A Built–In Testable Architecture and Design Method for On–Chip Error Correction Circuits of Embedded Memories in VLSI/ASIC Systems

by

Lianfa Cui

A thesis

presented to the University of Manitoba

in partial fulfillment of the

requirements for the degree of

Master of Science

in Electrical and Computer Engineering

Winnipeg, Manitoba, Canada

ISBN 0-315-99037-6

Name Lianfa Cui

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Applied Sciences — SUBJECT TERM

0544 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

A BUILT-IN TESTABLE ARCHITECTURE AND DESIGN METHOD FOR ON-CHIP ERROR CORRECTION CIRCUITS OF EMBEDDED MEMORIES IN VLSI/ASIC SYSTEMS

BY

LIANFA CUI

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis presents the development of an on–chip built–in testable error correction circuit (on–chip BIT ECC) for embedded memories in VLSI/ASIC systems. First, error control coding (ECC) for high–speed memories is briefly reviewed, focus is placed on the modified Hamming (n,k) SEC–DED codes and a parallel VLSI implementation. Second, built–in test as a solution to VLSI/ASIC designs is studied, a built–in testable architecture and design method for on–chip ECCs for embedded memories is proposed. The design represents a solution for built–in self testing of embedded memories with on–chip ECC in a divide–and–conquer strategy. It features: i) >99% fault coverage for the stuck–at faults in the on–chip ECC circuitry; ii) integrates the advantages of multiple BIST technologies, such as pseudorandom test, pseudoconcurrent/interleaved test, and scan test; iii) impacts the system performance by only one gate delay regardless of the size of the on–chip ECC implemented; and iv) uses up to 50% less on–chip test generation hardware and reduced testing time overhead than customary BIST implementations, as a result of test stimulus compaction and mappings dictated by the ECC codes implemented. The same built–in test circuitry can be dynamically reconfigured under software control to support four self test modes, corresponding to different BIST environments. The design method has been implemented and verified, through the development of a built–in testable Hamming (22,16) SEC–DED circuit, using the Cadence™ CAD tools, Verilog HDL, and Verilog–XL simulator. A prototype was produced using Xilinx FPGAs under the same EDA environment.

ACKNOWLEDGEMENT

I wish to express my sincere thanks to Dr. Robert D. McLeod, my advisor, for his excellent guidance, thoughtful discussions and suggestions throughout the course of this thesis research. I am also very much grateful for his financial support for my graduate study.

Special thanks go to many of my colleagues with the VLSI Research & Systems Group at University of Manitoba. I am grateful, in particular, to Ph.D. candidate Zaifu Zhang for his comments and suggestions on the subject of the study, to Dave Blight and Tapas Shome, and Mr. Martin Meier for their constant willingness to help in various VLSI/ASIC EDA tools that are extensively used in this thesis work.

My supervisor, Mr. Bradley Brown, and my colleagues with IDers Incorporated deserve much the same gratitude from me. The ACORN ASIC project I was engaged in as ASIC project engineer, interleaved with my graduate research at the University, rewarded me with invaluable experience in commercial testable VLSI/ASIC design.

I am also indebted to my wife, Shuyun, and my daughter, Xiao, for their consistent encouragement and support in many ways for my graduate study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BIT	Built–In Test
BIST	Built–In Self Test
BISD	Built–In Self Diagnosis
BILBO	Bidirectional Internal Logic Block Observation
COR	(error) Corrector
CUT	Circuit Under Test
DbED	Double–byte Error Detection
DFT	Design For Testability
DED	Double Error Detection
DSP	Digital Signal Processor
ECC	Error Control Circuit
EEROM	Electrically Erasable Read Only Memory
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
GUI	Graphics User Interface
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
LFSR	Linear Feedback Shift Register
MISR	Multiple Input Signature Register
PG	Parity Generation
PRTG	Pseudorandom Test Generation
PUST	Power Up Self Test
RISC	Reduced Instruction Set Computers
SbEC	Single–byte Error Correction
SD	Syndrome Decoder
SEC	Single Error Correction
SEC–DED	Single Error Correction and Double Error Detection
SG	Syndrome Generation
SST	Simultaneous Self Test
TAP	Test Access Port
TMR	Triple Modular Redundancy
VLSI	Very Large Scale Integration

CHAPTER 1

INTRODUCTION

The evolution of computer architecture and the advance of VLSI technology have enabled advanced features, such as higher integration of VLSI processor (signal processor) chip sets, and system–on–a–chip ASICs. VLSI/ASIC solutions are key to many technology companies keeping pace in today's competitive market. One of the dominant characteristics of a VLSI system solution is that it may include processors (RISC processors, DSP cores), memories, system control and interface logic, and perhaps most of the system peripherals on the same chip [WILS93]. This dramatically improves system functionality, speed, reliability, and serviceability of modern computer/communications systems. At the same time, this has presented serious testing obstacles. The design may be beyond the reach of design simulation tools, making it virtually untestable.

Fault tolerant VLSI/ASIC systems trade–off hardware/software redundancy and operating speed to achieve a higher degree of system reliability and availability [WALT90]. The use of error control coding (ECC) techniques in high speed memories (i.e. cache memories, private memories, embedded memories) may be one of the simplest ways in which this redundancy–speed–reliability trade–off is implemented. Error control codes (ECC) have been applied in various aspects of the modern computer systems: memory, arithmetic, logic, and communications [RaFu89]. For high performance embedded memory units in VLSI/ASIC systems, on–chip ECC is the principle candidate to ensure data integrity of the memory internal to the system. On–chip ECC circuits consume about 10% to 20% of the total on–chip memory area, and help enhance theoretical yield by a factor of 3 (at about 2.5 particles/cm^2 defect density), and depress the soft error rate at a factor of 10 (at an α–flux

density of 1 cm^{-2} h^{-1}) [FaSa91, RaFu89, FuAr89, MANO83]. On–chip ECC is also regarded as one of the concurrent test methods, based upon its capability of detecting and/or correcting errors on the fly and reporting error occurrence synchronously. Still, questions may exist such as: What if on–chip ECCs fail due to faults on the on–chip ECC circuit itself with the same error rate as on the memory cell arrays? What effective measures should be taken to capture a faulty on–chip ECC, if any, at run times to ensure the concurrent judge of data integrity is judging correctly?

In any event, embedded memory is becoming a key component of VLSI/ASIC system chips, as memory bandwidth is one of the most serious bottlenecks to the system performance. In addition, today's silicon VLSI/ASIC process technology can afford large memory integration with other logic circuitry. For the sake of improved system reliability, embedded memories with built–in ECC (or on–chip ECC) are integrated in VLSI/ASIC system chips. The challenging task is the vendor and manufacturing test of those embedded memories with built–in ECC. In embedded memories, the address, data, and read/write inputs may not be directly controllable and the data output may not be directly observable through the I/O pins of the VLSI/ASIC chip. As a matter of fact, on–chip ECC is internal to the embedded memories, and transparent to the user. Though self testing is a solution to the problem, the issue of testing embedded memories with on–chip ECC in its entirety has not yet been addressed in the literature. Built–in self–test (BIST) methods for embedded RAMs or ROMs have been proposed, which aim at self testing of the memory cell arrays [JaSt86, SuWa84, FASA90, ZoIv92], or aim at self testing of the information bits and check bits of memory cell arrays when on–chip ECC is employed [FrSa91].

On–chip ECC circuitry is sizable enough to itself warrant provisions for testability. Being fully combinational, as the parallel implementation of Hamming (n,k) SEC–DED code in this thesis, the on–chip ECC circuitry is a prime candidate for built–in self test. The

ultimate objective of this thesis study is, therefore, to develop a built–in testable architecture and a design method for the on–chip ECC. It shall provide a design–for–testability method for modular development of on–chip built–in testable ECCs (on–chip BIT ECCs) for embedded memories in VLSI/ASIC systems. It shall employ a divide–and–conquer strategy for self testing embedded memories with on–chip ECC in its entirety. It shall fully support dynamic system reconfigurability by taking part in automatic fault reporting logic of the system. Furthermore, it shall support self–testing an array of homogeneous embedded memories effectively under a single or multiple–processor based system environment. In a homogeneous multi–processor system all local (private) memories with built–in ECCs may have same configuration, test vectors can be supplied to all on–chip ECC circuits on the system chip in parallel from a single on–chip test generation circuit, and self testing of those on–chip ECCs for the embedded private memory arrays can all be done in parallel. Another objective of this thesis study is to minimize the size of this on–chip test generator, given an on–chip ECC of any size.

The test can then be used on demand, as in power–up self test (POST) or diagnostic test. It can also be repeated at dedicated intervals of time interleaved with the process. In the case of interleaved self test, a very short test sequence is desirable in order to minimize the impact on processor performance. Another objective of this thesis study is to minimize the test time.

The interleaved self test can be taken as pseudoconcurrent test. It allows the on–chip ECC to make use of the processor's idle time to perform partial self test, without stealing any time away from normal processor operation. In a scan test environment, the same built–in test hardware should be reconfigured under software control to form a portion of register chain to support scan test of the on–chip ECC circuit itself or other system modules on the same register chain in the system. Another objective of this thesis study is, therefore,

to develop dynamically reconfigurable (programmable) built–in test circuitry to support multiple self test methods integrated in the proposed built–in testable architecture.

In this thesis, Chapter 2 will briefly review error control coding (ECC) theory and techniques for high speed memories. Generic memory–chip codes and on–chip ECC structure shall be described in the light of improving yield and error rate. In addition, a modified Hamming (n, k) SEC–DED codes and an optimized (parallel, bit–sliced, and fully combinational) VLSI implementation are examined in detail.

Chapter 3 gives an overview of built–in test for VLSI/ASIC systems under the heading of design–for–testability. Fundamentals in modern test theory such as structural test, fault models and fault coverage, fault grading methods are studied; followed by discussions of some of the built–in test technologies and structures in testable VLSI/ASIC designs. It shows that built–in test has become a portion of VLSI/ASIC design.

Chapter 4 starts with testability analysis on the major components of an optimized on–chip ECC implementation for the modified Hamming (22, 16) SEC–DED codes. A natural partition for testability is then proposed. Different stages in developing a compact set of test vectors that guarantees >99% coverage of single stuck–at faults in the on–chip ECC circuitry is described. It is noted that the test vector matrix compaction rule, dictated by the ECC codes, resulted in up to 50% less on–chip test generation hardware and a reduced testing time than conventional BIST implementations.

Also included in Chapter 4 is a BIST architecture for an on–chip ECC that integrates the advantages of multiple BIST technologies: pseudorandom test, pseudoconcurrent test, scan test and scan path. This feature is facilitated through dynamic reconfigurable hardware design of the built–in test circuitry. Four programmable self test modes are dedicated

to different BIST environments under which the VLSI/ASIC system may operate. The control signal and intermittent status for coordinating the self tests are explicitly defined in this chapter.

Chapter 5 presents a structured modular VLSI implementation of all components and the integrated built–in testable on–chip ECC module itself. Considerations behind each of the configurations are presented. The measure taken to feature particular logic designs, for instance, to suppress aliasing problems inherently with signature generation techniques, are described.

The integrated VLSI EDA tools from Cadence Design Systems Inc. are extensively used throughout from design implementation, simulation and verification, to rapid prototyping. It is worthwhile to mention that the hardware description language (HDL) Verilog (as high level simulation language) was used in the development of stimulus for design simulations using the high performance Verilog–XL simulator; Xilinx FPGA development package, and Xilinx 4000 series technology library cells and devices were used, which enabled us to implement, verify and validate the built–in testable design method in a well–structured and easy–to–implement way. A brief description of using the Verilog HDL and high performance Verilog–XL simulator under the integrated Cadence™ (V4.2) EDA environment is also included in Chapter 5.

Chapter 5 concludes with rapid design prototyping and validation. After the design implementation has been verified through logic and timing simulation, design prototyping is illustrated by downloading the netlist information of the design onto a target Xilinx field programmable gate array (FPGA) chip on a demonstration board. The Xilinx Development System with Xilinx 4000 family demonstration board integrated into the Cadence™ (V4.2) on a desktop of SparcStation 10 under UNIX operating system was used. Evaluations of

the prototype on the FPGA device XC4003b are presented, which help to validate the solution previously proposed and implemented.

Chapter 6 draws the conclusions of this thesis research work. Further development of this research work are proposed. A direct work plan may include the investigations of approaches and techniques to convert the modular built–in testable design method into VLSI/ASIC design aids in EDA systems.

CHAPTER 2

ON–CHIP ERROR CHECKING AND CORRECTING

2.1 Introduction

In computer systems, large amounts of data move between various subsystems. For instance, the data traffic between the CPU and main memory may be of the order of 100 million bits every second. Even though the system is designed for very high reliability, there are bound to be a few errors in these communications caused by such things as the atmospherics noise, electrical noise, component or device malfunctions, or sometimes design or program faults. It is imperative that the system detects/corrects these errors as and when they occur. Some remedial action such as error correction or error recovery must take place before a more serious situation like a system crash arises.

Error control coding for computer systems is now an established field of study [RaFu89]. It is an extension of error control coding for communications, but stresses the problems of reliable computation which significantly differs from the problems of reliable communications. For instance, the later assumes perfect reliable computing and processing at the transmitter and receiver, and has less severe restraints on computation time for error correction. Further, in communication systems error detection combined with acknowledgement and retransmission protocols often provides a satisfactory method of obtaining reliable communication in the presence of communication channel errors. However, in computing if the presence of errors is detected in a word retrieved from memory, there may be no way of determining what the correct word is.

In this chapter we shall briefly review various aspects of the on–chip ECC codes for high speed memories. Generic on–chip ECC structures are described with respect to chip yield and error rate improvement. The modified Hamming (n, k) SEC–DED codes and a parallel, bit–sliced, and fully combinational VLSI implementation are to be examined in detail, illustrating that self testing is a solution for embedded on–chip ECC logic. We shall also take a brief look at some other fault tolerant methods such as self checking for on–chip ECCs. We defer the testability analysis of the optimized implementation of the Hamming (n,k) SEC–DED codes to Chapter 4 where we shall elaborates the controllability and observability (testability) of each of the major components of the on–chip ECC circuit and propose a design–for–testability (DFT) solution to a modular on–chip built–in testable ECC design.

2.2 Codes for high speed memories

Several types of error correction codes (ECC) have been successfully applied to memory systems, including high speed memories (i.e. control memories, cache memories, private memories, and embedded memories). Every memory designer has adopted for some form of error checking or correcting (ECC) code in order to enhance reliability. The ECC codes used at present for on–chip ECC are from the class of linear codes, and involve adding check bits to the information bits. The embedding ECC logic is transparent to the user. For improved reliability, concurrent ECC capacity has been found essential for over megabyte–level chip designs [FuAr89, KALT90].

There are basically two categories of ECC codes for high speed memories: bit error correcting/detecting codes and byte (burst) oriented error correcting/detecting codes. In each category, code classes are designated based on the capacity features of the codes. There exist many resources in the literature on coding theory and techniques for high speed

memories and embedded memories [FuPr90, YAMA88, PETE61, RaFu89, LoHu 88, FASA90]. In [RaFu89] two chapters are dedicated to error control codes for high speed memories. Included are modified Hamming SEC–DED codes, memory chip codes, double–bit–error correcting codes, code design techniques for high speed memories, and single byte–error detecting (SbEC) codes, single–byte–error correcting and double–byte–error detecting (SbEC–DbED) codes, and byte/burst–error detecting SEC–DED codes.

One of notable features of the codes for high speed memories is that parallel encoding and decoding is required to maintain high rates of throughput. Therefore, the encoding and decoding circuits are implemented by combinational logic instead of linear feedback shift registers (LFSR).

We restrict our discussion to the class of modified Hamming (n,k) SEC–DED codes, one of the most widely used on–chip (n,k) codes for high speed memories. A modified Hamming (22,16) SEC–DED code and its parallel VLSI implementation had been selected in this thesis as a starting point for developing a built–in testable architecture and design–for–testability method for an on–chip Hamming (n,k) SEC–DED circuit.

2.3 Modified Hamming (n,k) SEC–DED codes

A Hamming SEC–DED code can correct single–bit errors and detect double–bit errors. This code can be formed by extending a Hamming SEC code with an overall parity check, that is, a check on all the symbols in the code. The H matrix of an original Hamming (n,k) SEC–DED code is shown in Fig. 2–1. Here n represents the number of bits in a coded memory word, k the number of bits in original memory word. The original Hamming (n,k) SEC–DED code can be modified and optimized, by applying algebraic operations over the rows and columns of its H matrix. The resulting code is called a modified Hamming (n,k)

SEC–DED code. The H matrix of modified Hamming (22,16) SEC–DED code (also known as Hsiao code) is shown in Figure 2–2.

$$\mathbf{H} = \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0 \\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0 \\ 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \end{bmatrix}$$

Fig. 2–1 H matrix of original Hamming (22,16) SEC–DED code

$$\mathbf{H} = \begin{bmatrix} 1&1&0&1&1&0&0&0&1&1&1&0&0&1&0&0&1&0&0&0&0&0 \\ 1&0&1&1&0&1&1&0&1&0&0&1&0&0&1&0&0&1&0&0&0&0 \\ 0&1&1&0&1&1&0&1&0&1&0&0&1&0&0&1&0&0&1&0&0&0 \\ 1&1&1&0&0&0&1&1&0&0&1&1&1&0&0&0&0&0&0&1&0&0 \\ 0&0&0&1&1&1&1&1&0&0&0&0&0&1&1&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&1 \end{bmatrix}$$

Fig. 2–2 H matrix of modified Hamming (22,16) SEC–DED code

The minimum distance of a SEC–DED code is at least 4. Since an n–tuple of weight 3 or less is not a codedword, any set of three columns of the H matrix should be linearly independent. Note that the sum of two odd–weight r tuples is an even–weight r tuple. For this property, a SEC–DED code with r check bits can be constructed with its H matrix being constituted of distinct nonzero r–tuples of column vectors having odd weight [HSIA70]. This code is different from the original Hamming SEC–DED code whose H matrix has an all 1's row vector in addition to the SEC code H matrix. The modified Hamming code, more specifically, is an odd–weight–column SEC–DED code, for every H matrix column vector is odd weight. It is also noted the modified Hamming (22,16) SEC–DED code has a optimal

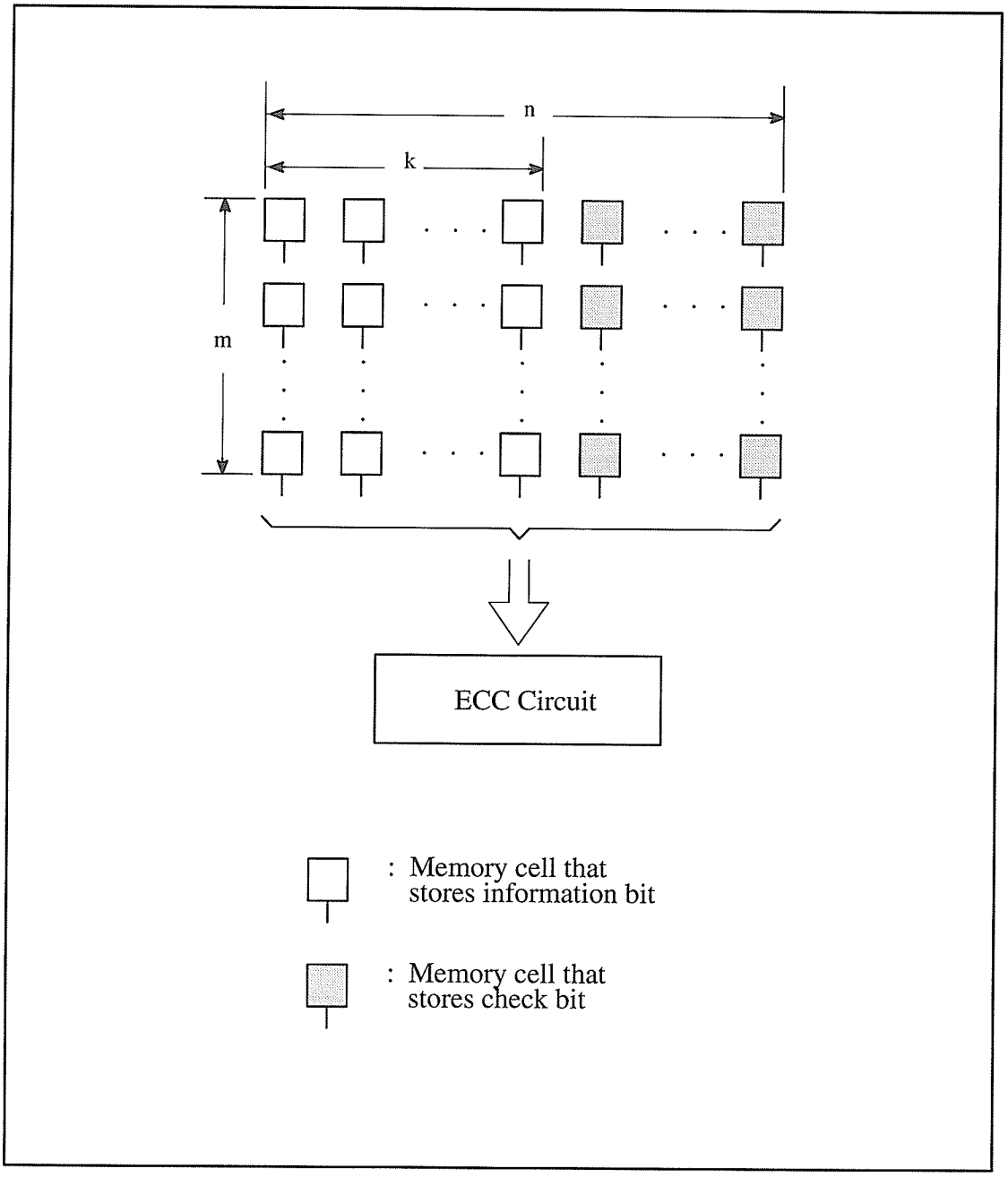


Fig. 2–3 Memory model using on–chip (n,k) codes

minimum number of 1's in its H matrix, which makes the hardware implementation and the speed of encoding/decoding circuit optimal. It satisfies the condition of minimum–equal–weight code, and hence it is called an optimal code from the practical point of view.

