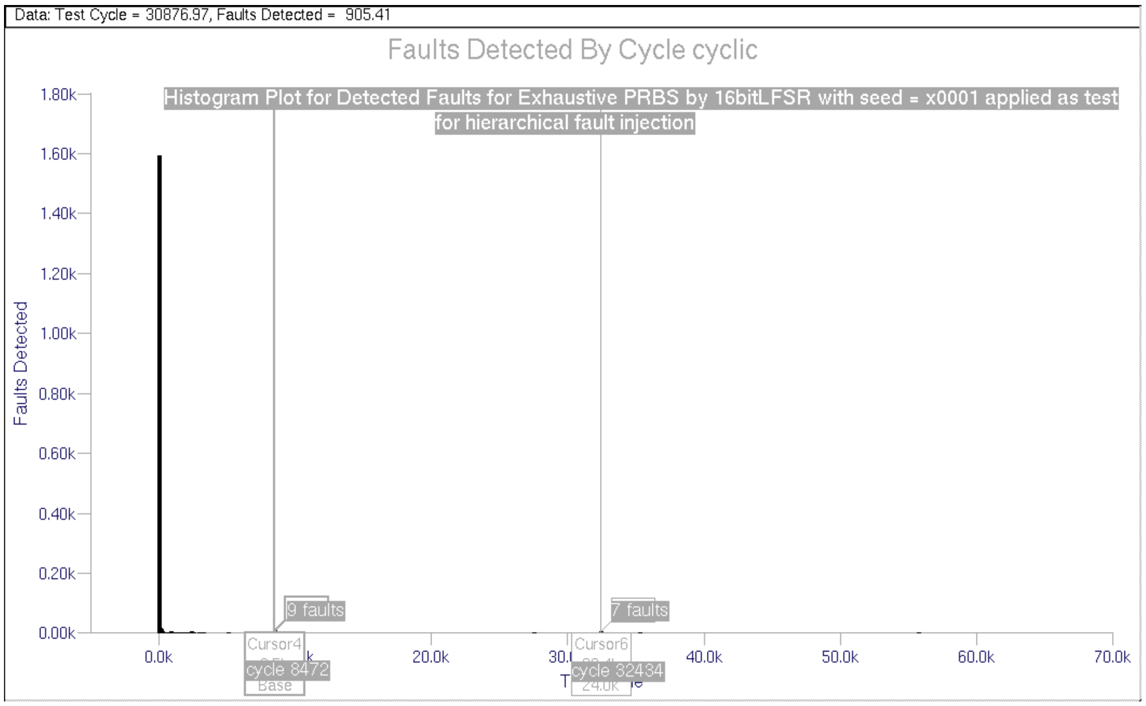


Figure VI-34, Histogram for 16 bit LFSR with seed=x0001

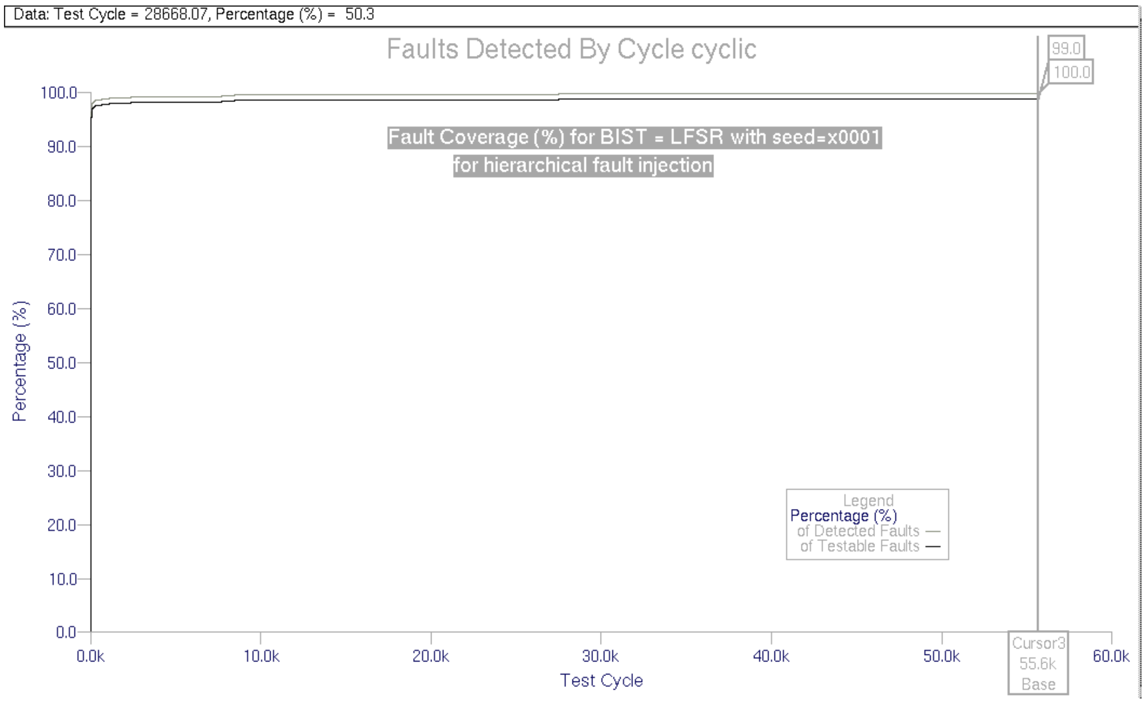


Figure VI-35, Fault coverage (%) for 16 bit LFSR with seed=x0001

As seen in the fault coverage plot, the curve is almost like a step function with almost all faults detected just at the start of the test. We include two more cursors in the histogram in order to demonstrate the cycles with high detection after the hyperactive start phase. These will be used in the determination of seeds for the LFSRs. In order to compare the result with the repetitive upcounter with k=4 in previous section, we zoom at the fault coverage plot to the initial jump, and determine at which cycle fault coverage reaches 97.02%. The zoomed fault coverage plot is shown in figure VI-36.

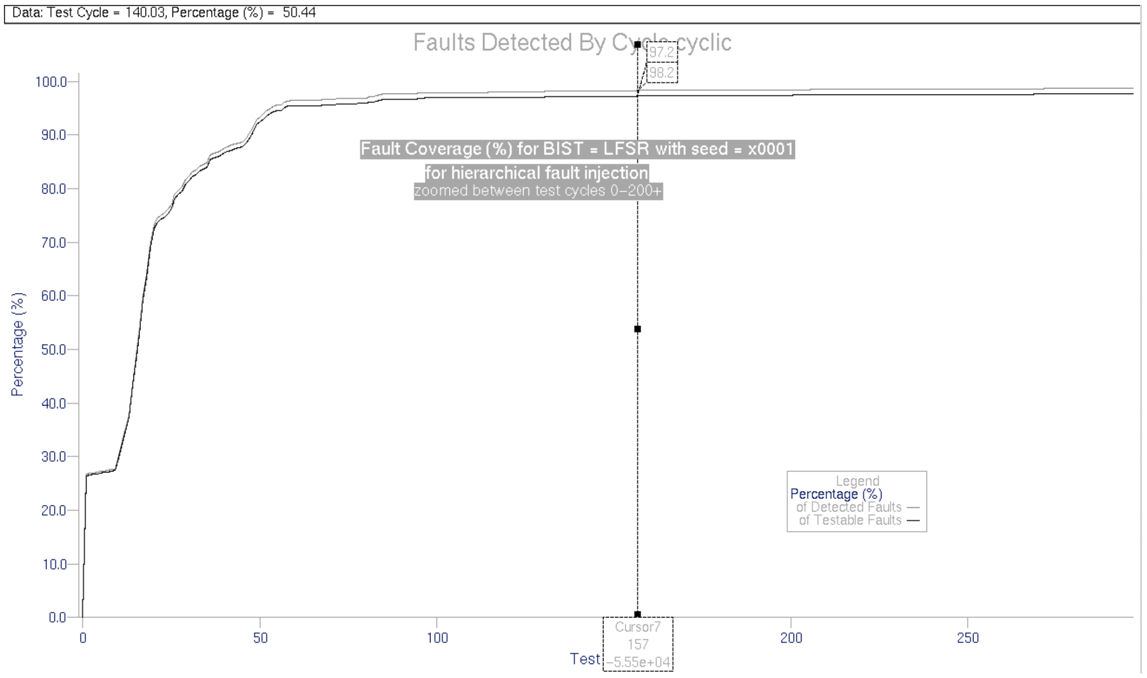


Figure VI-36, Zoomed Fault coverage (%) for 16 bit LFSR with seed=x0001

As seen, with just 157 cycles, the fault coverage reaches 97.2%[2]. We also include a zoomed portion of histogram in figure VI-37 to demonstrate the initial hyperactive phase and to point some of the redundancies in the test. As seen in the histogram, there is a long inactive phase right after start, and then the hyperactivity restarts, which is also indicated by the initial flat region in the zoomed fault coverage curve in figure VI-36. Moreover, the full histogram in figure VI-34 reveals two very important cycles in the detection after the initial phase, as the cursors show, these are cycles 8472 which detects 9 faults and 32434, which detects 7 faults, making the peaks in the almost saturated region of detection. A good methodology for seed determination might then be starting from these values as seed and *collapse the initial hyperactive regions with the late peaks.*

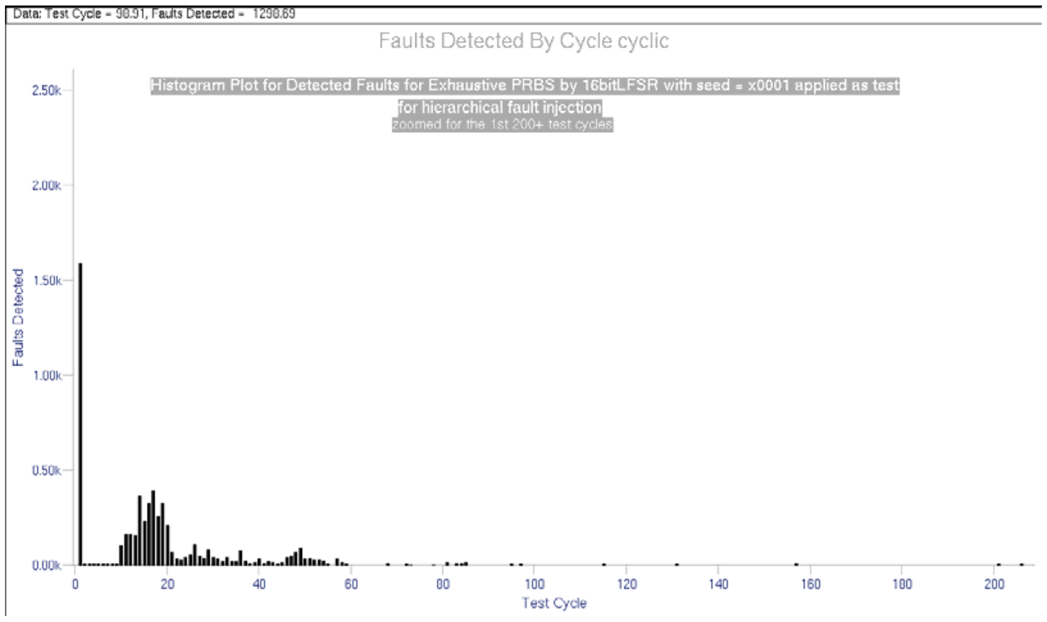


Figure VI-37, Zoomed Histogram for 16 bit LFSR with seed=x0001

As the LFSR lacks the all 0 pattern, we also applied this pattern to the end of test and it is seen to have no effect on fault detection. Therefore, application of DeBrujn counter is not needed to improve performance.

As stated, the two late peaks are used to determine an effective seed for the LFSR. First of these peaks is at cycle 8472, which detects 9 faults. To determine the corresponding LFSR value at this cycle, we determine the simulation time at this cycle and we refer to the simulation list file in:
"CD/MSc/Results/mult8x8/LFSR/LFSR16bit/seed0001/LFSR16bitsimlist.txt"

The corresponding LFSR value is then applied as the seed. For 8472nd cycle, as we strobe the outputs at +90ns after application of each vector:

Simulation time = 8472*100 – 10 = 847190 ns

And from the simlist file:

[2] Hence this is 97.2% rather than 97.02% of previous test, so even smaller cycles can achieve 97.02%

```
                •
                •
846990.0 A9   61   DF09
847090.0 D4   B0   0DC0
847190.0 EA   58   F870
847290.0 F5   2C   FE1C
847390.0 7A   96   CD7C
                •
                •
```

Time(ns) **^/b(7:0)**

^/a(7:0)

^/product(15:0)

➔ seed = B&&A = EA58

Seed = xEA58: With this new determined seed, the performed fault simulation data and results are in: "CD/MSc/Results/mult8x8/LFSR/LFSR16bit/seedEA58/". The data is not included in the report for brevity except for the zoomed fault coverage plot, which is in figure VI-38.