Figure 2–3 illustrates a memory model using on–chip (n,k) codes. For simplicity, we have illustrated an example H matrix (with n=22 and k=16), rather than to give complex mathematic expressions to construct this code. From practical view of point, it is important to realize how the ECC structure effects real hardware implementations of the code concerned; and how the ECCs such as the Hamming (n,k) SEC–DED code minimizes the real probability of miscorrection whenever triple or more errors occur. A miscorrection here refers to an erroneous decoding that results in an actual increase in the number of errors in the decoded word. This may happen when a syndrome pattern coincides with some column of H, then the decoder mistakes it for a single error and applies a miscorrection.

It is found that odd–weight–column SEC–DED codes have practical advantages and also a lower probability of erroneous decoding and are widely implemented in such computer systems as IBM 370/168, 303X, 308X, 4300 series, Cray 1, Tandem, and so on [CHEN84].

2.4 Generic on–chip ECC structure and logic implementation

From this section on, we will refer discussions of on–chip ECC only to its decoding architecture and parallel logic implementation. A block diagram of a generic on–chip ECC (decoding) circuit is shown in Fig. 2–4. This circuit consists of a parity generator (PG), a syndrome generator (SG), a syndrome decoder (SD), and a correction circuit (COR).

Assume a (n,k) code is to be implemented, PG will generate r (r = n–k) parity bits,

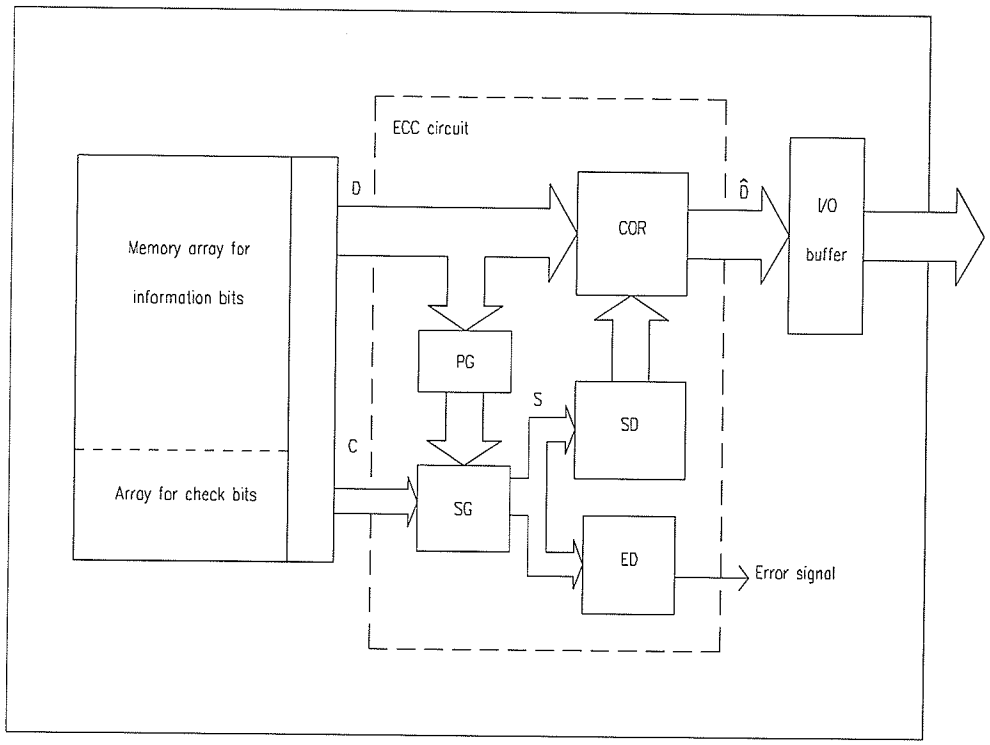


Fig. 2–4 A generic on–chip ECC model

and SG will generate r syndrome bits from the n–bit input data (k information bits and r check bits) simultaneously. SD will decode the syndromes and generate k error patterns that indicate if an error occurred in any bit, and COR will generate the corrected data using the error patterns. Usually, COR does not generate the corrected check bits. ED belongs to the synchronous error/fault reporting system and will generate the error signals to indicate the occurrence and the types of memory error(s).

As mentioned earlier, an on–chip ECC circuit is composed mainly of extra memory

cells and the error decoding circuit. In order to implement an on–chip ECC circuit, it is necessary to reduce the silicon area used for the ECC circuitry and the encoding, decoding, and correcting logic delay. Figure 2–5 shows an implementation with logic diagram of an on–chip ECC decoding circuit for a modified Hamming (22,16) SEC–DED code.

2.5 Self test solution vs. self checking method

It is a fact that an increase in the memory readout data size will increase the size of the on–chip ECC circuit, and therefore increase the probability of circuit failure. The ECC portion of the system is exposed to the same error prone environment as the memory cells due to physical defects or operational conditions. Thus the ECC circuit has to withstand failure in itself without transmitting erroneous data. Research work on fault tolerant ECC designs using built–in self checking have been reported [GAIT 88]. The simplest method for self–checking of an ECC circuit is to re–enter the corrected output, including corrected check bits, to another syndrome generator; if the syndromes are all zeros, then it is assumed the ECC circuit is fault free. The built–in test solution represents more recently developed self–testing methods for VLSI/ASIC designs. It holds the same capacity of fault detection, but emphasizes structured modeling of the type of faults expected and providing test vectors with high fault coverage. Fault analysis is performed on a nodel bases for the circuit under test. This feature is imperative in the initial stages of design development, furthering the quality of the design. It may also be extended to support built–in self–diagnosis (BISD) methods in which self–repair may be undertaken if the VLSI/WSI design itself is internally repairable [TrAg93, LoHu88].

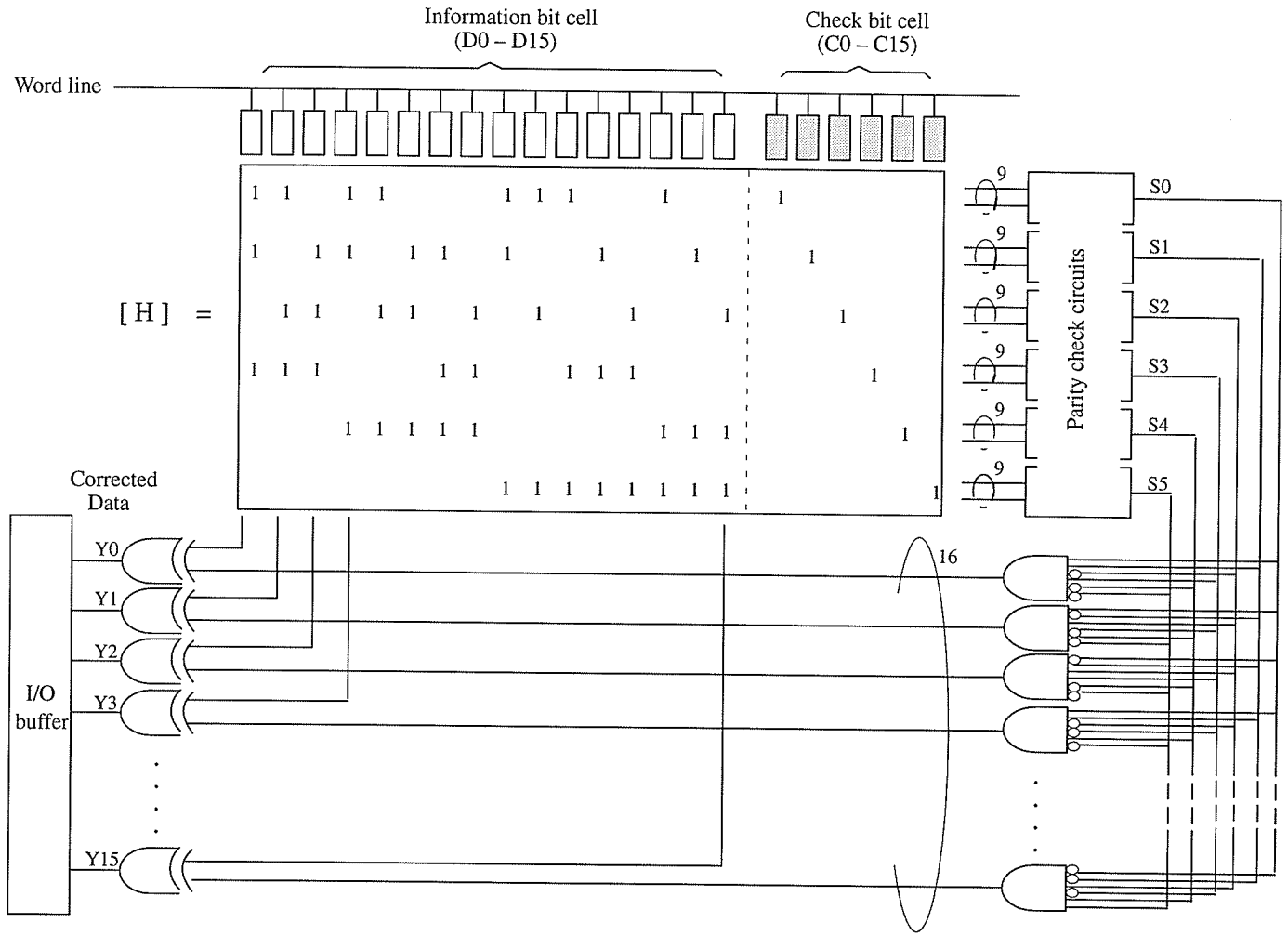


Fig. 2–5 Implementation scheme and logic diagram of an on–chip ECC circuit for a modified Hamming (22,16) SEC–DED code

CHAPTER 3

BUILT–IN TEST FOR VLSI/ASIC

3.1 Introduction

Testing of VLSI/ASIC designs is a major portion of the effort in their design, and manufacturing, and even in their operations at run time. Ever–increasing levels of integration, and innovations of fault tolerant VLSI/ASIC system architecture demand that VLSI/ASIC test technology (i.e. test hardware, test software, and test theory) evolve to a high degree of sophistication. As the number of transistors that can be integrated into one piece of silicon approaches millions, it would seem that some small portion of those circuits could be devoted to testing the function the remainder are to implement. This concept is called "Built–In Test" [McCL86, BARD87, MILL88, ARGA93]. Built–in test (BIT) represents a philosophy in VLSI, namely, design for testability (DFT).

In this chapter, we shall start with a brief review of some of the fundamental but important concepts related to the modern test theory and technology [WiPa83, MCCL86]. We shall study general guidelines, applied techniques and evaluation methods for built–in testable VLSI/ASIC design. We shall proceed with built–in test architectures or circuit structures, in respect to test generation, test response compression and analysis, and test control and support integration. This study would serve as theoretical and technology preparation for developing the proposed BIT solution to on–chip ECC design for embedded memories. The design method, implementation, verification, and validation are to be covered in the next two chapters.

3.2 Built–in test and external–applied test

Built–in test as a major design–for–testability technology addresses the test problems associated with digital networks of the size that are encountered in working with VLSI/ASIC systems. As increased circuit density continues to force very large scale integration (VLSI) to grow, conventional externally applied tests, structural or functional, become less and less satisfactory. Alternative to the externally applied tests is built–in test. Built–in test refers to any digital test technique where the generation of the tests and the mechanism for analyzing the responses to these tests is an integral part of the circuit being designed. The dependence on the latest, most sophisticated and costly test systems are lessened to a great extent [ABRA90].

Providing built–in test has now become a part of the VLSI/ASIC design process. As VLSI reaches commercial applications, quality demands will force some form of design–for–testability into the system. Taking silicon from the functionality of the design into built–in test functions is greatly enhanced by the capacity of VLSI digital circuits themselves. The incremental circuit and area overhead taken by the built–in test structures can be a small (or affordable) price to pay for the assured testability and the resultant quality that ensues.

3.3 Structural test and functional test

In 1959, R. D. Eldred showed that there was an effective way to test the hardware of a system rather than its function [BARD87]. This was the beginning of structural test as we know it today. The structural approach proposes that a digital network or system can be described in terms of logic primitives and macros (AND, OR, NOT, XOR, FLIP–FLOP and ADDERS). If a test is generated to test the fault of each logic elements in turn, or to

make the test pattern more effective and practical according to test algorithms, the faulty component(s) could be detected and located, and then be removed or substituted by a stand–by. In contrast, functional test demonstrates an adder could add and so forth, which provides a more direct measure in design verification and manufacturing test of a digital network or system. It is recognized that in some cases it is not practical and economically impossible to exhaustively test all intended functions of a VLSI/ASIC system of sophisticated complexity or of poor testability.

Built–in test can be either structural or functional in nature. Pseudorandom test and scan test are types of built–in test that perform a structural test of the digital network or system. A particularly appealing feature of psudorandom test is that the test patterns can be generated by simple built–in generators. There are other built–in test methods which perform either functional or structural tests, they either work with stored deterministic test patterns or execute functional programs that are designed to exercise specific parts of the digital network or system.

3.4 Fault models and fault coverage

The structural approach vs. the functional test also suggests that there is a model of faults of faults to be detected. Defects introduced during manufacturing, such as, open interconnections, shorts between conductors, excess leakage current and others, or from service–related problems such as electromigration, overloading, or burnout will all affect the logical behavior of the network. Fault modeling in a manner that is consistent with the representation of the network abstracts the effect of a physical defect into a stuck or faulty condition on one of the terminals of the gate that hosts the defect. This represents a systematic way to assess the malfunction of the network caused by a variety of actual defect(s).

The circuit fault model specifies the range of physical defects that can be detected by a given test procedure. Take the classic stuck–at model as an example, it is generalized to apply to any fault condition that causes a logic gate to behave as though one of its inputs or outputs is stuck at logic 1 or 0. Stuck–at faults affect only the interconnections (or lines) between gates in a logic circuit. The defect can be caused by connections within the gate being open, as well as by shorts to ground (stuck–at–0), or to the potential of the power supply (stuck–at–1) and so forth.

The stuck–at model has been the mainstay of the development of test theory. More complexed fault models have been proposed [MoTh86, ZHAN94]. Some of them are as follows:

- Sequential Fault Model

  If two or more lines that are shorted together and form a feedback path that creates a new state in which the network can exist, the fault is called a sequential fault.

- Combinational Fault Model

  If two or more lines that are shorted together do not form a feedback path that creates a new state in which the network can exist, the fault is called a combinational fault.

- Parametric Fault Model

  For instance, if a fault causes abnormal currents the fault is classified as a parametric fault.

- Pattern Sensitive Fault Model

If the effect of the fault is dependent on the state of some other circuit in the network, the fault is called pattern sensitive fault.

- Delay Fault Model

  If a fault causes a combinational circuit to fail to propagate data in time for clocking into next stage, the fault is classified to be a delay fault.

Fault coverage is a measure of test quality. Fault coverage analysis requires fault modeling. Fault models help in the generation of test patterns and in the evaluation of test quality in terms of coverage (the percentage of faults detected by a set of test patterns) of modeled faults [AgSc88]. Fault coverage can be determined by fault simulation, which may take a prohibitive amount of computer time for simulation of all faults in a large circuits in a VLSI /ASIC design. Recent EDA systems provide fault grading tools, such as Verifault from Cadence, QuickGrade II from Mentor Graphics. Some of those tools come with the graphics user interfaces (GUI), making them easier to use.

3.5 Design verification and test

Modern logic and timing simulation tools play an increasing important role in VLSI/ASIC design verification. During a simulation, the designer can apply many stimuli to the software model of the digital system and evaluate the responses. In this way, the designer can assure the design meets the logic and timing requirements. The simulation may be entirely in software on a general purpose computer workstation. Alternatively, the simulation can be done in a hybrid fashion, where those portion of the system available in hardware are used in conjunction with software models of the portion being evaluated [WIER93]. In either way, modern simulation tools associated with sophisticated VLSI/ ASIC EDA tools are now available to VLSI/ASIC designers for design verification.

Often the simulation is based on a register transfer level (RTL) of the design. Since this design must be translated to devices and interconnections at the silicon level, several sources of errors must be dealt with. Automatic programs that translate from RTL to gate level description and gate level descriptions to mask level layout descriptions are available in varied degree of sophistication. These automatic programs as CAD tools for VLSI/ ASIC design preserve the correctness of the design, thus allowing the design to rely heavily on the software verification done at the higher level. Simulation at lower levels, such as the simulation after the place and route would provide the design verification with a collection of more realistic parameters of the design such as the real circuit delays, and offers more sufficient timing verification of the design. A unit–delay model is often used in simulations at higher levels.

If sufficient design verification is done in the software stages of implementation, the purpose of the test is to determine that the hardware has been properly fabricated to the specification. The strict sense of test in this context is by no means to mislead to a "begins with design and ends with test" conjecture. In more and more technology companies, market demands for ever higher quality products are forcing a high testability requirement into initial specifications. When built–in test is incorporated in the VLSI/ASIC designs to meet testability specification, the distinction between design and test become moot. At that point, test is a part of design [BARD87].