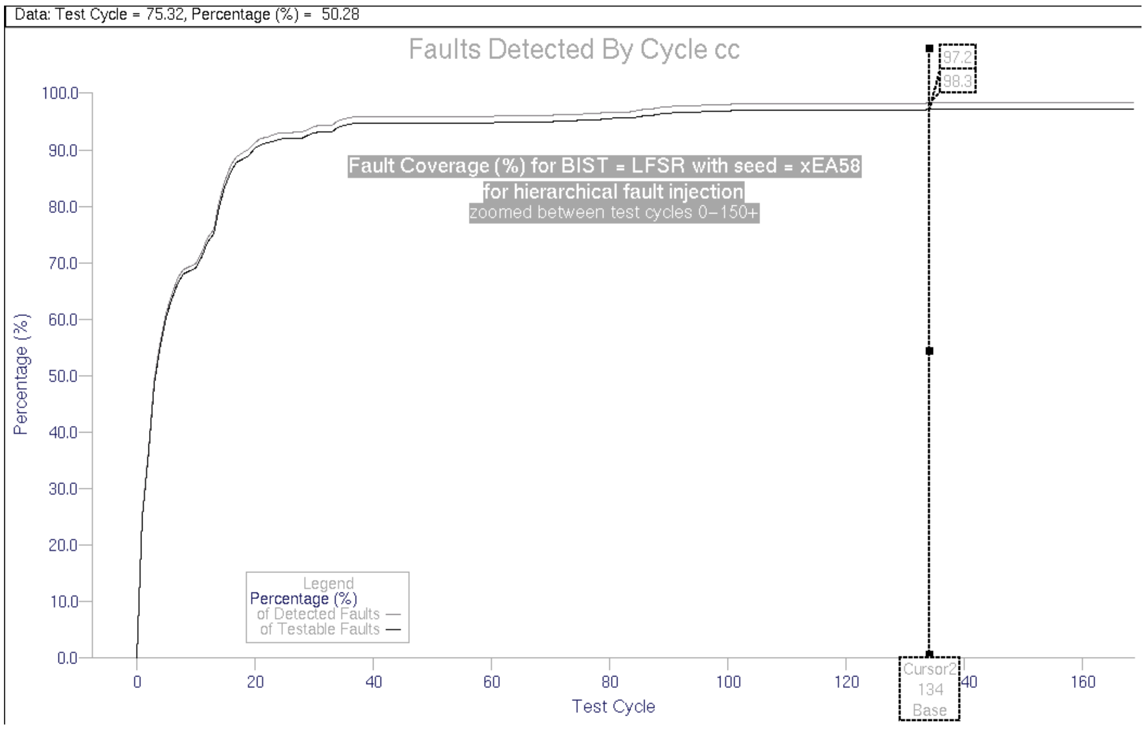


Figure VI-38, Zoomed Fault coverage (%) for 16 bit LFSR with seed=xEA58

As seen in the fault coverage figure, 97.2% fault coverage is achieved with only 134 patterns and the flat region of the previous fault coverage curve is not observed any more. The histogram plot, which can be referred from the CD shows no new late peaks, except for the obviously expected one in 32434 – 8472 +1 = 23963. cycle, which is displayed in the histogram plot.

Hence the other peak is at 32434th cycle corresponding to a simulation time of:

32434*100 – 10 = 3243390ns

The corresponding seed from the simulation list is x8080.

Seed = x8080: The fault simulation data and results for seed x8080 are in: "CD/MSc/Results/mult8x8/LFSR/LFSR16bit/seed8080/". The fault coverage curve reveals 97.2% fault coverage in 153 cycles, which can be referred from the CD. Although still slightly better from the x0001 seed, this seed does not provide as an efficient fault detection as seed xEA58.

This concludes the investigation for 16 bit LFSR and we move forward to observe fault coverage behavior of LFSR used as a repetitive pattern generator as BIST.

VI.4.5 –BIST = 8 bit LFSR – Repetitive Pattern:

In the previous two sections we have observed two very efficient techniques; in section VI.4.3 we have seen how repetitive patterns are effective in fault detection with only a small number of test vectors while in section VI.4.4 we have seen how pseudorandom techniques improve fault detection compared to the deterministic techniques. In this section we try to combine the two techniques to improve performance even further. We use an 8 bit LFSR to generate input patterns for a and b inputs of multiplier. The pattern repetition length is 4, and therefore, a(3:0) and a(7:4) have the same inputs: the rightmost 4 output bits of the LFSR and similarly, b(3:0) and b(7:4) have the leftmost 4 bits of LFSR output.

We once again use a similar Matlab script to generate the dofile for stimuli with different seed and taps. The generated Matlab script is in appendix E-3 and also can be referred from "CD>MSc/DesignFiles/matlab". The generated output stimuli for different seeds are in " CD/MSc/Results/mult8x8/LFSR/LFSR8bit/seedxx" directories. The schematic representation of the Matlab script, with the labeled Matlab variables is in figure VI-39. As can be deduced from the figure, the taps are at flip-flops 8,6,5 and 1 and the below case represents a seed of “01111011”.

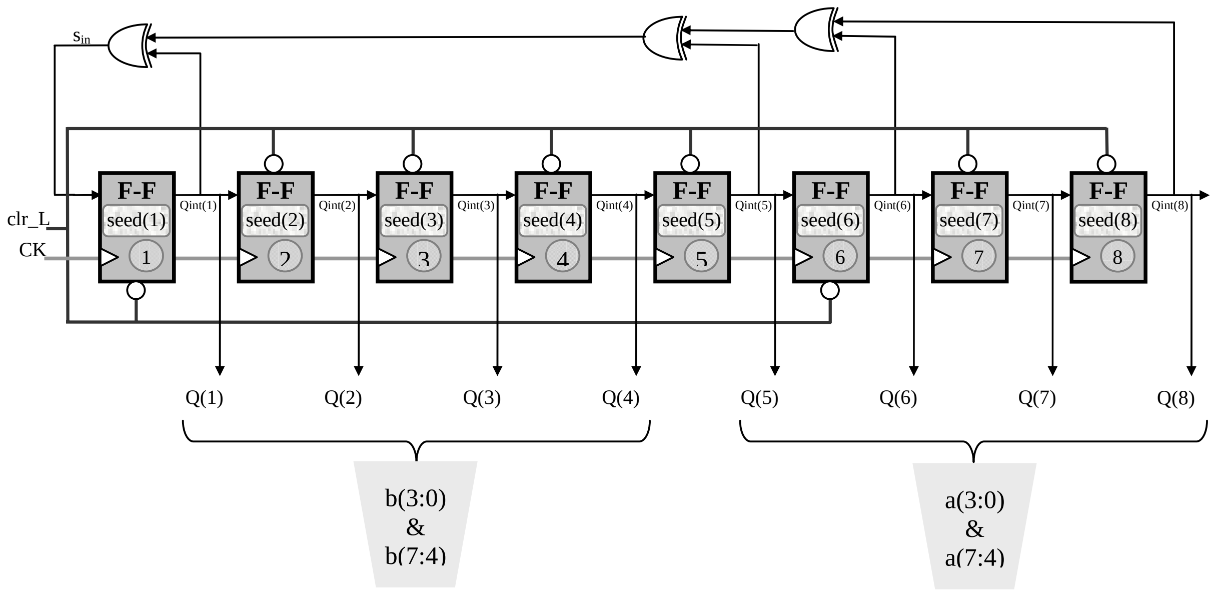


Figure VI-39,The emulated 8 bit LFSR by the Matlab Script

With this repetition length 4, the maximum number of different patterns = $2^4 * 2^4 = 2^8$ minus the all 0 forbidden case of LFSR, summing up to total of 255 patterns, which is already very compact for a 16 input circuit test. We start with a trivial seed of x01 and then follow the 'late peaks' procedure discussed in previous section.

Seed = x01: The first fault simulation with seed x01 is performed as a start point to determine an effective seed. The simulation data and results are in: "CD/MSc/Results/mult8x8/LFSR/LFSR8bit/seed01/". As seen in the fault coverage plot in figure VI-40, 97.12% fault coverage is achieved in 228 cycles, which is more than 70% worse than 16 bit LFSR with seed xEA58, however, the BIST hardware is reduced to half. The histogram plot is displayed in figure VI-41 and a few informative observations can be deduced. Firstly, as in the 16 bit LFSR case, the 00…01 seed causes a redundant period right after the start of test, secondly, the histogram indicates another redundant region between cycles 60 and 70. Afterwards, a strong burst of detection starts around cycle 75, which might be a good starting point for the test as well as another significant peak – yet not a long burst – in 166th cycle. The patterns corresponding these two cycles are good candidates of possible effective seeds.

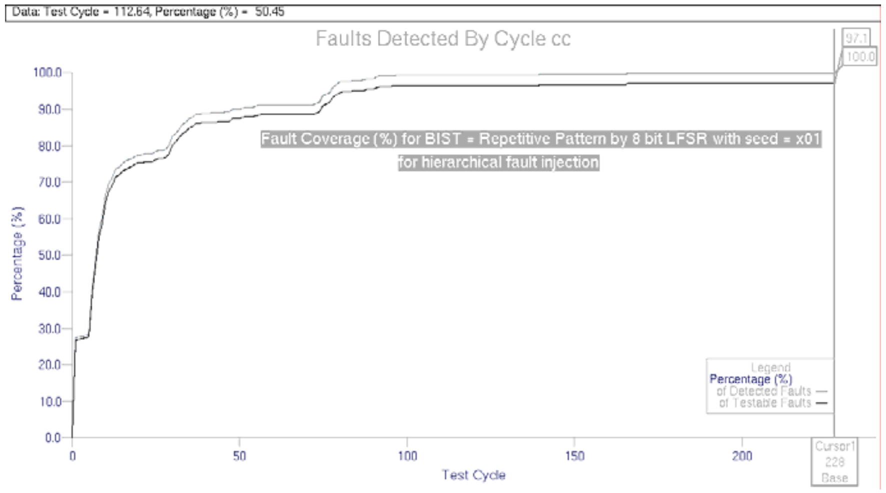


Figure VI-40, Fault coverage (%) for 8 bit LFSR with repetition length, k =4 and seed=x01

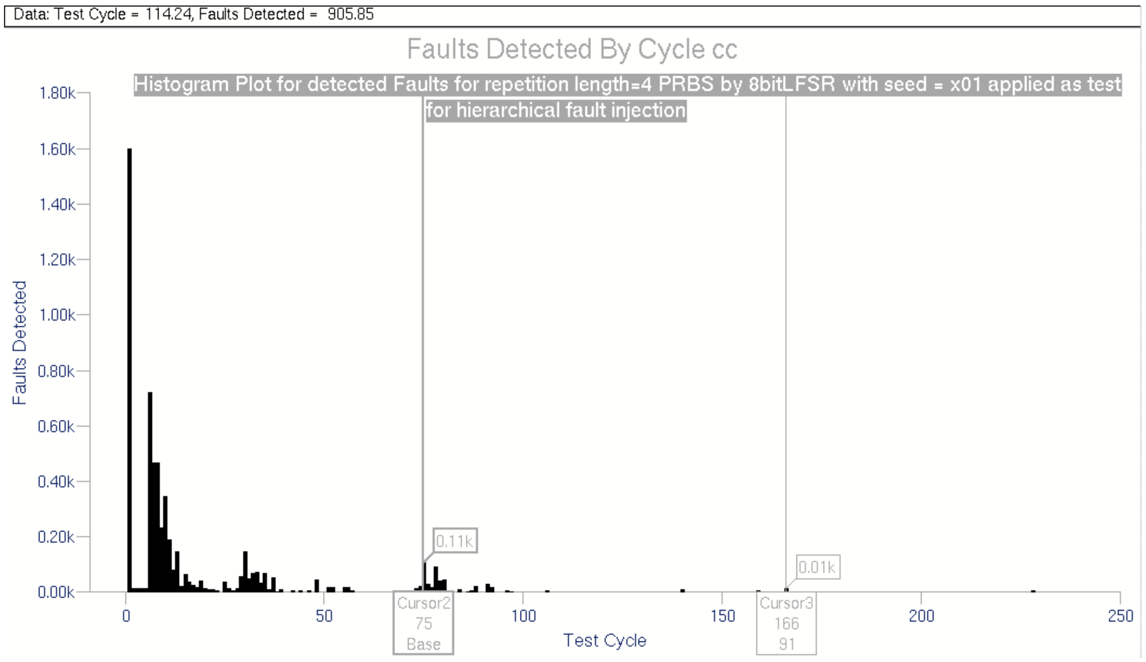


Figure VI-41, Histogram for8 bit LFSR with repetition length, k =4 and seed=x01

Once again from the simulation list, 75th cycle corresponds to 7490ns and 166th cycle corresponds to 16590 ns. The corresponding b&&a values are 77&&BB and BB&&77. Thus, the corresponding seeds are 7B and B7.

```
        •
        •
7390.0 FF  66  FF9A
7490.0 77  BB  DFED
7590.0 33  DD  F907
        •
        •
16490.0 66  FF  FF9A
```

```
16590.0 BB  77  DFED
16690.0 DD  BB  096F
        •
        •
Time(ns) ^/b(7:0)
              ^/a(7:0)
                    ^/product(15:0
```

Seed = x7B: The simulation results and data for seed = x7B are in: "CD/MSc/Results/mult8x8/LFSR/LFSR8bit/seed7B/". The fault coverage plot shown in figure VI-42 reveals 97.2% fault coverage in just 154 cycles and 97.1% fault coverage in just 109 cycles.

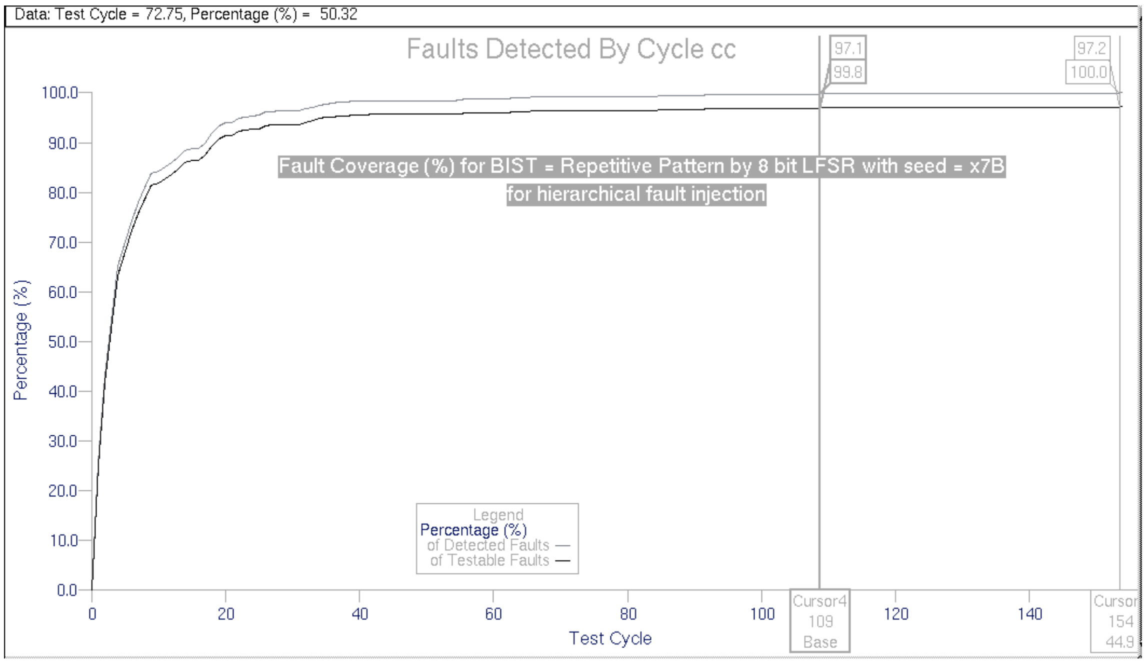


Figure VI-42, Fault coverage (%) for 8 bit LFSR with repetition length, k =4 and seed=x7B

As can be seen in the figure, the fault coverage is almost saturated after around 40 cycles and there is almost no redundancy as there are no flat regions in the ascending part of the curve, which can be verified from the histogram plot in the CD. This test reveals very good test coverage with minimal hardware and test complexity.

Seed = xB7: The simulation results and data for seed = xB7 are in: "CD/MSc/Results/mult8x8/LFSR/LFSR8bit/seedB7/". The fault coverage plot shown in figure VI-43 reveals an excellent 97.0% fault coverage in just 82 cycles and 97.24% fault coverage in 230 cycles.

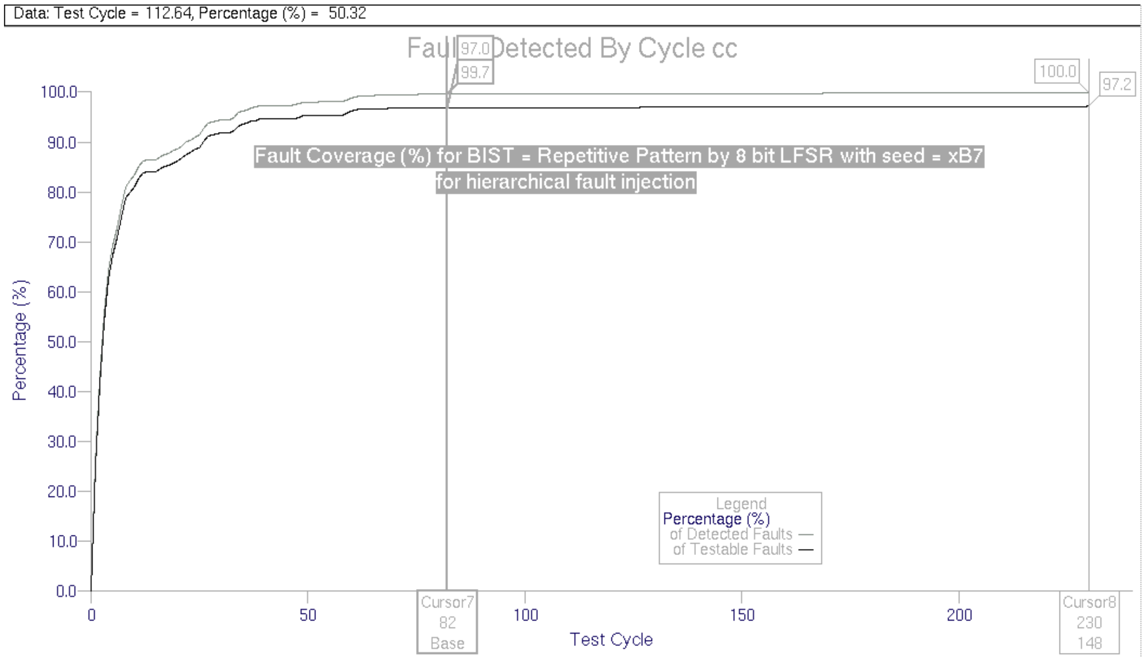


Figure VI-43, Fault coverage (%) for 8 bit LFSR with repetition length, k =4 and seed=xB7

Hence, these two last experiments are extremely efficient and they have comparable if not better results to 16 bit LFSR with seed xEA58 and either of them can be the implementation choice if LFSR is the final BIST decision.

VI.4.6 –BIST = 16 bit CA:

As described in section III, CA are the least known of the two described PRBS generation techniques and have preferable more random characteristics compared to LFSRs despite the higher hardware cost. In this section we scrutinize several CA with different seeds as an alternative to BIST implementation. Once again we generate a Matlab script to produce the stimulus of a hypothetical CA. However, although in the LFSR case the matlab implementation is simply based on modulo 2 summation of tap values and then shifting, the CA computation is a little bit more involved, yet mathematically perfectly described. The schematic model for the matlab implementation is as shown in figure VI-44. The figure displays a CA with 150 cells at 1st and 15th locations, which is an appropriate combination for max-length sequence generation. The details of mathematical computation of CA output is described in section III, pages 10-12, however a slight modification to the matrix manipulations is performed as the Q outputs are represented as a row vector rather than a column vector in matlab. However, the equations hold as the tridiagonal matrix $T^T = T$.

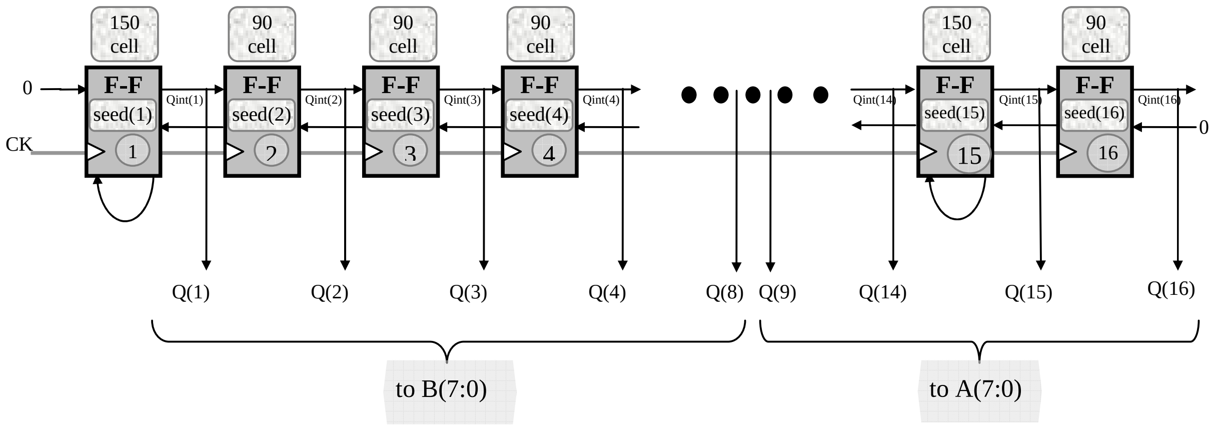


Figure VI-44, Matlab Emulated 16 bit CA circuit

The generated Matlab script is in appendix E-4 and also can be referred from “CD>MSc/DesignFiles/matlab”. The generated output stimuli for different seeds are in "CD/MSc/Results/mult8x8/CA/CA16bit/seedxxxx" directories.

Seed = x0001: Having already become the general initiation point, we once again start with a seed of x0001 for the CA. The simulation data and results are in: "CD/MSc/Results/mult8x8/CA/CA16bit/seed0001/", the simulation list is saved in binary radix to demonstrate the nonexistent shifting property in CA contrary to the LFSR case. The full exhaustive simulation revealed a 99.12% fault coverage – once again inconsistent with the other full exhaustive cases for an indeterminate reason! – and the fault coverage curve is once again like a step function, very much like desired. However the zoomed fault coverage plot in figure VI-45 reveals the fault coverage reaches 97.2% in 159 cycles, which is worse than all the LFSR cases. The plot shows once again the characteristic property of seed x0001, a flat region right after the start. This plot is very similar to the x0001 case of the LFSR. The histogram shown in figure VI-46 reveals a superior detection at the start as usual however, there is a very strong spread within the first 10K. There are also 2 strong late peaks, one in cycle 34569 and the other in cycle 55870, each detecting 7 faults, which will be our next starting points. Once again referring to the simulation list, the 34569th corresponds to a seed of x8080 and 55870th cycle corresponds to a seed of x534F.

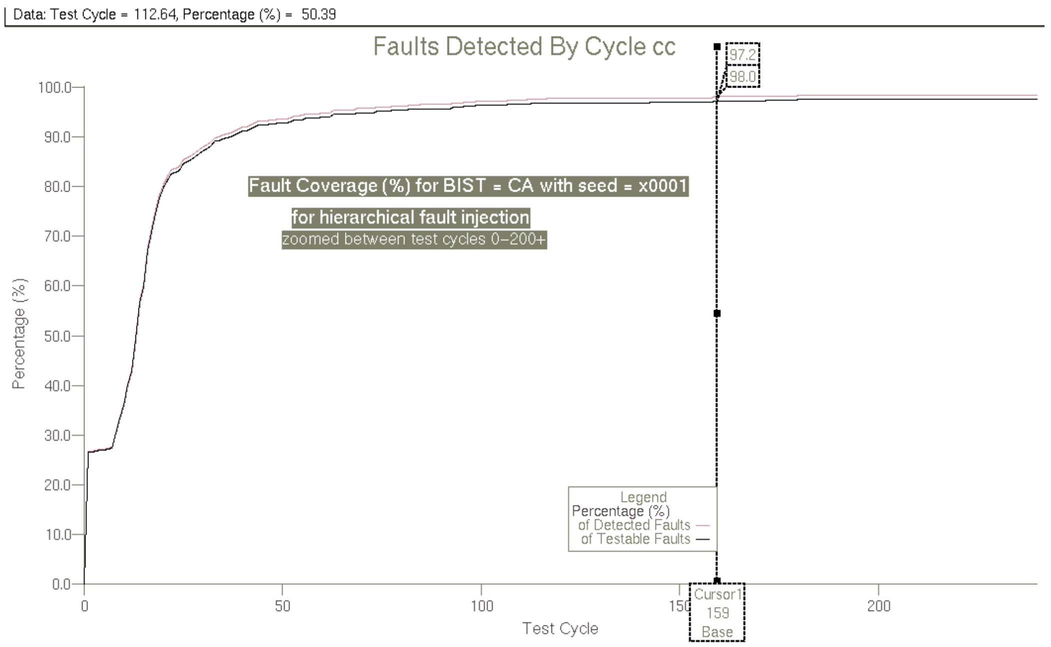


Figure VI-45, Fault coverage (%) for 16 bit CA with seed=x0001

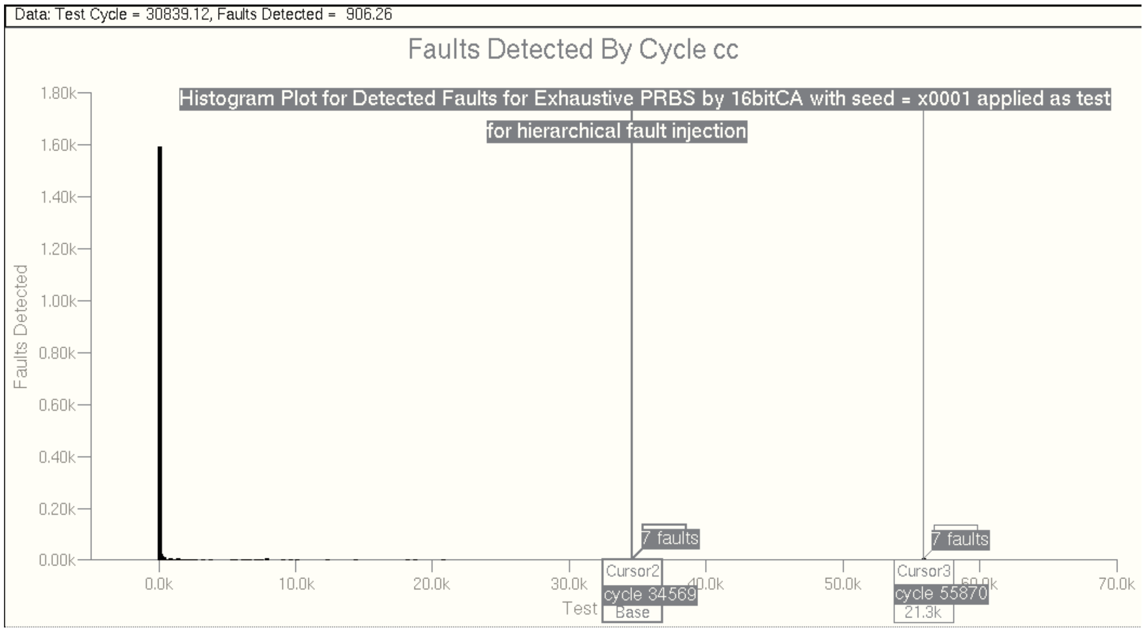


Figure VI-46, Histogram for 16 bit CA with seed=x0001

Seed = x8080: The simulation data and results for this seed are in: "CD/MSc/Results/mult8x8/CA/CA16bit/seed8080/". Fault coverage curve reveals that 97.2% fault coverage is achieved in 138 cycles, which though very good cannot overperform LFSR. The redundant regions are removed with the seed selection but histogram still shows a strong spread up to 5K. The significant peak at cycle 21302 is exactly the peak that corresponds to the peak at 55870 for seed = x0001.

<u>Seed = x534F:</u> The simulation data and results for this seed are in: "CD/MSc/Results/mult8x8/CA/CA16bit/seed534F/". The fault coverage shown in figure VI-47 reveals <u>97.2% fault coverage in just 122 cycles</u>, which is the best achieved result so far. the histogram plot shown in figure VI-48 produces a new peak at cycle 6052 with 6 faults. From simulation list, this corresponds to a seed of D5D5, which is the next candidate for the optimal seed.