3.6 On–line test and off–line test

When the system is operated as intended to perform some assigned function, it is said to be "on–line". Often it is desired to test the system after it has been put on–line. This may be to assure readiness for a critical mission or job, or may be required to monitor error recovery procedures, or to measure the fault–tolerant status of the system. If a procedure

to test the system is performed as a task in the job stream of the system while other tasks are ongoing, the test is said to be an on–line test. On the other hand, if the system must be shut down and/or dedicated to the test procedure in such a manner that normal system function stops completely, the test procedure is said to be an off–line test. In other words, the system must be taken off–line for the test to be performed.

The on–line test implies the test runs at full speed of the system that is under test and in normal operation. The off–line test may be performed at a speed that is lower than the one at which the system runs in normal operation.

3.7 Built–in test as a solution to testable VLSI/ASIC designs

The ultimate in testable VLSI/ASIC design is to make the design test itself. To meet testability specification, a built–in test architecture is a central task in developing any of testable VLSI/ASIC designs. Building test into the design, as expected, consumes added circuit and I/O overhead, but at the same time results in visible reductions to the costs of testing when compared with an external test using automatic test equipment. Built–in test achieves these savings through the following factors:

i) eliminating (or at least reducing) the costs of test pattern generation and fault simulation,
ii) shortening the time duration of tests (by running tests at circuit speeds),
iii) simplifying the external test equipment, and
iv) easily adapting to engineering changes.

Concurrent or on–line built–in test includes such methods as error detection and correction circuitry, totally self–checking circuits, self–verification, and others. It is worth

noting that some of the important techniques, such as triple modular redundancy (TMR), provide an instantaneous correction of errors caused by either permanent or intermittent faults. An off–line or built–in test of these circuits must ensure that the redundancy exists and is active. TMR is less reliable than its simplest version if it does not begin operation in a fault–free condition. A complete test of error correction circuitry (ECC) in this context is a mandatory requirement.

A nonconcurrent built–in test requires a mechanism for supplying test patterns to the circuit being tested and a means for comparing the responses from the circuit under test (CUT) to the known good responses as suggested in Figure 3–1. Both mechanism and means must be compact enough to reasonably be built into the circuit.

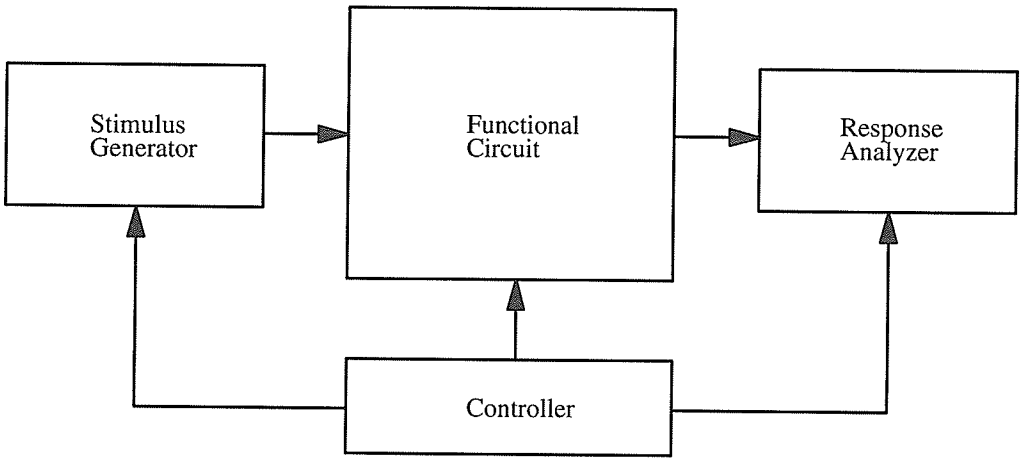


Fig. 3–1 A general nonconcurrent BIST structure

3.7.1 Built–in test methodologies

There are many ways to generate the tests, with the simplest categorization being

in terms of the type of testing used: 1) exhaustive test; 2) random test; 3) pre–stored test; and 4) functional test.

In exhaustive test, the test length is 2^n, where n is the number of inputs to the circuits under test (CUT). Since all possible test patterns are applied, all possible single and multiple stuck faults are detected (redundancies excepted). The tests are generated with any process that cycles exhaustively through the circuit input space, such as a binary counter, a Gray code generator, a LFSR, or a n–stage nonlinear feedback shift register. Exhaustive test for high input pin count structures requires relatively long test times, but it has been suggested that circuits can be added to partition such structures into subcircuits, each of whose input pin count is low enough to permit exhaustive test in a reasonable amount of time.

Random test implies the application of a randomly chosen subset of the 2^n possible input patterns. (Random testing is a misnomer because the tests are actually chosen pseudorandomly so that the test set is repeatable). A guarantee of the test coverage for the subset can be obtained by running the test against a fault model of the circuit, or a probabilistic measure of coverage can be obtained by analytical methods. The number of applied tests or the size of the subset is constrained by the economically allowable test time. While circuit partitioning is not needed, some logic modification may be necessary to ensure adequate coverage from the limited test set. Linear feedback shift register (LFSR) or cellular automata circuits [HoMc90] are the usual choices for a random test generator.

Pre–stored testing, on the other hand, requires a preliminary step of test generation. The cost of test generation can be offset by the savings in the test time resulting from a much smaller number of applied test. The certainty of a know test coverage is an added bonus. Given this test set, pre–stored test can be achieved in several ways. The simplest approach

is to store the test patterns in an on–chip ROM and to use a counter to cycle through the ROM addresses. For relatively complex circuits, the ROM may be rather large. Another approach to pre–stored test is a technique called "store–and–generate". A much smaller on–chip ROM, an address counter, and a linear feedback shift register (LFSR) are used, with the ROM contains r words of n bits each. Each of these r words is used sequentially as a string value (the seeds) for the n–stage LFSR. For each of the seeds, the LFSR will generate s vectors (of n bits each) that are applied to the n–input circuit as test patterns. The counter steps the LFSR s times for each address of the ROM. This total of r X s patterns is presumed to be a complete test set for the structural fault model used.

Functional test is a verification of the intended functions of the circuit. In a complex digital circuit many different failure models are possible. The assumption that faults can be modeled as logic gate inputs or output fixed to either a logic 0 or logic 1, as in the single stuck–at fault model, admittedly does not cover all these failure modes but has remained successful both because it is computationally feasible (for small circuits) and because it results in a qualitative measure of test coverage. Functional testing, being an alternative to the single stuck–fault model, is an approach that has been suggested to more realistically account for the actual effects of physical failures on logic. Failures are then modeled at the register transfer level (RTL) or the functional level in terms of variations in expected function. One requirement for successful functional test generation is that it isn't sufficient for a functional test to determine whether the intended function has been performed correctly; a functional test should also verify that no unintended function was additionally performed.

3.7.2 Built–in test response analysis

Built–in test requires a method of checking the output response of the circuit under

test that is simpler and is less storage–intensive than the conventional bit–by–bit comparison of the actual test output with the expected correct values. The usual method is to perform some form of data compression on the test responses before making the reference comparison. The compressed response is referred to as the "signature" of the circuit under test, and comparison is made to the precomputed signature of a fault–free version of the circuit.

The signature and its collection algorithm should meet the following qualitative guidelines:

1). The algorithm must be simple enough to be implemented as part of the built–in test circuitry;

2). The implementation must be fast enough to remove it as a limiting factor in test time;

3). The algorithm must provide approximately logarithmic compression of the test response data to minimize the reference–signature storage volume.

4). The compression method must not lose information. Specifically, it must not lose any evidence of a fault indicated by a wrong response from the circuit under test.

There is no algorithm that unambiguously meets all of these criteria. The greatest problem is the possibility that the error pattern from a faulty circuit may be compressed to the same signature code as the fault–free circuit. Since only a function of the test response sequence is verified rather than the sequence itself, there is a loss of information

that can "mask" errors in the sequence. Fault masking in built–in test is measured by the probability that a compression of a possible error sequence of test response produces the same signature as the fault–free circuit.

Different compression techniques and methods have been suggested and some are in actual use. One can refer to [BARD87] for an introduction of the following methods of test analysis: 1) parity checking; 2) transition counting; 3) syndrome generation; 4) signature analysis; 5) Walsh spectra; and 6) cyclic codes.

Among these methods, signature analysis has been the most widely used data compression method for built–in test designs. It uses a shift register with various stages tapped and fed back to an exclusive OR (XOR) gate that in turn feeds the register input. The feedback network configuration of the shift register is determined by a primitive polynomial which has a small number of terms. This implies a simple and effective hardware implementation of the signature generation circuit. A modified multiple (parallel) input shift register (MISR) structure is studied and applied in our project.

3.7.3 Built–in test structures

Built–in test is a collection of possibilities, the choice of which depends upon the application. There is not one best built–in test structure, however, there are factors to consider in relation to evaluation of built–in test structures. These factors are: i) fault coverage required; ii) system overhead which is tolerable; iii) the impact of the built–in test on the system performance; and vi) the test time that is allowable.

Implementation approaches are different and versatile in a variety of built–in test architectures. In this section we can not mention all of them with the rewards and hazards

of each one. Some of those that have been suggested or used in the past are: LFSR or CA structures for probability–based (pseudorandom) test generation [HoMc90], and MISRs for parallel test response compression (as shown in Figure 3–2); Scan–based built–in test structures (to be briefly described in the next subsections), may use LSSD or BILBO based structures; Others are for simultaneous self–test (SST) and Fast–forward test. Refer to [MCCL86] and [BARD87] for detailed circuit structures for built–in test.

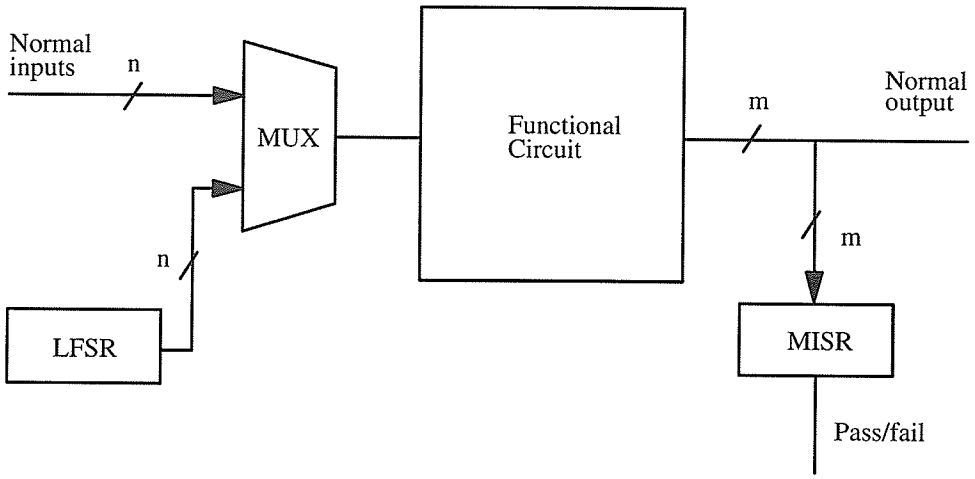


Fig. 3–2 BIST circuit with LFSR and MISR

3.7.3.1 Scan–path structures

Scan path refers to a disciplined design standard for all storage elements (other than memory arrays) which has the express purpose of making the stored values easy to control and easy to observe. With this facility the storage element becomes in effect both a primary input and primary output. Test input signals can be clocked in through one or multiple scan paths (scan chains) internal to the circuit under test, and test results can be observed wher-

ever one of the storage elements occurs in the logic circuit. Thus the test problem reduces to one for the combinational logic between the storage elements. The test results can be scanned (clocked) out along the internal scan path for verification. It is worth noting that advanced EDA tools today offer scan insertions for gate level VLSI/ASIC design implementations. Thus, the configuration needed to convert the storage elements into a chain of scan registers when in test mode is automated using scan insertion tools.

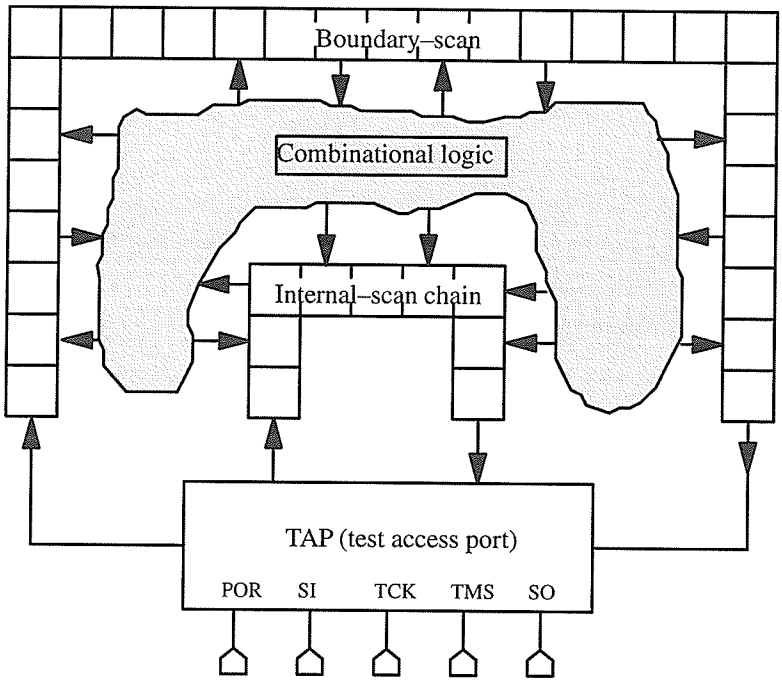


Fig. 3–3 A general scan–based BIST design model

3.7.3.2 Boundary scan test

Boundary scan test, the inclusion of shift–registers latches around the periphery of a VLSI/ASIC chip (or a multiple–chip package), originated from a useful concept for diag-

nosis at the system or subsystem level. In the most general sense of hardware or system diagnosis, different diagnostic demands are placed on the testing facility at different packaging levels [GlBr89]. At the chip level, diagnosis of the failing net (or the circuit) is required for resolution of zero yield situations at first silicon and later for yield improvement. At the subsystem level the diagnostics must identify a repair action since it is unlikely that the package is a throwaway item. At the system level, diagnosis must be able to identify a field replaceable unit. Because a VLSI/ASIC chip is likely to be at any of the levels concerned, boundary scan support has become a standard feature in almost every VLSI technology devices. Figure 3–3 shows a general scan design including boundary scan and internal scan chains.

IEEE–1149 boundary scan standard is the documentation of references for all boundary scan design and implementations. It is worth noting that advanced EDA tools and many VLSI/ASIC vendors provide ready–to–use boundary macrocells or boundary scan insertion tools [DONN91]. Unless optional boundary scan features are of concern the designer would not be involved in detailed implementations of the boundary scan circuitry for the VLSI/ASIC design. A proprietary scan design methodology with ATPG (automatic test pattern generation) tools at Bell–Northern Research is given in [DOSI 92]. Refer to [MaTu90] for the boundary scan architecture and information on the test access port.

CHAPTER 4

A BUILT–IN TESTABLE ON–CHIP ECC

4.1 Introduction

In a system VLSI/ASIC chip, embedded memories with on–chip ECC presents a solution for wide memory bandwidth and reliable data transfers. It also presents a challenge to testing these functional modules internal to the system chip. In the previous chapters, we have studied error control coding for memory systems, the design for testability (DFT) philosophy, and the built–in test (BIT) methodology. In this chapter, we shall present a design–for–testability method for on–chip built–in testable ECCs. First, we shall start with testability analysis of the on–chip ECC implementation presented in Chapter 3 and propose a natural partition for testability. Second, we shall demonstrate a procedure of test development, mapping, and compaction, resulting in reduced hardware overhead and test time associated with built–in test. Then, we shall describe a built–in testable architecture that integrates the advantages of multiple self test methodologies, and supports up to four BIST environments. Finally, a viable BIST structure, intraconnections, and control signals are explicitly defined for modular design implementations.

4.2 Circuit partitioning for testability

The on–chip ECC can be partitioned as in Fig. 4–1 into two parts: PG/SG and SD/DC. With reference to logic diagram of the on–chip ECC circuit in Fig. 2–5, the following analysis can be made with regard to the testability of the ECC circuit.

4.2.1 The PG/SG Circuit

The PG/SG blocks generate parity and syndromes from the coded incoming data. The testability features of this portion of the ECC circuit are as follows:

1) It consists of only linear circuity (XOR logic) with high controllability and observability;
2) It is a fully bit–sliced implementation, such that every syndrome bit is generated by its own circuitry;
3) Each information data bit has a fan–out of 3, which may cause some random pattern resistance that counteracts the high controllability of the XOR tree.

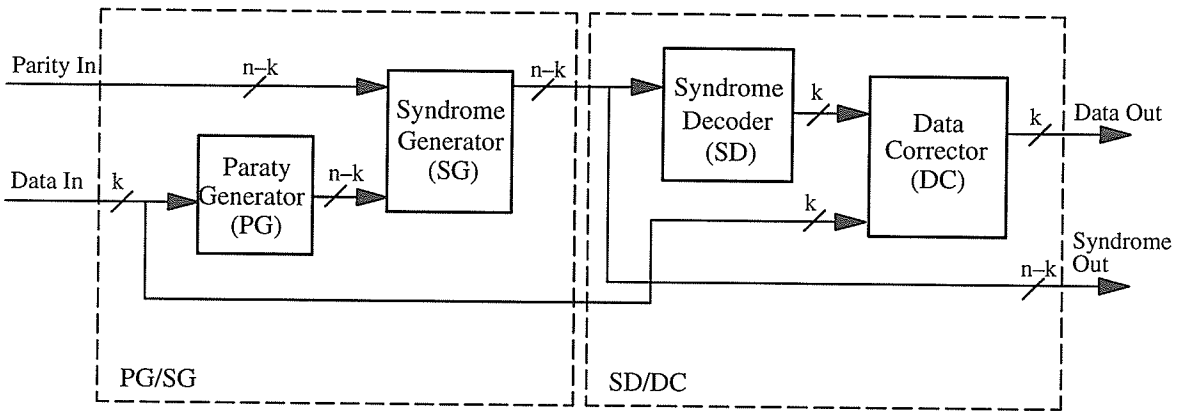


Fig. 4–1 A natual partition of the error correction circuit

4.2.2 The SD/DC Circuit

The SD/DC blocks decode the syndrome and corrects the incoming data when applicable. The testability features of this portion of the ECC circuit are as follows:

1) It consists of non–linear circuits;

2) It has high fan–in;

3) These two properties make for low controllability.

In our BIST scheme we first develop a test generator that provides the needed but perhaps a minimal set of test vectors for testing the PG/SG. The problem on how well this set of test vectors covers the faults in SD/DC is addressed later in this Chapter.

4.3 Testing PG/SG

The modified Hamming (22,16) SEC–DED code is embodied in the syndromes equations (S_i), as functions of the data bits (D_i) and the check bits (C_i):

$$
\begin{aligned}
S_0 &= C_0 + D_0 + D_1 + D_3 + D_4 + D_8 + D_9 + D_{10} + D_{13} \\
S_1 &= C_1 + D_0 + D_2 + D_3 + D_5 + D_6 + D_8 + D_{11} + D_{14} \\
S_2 &= C_2 + D_1 + D_2 + D_4 + D_5 + D_7 + D_9 + D_{11} + D_{15} \\
S_3 &= C_3 + D_0 + D_1 + D_2 + D_6 + D_7 + D_{10} + D_{11} + D_{12} \\
S_4 &= C_4 + D_3 + D_4 + D_5 + D_6 + D_7 + D_{13} + D_{14} + D_{15} \\
S_5 &= C_5 + D_8 + D_9 + D_{10} + D_{11} + D_{12} + D_{13} + D_{14} + D_{15}
\end{aligned}
$$

Primary input to the on–chip ECC consists of 16 data bits and 6 check bits that fan out to the six PG/SG bit slices according to the above code. Fig 4.2 shows one of the six syndrome bit slices and the structural test vector set $\mathbf{T_o}$ for testing the bit slice.