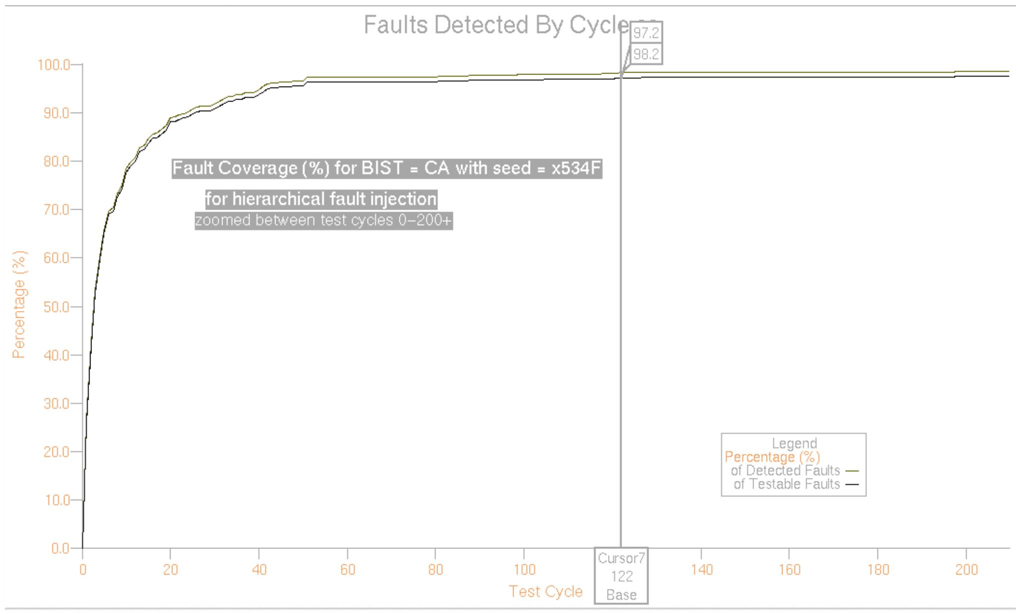


Figure VI-47, Fault coverage (%) for 16 bit CA with seed=x534F

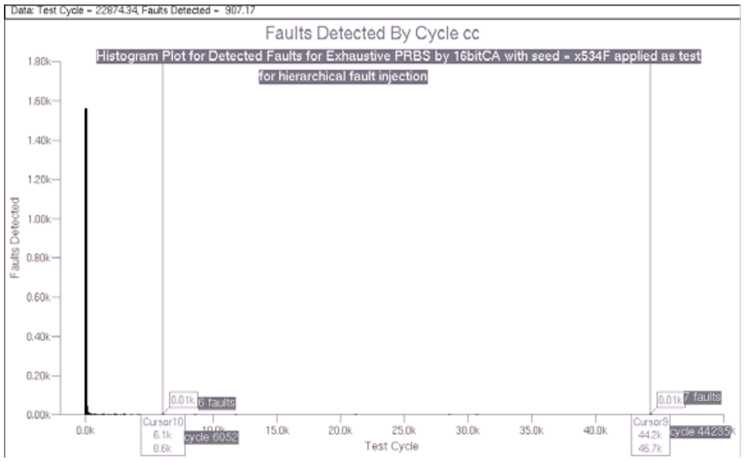


Figure VI-48, Histogram for 16 bit CA with seed=x534F

Seed = xD5D5: The simulation data and results for seed = xD5D5 are in: "CD/MSc/Results/mult8x8/CA/CA16bit/seedD5D5/", it revealed 97.2% fault coverage in 127 cycles as can be seen from the plots in CD, which is the second best result achieved after 16bit CA with seed x534F.

With these experiments we conclude the investigation of 16 bit CA as BIST, and achieve slightly better results compared to LFSR at some specific cases, however as the general engineering practice the trade-off is in hardware cost. Therefore, there is no single decision that can be made at this point that can state one scheme is "better" than the other one. These relations are based on either performance or cost.

VI.4.7 –BIST = 8 bit CA – Repetitive Pattern:

Similar to the LFSR case, we also investigate the effectiveness of repeated patterns when CA are chosen as the pattern generator circuit. We once again use a repetition length of 4 and consider an 8 bit CA for BIST circuit. We make use of the Matlab script to emulate the outputs of the 8 bit CA with 150 cells located at 2nd and 3rd positions, as shown in appendix E-5 and included in “CD>MSc/DesignFiles/matlab”. The generated output stimuli for different seeds are in "CD/MSc/Results/mult8x8/CA/CA8bit/seedxx" directories.

Similar to figure VI-39, the Q(1:4) outputs are input to both b(3:0) and b(7:4); and Q(5:8) is input to a(3:0) and a(7:4). As usual we start with a seed of x01.

Seed = x01: The simulation data and results are in: "CD/MSc/Results/mult8x8/CA/CA8bit/seed01/". As can be seen from the fault coverage plot, it achieves 97.1% fault coverage in 139 cycles and 97.2% fault coverage in 179 cycles. The results are not better than the LFSR with seed x7B, but as the initial starting point, it gives sufficient information for seed determination. The histogram, shown in figure VI-49 is quite spread and there are several redundant regions. Moreover, the spread consists of few bursts at various locations. Some of the significant cycles are, cycle 50 with 41 detected faults, cycle 60 with 64 detected faults, cycle 138 with 7 faults and cycle 180 with 9 faults. There is a very spread burst between cycles 60-100, starting at 57th cycle. Therefore, one of the good starting points might be 57th cycle as it will cover up to 100 burst at the hyperactive startup phase. In the same fashion, starting from cycle 138 will also cover the cycle 180 peak. As can be deduced, we do not take any precaution for cycle 50, under the hope that, starting later than 50 might have the chance to cover the faults detected by 50 with another vector, thus reducing any adverse effect leaving 50 uncovered might cause. Nevertheless, our expectation is, if cycle 50 pattern cannot be compensated with the new start points, we'll observe a peak at

255 + 50 – 57 + 1 =249th cycle for 57th cycle as start point

and

305-138 + 1 = 168th cycle for 138th cycle as start point

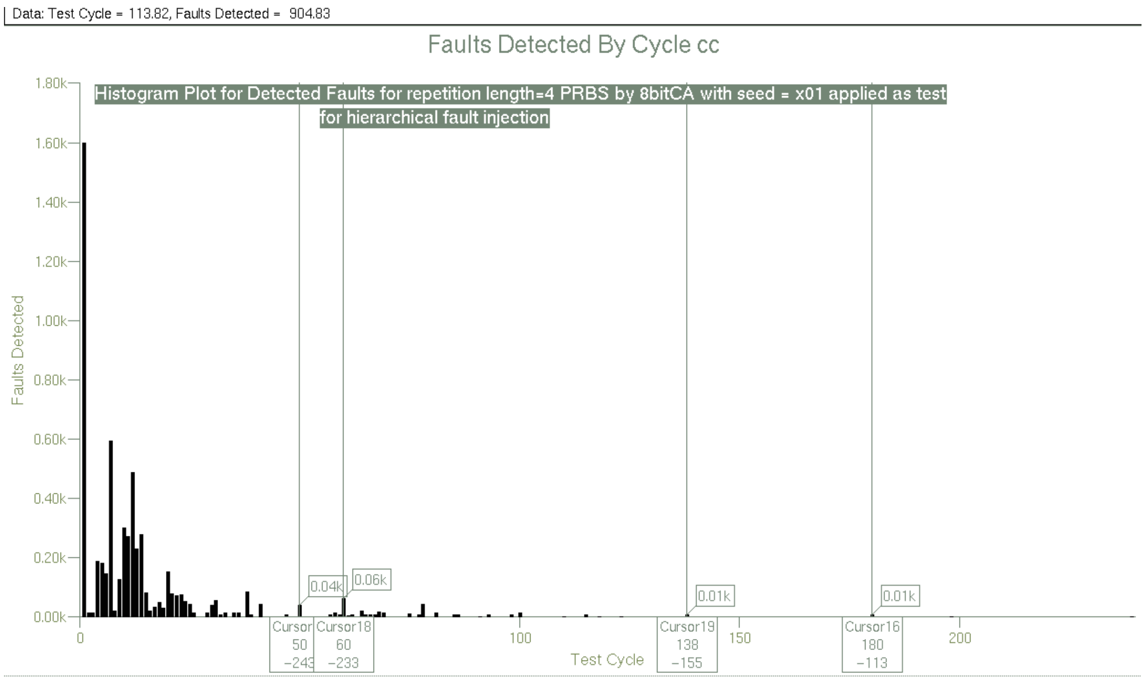


Figure VI-49, Histogram for 8 bit CA with seed=x01

Once again, referring to the simulation list file, the seed corresponding tom57th cycle is: x6D and to 138th cycle is: xAB.

Seed = x6D: The simulation data and results for seed = x6D is in: "CD/MSc/Results/mult8x8/CA/CA8bit/seed6D/". The fault coverage plot in figure VI-50 shows it achieves 97.2% fault coverage in just 124 cycles, while reaches 97.0% in 123 cycles. It can be verified that the 124 cycles result is the best achieved so far, however, seed=x7B exceeds 97.0% before 109th cycle. Therefore, we face the controversy once again, that different BIST strategies perform better for different target fault coverages. ⇐

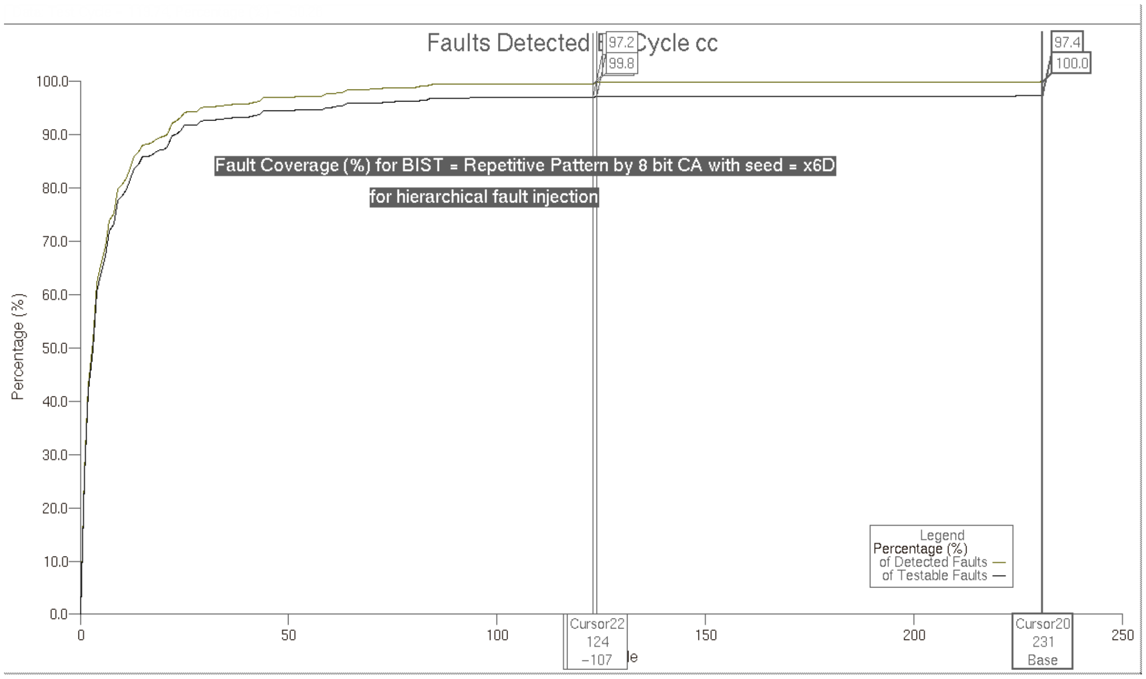


Figure VI-50, Fault coverage (%) for 8 bit CA with seed=x6D

The histogram plot that can be accessed from the CD shows a very high detection efficiency, with the highest peak at 124th cycle – where the fault coverage jumps from 97.0% to 97.2% -, which was to be expected as the highest late we had considered in previous seed=x01 case was at 180th cycle. To verify this, 57+124-1 = 180, shows the 180th peak is the 124th peak of this simulation.

<u>Seed = xAB:</u> The simulation data and results for seed = xAB is in: "CD/MSc/Results/mult8x8/CA/CA8bit/seedAB/". As can be verified from the fault coverage plot in figure VI-51, the <u>fault coverage reaches 97.19% in 125 cycles and 97.1% in just 110 cycles</u>.

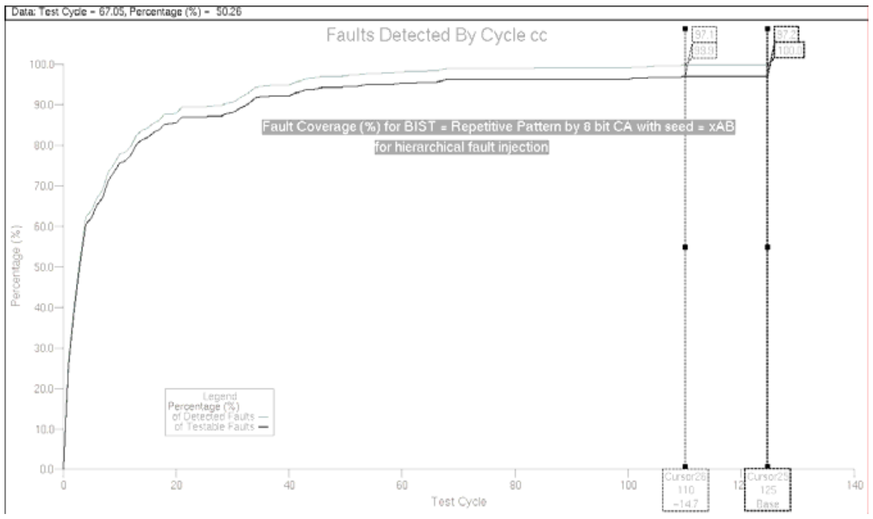


Figure VI-51, Fault coverage (%) for 8 bit CA with seed=xAB

This concludes our investigation of different BIST techniques and in the next subsection, we discuss the results of these techniques.

VI.4.8 – Summary of BIST Techniques:

As a brief sum up, it is observed that, LFSR and CA pseudorandom techniques perform better than deterministic techniques, due to their almost random nature of pattern sequence. The fault coverage builds up much quickly. In regard of this opinion, it is expected that CA perform better than LFSR as they are free from the shifting behavior of LFSRs, however, very slight improvements if any are observed with application of CA and therefore it is not very feasible to incur the additional hardware cost of CA for this insignificant improvement. On the contrary, the application of repetitive patterns is seen to provide excellent fault coverage with only a small set test vectors. As thoroughly scrutinized, the 16bit full exhaustive patterns are not significantly better than 8 bit repetitive patterns for both LFSRs and CA with the test length significantly reduced.

Yet, it is not very obvious which BIST scheme is more appropriate for implementation. To have an overall view of the results, we repeat the outcomes of each section in here in table 20. Although there is no such measure that can be induced to state one method is 'better' than the others, one of the evident observations is, repetitive patterns with only 8 bit pattern generators are almost equally efficient as 16 bit pattern generators. Therefore, the BIST choice can safely be an 8 bit pattern generator. Among the 8 bit generators, upcounter is out of scope and the choice is between CA and LFSRs. The BIST implementation can be either LFSRs with seeds x7B or xB7, or CA with seeds x6D and xAB. However, due to the additional hardware cost of CA, the probable choice will be between the two LFSRs.

BIST method	**Fault Coverage vs. # of patterns**
8 bit Upcounter Using Repetitive patterns with k=4	97.02% with 228 patterns
16 bit LFSR with seed=x0001	97.2% with 157 patterns
16 bit LFSR with seed=xEA58	97.2% with 134 patterns
16 bit LFSR with seed = x8080	97.2% with 153 patterns
8 bit LFSR using repetitive patterns with k =4 and seed=x01	97.12% with 228 patterns
8 bit LFSR using repetitive patterns with k =4 and seed=x7B	97.2% with 154 patterns 97.1% with 109 patterns
8 bit LFSR using repetitive patterns with k =4 and seed=xB7	97.0% with 82 patterns 97.24% with 230 patterns
16 bit CA with seed=x0001	97.2% with 159 patterns
16 bit CA with seed=x8080	97.2% with 138 patterns
16 bit CA with seed=x534F	97.2% with 122 patterns
16 bit CA with seed=xD5D5	97.2% with 127 patterns
8 bit CA using repetitive patterns with k =4 and seed = x01	97.1% with 139 patterns 97.2% with 179 patterns
8 bit CA using repetitive patterns with k =4 and seed = x6D	97.2% with 124 patterns 97.0% with 123 patterns
8 bit CA using repetitive patterns with k =4 and seed = xAB	97.19% with 125 patterns 97.1% with 110 patterns

Table 20, Summary of BIST Results

Before implementing the BIST generator, being very cautious, we first decide to implement the output data compressor, with the initial and hopefully last choice of a signature analyzer in order to make sure the data compression stage does not change the effectiveness relation between the investigated pattern generation schemes.

VI.5 – MISR IMPLEMENTATION FOR BIST

Having finalized the investigation for pattern generation, we move to output data compression. Signature analysis is the generally applied technique in data compression and has been our first choice of implementation. The regular multiplier structure suggests that it is not probable to have a single stuck at fault that causes two distant separate multiplier outputs change at the same time and therefore we expect a very low fault masking probability with signature analyzer. The signature analyzer is designed as described in design entry section and synthesized into eddm-schematic. The synthesized 16 bit signature analyzer is in:
" CD/MSc/DesignFiles/Renoir/Eddm_sch/signalyzer_16_10110111101100011/"

The generated schematic for signature analyzer is shown in figure VI-52 and can be accessed from "CD/MSc/Results/Sign16/Sign16sch.gif".

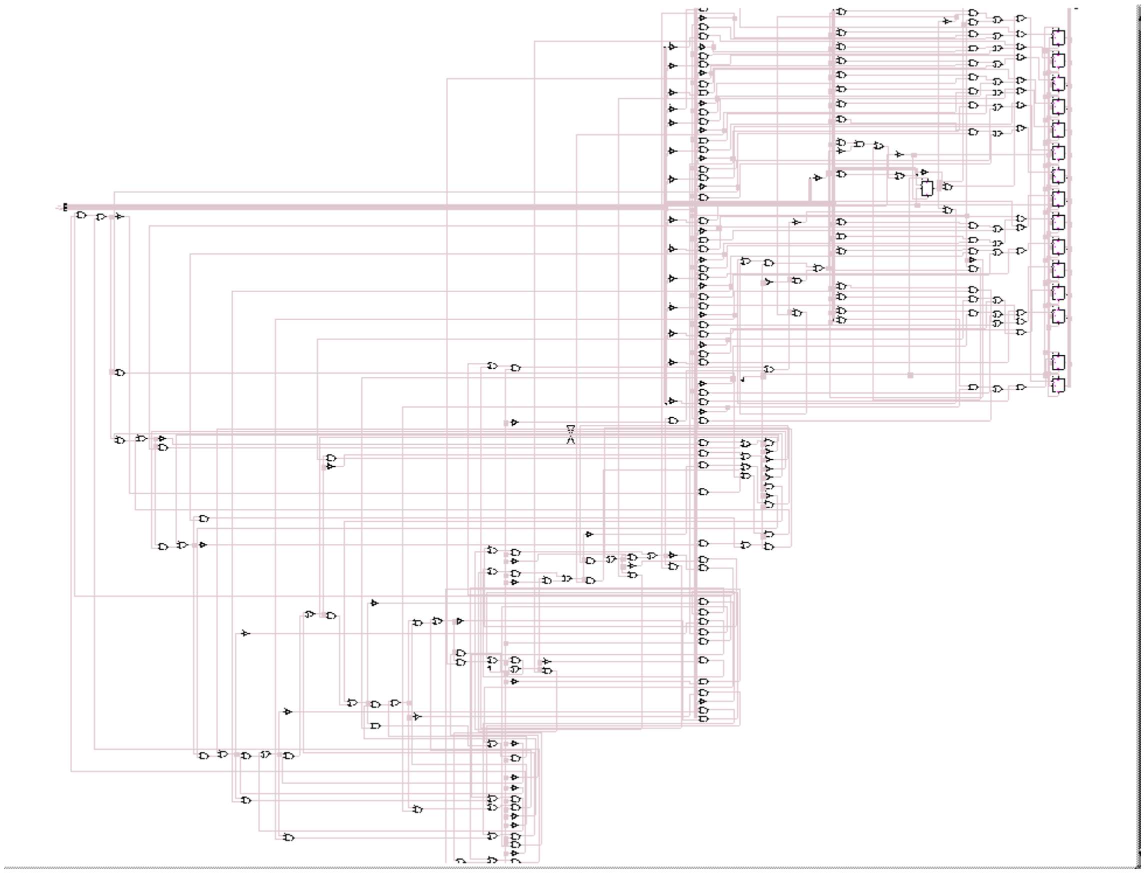


Figure VI-52, Generated schematic for 16 bit signature analyzer

As described in design entry section, the signature analyzer is assured for correct functionality in VHDL level, then it is combined with the multiplier circuit at one higher level into MultSign8x8, which is then synthesized into eddm-schematic as in:
"CD/MSc/DesignFiles/Renoir/Eddm_sch/multsignnxn_10110111101100011_8/"
The schematic for MultSign8x8 is as shown in figure VI-53 and in:
"CD/MSc/Results/MultSign8x8/multsignsch.gif".

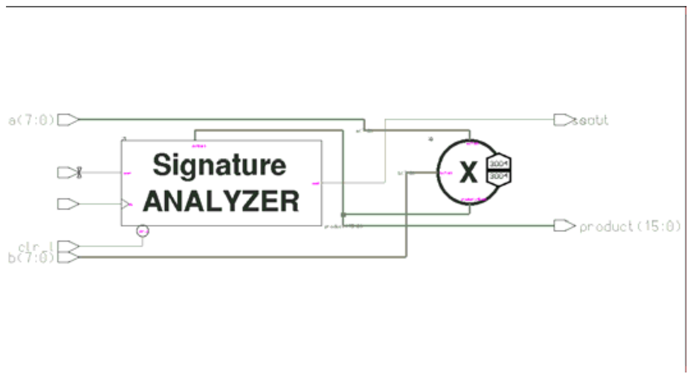


Figure VI-53, Generated schematic for 16 bit signature analyzer + 8x8 multiplier

Now, in order to observe how much fault masking occurs with signature analysis, we used the stimulus for 8 bit LFSR with seed=x7B to drive the multiplier. We have also included 3 signature analyzer signals as shown in figure VI-54.

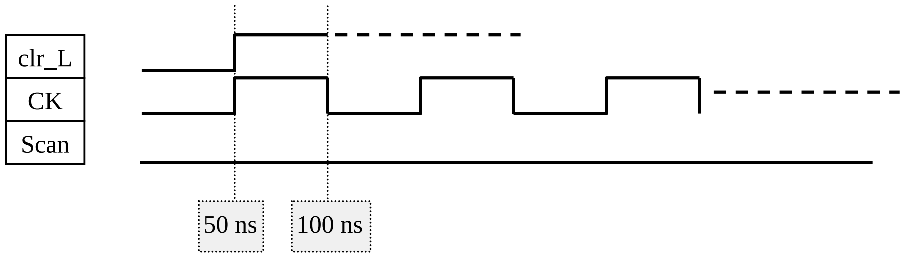


Figure VI-54, Signature analyzer input signals

As also shown in figure VI-53, we used the same fault dictionary to generate the same faults in the multiplier only to be able to make a comparison between the two cases. Clearly, we only strobed the single bit signature analyzer output, 'sout' to detect any faulty output. As a result of the fault simulation, <u>96.80% fault coverage</u> is achieved by observing only the signature analyzer output, which is very slightly lower than the result we achieved by observing all 16 multiplier outputs. All the simulation data and results are in: "CD/MSc/Results/MultSign8x8/", and they are not included in the report for brevity. However, we show the histogram in figure VI-55, as it bears an important information about the minimal effective test set.

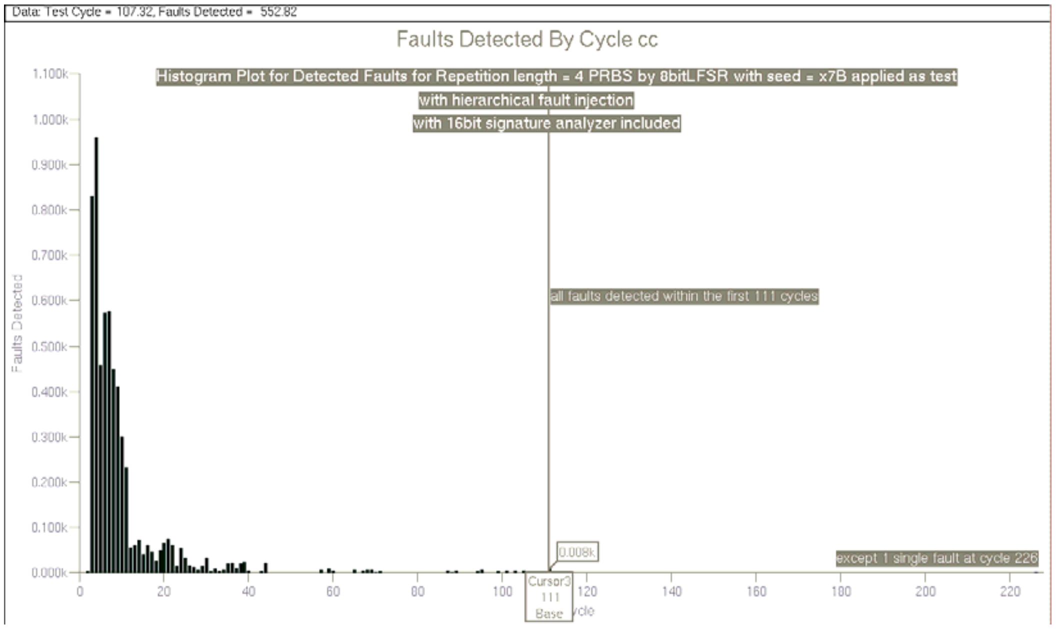


Figure VI-55, Histogram for MultSign8x8, with 8 bit LFSR with seed=x7B as input

As seen in the histogram, all the faults are detected in the first 111 cycles, except for only 1 at cycle 226, which has a contribution: $1/6008 = 1.66*10^{-4}$ to the total fault coverage. Therefore we can achieve the same fault coverage with only 111 patterns.

This immediately concluded our output compression investigation due to the excellent results, moreover the above outcome also suggested we use seed=x7B LFSR as the pattern generation implementation of choice.

VI.6 –LFSR IMPLEMENTATION FOR BIST

As the last and most elaborated portion of the BIST circuitry, we designed the LFSR as described in the design entry section. We then synthesized the parameterized LFSR into an 8 bit LFSR with seed x7B and taps at 1,5,6 and 8, which corresponds to the LFSR shown in figure VI-39. The synthesized LFSR is in:
"CD/MSc/DesignFiles/Renoir/Eddm_sch/lfsr_8_01111011_10001101/"

The generated schematic for the LFSR is shown in figure VI-56 and can be accessed from " CD/MSc/Results/LFSR8/LFSR8sch.gif".