The test vectors in test matrix $\mathbf{T_o}$ cover all single stuck–at faults [BOSS70] in the bit slice. For every test in $\mathbf{T_o}$ to reach every one of the six bit slices in PG/SG, we need to

expand each test in $\mathbf{T_o}$ to six different tests according to the code. This would establish a set of test vectors in test matrix **T** for testing PG/SG as shown in Fig. 4.3.

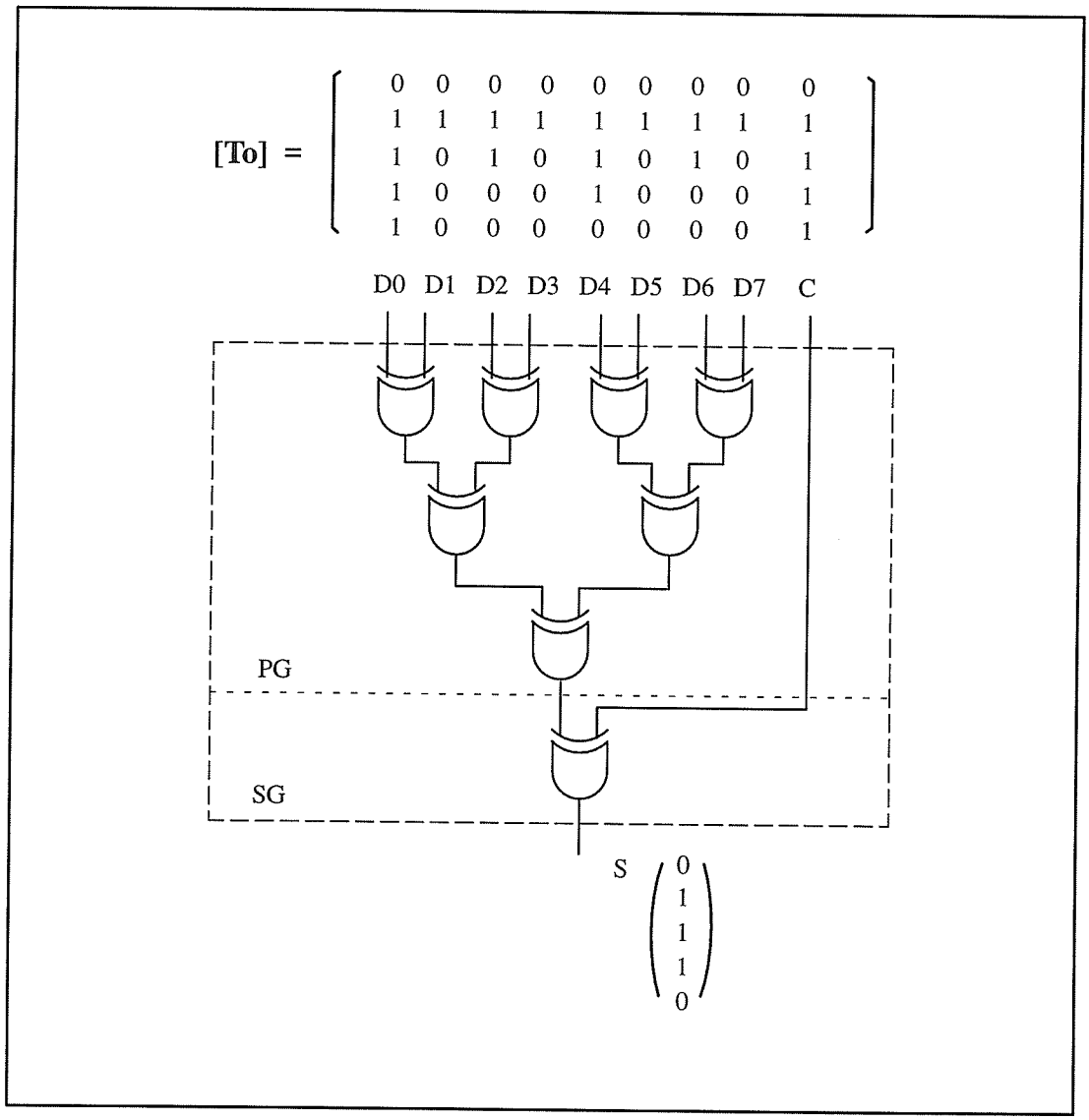


Fig. 4–2 A PG/SG bit slice and bit slice tests

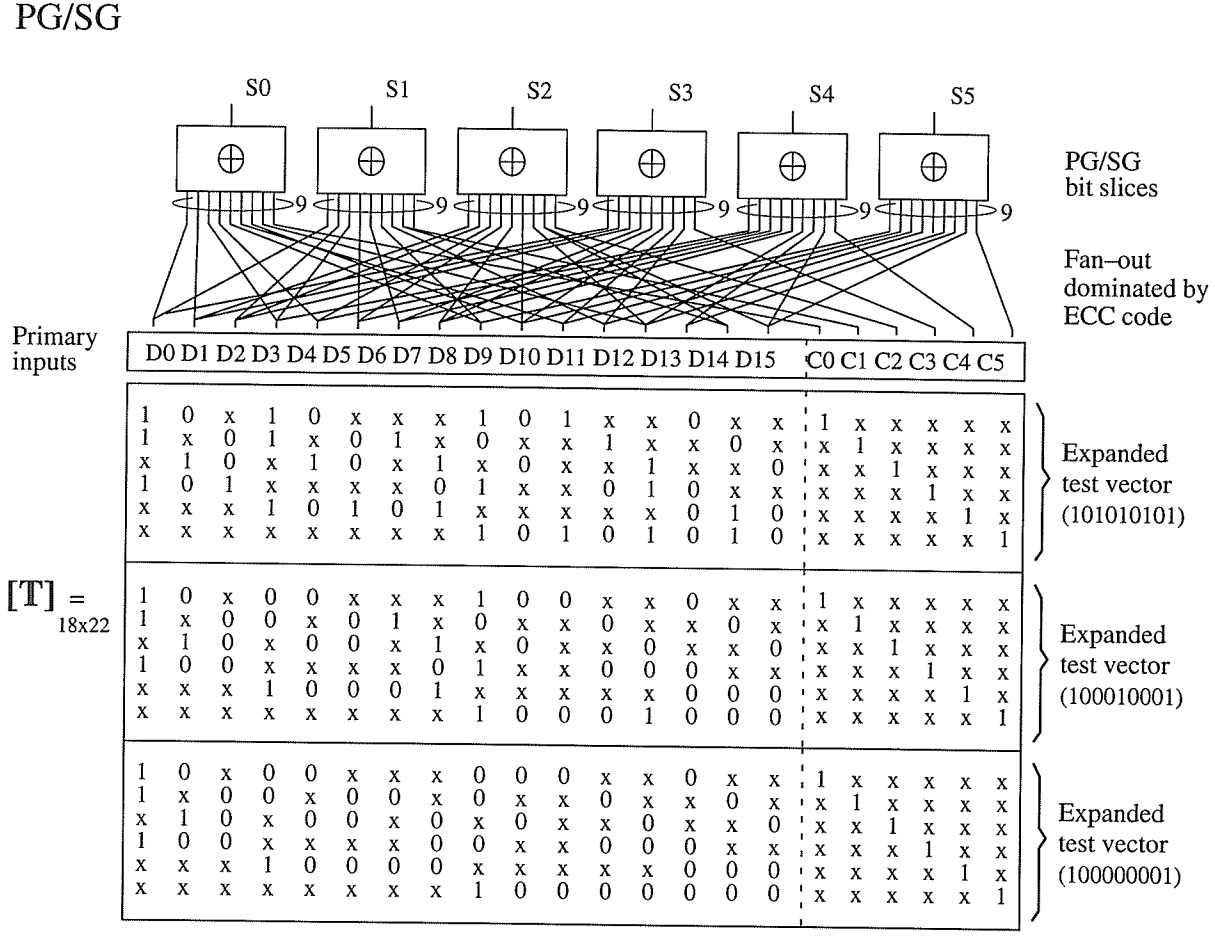


Fig. 4–3 Mapping (expanding) $\mathbb{T}_o$ to six bit slice tests

4.3.1 Test matrix reduction

As shown in Fig. 4–3, the mapping from test matrix $\mathbf{T_o}$ to test matrix **T** is according to the fan–out of the primary inputs dominated by the ECC code. The resultant test matrix **T** has a dimension of 18 X 22. This is because the set, reset vectors, i.e., (11111111), (00000000) are not included. There is no code dependence for these two global operations, and set or reset tests can be done to all six slices simultaneously.

We shall compact the rows and columns of matrix **T**, in section 4.3.1.1 and section 4.3.1.2, respectively. The resulting matrix will be reduced to only eight 11–bit test vectors.

4.3.1.1 Row compaction

The rows of **T** are the test vectors needed for all stuck–at faults in PG/SG. Due to the high sparsity of **T** with respect to don't cares, the rows of T can be compacted. The result of row compact of **T** is test matrix $\mathbf{T_c}$ as in Fig. 4–4.

$[\ T_c\ ]_{8x22} =$

D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	C0	C1	C2	C3	C4	C5
1	0	x	1	0	1	0	1	1	0	1	0	1	0	1	0	1	x	x	x	1	1
1	1	0	1	1	0	1	1	0	0	x	1	1	x	0	0	x	1	1	x	x	x
1	0	1	x	x	x	0	1	x	x	0	1	0	x	x	x	x	x	x	x	1	x
1	0	0	0	0	x	0	1	1	0	0	0	0	0	x	x	1	x	x	1	x	x
1	1	0	0	0	0	1	1	0	0	x	0	0	x	0	0	x	1	1	x	x	x
x	x	x	1	0	0	0	1	1	0	0	0	1	0	0	0	x	x	x	x	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	1	1	x	1	x	x
x	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	x	x	1	x	1	1

Fig. 4–4 Test matrix resulting from row compaction

In comparison with **T** that has 18 rows, the test matrix $\mathbf{T_c}$ has only 8 rows. This represents a > 50% reduction in the needed test inputs. Still, test vectors in $\mathbf{T_c}$ are 22 bits long, the same as the test vectors in test matrix **T**. Each bit corresponds to a separate bit of test inputs or the outputs of a on–chip test generator. This suggests that a 22 bit built–in test generator would still be required should an on–chip test generation be implemented.

4.3.1.2 Column compaction

To minimize the size of the on–chip test generator, we noted that two columns of the matrix can be supplied from the same bit of the test generator if they represent exactly

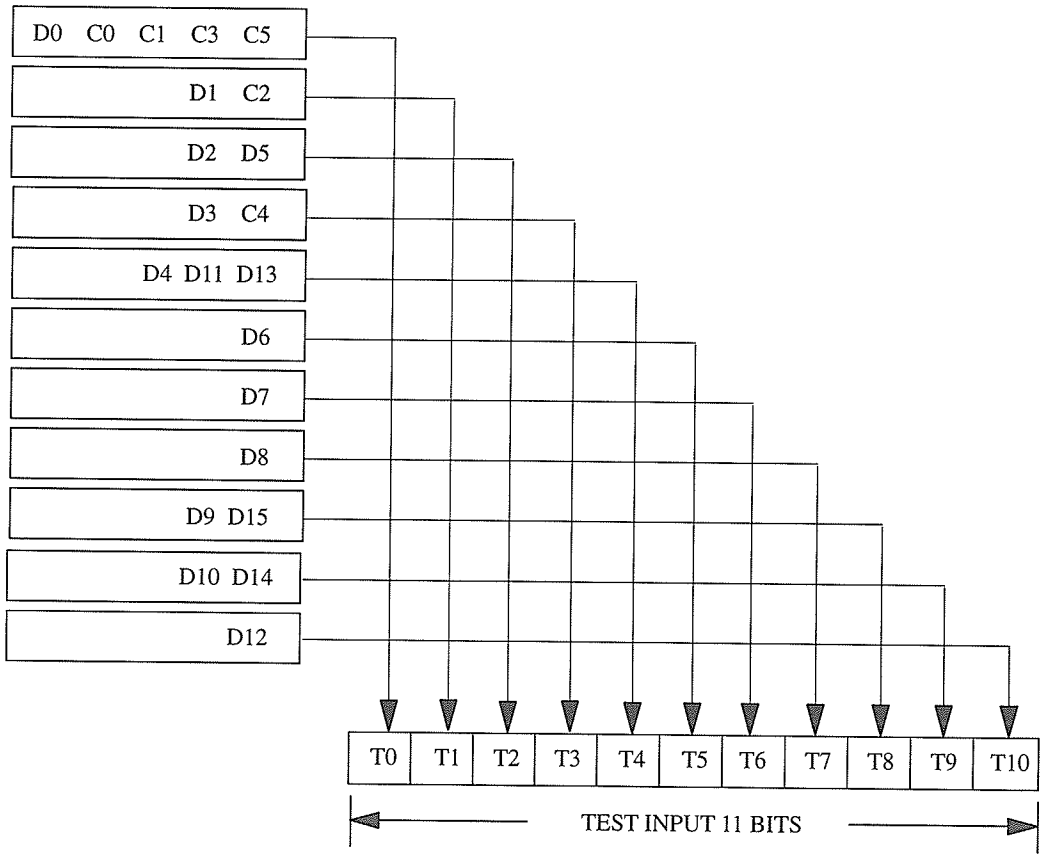


Fig. 4–5 Eleven (11) groups merged

the same binary (column) vector. Therefore, all we need to do to compact the columns of **Tc** is to identify the the columns that can be merged by this rule. For instance, it is easy to see that the columns D_0 and C_0 can be merged into one column. We ended with 11 groups of columns in **Tc** merged as shown in Fig. 4–5, and finally the test matrix $\mathbf{T_t}$ for testing PG/SG in Fig. 4–6.

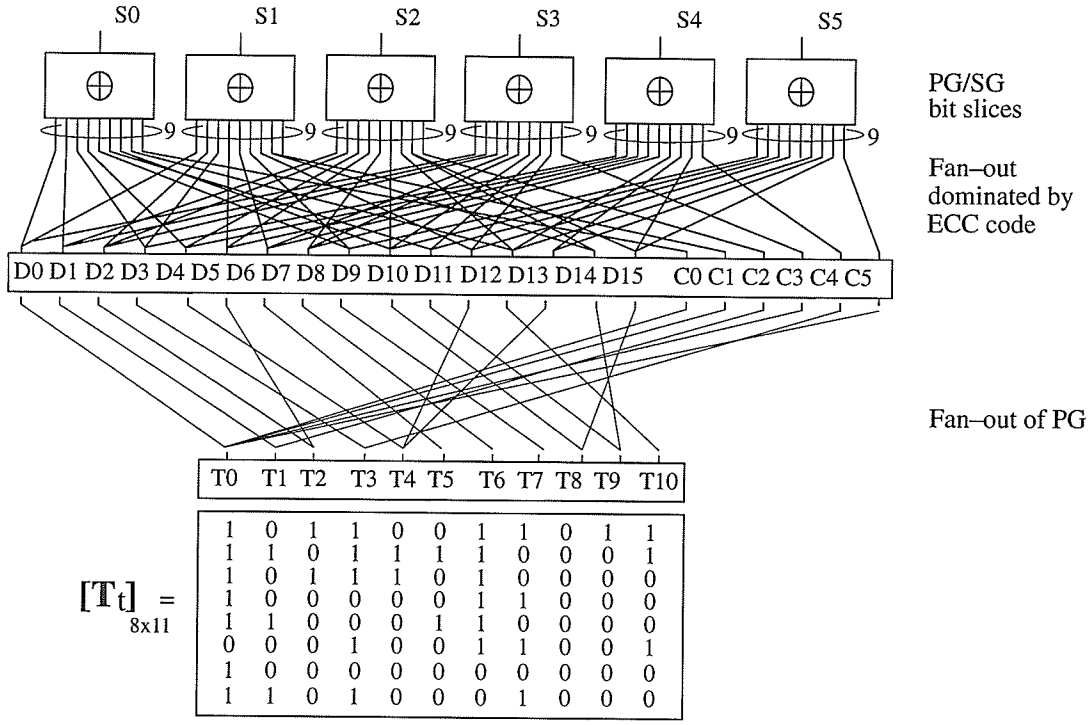


Fig. 4–6 A compact set of test vectors for testing PG/SG

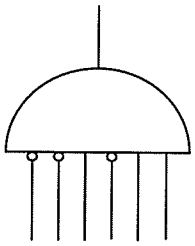
4.3.2 Discussion

For testing PG/SG, the original test matrix **T** of 18 22–bit test vectors was obtained by expanding the test vectors for one bit–slice; **T** can then be downsized, by row compaction, to a 8 22–bit test vector matrix $\mathbf{T_c}$. These 8 test vectors cover all stuck–at faults in

the PG/SG. They can either be scanned in as test inputs to the circuits in scan test or stored in a on–chip ROM and applied vector–by–vector by reading from the ROM. The column compaction concluded with the 11 groups of columns merged. This implies that an 11–bit (instead of 22–bit) on–chip test generator is sufficient for testing the PG/SG, given the fan–out of each bit of the test generator determined by the mapping rules. Practically, this means 50% less built–in test generation hardware overhead can be achieved. The uneven fanout (maximum 5) should not cause any problems with respect to circuit operation.