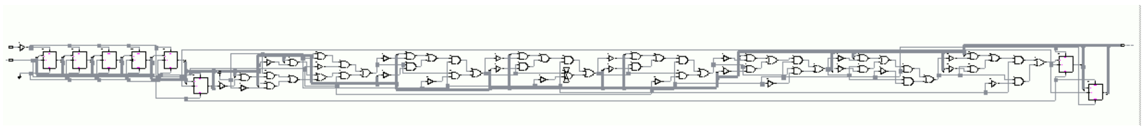


Figure VI-56, Generated 8 bit LFSR

Finally, we connect all the 3 parts of the BIST + multiplier circuit to achieve the final multiplier circuit with included input pattern generation and output compression. The Renoir design can be referred from appendix B-4, which is the implementation of blocks as described in repetitive pattern LFSR pattern generation and output compression phases. The final synthesized design is in: "CD/MSc/DesignFiles/Renoir/Eddm_sch/lfsrmultsignnxn_
10110111101100011_8_10001101_01111011_8/

The top-level schematic for the complete design is shown in figure VI-57 and can be accessed from: " CD/MSc/Results/LFSRMultSign8x8/LFSRMultSignsch.gif"

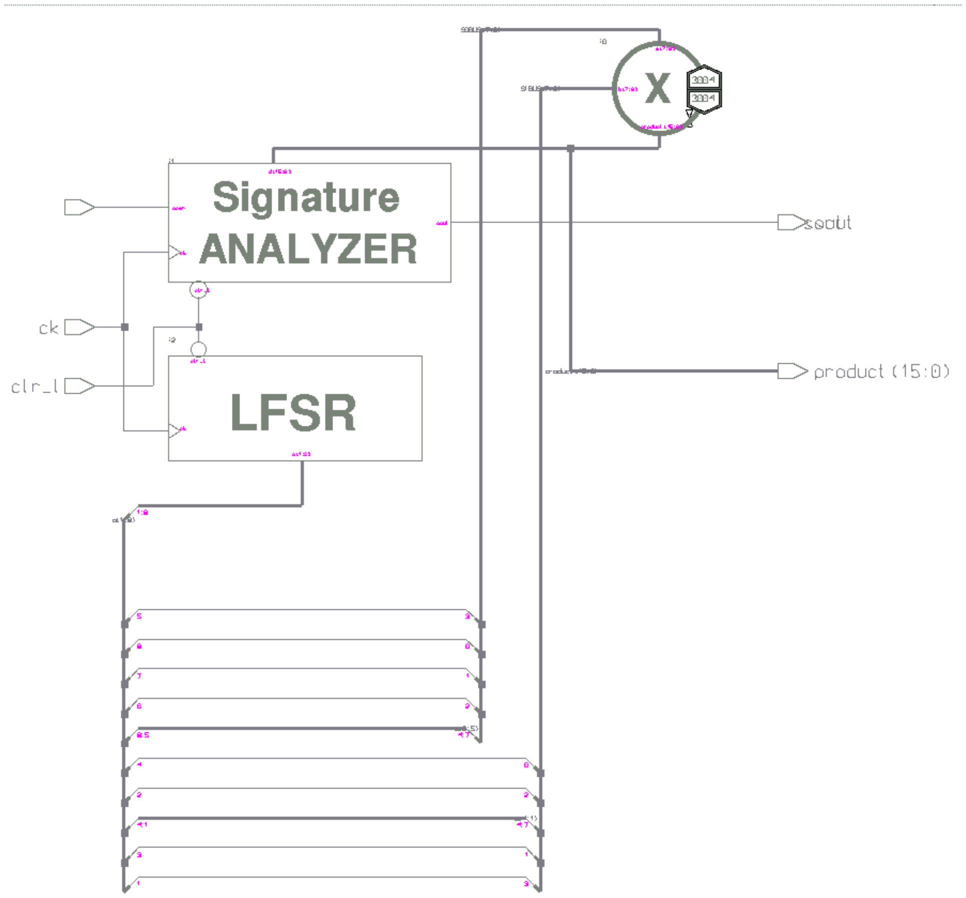


Figure VI-57, Generated schematic for complete design for mult8x8

In order to assure our software based observations match with the hardware implementation, we perform fault simulation for the final circuit by applying only the clk, clr and scan signals as the stimuli, as in figure VI-54. We only strobe the sout output of the signature analyzer and with the 6008 faults injected into the multiplier as shown in figure VI-57, we achieve a 97.07% fault coverage as shown in figure VI-58. Once again, the results are not exactly the same as the 2 previous simulations performed under the same scheme. In order to alleviate any suspicion about any flow in the experiment, we wrote a small UNIX script that takes the two simulation list files and compares the input output fields for the files, which is in "CD/MSc/Results/convcomp". The resultant compare as shown in: "CD/MSc/Results/compresult" reveals, the simulation stimuli are exactly the same throughout the simulation, therefore, the slight difference in fault coverage measurements is attributed to the CAD tool again.

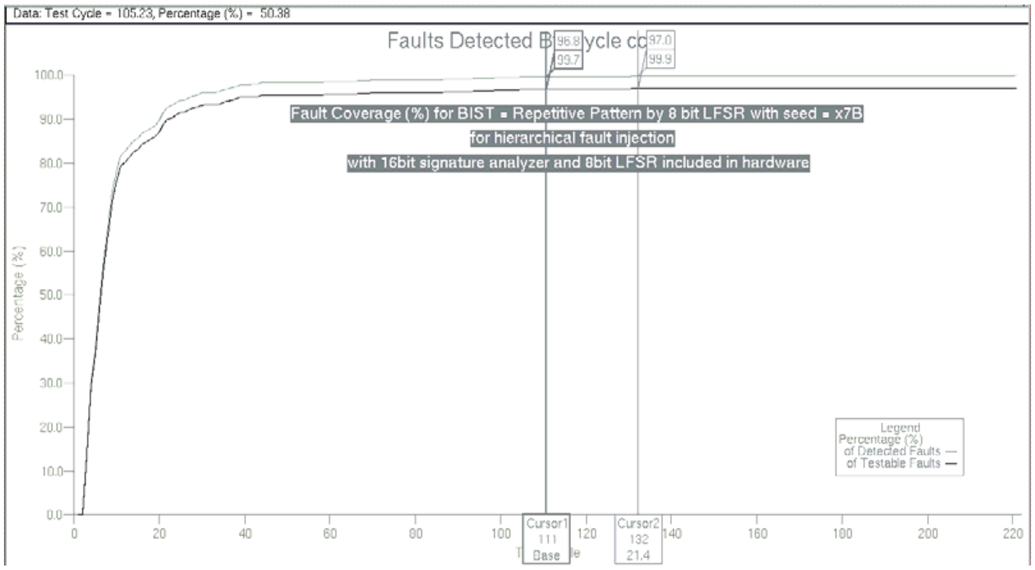


Figure VI-58, Fault coverage (%) for LFSRmultSign8x8

As one last step to the investigation, we include the BIST circuitry in the fault simulation and inject hierarchical faults to the LFSR and the signature analyzer, however, as the D-flip-flop primitives in QuickFault have one inverted output as well as the noninverted ones, we have several untestable pins in the design, which must be nofaulted. Moreover, the set and reset pins of the flip-flops are also problematic one for each flip-flop is hardwired or dangling. Nevertheless, the fault injection increases total fault count to 8016, with 6008 faults in multiplier, 448 in LFSR and 1560 in signature analyzer. The final fault grade, which is in: "CD/MSc/Results/LFSRMultSign8x8/allfaulted/faultsummary.txt" reveals 96.95% fault coverage, which is very close to the only multiplier case.

This finally concludes the whole BIST investigation, we observe very satisfactory results with LFSR as pattern generator and signature analyzer as output data compressor. As the next step in the project – which is now more of curiosity due to the repetitive patterns – we deploy larger multipliers and perform fault simulation with the same amount of input test vectors, using repetitive patterns with a repetition length of 4.

VI.7 –16X16 MULTIPLIER WITH BIST

We move from the 8x8 multiplier to 16 by 16 multiplier as the new circuit to be tested with repetitive patterns. As with the circuit size, we expect the gate count to quadruple and thus we expect to have 4 times as much faults on the circuit as the 8x8 multiplier. The synthesized LFSRmultSign16x16 is in "CD/MSc/DesignFiles/Renoir/Eddm_sch/lfsrmultsignnxn_16_1000101000101010100010100 00000101_16_10001101/" and the new generated schematics for 16x16 multiplier, 32 bit signature analyzer and top level circuit are shown in figures VI-59 to VI-61, which can be also accessed from "CD/MSc/Results/LFSRMultSign16x16/"

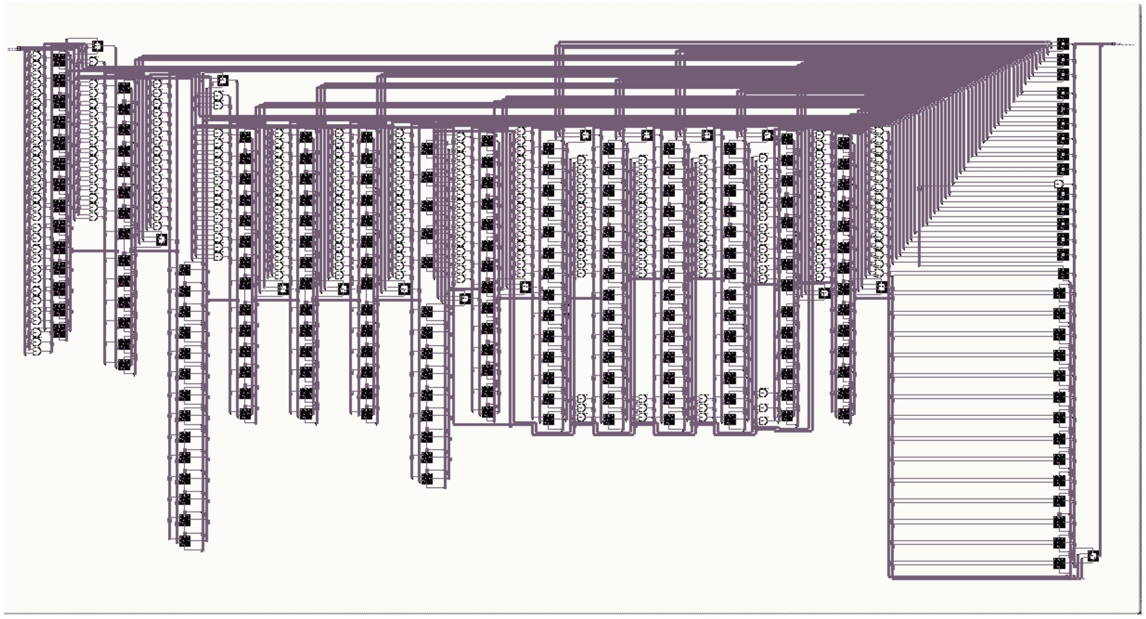


Figure VI-59, Generated 16x16 schematic

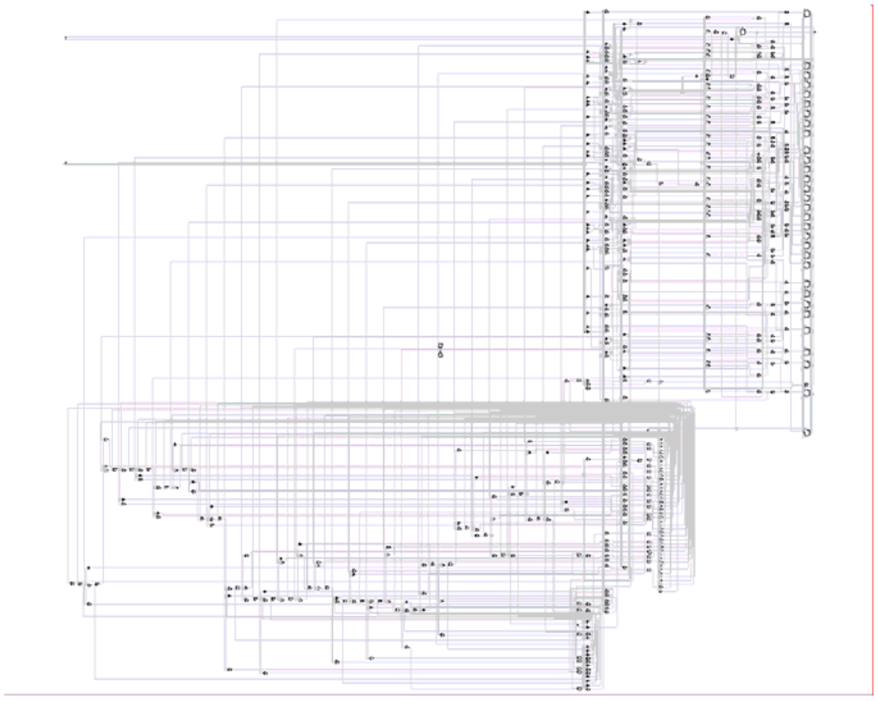


Figure VI-60, Generated 32 bit signature analyzer

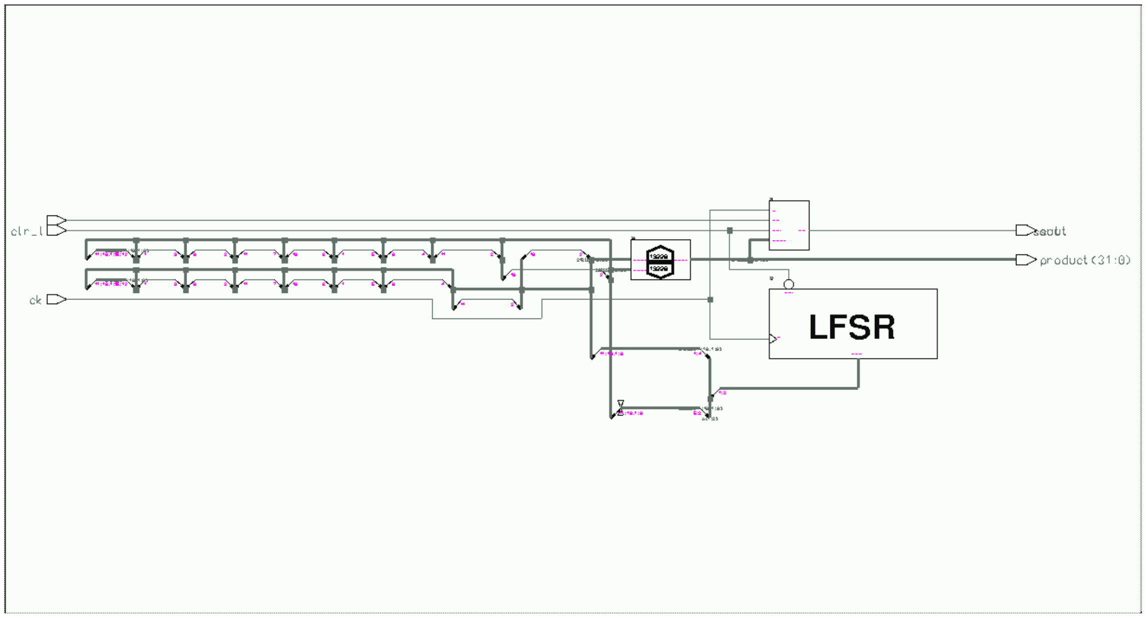


Figure VI-61, Top level circuit

Here we observe the advantage of making separate symbols for each block, as the component counts increase, this turns out to be only feasible way to distinguish between blocks in the schematic. Hence there is no new schematic for the LFSR as we still use a single 8 bit LFSR for repetition length 4 patterns.

The simulation data and the results for the fault simulation of 16bit top level circuit are in "CD/MSc/Results/LFSRMultSign16x16/" and as seen in the grade and fault list, the total number of faults exploded up to 26456, which is almost 4 times the 8x8 multiplier faults.

We perform fault simulation, the same way we have done for 8x8 multiplier, with repetitive PRBS with k=4, and obviously now all the 4 bit b inputs b(15:12), b(11:8) , b(7:4) , b(3:0) are connected to LFSR output (1:4) and all the 4 bit a inputs a(15:12), a(11:8) , a(7:4) , a(3:0) are connected to LFSR output (5:8). As shown in figure VI-62, we achieve an unexpectedly good result. The fault coverage reaches 97% in just 57 cycles and it climbs up to 98.83% in the whole test. This extremely satisfactory result reveals an excellent conclusion, the repetitive patterns provide very high fault coverage with a fixed number of patterns independent of the size of the multiplier. $\Leftarrow$

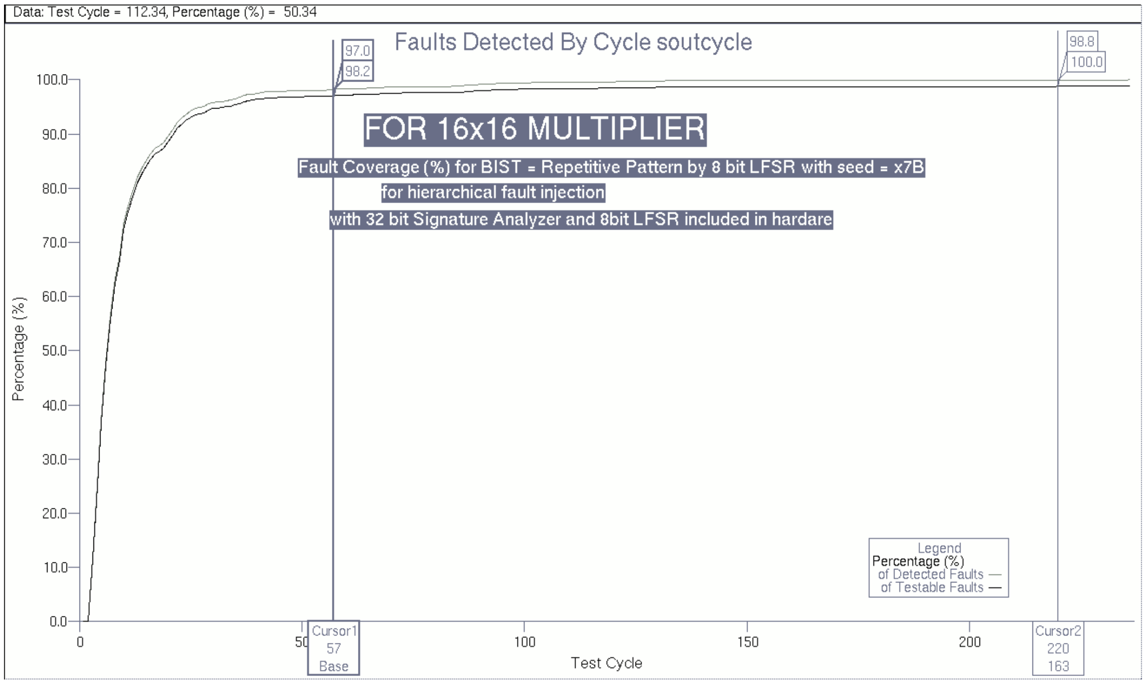


Figure VI-62, Fault coverage (%) for LFSRmultSign16x16

Despite it looks counterintuitive, the fault coverage even improved as we increased multiplier size, when compared to figure VI-58 of 8x8 multiplier.

VI.8 –LARGER MULTIPLIERS WITH BIST

To verify our observations we increase the circuit size to a 32x32 multiplier, still with the 8 bit single LFSR for input test generation. The synthesized circuit is in: "CD/MSc/DesignFiles/Renoir/Eddm_sch/lfsrmultsignnxn_32_10000000100010000000100010001000100000000010001/". Once again the faults are expected to quadruple over 16x16 multiplier and as can be verified from "CD/MSc/Results/LFSRMultSign32x32/", the total fault count increased to 111128.

The new generated schematics for 32x32 multiplier, 64 bit signature analyzer and top level circuit are shown in figures VI-63 to VI-65.

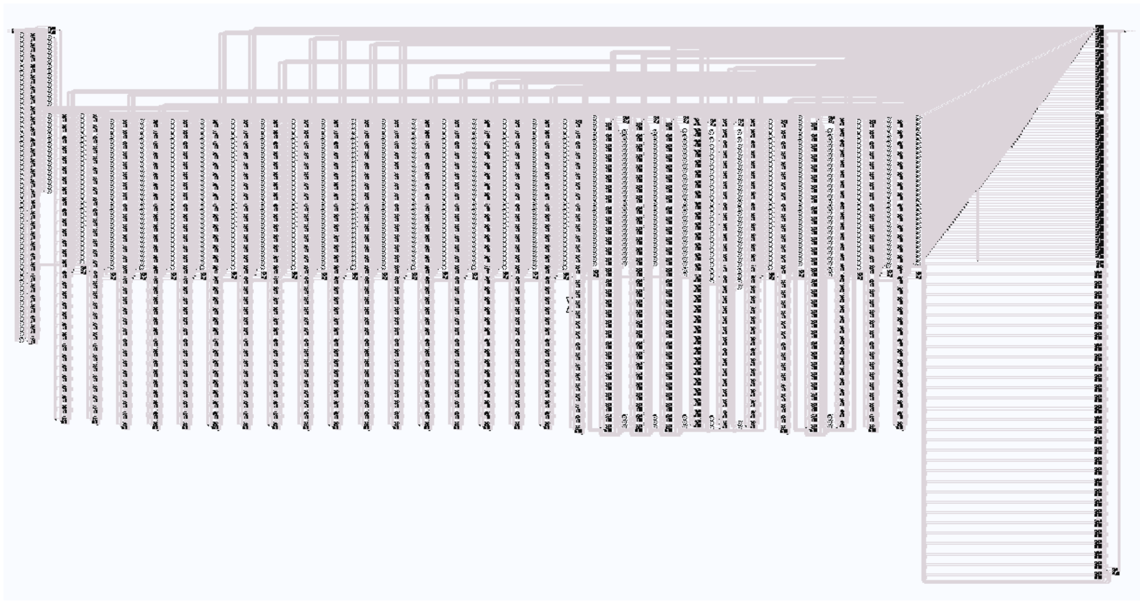


Figure VI-63, Generated 32x32 multiplier schematic

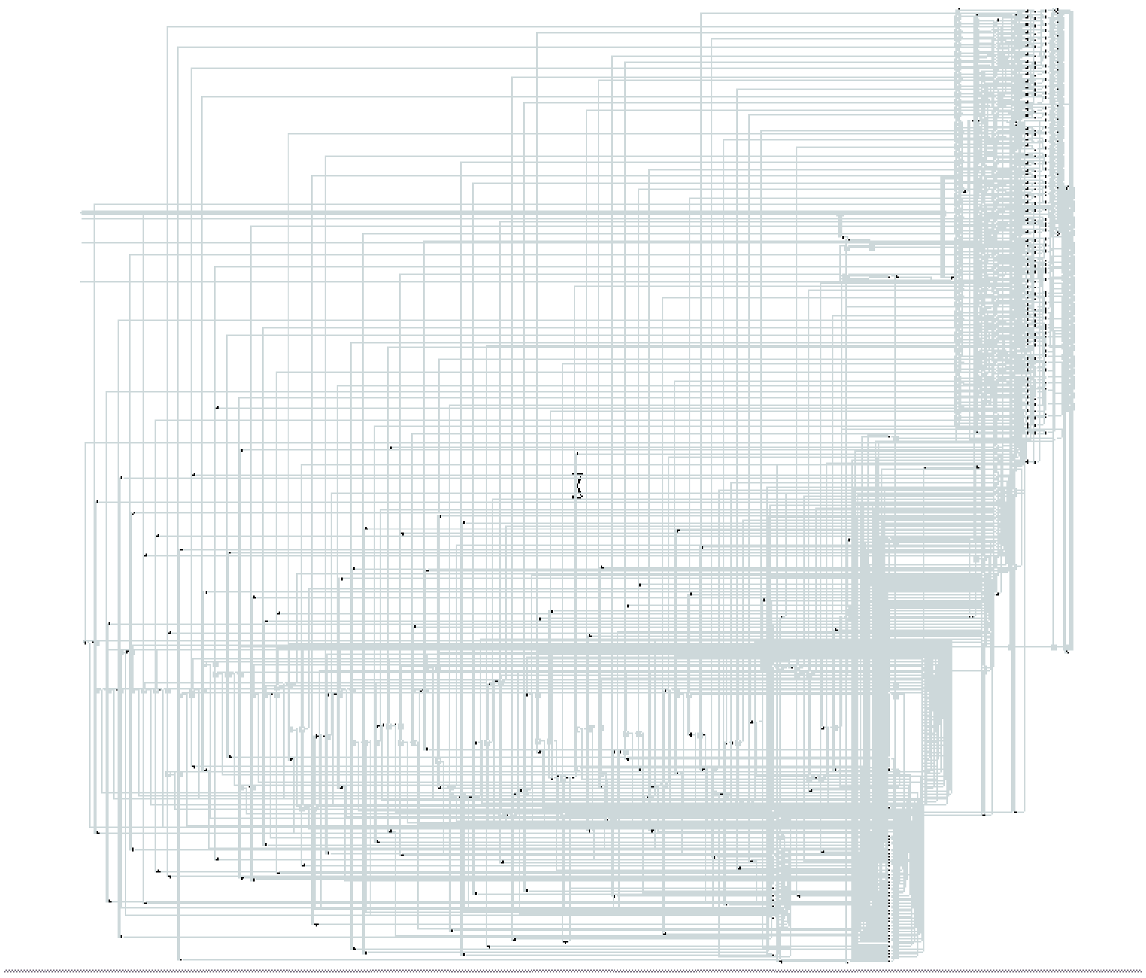


Figure VI-64, Generated 64 bit signature analyzer

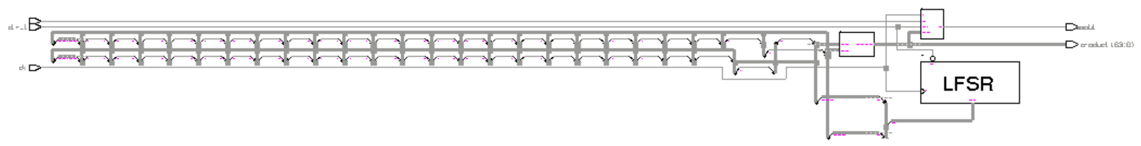


Figure VI-65, Top level circuit schematic for 32 bit multiplier

Unfortunately, with this much faults, the fault simulation could not complete and crashed due to insufficient memory. The post synthesis simulation results could be obtained, and can be referred from CD for concordance, but no fault simulation data except for fault list is available.

We have also designed a 24x24 multiplier and synthesized into eddm-schematic, but the fault simulation could only continue for 112ns, which is it crashed while evaluating the 2nd input pattern. Therefore, we ere unable to achieve any results beyond 16x16 multiplier.

Thus, we conclude our investigation for BIST and reach the conclusions repeated a few times within the text. PRBS techniques, in combination with repetitive patterns are seen to be unbelievably efficient in fault detection, regardless of the size of the multiplier. Signature analyzer, reduces the output data volume to a single bit, yet incurs insignificant loss in fault detection. CA, though the more random output nature promises significant enhancements, is not seen to be worth sacrificing the additional hardware cost for the very slight to no improvement in fault detection.

VII-WORKPLAN

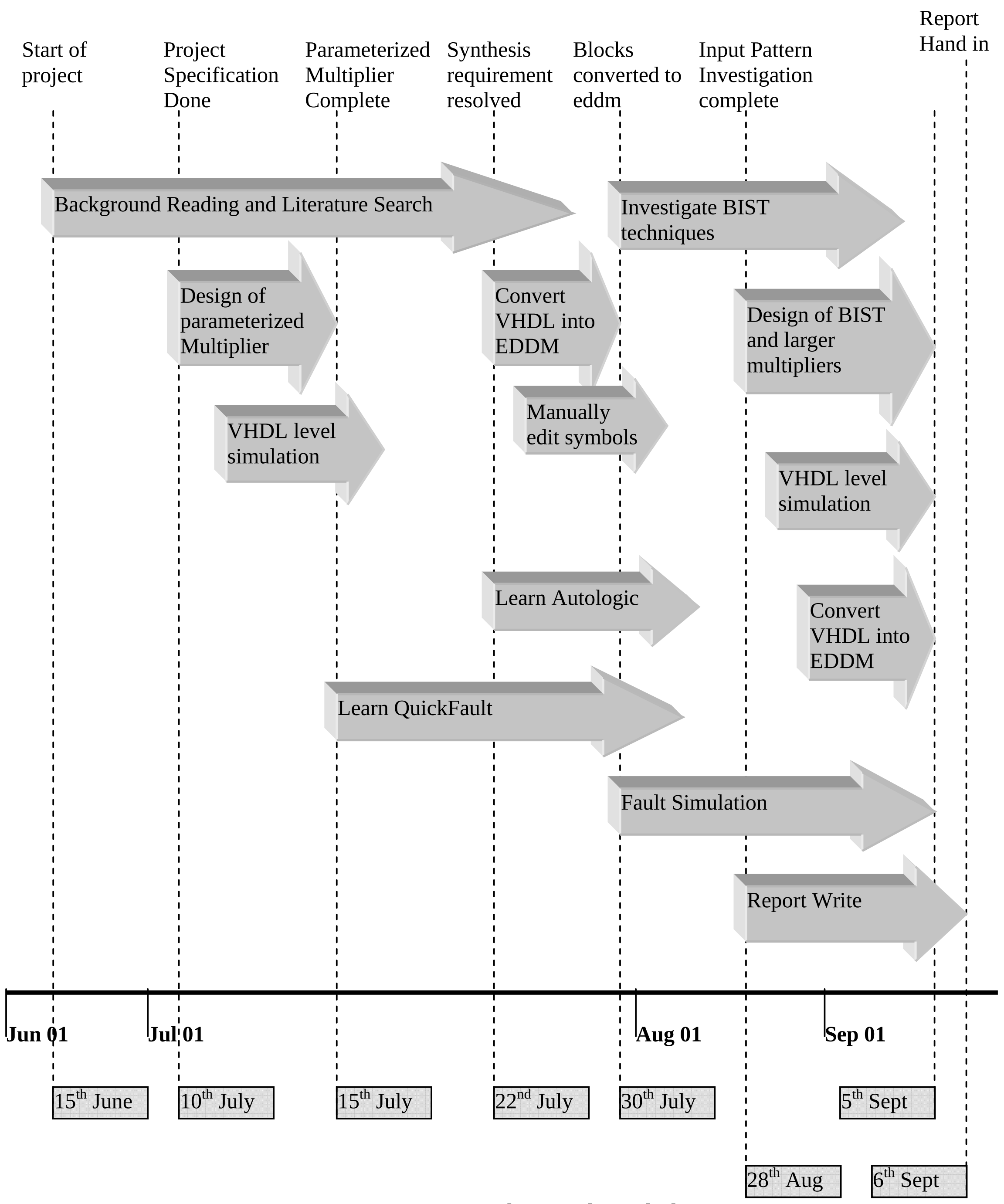


Figure VII-1, Milestoned Workplan

VIII-CONCLUSIONS

In this project we have designed a parameterized complete multiplier + BIST system, including the parameterized signed parallel CPA multiplier, parameterized LFSR for input pattern generation and parameterized signature analyzer for output data compression. Several input pattern generation techniques with very common and original methodologies proposed are investigated. Exhaustive pattern generation with board level fault modeling revealed 100% fault coverage at around 50% of both upcounter and downcounter pattern generators, interestingly, both with exactly 32897 vectors. The observation that, the primary inputs are reduced to half number after the bit product generation and a rolling 0 tests all HHandH gates' faults led to the application of a rolling 0 as input test set and 98.07 % fault coverage achieved with just 16 cycles. Further observing that the HLandH gates' all undetected faults can be tested with an all 0s pattern, led to a rolling 0s + all 0s test set which revealed a prominent 100 % fault coverage with 17 patterns.