4.4 Testing SD/COR

We now derive all the test inputs needed to cover the stuck–at faults in SD/DC from the test vectors needed for testing a bit slice of the syndrome decoder. Figure. 4–7 shows the one main test and six auxiliary tests which are all needed for testing one bit slice of the



Main Test	0 0 1 0 1 1
Auxillary Test	1 0 1 0 1 1
	0 1 1 0 1 1
	0 0 0 0 1 1
	0 0 1 1 1 1
	0 0 1 0 0 1
	0 0 1 0 1 0

Fig. 4–7 Test vectors for one syndrome decoder bit slice

syndrome decoder circuit (For each bit slice the expected output of the main test is 1, whereas the expected test output is 0 for all auxiliary tests). This leads to a set of 112 (7 X 16) test vectors in total for testing the 16 bit slices. Figure 4–8 explicitly shows the 112 test vectors with each of 16 main tests referred to as Di (i = 0, 1, .. , 15). It is noted that

$T =$

D0	D1	D2	D3	D4
1 0 1 1 1 1 1	1 0 1 1 1 1 1	0 1 0 0 0 0 0	1 0 1 1 1 1 1	1 0 1 1 1 1 1
1 1 0 1 1 1 1	0 0 1 0 0 0 0	1 1 0 1 1 1 1	1 1 0 1 1 1 1	0 0 1 0 0 0 0
0 0 0 1 0 0 0	1 0 1 0 1 1 1	1 1 1 0 1 1 1	0 0 0 1 0 0 0	1 1 1 0 1 1 1
1 1 1 1 0 1 1	1 1 1 1 0 1 1	1 1 1 1 0 1 1	0 0 0 0 1 0 0	0 0 0 0 1 0 0
0 0 0 0 0 1 0	0 0 0 0 0 1 0	0 0 0 0 0 1 0	1 1 1 1 1 0 1	1 1 1 1 1 0 1
0 0 0 0 0 0 1	0 0 0 0 0 0 1	0 0 0 0 0 0 1	0 0 0 0 0 0 1	0 0 0 0 0 0 1

D5	D6	D7	D8	D9
0 1 0 0 0 0 0	0 1 0 0 0 0 0	0 1 0 0 0 0 0	1 0 1 1 1 1 1	1 0 1 1 1 1 1
1 1 0 1 1 1 1	1 1 0 1 1 1 1	0 0 1 0 0 0 0	1 1 0 1 1 1 1	0 0 1 0 0 0 0
1 1 1 0 1 1 1	1 1 1 0 1 1 1	1 1 1 0 1 1 1	0 0 0 1 0 0 0	1 1 1 0 1 1 1
0 0 0 0 1 0 0	1 1 1 1 0 1 1	1 1 1 1 0 1 1	0 0 0 0 1 0 0	0 0 0 0 1 0 0
1 1 1 1 1 0 1	1 1 1 1 1 0 1	1 1 1 1 1 0 1	0 0 0 0 0 1 0	0 0 0 0 0 1 0
0 0 0 0 0 0 1	0 0 0 0 0 0 1	0 0 0 0 0 0 1	1 1 1 1 1 1 0	1 1 1 1 1 1 0

D10	D11	D12	D13	D14
1 0 1 1 1 1 1	0 1 0 0 0 0 0	0 1 0 0 0 0 0	1 0 1 1 1 1 1	0 1 0 0 0 0 0
0 0 1 0 0 0 0	1 1 0 1 1 1 1	0 0 1 0 0 0 0	0 0 1 0 0 0 0	1 1 0 1 1 1 1
0 0 0 1 0 0 0	0 0 0 1 0 0 0	1 1 1 0 1 1 1	0 0 0 1 0 0 0	0 0 0 1 0 0 0
1 1 1 1 0 1 1	1 1 1 1 0 1 1	1 1 1 1 0 1 1	0 0 0 0 1 0 0	0 0 0 0 1 0 0
0 0 0 0 0 1 0	0 0 0 0 0 1 0	0 0 0 0 0 1 0	1 1 1 1 1 0 1	1 1 1 1 1 0 1
1 1 1 1 1 1 0	1 1 1 1 1 1 0	1 1 1 1 1 1 0	1 1 1 1 1 1 0	1 1 1 1 1 1 0

D15
0 1 0 0 0 0 0
0 0 1 0 0 0 0
1 1 1 0 1 1 1
0 0 0 0 1 0 0
1 1 1 1 1 0 1
1 1 1 1 1 1 0

Fig. 4–8 Original test vectors for testing SD/COR

each auxiliary test vector is one bit different from the main test for the bit slice and, auxiliary test vectors may overlap in the test vector set **T** although the 16 main tests are distinct.

By combining these 'overlaps' in **T**, a set of 46 test vectors result. Figure 4–9 explicitly shows the non–overlapped 46 test vectors needed for covering the stuck–at faults in SD/COR circuit.

$$\mathbf{T'} = \left[\begin{array}{cccccccccccccccccccccccccccccccccccccccccccc}
1&1&0&1&1&0&0&0&1&1&1&0&0&1&0&0&0&1&1&1&1&0&1&1&1&0&0&0&0&1&1&1&0&1&0&0&0\\
1&0&1&1&0&1&1&0&1&0&0&1&0&0&1&0&1&0&1&1&1&0&0&0&0&1&1&1&1&0&1&1&0&0&1&0&1\\
0&1&1&0&1&1&0&1&0&1&0&0&1&0&0&1&0&0&1&0&0&1&1&1&1&1&1&1&1&0&0&1&0&1&1&1&0&0\\
1&1&1&0&0&0&1&1&0&0&1&1&1&0&0&0&1&1&1&1&1&1&1&0&1&1&0&1&1&0&0&0&0&0&0&0&1&1\\
0&0&0&1&1&1&1&1&0&0&0&0&0&1&1&1&0&0&0&1&0&0&0&1&0&0&1&0&1&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&1&1&0&1
\end{array}\right]$$

$$\left[\begin{array}{ccccccccc}
0&0&1&1&1&0&0&0&1\\
0&1&0&1&1&0&0&0&0\\
1&0&0&1&0&1&0&0&0\\
1&0&0&0&0&0&1&0&1\\
1&0&0&0&0&0&0&1&1\\
1&1&1&1&0&1&1&1&1
\end{array}\right]$$

Fig. 4–9 Non–overlapped test vectors for testing SD/COR

4.4.1 Pipelining the tests for SD/COR and PG/SG

The prime question here is whether the test output from PG/SG form a complete test set for SD/DC when tests are generated using an on–chip 11 bit PRTG. It is noted that the fanout derived in the previous section maps the test generator outputs of an n/2 bit wide test vectors to a much larger space of n–bit vectors; and the ECC code implemented by PG/SG in turn maps this space into the much smaller space of n–k bit syndrome vectors. The product of these maps acting on the initial test set does not always generate all of the

possible 2^{n-k} syndromes vectors, even if the initial test set is exhaustive (i.e.,$2^{n/2}-1$ vectors in size). When the vectors in the syndrome space are left out by the above mappings and happen to coincide with the ones needed by SD/DC, a reduced fault coverage will result. We simulated the syndrome vectors generated by the PS/SG when fed by $2^{11}-1$ pseudorandom test vectors to analyze the fault coverage.

It is noted that the 46 test vectors needed occupies 72% of the compete syndrome space(i.e., $2^6 = 64$). We observed that the 46 syndrome vectors needed for testing the SD/DC circuit were included in the syndrome vector set of the test responses from the PG/SG circuit when initial test vectors are applied from an on–chip 11 bit PRTG unit with random seeds. As such, we anticipate a >99% fault coverage for all stuck–at faults in SD/DC as well as in PG/SG when pipelining the tests for PG/SG and SD/DC on on–chip 11 bit test generator PRTG circuit.

4.4.2 Linear independency of syndrome equations

Refer to figure 4–5, the syndrome equations in section 4.3 now can be expressed as functions of the original 11 bit test inputs T[0:10] :

$$S_0 = T_0 + T_1 + T_3 + T_4 + T_7 + T_8 + T_9$$
$$S_1 = T_0 + T_2 + T_3 + T_4 + T_5 + T_7 + T_9$$
$$S_2 = T_1 + T_2 + T_4 + T_6 + T_8$$
$$S_3 = T_0 + T_1 + T_2 + T_4 + T_5 + T_6 + T_9 + T_{10}$$
$$S_4 = T_2 + T_3 + T_4 + T_5 + T_6 + T_8 + T_9$$
$$S_5 = T_0 + T_4 + T_7 + T_8 + T_9 + T_{10}$$

It is noted that these syndrome equations are lineally independent. Therefore, all syn-

dromes generated corresponding to a given set of $2^{11}-1$ test inputs to PG/SG will be applied to testing SD/COR. This linear independence implies the possibility of obtaining 100% fault coverage for SD/DC.

The linear independence of the syndrome equations in our case does not imply the linear independence of the syndrome equation holds true for all ECC codes. A different ECC code presents a different map and changes the syndrome equations. However, since every syndrome equation includes exactly one check bit input, this may be a good tool for eliminating linear dependencies and increase the fault coverage at no extra cost. One can redo the second map, namely the fan–out of the test generator, and regroup and merge the check bit of Figure 4–4 into different groups as in Figure 4–5. In general, given any linear dependence $\sum S_i = 0$, one needs to pick any one of the syndromes in the sum (e.g. S_j) and provide its check bit from another possible test generator bit, say the bit T_m instead of T_n, by taking advantages of the don't–cares. This is equivalent to replacing S_j with $S_j' = S_j + T_m + T_n$, which will change the linear sum of syndromes to $\sum S_i + T_m + T_n \neq 0$.

4.5 Best testable codes

The syndrome decoder has the lowest controllability of all elements in the ECC. Hence it sets the cap on test time. Even if the syndrome equations is linearly independent, the 100% fault coverage is only guaranteed if the test generator runs exhaustively ($2^{11}-1$ test vectors in our context). During one test cycle the test generator may not go through all combinations of inputs exhaustively. The best testable codes are, therefore, defined as one that requires the least time to test its hard–to–test section, i.e. the syndrome decoder. From the analyses in section 4.3. it is noted that the more auxiliary vectors overlap, the more effective the testing the syndrome decoder becomes. An auxiliary test is the main test with one of its bits flipped. Therefore, a short test time can be achieved by using a code

whose syndrome space consists of vectors with maximum of Hamming distance of 2 between them.

4.6 The BIST architecture

The block diagram of the built–in testable ECC module developed in this thesis is shown in Fig. 4.10. The on–chip ECC core, the PRTG unit, the MISR unit, and the test controller unit constitute a built–in test structure, which seems to be a conventional BIST structure that implements built–in pseudorandom test on the on–chip ECC. In this BIST architecture, however, the capacity of each functional block has been defined not only to implement the pseudorandom test, but also the pseudoconcurrent test, and the scan test of the on–chip ECC and other system modules located in a same scan chain when in test mode.

This proposed BIST architecture is designed to support self testing of the on–chip ECC under a multiple BIST environment. So it is imperative that a minimal hardware overhead should be achieved by the BIST architecture. It is noted that internal control signals and status, combined with the intraconnections of the BIST system structure, inherently demands advanced hardware design methods such that the same built–in test hardware can be dynamically configured under software control for each of the four self test modes: 1) on–demand test mode, 2) pseudoconcurrent (interleaved) test mode, 3) scan test mode, and 4) scan–path mode.

The BIST architecture also specifies that test stimulus are sourced from either the on–chip PRTG–based test generator or from the test vectors from a scan test input port.

Before we get to define the independent self–test modes and evaluate the performance aspect of the BIST architecture in detail, we need to present some general descriptions

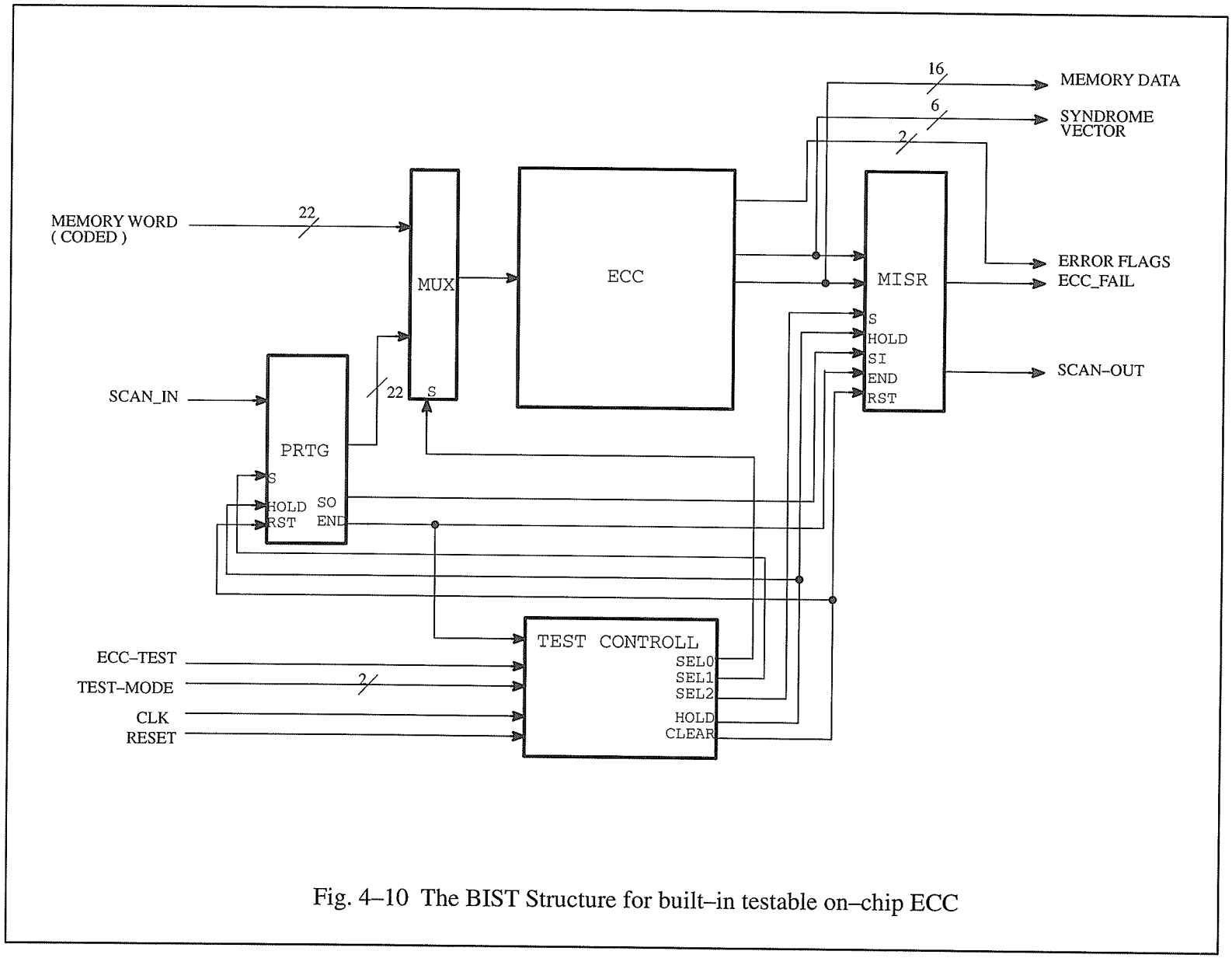


Fig. 4–10 The BIST Structure for built–in testable on–chip ECC

about the operational and control aspects of the built–in test structure as a system.

4.6.1 System constructs and operations

The built–in testable ECC is constructed by five (5) functional blocks: 1) a PRTG–based test generation or scan test input unit; 2) a multiplexer unit for selecting from normal memory operation input or self test input to the ECC circuitry; 3) the on–chip ECC core; 4) a MISR–based parallel signature analyzer or scan output unit; and 5) a high performance self test controller.

The only external controls to the BIST on–chip ECC system are: a dedicated CLOCK signal to run the test function only; a two–bit signal TEST_MODE[0:1] to chose one of four self test modes; an ECC_TEST signal to signal the system into and out of self–test operation; and a global RESET signal for presetting the system to a defined initial state.

During normal memory operation of the system, the coded memory words (22 bits) are selected by the multiplexer (MUX) as the inputs to the ECC circuits for error checking and correction. The corrected memory words (16 bits) are send out directly via the memory output buffer (not shown in Fig. 4–10). The syndrome output bits are optional. They could be useful in tracking the errors on a production line for fault location purposes.

When the system is called up by the processor for a self test in a given test mode, such as the on–demand test mode, the MUX would select the pseudorandom test stimulus generated by the on–chip PRTG to the ECC circuitry. During the test, the outputs from the ECC would in turn be routed to the on–chip parallel signature analyzer (MISR). The MISR has the circuitry to generate and match a (final) signature with the known correct signature in each test cycle. It sets a one–bit ECC_Failure flag if a different signature pattern results

on the completion of a built–in test cycle.

The SCAN_IN and SCAN_OUT terminals on the PRTG and the MISR units are dedicated to the two scan–based test modes. In scan–based tests, the test stimulus is serially shifted from outside of the system and test results are shifted out for analysis. This is basically accommodated by configuring the flip flops in the PRTG and the MISR into shift registers. There are differences between these two scan–based test modes. In scan test mode, the ECC circuit is the circuit under test (CUT); while in the scan path mode, the bit streams (either as test input or test response) will simply by–pass the ECC model to support scan test operations on other system modules located in the same scan chain as the ECC model. Internally in our on–chip ECC, these two scan–based test operations are treated as two individual self test modes, interpreted as two distinct processor commands.

One major difference in circuit configurations is that in scan path mode the SCAN–OUT from PRTG is connected to the SCAN_IN on MISR, enabling a chain of by–pass shift registers. In scan test mode, this net is disconnected. As such, the scan test input will never be shifted beyond the test input shift register in PRTG, and the test output bit stream will have a constant extension of its last significant bit. In PRTG, the bit stream of test stimulus is aligned and converted into a 11 bit test vector, which is selected by the MUX to feed into the ECC. In the MISR, the test response is captured in parallel and in turn converted back into a serial bit steam and shift out via the SCAN_OUT port.

As the BIST architecture integrates the advantages of multiple BIST technologies, it demands an integrated test control unit. Again, minimal and programmable features of the test controller is desired. The test controller receives from the processor a command in a 3 bit format, which specifies a test mode in 2 bits and orders assertion or termination of the self test in 1 bit. These three control bits may be latched into a control register such

that the host processor can still hold the control bus for other tasks while the self testing of the on–chip ECC is in progress. The test controller will put the built–in test circuitry as a whole into a configuration dedicated to the given test mode and operation. For instance, it will set up the proper test data flow path, manage the source of test input, and so forth. During a self test, the test controller will monitor the internal status or states, generate the appropriate internal control signals to coordinate or synchronize the operations of each of the functional units in the BIST structure. In addition, it will manage the timing of a proprietary system initiation to set the built–in test circuitry to a predefined state before a new self test cycle begins.

During a self test, the test controller will also be alert to control command changes (or new commands) to actively respond to the host processor. Take an interleaved test as an example. The test controller will generate a HOLD signal when the system is called back to the normal operation from an on–going pseudoconcurrent test. The HOLD signal will freeze all the built–in test circuitry so that the system can resume the interleave test from where it left off.

The on–line error/fault reporting system is composed of three flag signals: DOUBLE_ERROR, COR_ERROR and ECC_FAILURE. During normal operation the first two error flags can be monitored concurrently for error detection and correction. The ECC_FAILURE signal is latched into an one bit flag register on completion of each self test cycle. The information stored in this register is the conclusion of the signature comparison, and updated in the end of one test cycle. The information stored by this flag can be used by the system processor to decide whether or not a repair action needs to be taken, for instance, to replace a faulty on–chip ECC with a stand–by.

In the next sections, we shall define the alternative self test modes and evaluate the

performance of the built–in testable architecture from a system view. The implementation of built–in test circuitry and control will be detailed in Chapter 5 of this document.

4.6.2 Alternative self test modes

The definition given below will explicitly specify each individual self test mode and serve as the guideline for the circuit design of each internal functional unit of the on–chip BIT ECC.

4.6.2.1 On–demand mode

The on–demand mode is invoked by asserting the ECC_TEST signal with the two–bit signal T_MODE [1:0] set to $(11)_2$. That is, T_MODE0 = T_MODE1 = 1. The test circuitry is then automatically cleared and a test cycle begins. The built–in PRTG circuit with a hardcoded seed pattern will serve as the source of test inputs. Upon the completion of one test cycle with $2^{11}-1$ test vectors, the final signature is generated by the MISR and compared with a hardcoded correct signature, and a single bit pass/fail result is logged in to the flag register (0 = pass, 1 = fail). Test circuitry is then cleared and the test cycle repeats until the system is signaled out of test mode by flipping over the ECC_TEST, in which case the last completed test result is retained.

4.6.2.2 Pseudoconcurrent mode

Self–test in pseudoconcurrent mode is asserted with the control signals ECC_TEST=1, T_MODE0 = 1, T_MODE1 = 0. Since the ECC core circuitry does not have a stored state, it can be summoned to serve the system while the information in its test circuitry is held unchanged. The ECC may enter into self test whenever the system is idle,

and is allowed to complete as much of the current test cycle as it can before being asked to serve the system again. The system can leave an interleaved self test cycle by flipping over the control bit ECC_TEST, and resume the interleaved test by setting up the ECC_TEST bit again. These test subcycles continue to run, interleaved with the process, until one cycle is completed. The result is then latched and the test circuitry is reset for a new cycle to begin. The last completed test result is always available to the processor.