Board level faulting being too abstract, we moved to hierarchical faulting and to the other extreme; the hierarchical primitives are chosen as the MGC_Genlib primitives, including only AND, OR, INV gates and registers for the designed blocks. With this level of hierarchy, the fault detection of rolling zeros and all 0s dropped down to 81.61%, and modifications such as two rolling 0s applied simultaneously didn't alleviate the downgrade. As a benchmark for the rest of the simulations, a full exhaustive testing with downcounter is performed hierarchically and although all possible 2^{16} input combinations are applied to the multiplier, the fault coverage percentage reached to 98.48% rather than 100%, due to the probable logic redundancies in the unoptimized, unflattened hierarchical design. As of importance to note, most of the undetected faults in hierarchical faulted circuits resided in the modified MSB FA's for the signed multiplier, hinting a probable inherent redundancy in this modification. Yet, for a plausible statement, an unsigned multiplier must be simulated in the same scheme, as the high fault population in the MSB might be due to less frequent occurrence of the propagation of partial sums up to the MSBs in both multipliers. However, signed multiplier, making use of the MSB only when both inputs are most negative seems to bear a correlation with this although there is no theoretical foundation. Another observation, which supports this hypothesis has been, the 32897. pattern in the upcounter stimulus, which was seen to be the last effective pattern, is 32896_{10} = 10000000 10000000_2.

After setting up the benchmark for hierarchical faulting, a non-exhaustive upcount test using repetitive patterns with repetition length=4 is applied, and a 97.02 % coverage with only 256 cycles is achieved – as the fault coverage for full exhaustive test is 98.49 %, this might be considered as a 97.02 %/ 98.49 % fault coverage for the actual functional gates for the multiplier-. PRBS generation schemes are extensively investigated and a successful seed determination technique is demonstrated. For exhaustive testing 16 bit LFSRs and CA are applied for different seeds and better fault coverages are achieved in less than 256 cycles, yet with the expense of hardware cost. Applying repetitive pattern technique with PRBS generators, still better results are achieved with the 8bit LFSR with seed x7B revealing 97.1 % fault coverage in just 109 patterns and 97.2 % fault coverage in 154 patterns. Applying the 8 bit CA with seed xAB revealed 97.1 % fault coverage in 110 patterns and 97.2 % fault coverage in just 125 patterns. Comparing LFSR with CA, for both exhaustive and non-exhaustive tests, CA are seen to perform slightly better, due to their much random-like output in nature, but the improvements were very insignificant. As the implementation choice, 8 bit LFSR with seed 7B is chosen due to less hardware cost compared to CA.

After the completion of input pattern generation techniques, output compression techniques are investigated, and for the 255 pattern input test set, a hamming distance 5, high weight polynomial is used as the characteristic polynomial of signature analyzer. The characteristic polynomial $1+x^2+ x^3+ x^5+ x^6+ x^7+ x^8+ x^{10}+ x^{11}+ x^{15}+ x^{16}$ is observed to have almost no fault masking during compression, which is actually polynomial multiplication of 2 8th order polynomials.

Of the designed larger multipliers, 16x16 multiplier is interestingly seen to have even higher fault coverage with the constant 255 test vectors, and repetitive patterns is confirmed to be very effective for multiplier testing regardless of the size. However, 24x24 and 32x32 multipliers could not be fault simulated due to insufficient memory in fault simulation and the results are not generalized.

To sum up, multiplier structures, regardless of size, are seen to be very effectively tested with repetitive patterns and pseudorandom sequences are seen to be much more effective than regular patterns. Hierarchical faulting is seen to be more pessimistic than board level faulting, and the importance of hierarchical primitive levels is seen to be of great influence on the fault coverage results. A high weight signature analyzer is seen to be very effective in output data compression, with almost no difference from direct measurement.

IX-SUGGESTIONS FOR FUTURE WORK

Due to long time overhead of synthesis process and fault simulation, the achieved goals within the allocated time slot for the project do not cover all the desired aims. Some of the future work that can be performed along this project must be investigation of BIST techniques for MAC structures and design and synthesis of a complete system with MAC circuit and the BIST circuit, preferably again completely parameterized.

Another point of direction is determination of a deterministic set of vectors that perform a compact test. An initiation to this might be [26], for a c-testable design. Yet, another approach for a deterministic set of tests might be experimentally observing the already documented results and begin with a good starting point for the test and with exhaustively searching the test vector space and observing the detection ability of individual patterns in the histogram plot provided by QuickFault.

Other multiplier structures can also be investigated, but as already stated in [3], different structures have very similar fault coverage properties and the results are already well anticipated.

As the hierarchical fault models do not provide a generalized fault coverage characteristic due their variety, a standard fault model similar to CFM described in [23] can be investigated and the possibility of applying this model via a single stuck at model fault simulator, like QuickFault can be discussed. A starting point to this might be [24].

X-References and Bibliography

X.1 - References

[1] Russell, G. and Ian L. Sayers, "*Advanced Simulation and Test Methodologies for VLSI Design*", London: Van Nostrand Reinhold (International), 1989, ISBN 0-7476-0001-5

[2] Psarakis, M., D. Gizopoulos, A. Paschalis, N. Kranitis and Y. Zorian, "*Robust and Low-Cost BIST Architectures for Sequential Fault Testing in Datapath Multipliers*", VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001, pp. 15-20, 2001

[3] Gizopoulos, D., A. Paschalis and Y. Zorian, "*An Effective Built-In Self-Test Scheme for Parallel Multipliers*", IEEE Trans. on Computers, vol. 48, no. 9, pp. 936-950, Sep. 1999

[4] Raghunath, K. J., H. Farrokh, N. Naganathan, M. Rambaud, K. Mondal, F. Masci and M. Hollopeter, "*A Compact Carry Save Multiplier Architecture and Its Applications*", Circuits and Systems, 1997. Proceedings of the 40th Midwest Symposium on, vol. 2, pp. 794 –797, 1998

[5] Gizopoulos, D., A. Paschalis and Y. Zorian, "*Effective built-in self test for Booth multipliers*", IEEE Design & Test of Computers, vol. 15, no. 3, pp. 105 -111, July-Sep. 1998

[6] Weste N. H. E. and K. Eshragian, "*Principles of CMOS VLSI Design, A Systems Perspective*", Massachusetts: Addison-Wesley, 1993, ISBN 0-201-08222-5

[7] Pirsch, P., "*Architectures for Digital Signal Processing*", Chichester: John Wiley & Sons, 1998, ISBN 0-471-97145-6

[8] Morling R. C. S. And I, Kale, lecture notes, "*DSP and Communication Processor Design*", MSc VLSI System Design, Univ. of Westminster, Feb. 2001

[9] "*DSP Design Slides, Chapter 13*", Lund University, accessed via Netscape, 12nd July 2001, at http://www.tde.lth.se/ugradcourses/DSPDesign/slides/chap13.pdf

[10] Omondi, A. R., "*Computer Arithmetic Systems Algorithms, Architecture and Implementations*", New Jersey: Prentice Hall, 1994, ISBN 0-13-334301-4

[11] Roth, C. H., "*Digital Systems Design Using VHDL*", Boston: PWS, 1998, ISBN 0-534-95099-X

[12] Pucknell, D. A. and K. Eshragian, "*Basic VLSI Design*", Australia: Prentice Hall, Third Edition, 1994, ISBN 0-13-079153-9

[13] Hurst, S. L., “*VLSI Testing, Digital and Mixed Analogue/Digital Techniques*”, London: IEE Circuits, Devices and Systems Series 9, 1998, ISBN 0-85296-901-5

[14] Bardell P. H., McAnney W. H. and Savir J., “*Built-In Test for VLSI: Pseudorandom Techniques*”, New York: John Wiley & Sons, 1987, ISBN 0-471-62463-2

[15] Lala, P. G., “*Digital Circuit Testing and Testability*”, San Diego: Academic Press, 1997, ISBN 0-12-434330-9

[16] Eldred, R.D., "*Test Routines Based on Symbolic Logical Statements*", Journal ACM, vol.6, no.1, 1959, pp.33-36

[17] Wu, C. W., EE6250: VLSI Testing and Design for Testability course notes, “*VLSI Testing and Design for Testability, Chapter-3: Testability Measures”*, Department of Electrical Engineering, National Tsing Hua University, Taiwan, accessed via Netscape, 11th Aug 2001, at http://larc.ee.nthu.edu.tw/~cww/n/625/6250/03.pdf

[18] Vemuri, R., ECECS 682: VLSI Test and Validation course notes, “*Design for Testability*”,Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, US, , accessed via Netscape, 11th Aug 2001, at http://www.ececs.uc.edu/~ranga/courses/682/slides-spring-2001/DFTnew.pdf

[19] Goel, P., "*Test Generation Costs Analysis and Projections*", IEEE Proceedings on the Design Automation Conference, June 1980, pp. 77-84

[20] Mentor Graphics Corp., “*Fault Analysis User’s Manual: Software version V8.5_1*”, Unpublished, Oregon, 1995

[21] Mentor Graphics Corp., “*Synthesizing with Autologic II: Software version V8.5_1*”, Unpublished, Oregon, 1995

[22] Shah, S., A. J. Al-Khalili, D. Al-Khalili, “*Comparison of 32-bit multipliers for various performance measures***”,** Microelectronics, 2000, ICM 2000, Proceedings of the 12th International Conference on, pp. 75 – 80, Nov. 2000

[23] Kautz, W. H., “*Testing for Faults in Cellular Logic Arrays*”, proceedings of the 8th Annual Symposium on Switching and Automata Theory, pp. 161-174, 1967

[24] Psarakis, M., D. Gizopoulos and A. Paschalis, “*Test Generation and Fault Simulation for Cell Fault Model Using Stuck at Fault Model Based Test Tools*”, Electronic Testing Journal: Theory and Applications, vol. 13, no:3, Dec. 1998

[25] Morling, R. C. S., “*Production Test of Digital Integrated Circuits*”, MSc VLSI System Design, Univ. of Westminster, Dec. 2000

[26] Friedman, A. D., “*Easily Testable Iterative Systems*”, IEEE Trans. on Computers, vol. C-22, no. 12, pp.1061-1065, Dec. 1973
(Theoretical Information about C-testability and iterative arrays)

[27] Roth, J. P., “*Diagnosis of Automata Failures: A calculus and a method*”, IBM Journal of Research and Development, pp. 278-291, 1966

[28] Eichelburger, E. B. and E. Lindbloom, “*Random Pattern Coverage Enhancement and Diagnostics for LSSD Self-Test*”, IBM Journal of Research and Development, pp. 265-272, 1983

X.2 - BIBLIOGRAPHY

[1] Bennets, R. G., “*Introduction to Digital Board Testing*”, New York: Crane Russak, 1982, ISBN –8448-1385-0, pp. 107-114 & 220-223 & 165-172
(Information on Signature Analysis and Test Pattern Generation Problems)

[2] Kaniz M., M. Cole and S. Mourad, “*Mentor Graphics Quickfault II Tutorial*”, Department of Electrical Engineering, Santa Clara University, Santa Clara, US, accessed via Netscape, 27th July 2001, at http://www.scudc.scu.edu/mentortu/mg_quickflt.html
(Quick to apply, but incomprehensive information about fault simulation)

[3] Mentor Graphics Corp., “*SimView Common Simulation User’s Manual, Software Version 8.5_1l*”, Unpublished, Oregon, 1995
(Worthy information about dofiles, waveform databases, scripts and batch simulation)

[4] Bhasker, J., “*A VHDL Primer*”, New Jersey: Prentice Hall, Revised Edition, 1995, ISBN 0-13-181447-8
(Well presented information on VHDL for inexperienced users)

[5] Naylor, D. and S. Jones, “*VHDL: A Logic Synthesis Approach*”, London: Chapman & Hall, 1997, ISBN 0-412-61650-5
(Information about design → synthesis)

[6] Morling R. C. S., “*Designing for Testability*”, MSc VLSI System Design, Univ. of Westminster, Dec. 2000
(Very brief information on DfT and fault simulation terminology)

XI-APPENDICES

A – GENERATED VHDL CODES

B – STRUCTURAL DESIGN SCHEMATICS IN RENOIR

C – VHDL LEVEL SIMULATION RESULTS

D – FAULT SIMULATION DATA

E – MATLAB SCRIPTS