The only major difference between on–demand and pseudoconcurrent test is that in on–demand mode the test circuitry is reset every time the test mode is entered, but in pseudoconcurrent test, reset only occurs when one test cycle is completed. The test controller must have a means to detect whether or not a complete test cycle is done when the system quits the interleaved test mode. The control will branch into different procedures based on this detection or the state of a interleave test cycle. The intermittent information of an interleaved test should be held when the system exits from the pseudoconcurrent test mode and a test cycle is not yet completed. Hence, when the system reenters the self test in pseudoconcurrent test mode, the test can resume from where it left off. In the case that another test mode is entered, the test circuitry would be reset, with the last test result still stored in the flag bit.

4.6.2.3 Scan test mode

Scan test on the ECC circuit is asserted with the control signals ECC_TEST=1, T_MODE0 = 0, T_MODE1 = 1. The bit stream of a test vector will be scanned in serially from the SCAN_IN input port of the PRTG unit. Upon alignment of a test vector's bits with the PRTG register bits, the test vector will be purged into and through the ECC circuit. The outputs from the ECC will then be latched in parallel onto the MISR registers from where they will be scanned out through SCAN_OUT port on the MISR for verification. The test

using next test vector can then begin. In scan test mode both PRTG and MISR are configured into shift registers, no test inputs are generated internally and no test output analyzed internally. The termination of a scan test is effected when the system resets the control bit ECC_TEST.

4.6.2.4 Scan path mode

A scan path may be formed to feed the test input serially into "deep" circuitry. With the control signals ECC_TEST=1, T_MODE0 = 0, T_MODE1 = 0, the system will enter self test in the scan path mode, and configure the registers in PRTG and MISR into one shift register as a portion of a scan chain. Scan test vectors will be clocked in as in scan test mode, but bypass the ECC circuitry, for scan test of other system modules. This can be used to alleviate the complexity in system debugging tasks and enhance the diagnosability of the overall system.

4.6.3. Performance view of the BIST architecture

We shall now conclude the discussion of the BIST architecture with a brief system view of its performance. It is noted that one of the features of the proposed architecture is that, due to the need of selecting the system input or test input to the ECC core, the built–in tests affect the performance of the system operation by only one gate delay regardless the size of the on–chip ECC implemented. We had selected Hamming (22,16) SEC–DED code in this thesis study. With this feature we can anticipate that the BIST architecture can be equally effective to any modular on–chip BIT ECCs of Hamming (n,k) SEC–DED codes. Therefore, it can be applied directly to embedded memories in 32–bit, 64–bit or 128–bit systems for improved data integrity.

CHAPTER 5

VLSI IMPLEMENTATION AND FPGA PROTOTYPE

5.1 Introduction

One purpose of this thesis is to design and develop a built–in testable on–chip ECC for embedded memories under VLSI/ASIC system environment. In Chapter 4 we proposed a built–in testable architecture and defined high–level functionality and specifications for the design. In this chapter we shall present the design implementation, verification, and prototype aspects of the design project. First, we start with a brief review of basic VLSI/ASIC design flow. Second, we describe the modular design and test of each of the internal functional units and the integrated design using VLSI/ASIC CAE tools in Cadence^{TM} (V4.2) EDA environment on Sun SparcStations. We also discuss the design simulation method of using Verilog HDL and Verilog–XL. Finally, we describe rapid design prototyping using Xilinx FPGA technology under the same integrated Cadence EDA environment.

5.2 VLSI/ASIC design procedures

The design of a VLSI/ASIC circuit has several stages. A simplified VLSI/ASIC design flow is shown in Figure 5–1. The conceptual design refers to an abstraction of what is to be implemented. This is the creative part of VLSI/ASIC design where the designer must use his imagination as well as knowledge and experience to solve the problem in question. The expected output from this stage may include the fundamental schemes devised.

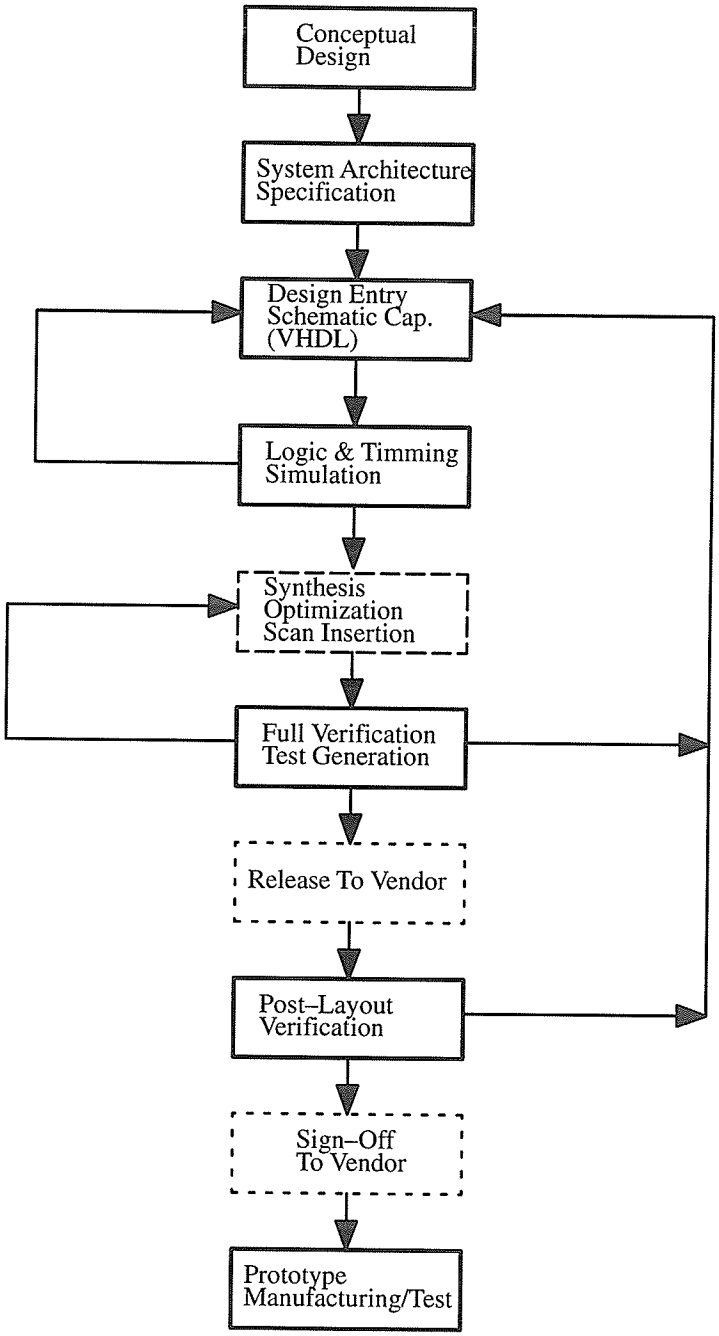


Fig. 5–1 A simplified VLSI/ASIC design flow

the prospective target VLSI/ASIC technology and vendor selected, the EDA environment and VLSI CAD tools to be used, the system view and specifications, and perhaps other system considerations such as the interfacing aspects of the design.

The next stage in the design procedure is designing the circuit to be implemented. A structured approach is usually taken to design the circuits in a hierarchical manner. A system architecture or structure with blocks of its subsystems is generated first. It specifies the functionality of the system and its subsystems, and their logic connections. Next, each of these subsystems are designed in a similar manner until the lowest level is reached, where the detailed circuitry consisting of the primary logic elements are configured and implemented. From this point, complex elements may be designed and implemented by combining the small or simple elements. This is the so called bottom–up design methodology. The design scripts generated in the early stages of the design procedure are done in a top–down fashion. The designer can concentrate on the features and specifications of the design rather than the quite detailed and low level considerations.

With sophisticated VLSI CAD/CAE tools available today, the next stage in the VLSI design procedure is called design entry. That is to get the circuit design scripts on the notebook of the designer to the workspace of the design entry tool on a workstation. The design methodology schematic capture was used in this project, other methods such as high level structured design using hardware description languages (HDL) are competitive alternatives. Schematic capture means that the CAD software can generated a series of netlists of the circuit design in specified target format. A netlist file is a data file that contains connectivity information of a circuit in specified data format. The netlist files generated or extracted (captured) from the schematic can be used to generated a layout of the design for silicon fabrication. The netlists can also be used in logic and/or timing simulations to test the design for verification before the fabrication process is launched. Entering

a schematic is just like drawing a schematic diagram in a window using the CAD tools. Library elements for certain technology such as Xilinx 3k or 4k technologies are available. Candence™ supports hierarchical design, as do most VLSI CAD tools. Thus, schematic design was done in a bottom–up fashion as mentioned previously.

The next procedure shown in the design process is simulate and generate the net-lists. Each abstraction entered using schematic capture was tested using a simulator such as Verilog–XL™ or SILOS ™. If it works correctly by itself, it passes the verification of functionality on this stage, and can be used in a higher level of the design. If errors in operation or design of the element were found during the simulation, then the circuit design and corresponding schematic would have to be corrected until the simulation yields the correct results. The simulation can accept the design entered using VHDL models to verify the functionality of the design. Logic synthesis and optimization tools then generate either the gate–level schematics or netlist of the design given the design model in VHDL.

As mentioned, pre–layout logic and timing simulation is done at this point to test the design before prototyping/manufacturing. This type of simulation models a circuit at the gate level instead of at the transistor level. Gate level simulation allow circuits to be simulated independent of the technology to be used in manufacturing. It is to test the functionality of gates and the circuits regardless of the fabrication process.

Normally, after the design has been completely entered and simulated, we are ready to release the design to a VLSI/ASIC vendor to generate a layout of the chip for prototype manufacturing. The generation of layout consists of placing and routing standard cells in a given technology. Placement and routing can be done automatically by routing programs, but sometimes it may be performed manually for performance reasons. After the generation of layout, we need to simulate the design annotated with the timing data to ensure that

capacitances due to layout have not violated the timing specifications or constraints. The final sign–off of the VLSI/ASIC design to the ASIC vendor is on the completion of this post–layout verification and a fault grading process for testability.

In our project, we chose a rapid prototype method using field programmable gate array (FPGA) technology from Xilinx to validate the BIST solution to on–chip ECCs for embedded memories. The major merits of the FPGA are its instant implementation and infinite reprogrammable capacity. It is this feature that provides the VLSI/ASIC designers with a powerful and economic approach in rapid new design prototyping. Instead of sending the data from the CAD system to drive the pattern generator that transfers the patterns to photo–sensitive masks used for fabrication process of the chip, and then waiting for a completed silicon wafer processing cycle before getting design samples, the designer can instead download the design through a communication cable from the workstation to a Xilinx FPGA device. This process uses the connectivity information extracted from the design and specifically attached data packages for device programing process and configures the FPGA device into the specified VLSI design as desired. The whole process is completed in minutes. The design obtained can soon be tested for verification, or applied in real application if applied.

The final stage of the design procedures is vendor/manufacturing test. The output from this stage may include the evidence that shows that both desired functionality and testability specification are met. The major objective of this thesis is to develop and implement a cost effective and high performance built–in test solution to designing on–chip BIT ECCs for embedded memories. It provides a design–for–testability method and modular approach that greatly reduced the complexity and the requirement imposed on sophisticated VLSI/ASIC testers.

5.3 Modular implementations

The memory data protection that the on–chip ECC can normally provide is through its on–chip encoding and decoding logic based on the ECC code implemented. In a RAM, the data to be stored in RAM cells is called codedword. When the data is retrieved, it is decoded back to the original data. In a ROM, the ECC circuitry is the decoding logic itself, with the encoding process done in some other way than incorporating the encoding logic on the memory chip. The decoding logic is closely coupled with the data corrector and error indicator flags. In the literature the decoder of an ECC circuitry often refers to the decoding logic, error detector/corrector and the error flags as well. Hence decoder is a major design of any on–chip ECC scheme. In our implementation, the on–chip BIT ECC does not include the encoder portion.

The design implementation is discussed in terms of its components: the ECC core, the PRTG, the MUX, the MISR and the test controller. A bottom–up methodology is used and the circuit design of subsystems is completed first. The bottom–up methods allows the designer to optimize the low–level implementations by using perhaps minimal hardware resources. The circuit design and simulation is performed in Design Composer™ and Verilog–XL™ and Xilinx 4k library cells integrated within Cadence™ (V4.2).

5.3.1 The Hamming decoder (the ECC Core)

The modified Hamming (n, k) SEC–DED code is implemented with n = 22, k = 16. The schematic of the decoding logic circuit is shown in Figure 5–2. In this section, we will briefly describe the features of the ECC core design, and present the design verification process.

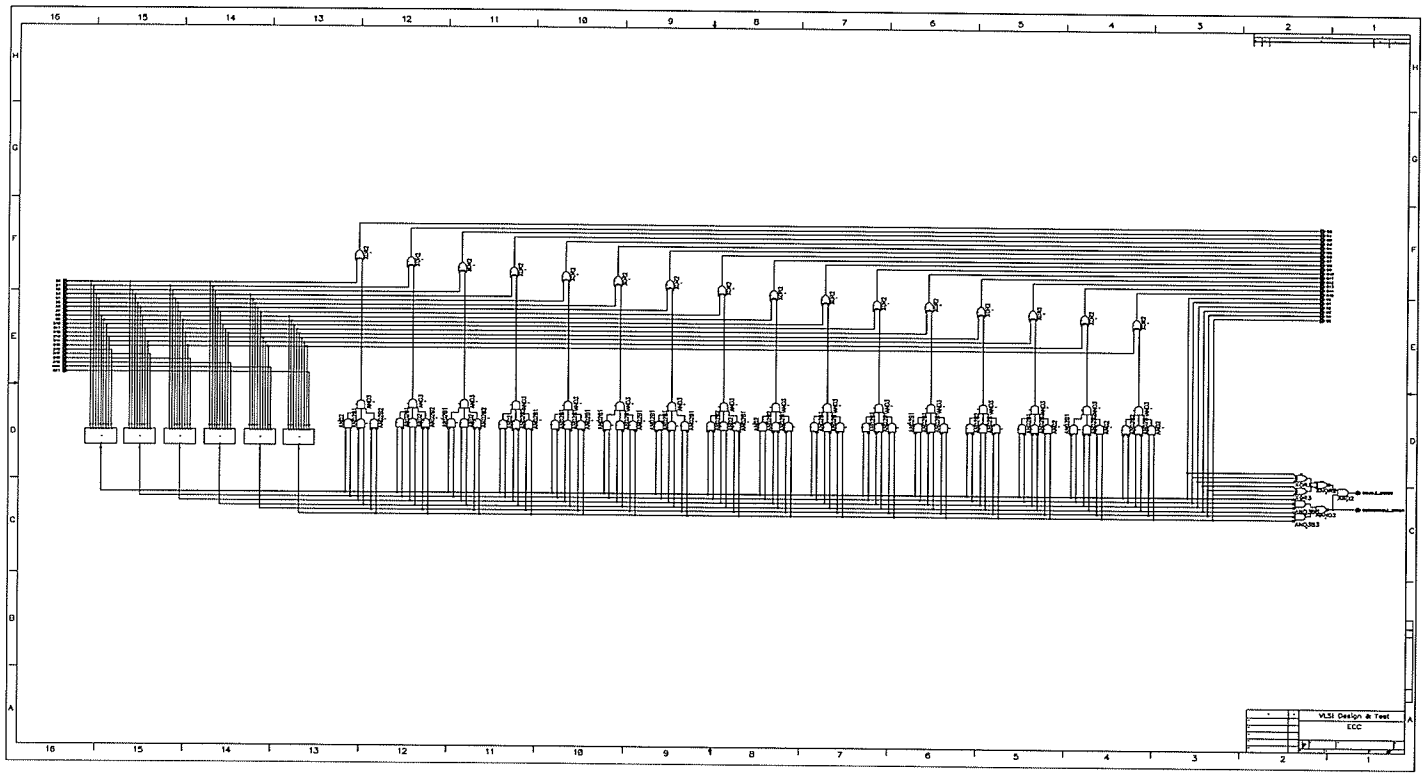


Fig. 5–2 Schematic of Hamming (22,16) SEC–DED circuit

5.3.1.1 Fully combinational and bit–sliced solution

This decoding circuit maps the coded memory word (n bits) into the corrected memory word (k bits), and also generates a two–bit error signal for reporting. It is noted that this implementation is a fully combinational and bit–sliced solution. It is faster in speed for it has parallel operations on each bit of the operand. Its principle design and implementation can be easily extended to on–chip ECC designs of any size using the modified Hamming (n, k) SEC–DED code.

5.3.1.2 The PG, SG and COR circuits

In the ECC core, the parity generator consists of 6 parity trees; The parity generator output is taken as the syndrome vector of the codedword in process. The 6–bit syndrome vectors are decoded by the 16–bit wide syndrome decoder, each bit of which performs a different function on the individual bits of syndrome vectors; The data corrector is an array of 22 XOR gates. If any bit of syndrome decoder output yields a 1, the corresponding data bit is to be flipped over (by being XORed with 1). In other words, it is corrected when it follows through the corresponding bit of the data corrector circuit. Should any bit on the syndrome decoder output be zero, it indicates the corresponding data bit is correct and then by–passed through the data corrector.

The error indicator flags will be set when an error or multiple errors are detected by the ECC circuits. It indicates whether the error detected is correctable or uncorrectable (multiple errors) using the dedicated flag bits.

At this point, it is worth mentioning that the modified Hamming SEC–DED code implemented deserves a faster speed implementation. Since it has minimum 1's in its H

matrix and each row of the matrix has the same number of 1's, the decoding circuit's 6 parity trees all have same number of inputs and hence the same depth. As for the original Hamming SEC–DED code, the last parity tree would have 22 inputs, hence the height of the parity trees are not even, and the data transfer over the PG circuit would be slower.

5.3.1.3 Design verification

To verify the correct operation of the ECC core itself, it was simulated with coded words, and some of which are error–injected with single or multiple errors. The simulation output concludes that the implementation works correct. The simulation files (the stimulus files and the simulation output files) are included in the Appendices of this document.

5.3.2 The pseudorandom test generator (PRTG)

The proprietary pseudorandom test generation circuit (PRTG) primarily provides a set of random and repeated test stimulus for testing the on–chip ECC circuitry in the on–demand mode and pseudoconcurrent (interleaved) test modes. It generates and receives internal control signals to monitor the test processes and coordinate with other blocks of the BIST system. A minor add–up to the premier PRTG configuration makes the circuit capable to support the other two scan–based self test modes easily. The schematic of the PRTG circuit is shown in Figure 5–3.

5.3.2.1 The PRTG core

The implementation of the PRTG circuit is based on a simplified BILBO structure. It mainly consists of a shift register of 11 bits with a multiplexed serial input, a serial output and multiple parallel outputs. The serial input is either from the scan input or from the feed

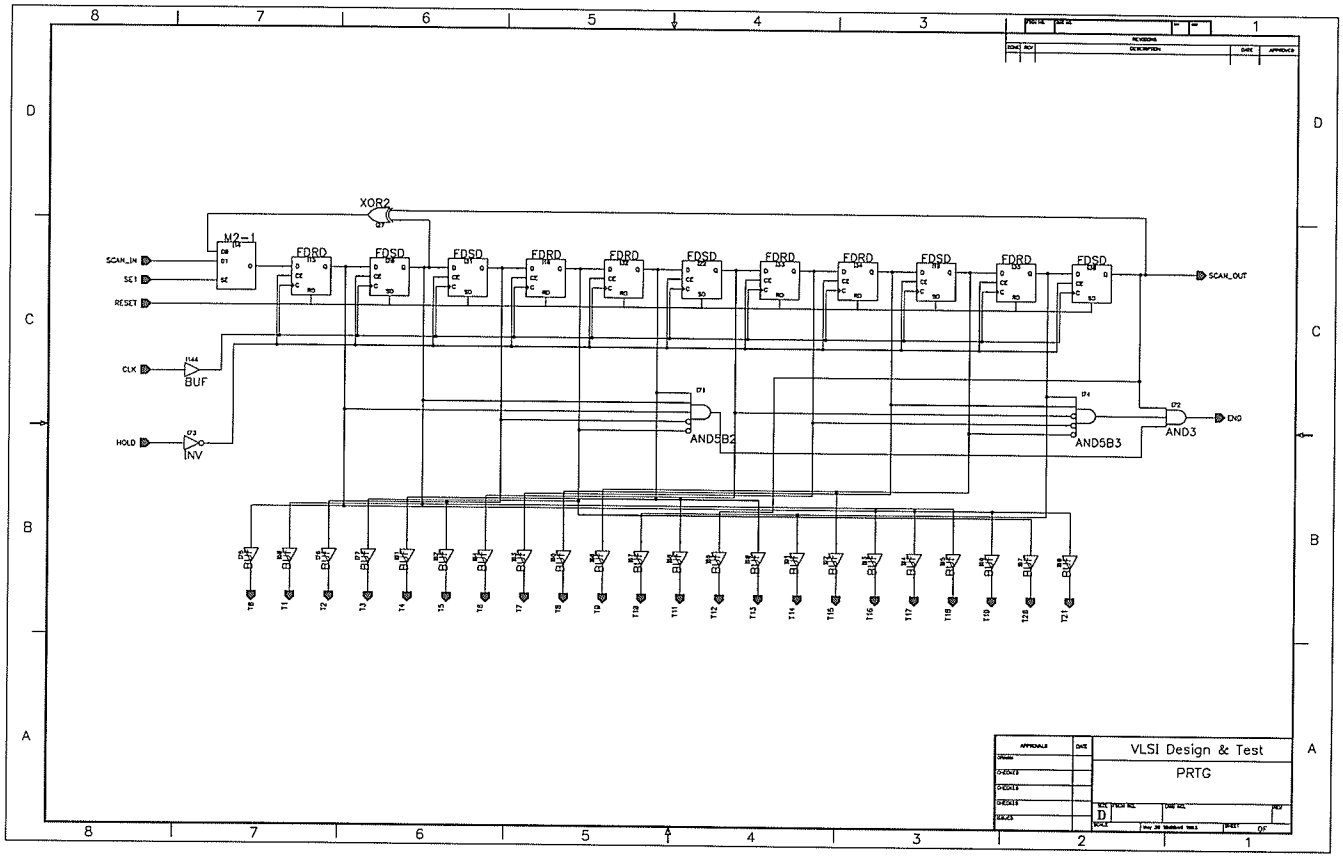


Fig. 5–3. Schematic of pseudorandom test generator (PRTG)

back network. The feed back network configuration is dominated by a primitive polynomial of degree 11: $X^{11} + X^2 + 1$. This primary polynomial has the least number of terms in the polynomials of same degree [PETE61]. This implies the simplest hardware configuration can be achieved for generation of the same random stimulus space of order $2^{11} - 1$.

The select input of the 2–to–1 MUX picks up the feed back network when its select input SE1 = 0 set by the test controller in on–demand test and pseudoconcurrent test; When the system is to perform scan–based tests, the signal SE1 is set 1 that allows the scan test vectors to be clocked in from the input port SCAN–IN, shifted through the PRTG register and clocked out at the port SCAN_OUT of the circuit.

5.3.2.2 The HOLD status

Besides the clock and reset inputs that are provided to each of the D flip–flops in the register, each flip–flop has a clock enable input CE that accepts a HOLD signal from the test controller. The HOLD signal is to hold or "freeze" the stimulus generator (and the signature analyzer discussed later) whenever an interleaved test is exited as the system is called back to normal operation while a test cycle is not completed yet. When the system reenters the self test in pseudoconcurrent mode, the HOLD signal will be released and the interleaved test can resume from the status it left.

5.3.2.3 The END signal

Also included in the PRTG circuit is a multibit AND function on the bits of output stimulus vectors to detect the end of each test cycle. Because any two adjunct vectors in a $2^n - 1$ vector space can be taken as the seed and the last test vector respectively, and a deterministic sequences of test vectors are to be generated repeatedly, both the seed and

the last test vector can be hardcoded into the PRTG circuit. As such, the test pattern generated always starts with the seed pattern and ended with the last test vector in one test cycle. In the PRTG implementation, the seed is realized by specifying a default binary value to the shift register. for instance, a combination of flip flops with either reset or preset terminals connected to the RESET signal will do the work. On the other side, an END signal is generated by the AND function whose inputs are conditioned according to the last test pattern. Whenever a test pattern matches the condition set up for the AND function, an END signal is asserted. The conditioned AND logic is to function as a simple on–chip comparator.

The END signal plays an important role in self test operations in on–demand mode or pseudoconcurrent mode. The signal END is forwarded to the test controller and the MISR to acknowledged the completion of a test cycle. Responding to the END signal, some actions would be taken in both test controller and the MISR circuitry with respect to flag logging and system clear–up for a new test cycle.

5.3.2.4 Fanout realization

The stimulus vector generated by the PRTG is 11 bits. Mapping the PRTG outputs onto the 22–bit test patterns required for testing the on–chip ECC required an array of buffers to be placed. The fanout configuration is dominated by the merged group table which resulted from the test matrix reduction in Chapter 4.

5.3.2.5 Design verification

Simulation on the PRTG circuit was performed in two stages. First, the PRTG was simulated to verify its functionality in generating the set of test stimulus with the seed con

dition defined in the hardware. This part of logic simulation can be used to determine and verify the last test pattern in a given test cycle. Second, the overall PRTG circuit was simulated to test the correctness of its operations and functionality. This included the timing and procedures for the END detection, the HOLD status and resumption from the frozen state, and the correctness of the test patterns mapped through the fanout circuitry. The PRTG simulation also covered the operations of serially shifting the possible scan test patterns in and through the PRTG register for scan tests. This includes the test of the 2–to–1 MUX function and appropriate selection of the input resources. The simulation results concluded that the PRTG implementation worked correctly. The simulation files are included in the Appendices of this document.

The step–wise partition coupled with the from–simple–to–complex philosophy embedded in the PRTG simulation process ensures not only a complete verification of the components and the whole functional unit, but also an easy–to–debug solution. So it is a well accepted method in engineering design practice. In some sense, by simulating the partly–done digital circuit, the simulation is interleaved with circuit design and development process. So it can serve not only as a verification tool, but also a circuit design aid in an incremental circuit development method.

5.3.3 The multiplexer (MUX)

The 22 2–to–1 multiplexer was implemented to select the input data to the ECC from two separated sources. The switching function depends upon the systems status or operations. That is, whether the system is in normal memory operation or called upon for self test. The schematic of the MUX is shown in Fig. 5–4.

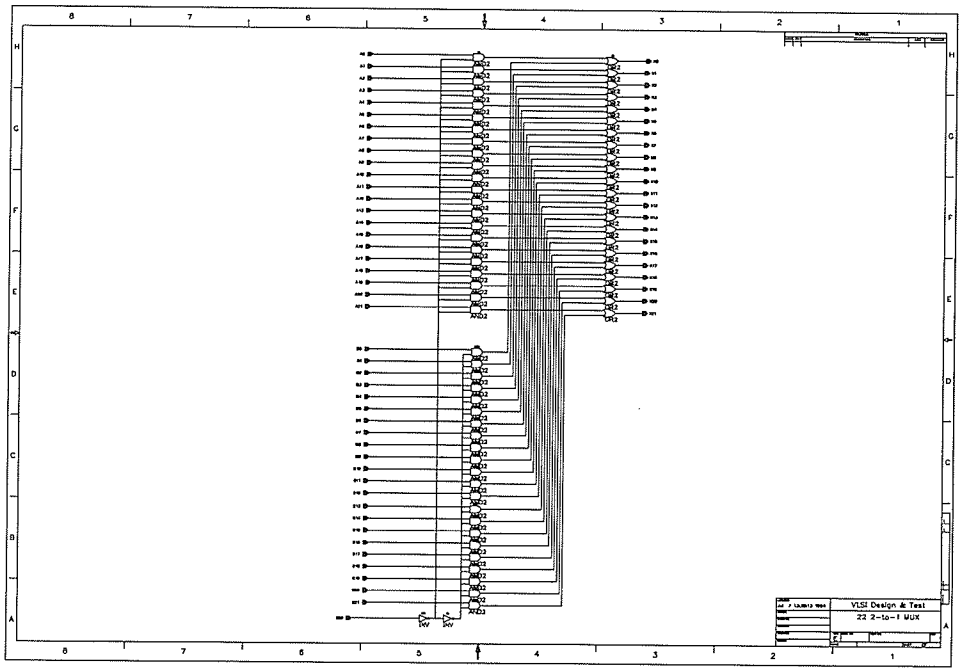


Fig. 5–4 Schematic of 22 2–to–1 multiplexer

5.3.3.1 Standard implementation

The implementation of the MUX here can be taken as standard. The 44 inputs are form the 22–bit wide coded memory word and the 22–bit wide self test stimulus. The select input SE0 is the control input that determines which input data path is to be selected active and routed out through the output OR gates. When SE0=1 it selects the coded memory word, with SE0=0 it selects the test inputs.

5.3.3.2 Design verification

The multiplexer unit is simulated to verify the correctness of its functionality. Two

groups of data with each having 22 bits can be multiplexed with the select signal set to 1 or 0 respectively. The 22–bit output is subsequently compared. The simulation results shows the MUX implementation works correctly. The simulation files are included in Appendices of this document.

5.3.4 Multiple input signature register (MISR)

The multiple input signature register (MISR) is capable of handling the on–chip test result analysis process in a parallel fashion. It takes a multiple bit pattern as its input to generate the signature. The signature is a test engineering term that was coined to simply mean a compressed word that represents the results of a particular test. In our design implementation, the fundamental MISR circuitry must be modified so that the same hardware block can be configured under software control to support not only the pseudorandom based self tests but also the scan–based self tests. To obtain the lowest possible error/fault masking rate in our MISR design, we need to implement a built–in solution to depress the aliasing problem effectively.

The schematic of the MISR unit is shown in Figure 5–5. The operation of the MISR consists of: 1) Compacting the test output data and producing the final signature; 2) Checking this signature against a hardcoded correct signature on the chip, and storing the result of this comparison in a single bit flag register that is put out to the processor; and 3) Scan test support.

5.3.4.1 BILBO–based MISR structure

The BIST structure BILBO is a prime candidate for the MISR that primarily acts as either a parallel signature analyzer or a simple shift registers. The configuration of this

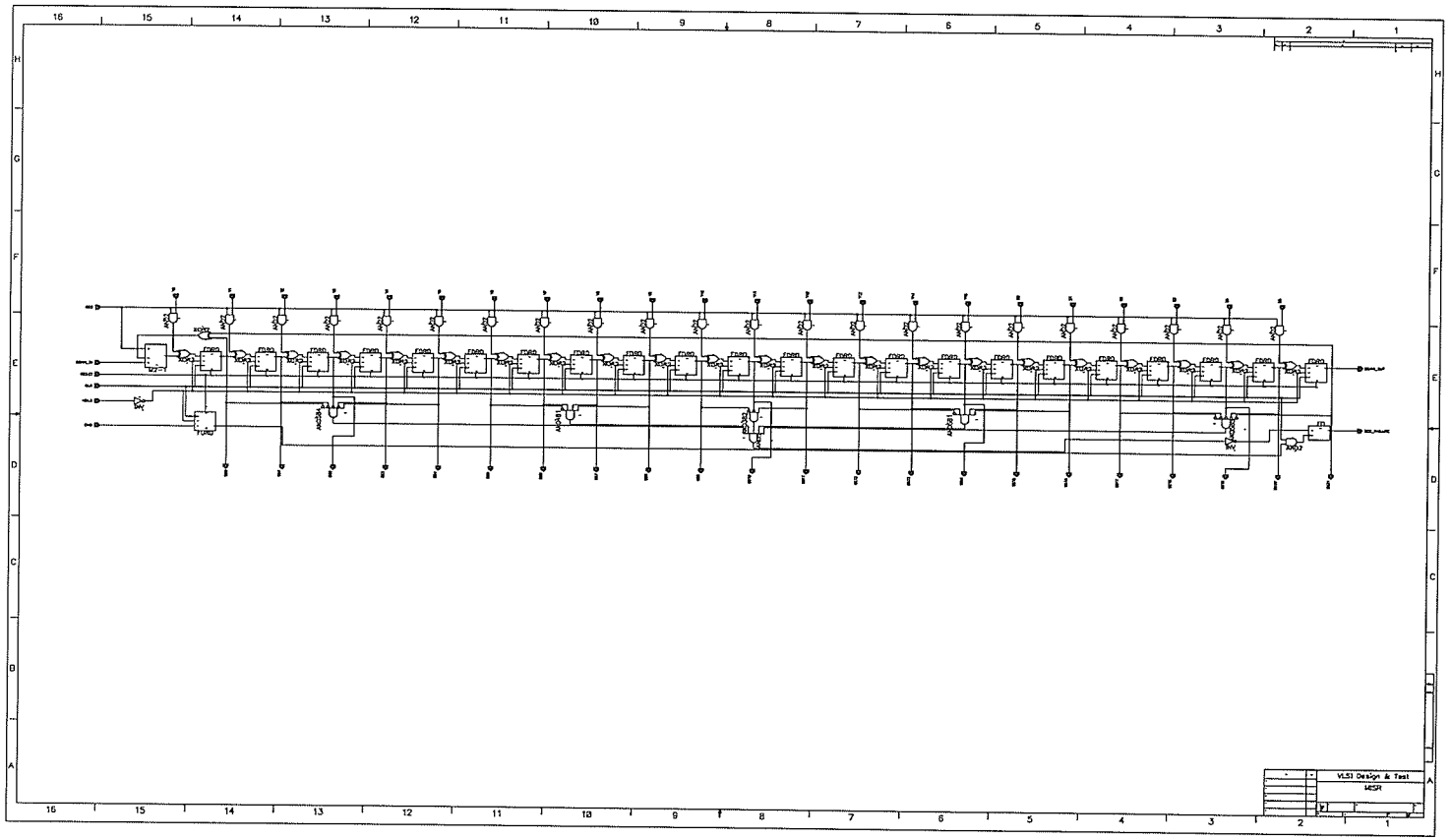


Fig. 5–5 Schematic of multi–input signature register (MISR)

22–stage register is basically determined with a primitive polynomial of degree 22: $X^{22} + X + 1$. To completely implement the functionality of the MISR unit, we added on a hard-coded comparator to realize the on–chip signature comparison, and a 1–bit flag register for automatic fault reporting. A couple of other gates are used to process the internal signals from other blocks of the system for timing purposes.

When the system is in a self test operation associated with the pseudorandom based test as in on–demand test or interleaved test, the control input SE2 is set to 1. This allows the circuit to function as a parallel read register with the signal from the feedback network being selected to pass through the 2–to–1 MUX and reach the XORed input of the flip flops at the first stage. Using this approach, a signature of the test in progress can be generated and read out in parallel.

In scan–based self test of other system modules, the register can be reconfigured into a serial read–in and serial read–out register with the control input SE2 set to 0. Test data can then be scanned in via the serial input port or scanned out via the serial output port. As for scan test of the on–chip ECC circuit itself, the MISR functions as a parallel read–in and serial read–out register. Hence, a relatively complicated control procedure is needed such that the control inputs SE2 must be flipped over and over for proper data flow during the test. A complete application program for the on–chip BIT ECC is out of the scope of this project due to the limited time frame. However, it is worth noting that the set of test stimulus for scan test of the on–chip ECC can be much smaller in size. In Chapter 4 we generated a set of 8 vectors for sufficient testing of PG/SG.

5.3.4.2 On–chip signature comparison

An AND gate structure with hardcoded final signature is used as a simple on–chip

signature comparator. The input of the AND gate is the 22 bits of the signature generated by the BILBO. Using this approach, an AND gate is the only cost for on–chip signature comparison. If the signature generated is the same as the hardcoded correct signature, the output of the AND gate should be 0; otherwise, it yields a 1 to indicate a faulty ECC is detected by the built–in test in the last test cycle just finished.

The hardcoded signature is supposed to be the correct one. With high performance simulation tools, the calculation of the correct signature can be part of logic simulation task. Upon the completion of the simulation on a partly–done implementation, the designer can proceed with the hardcoding of the final correct signature on the AND gate in the design. One may write a high level program in C or Pascal to perform the calculation as well.

5.3.4.3 Depressing the aliasing rate

The aliasing problem is the one inherent with signature compaction using structure such as the MISRs. The effect of the aliasing problem is the error/fault masking in built–in test systems [DAMI89, MiZh90]. This means that it is possible that the fault(s) can escape being caught by the test patterns applied which could have covered the fault(s) in the fault simulation runs. In reality, this problem will lower the fault coverage of certain test sets. Investigations and much research work have been done extensively in this area [WILL89, StWu90, IvAg88]. According to literature, the aliasing rate, as a probability of error/fault masking, is on the order of 2^{-n}, where n is the number of the bits in signature generation circuits.

It is noted that in our implementation of the MISR unit, we build a 22 bit signature register, by taking in the 6 bit syndrome vector as well as the 16 bit data output as the inputs to the signature register. The signature register may have only taken 16 bit data output for

generating the signature according to conventional MISR configurations. In fact, we used six (6) more flip flops in the signature register. The trade–off is that we can obtain a factor of six lower aliasing rate down to 2^{-22} from 2^{-16}, because the number of the bit in the signature generation circuit is 22 bits rather than 16 bits.

5.3.4.4 Automatic fault reporting

The last but not the least portion of MISR structure is the fault logging circuit based on the 1–bit flag register. It has a data input from the signature comparator, and a clock input from an AND gate where the clock signal is ANDed with the END signal from the PRTG unit. The one bit fault flag register actually serves part of the automatic fault reporting system. It latches in and stores the test result: when it stores a logic 0, the ECC_FAILURE=0 indicates fault–free; whereas if it stores a 1, it means a faulty on–chip ECC has been detected by the self test in the last test cycle finished. In next section, we shall discuss the implementation of the timing control in clocking the flag register.

5.3.4.5 The END signal

The output of the signature comparator represents the conclusion of one self test cycle, which is supposed to be latched into the flag register when the last test in a given test cycle is completed. In fact, the final signature is generated one clock cycle after the last test stimulus completes. In other words, it takes one extra clock cycle for the test output to be compacted and the final signature compared with the correct one. The END signal from the PRTG is primarily devised to signal the end of the test cycle, and used in the MISR unit to active the clock input on the flag register. This implementation serves the requirement of latching in the test result on each test cycle. Because the END signal is generated simultaneously with the last test stimulus, one flip flop was placed in the MISR for the END

signal to control the synchronization of the flag logging process.

5.3.4.6 The HOLD status

As discussed in the test pattern generator unit, a HOLD status which is critical for interleaved self test must be implemented in the MISR unit as well. The HOLD input of the MISR is connected to the clock enable terminal on each of the flip flops to freeze the status of the register whenever the system is called back for normal operation from an on–going interleaved test process whereas a complete test cycle is not yet finished. This holds up all the test data analysis associated with the hold of the test stimulus generation on the PRTG unit and ensures that the interleaved test can be resumed later from where it left. By disabling the clock input, all the intermittent data are stored in the MISR register and ready for use when the test is resumed by resetting the HOLD signal.

We shall have a detailed discussion on the generation of the HOLD signal and other internal control signals later in this chapter where the implementation of test controller is described.

5.3.4.7 Design verification

Simulation of the MISR design, using the integrated Verilog–XL simulator under Cadence EDA environment plays a triple–fold role in implementing the MISR unit. First, the BILBO–based 22–stage MISR register itself was simulated to verify the correctness of its operation, which leaving the implementation of rest of the MISR unit open. Second, simulation on an integration of the PRTG, the MUX, the ECC core, and the MISR core directly helped to deduce the correct final signature; It takes 2048 clock cycles to catch the final signature with the PRTG providing a set of 2^{11} –1 test stimuli for one complete test

cycle. Third, the full functionality of the MISR–based design was tested after the correct signature was hardcoded and the fault reporting structure was implemented. Again, the simulation process can be a powerful design aid in incremental VLSI/ASIC design process. The simulation on the MISR–based unit concluded that the MISR implementation worked correctly. The simulation files are included in the Appendices of this document.

5.3.5 Built–in test controller

Self testing of the on–chip ECC with multiple self test modes involves consistent

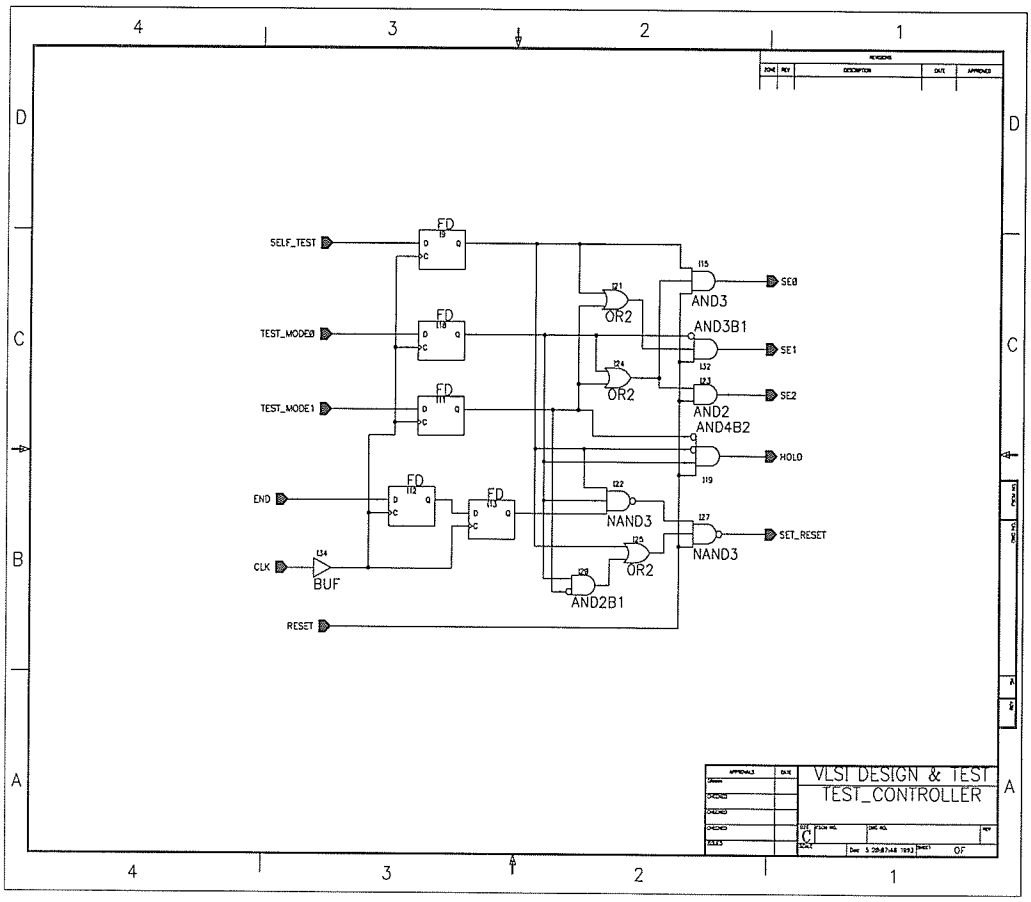


Fig. 5–6 Schematic of built–in test controller

operations of all the functional units of the BIST design. A test controller which coordinates the self test procedures was implemented following a simple yet effective theme. The schematic of the test controller is shown in Figure 5–6.

5.3.5.1 Control signals

Before we describe the circuit configuration and the operational aspects of the test controller, we need to collect all the control signals external and internal to the on–chip BIT ECC design.

i) The external control signals from the host processor are:

CLK: a clock signal used to run the self–test functions only;

TEST_MODE[1:0]: a two–bit signal to chose a test mode out of four;

SELF_TEST: a signal for signaling the module into and out of self–test, and

RESET: a global reset to place the built–in test circuitry into a known state.

ii) The internal signal to the input of the test controller is:

END: a signal from the PRTG unit, which signals the detection of the last test in a psudorandom based test cycle.

iii) The test controller provides five control signals:

SE0: select test input when true, otherwise select the coded memory word;

SE1: select Scan–In when true, otherwise select the feedback from PRTG;

SE2: Select feedback from the MISR when true, otherwise select the Scan–In;

HOLD: hold intermittent test results when true (Interleaved test only);

R/S: preset PRTG, clear MISR and other flip flops except the flag when true;

A state table of the built–in test controller is given in Fig. 5–7. The eight states are defined by SELF_TEST, TEST_MODE0, and TEST_MODE1. Please note that for sim-

plicity the D_0, D_1, and D_2, are used in the table to stand for the three bits of test commands, respectively.

D0	D1	D2	SE0	SE1	SE2	HOLD	R/S	
1	1	1	1	0	1	0	0	On–Demand Test Mode
0	1	1	0	0	1	0	1	Exit from On–Demand test
1	1	0	1	0	1	0	0	Psudo–Concurrent Test Mode
0	1	0	0	0	1	1	0	Exit from Psudo–Concurrent Test
1	0	1	1	1	1	0	0	Scan Test Mode
0	0	1	0	1	1	0	1	Exit from Scan Test
1	0	0	0	1	0	0	0	Scan Path Test Mode
(0	0	0)	0	0	0	0	1	Exit from Scan Path mode; Reset

Fig. 5–7 State table of the built–in test controller

5.3.5.2 The scheme for test control implementation

The operation of the controller can be easily verified against the definition of the 3 bit commands from the host processor and corresponding state that the test controller is in. Before we analyze its operation in terms of each of the alternative self test modes in next section, we briefly examine some aspects of the approach used in the implementation of the test controller circuit. In this implementation, three D flip flops are used for latching in the test commands from the processor. The other two D flip flops are for tuning up the timing sequences in establishing an SET_RESET (housekeeping) action based upon the END signaling. The combinational logic block generates the signals required to coordinate the subsystems. This approach, coupled with the straight forward configuration of the END signaling and the master RESET resulted in an efficient control logic implementation.

It makes a difference in reducing the hardware configurations complexity of the control logic, if we do the housekeeping at the end of each self test cycle instead of at the beginning of a new test cycle. The master reset is always available to reset the built–in test fixture to a predefined state. Suppose the last test vector is now detected. For a new test cycle to begin, a seed is to be placed in the test generator PRTG when the signature analyzer MISR and all internal flip flops except the flag register are reset. It is noted that the clearing of the MISR unit should not been done until the output data of the last test has been compacted to form the final signature, and the signature comparison result is latched into the flag register. Based on this analyses, a simple way to generate the housekeeping signal Set_Reset is to place two flip flops to postpone the housekeeping signal two clock cycles after the END signal.

Because the test result is logged upon completion of one self test cycle, and only overwritten after next test cycle ends, the system could be called upon to exit the self test in progress without affecting the current logged test result. By using three dedicated registers for logging the test command, the commands have been designated so that the processor can issue one of the self test commands on any clock cycle and the text fixture will respond to the command on the next clock cycle. Based on this approach of entering or exiting any one of the four self test modes, the housekeeping is also done each tim the system exits from a self test operation (i.e. SELF_TEST=0), with one exception being exiting from the interleaved self test mode. The HOLD signal is then generated to hold the intermittent self test results as well as to freeze the test generator and response analyzer. If the system reenters the interleaved self test, the self test state can be resumed instantly. However, if the system is to enter another self test mode, a reset command should be issued before the system enters the self test operation. The overall synchronous hardware realization of the built–in test controller is traded–off with the possibility of at most one clock cycle loss as the system exits from the pseudorandom based self test.

5.3.5.3 Design verification

The test controller implementation was simulated to verify its correctness. First, the correlation between the inputs and output signals and their timing relations was examined to ensure the dynamic reconfigurability is implemented correctly for each and all alternative self test modes as defined in the specification. Second, the operation of the controller in performing the coordination of each self test was verified. The stimulus was created to probe key procedures such as the sequential actions when the last test of one test cycle is reached, and the proper status control as the system exits and/or reenters the interleaved test, and so forth. Third, the housekeeping procedures were tested which may be invoked by three different sources. The simulation results concluded the test controller implementation worked correctly. The simulation files are included in the Appendices of this document.

5.4 System integration and verification

With the subsystem modules implemented and verified in a modular fashion, the top level of the on–chip BIT ECC can be produced easily with the integration of all internal functional units. The top level circuit design and integration is shown in Figure 5–8. As this is a full implementation of the built–in testable architecture and specifications described in Chapter 4. It realizes all intraconnections and its I/O signals and ports that interface the on–chip BIT ECC module with the embedded memory, the host processor, and other possible system modules, such as the ones that reside in the same scan test path and neighboring to the on–chip BIT ECC when the system is configured for scan based self test.

Before we describe the detailed simulation on the top–level design implementation, let us have a brief review on design simulation using Verilog HDL and high performance

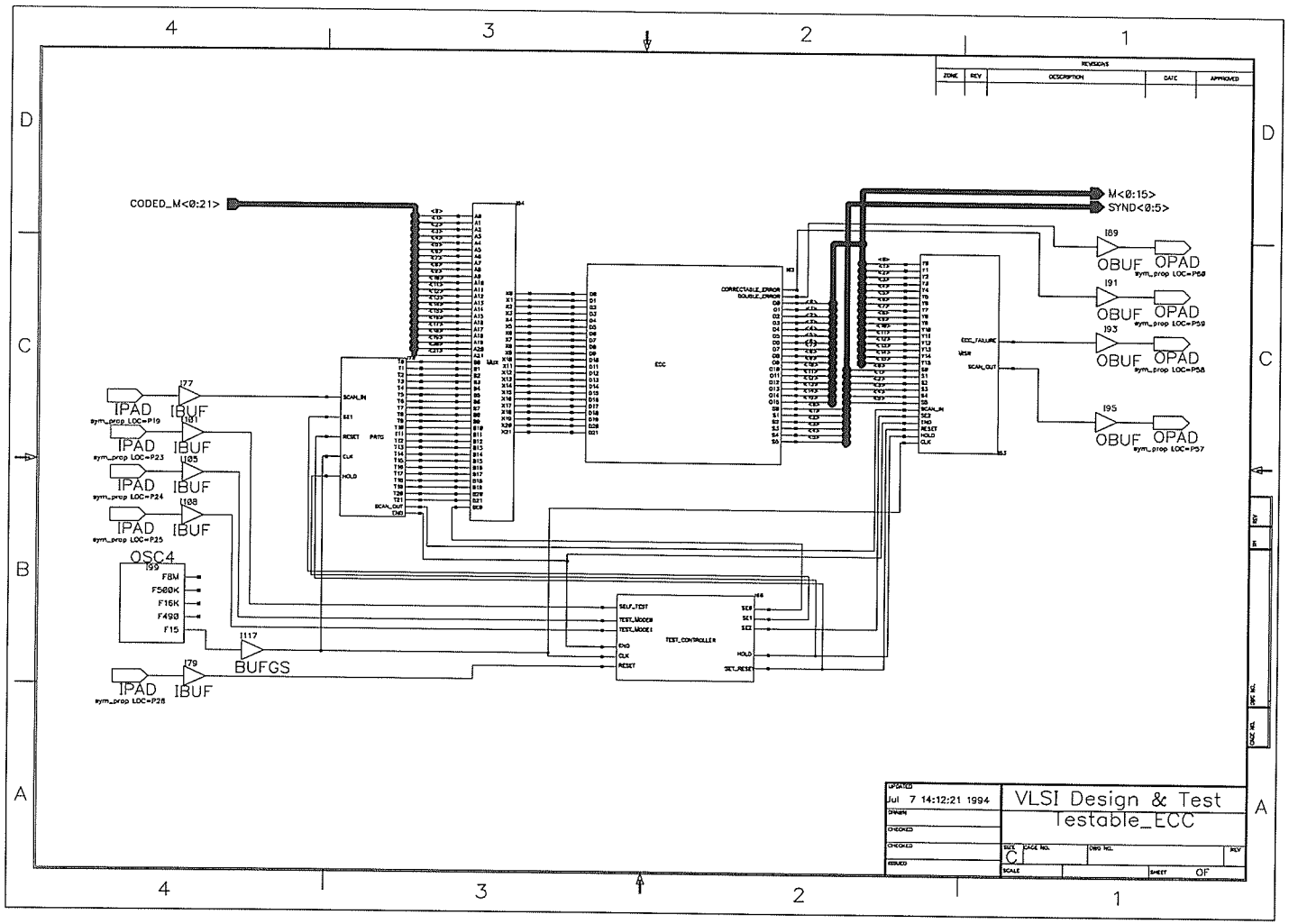


Fig. 5–8 Schematic of the built–in testable on–chip ECC

simulator Verilog–XL integrated within Cadence™ V4.2.

5.4.1. Simulation using Verilog HDL and Verilog–XL simulator

In this thesis, Verilog HDL as a high level stimulus language, is extensively used in developing all the stimulus for design simulation. A stimulus file written in Verilog is a format of inputs that are created to sense or probe the functions being verified. It is similar to the test patterns for testing but not the same. Test pattens are generated against a certain type of targeted faults represented by the fault models. Stimulus for simulation focus on the generating and verifying the intended functions to be tested. The high performance Verilog–XL™ simulator supports both logic and timing simulations, and can be called upon from within the Cadence Design Composer™ directly to perform the simulations on the circuits under design. In this environment, the stimulus must be written in Verilog HDL in a Verilog–XL recognized format. In order to run the simulation, a netlist file of the design in a Verilog description format recognized by the simulator is also required. Using Verilog–XL™ within Cadence, this conversion of the design netlist to Verilog description is automated. A template stimulus file is also generated by the software to speed up the stimulus development.

As our design prototype utilized the Xilinx FPGA, XC 4000 library cells were used in all circuit implementations. The Xilinx technology libraries are also available within Cadence™ (V4.2). As such, the netlist files extracted in Xilinx xnf format are converted into a Verilog description (.v file) internally within Cadence.

Along with the (.v file) is the template stimulus file(.stim file) for the given design to simulate. By default, the stimulus in the .stim file are null vectors, the designer is responsible for the real stimulus patterns needed for the simulation run. The time taken to run a

real simulation is quite dependent on the size of the design and the functions to be simulated. All relevant data to the simulation process are logged under the design directory for reference besides the simulation run output (.log file).

5.4.2 System level design verification

Simulation of the top level design implementation can be taken as an extension of the simulations on the subsystem blocks. However, in the previous simulations on lower level internal units, the correctness of internal control signals as inputs to the unit under simulation have been taken for granted in all separate or individual design verifications. Now, these internal signals are no longer assumed and only the external control signals (the processor commands) are given to exercise the functionality of the top–level design. The correctness of generating the internal signals and their real effects on the behavior of each subsystem and system are to be verified with special attention. The simulation stimulus injected to the top–level design implementation for simulation purposes may be restricted as well to a limited number of system input pins. In our case, this is the six input pins on the top level design shown in Fig. 5–8. In this manner the simulations reflect the system operations and functionality in a working environment that most closely resembles the reality. In other words, only given the commands from the processor, the top level circuitry should work out the all self test procedures by itself in any given self test mode arbitrarily assigned. The simulation files for the top level design verification are included in the Appendices of this document.

In the top level design implementation and verification, the .v file has 433 lines in it and the .stim file has 191 lines in it. The simulation output file (verilog.log) is 2,654,642 byte in size. In the .stim file there are no internal signals in the top level simulation. Only those input ports of the system are given the stimulus externally. These signals are

CLOCK, ECC_TEST, MODE0, MODE1, GLOBALSTRT, RESET, SCAN_IN. For each self–test mode, two test cycles (4096 clock cycle) are scheduled using a loop. This tests the behavior of the system entering the designated self test mode and between the test cycles, that is, when one completes and the next one is to begin. Following this test, exiting from the self–test mode back to the normal operation is verified. Procedures were designed to test the interleaved pseudoconcurrent self test with it exiting even though a completed test cycle has not yet finished. Subsequently, the system resumes the interleaved test. The simulation shows the intermittent test results were "locked" and served as starting point when the interleaved self test is resumed. The hardware reconfigurability of the built–in test circuitry to support the two scan based test modes were tested to ensure test patterns or the test output can be routed through the registers and the serial ports of the system to their correct destination. The differentiation between the scan test of the on–chip ECC and the scan path test mode is visible from the simulation output.

The independency of each self–test mode can be a major concern in such a system that is supposed to support multiple self test methods. This was given considerable attention and observation throughout the simulation runs as well as in creating stimulus to sense and capture the performance concerned. The hardware dynamic reconfigurability was verified for the worst cases, such as the system accidently enters and exits self test in a miscoupled test mode to ensure it is handled smoothly without crashing the system. The simulation results concluded that complete top level design implementation works correctly.

5.4.3 Discussion

It is noted that the built–in test system is designed such that the on–chip ECC both enters and exits the test mode synchronously. This means that when the module is called upon to serve the system normal operation, the present test vector is allowed to complete

and the next test vector is held in the test generator before serving the system. Also, when it is released by the system for self test, it does not start its test cycle until the start of the next clock cycle. This is a very safe design but it could result in the loss of one clock cycle every time pseudorandom based self test is exited.

5.5 Rapid prototyping with Xilinx FPGAs

To validate the BIST solution to on–chip BIT ECC for high speed and high performance embedded memories, rapid design prototyping using FPGA technology from Xilinx was implemented. With Cadence 4.2, the Xilinx FPGA development package is a built–in feature. Downloading the design to a Xilinx FPGA device is done through a communication cable (Xchecker) connected to a serial port on the workstation and a demo board from Xilinx. In this project, we used a Xilinx XC4000 family device (XC4003b) to implement the prototype. On the demo board an appropriate configuration is set up so as to activate the external control signals to the design and to capture and display the output signals and flag signals for design evaluation.

The XC4000 families of FPGA is of third generation FPGAs. It provides the benefits of custom CMOS VLSI/ASIC, while avoiding the initial cost, time delay, and inherent risk of a conventional masked gate array. Since a custom design can be created instantly by programming a FPGA chip from within the same workstation environment, and since the chip can be reprogrammed an unlimited number of times, FPGAs are ideal for rapid prototyping of virtually any innovative VLSI/ASIC design. This has the effect of reducing the time from specification to a conventional prototype.

We have experimentally analyzed the BIST on–chip ECC design prototype in the XC4003b chip and evaluated its integrated built–in test features. We primarily observed

the self test operations in on–demand and interleaved mode using pseudorandom BIST. As for the two scan based tests, although we did not provide the scan test input to perform the test on the Xilinx chip, we verified in detail the test fixture to make sure the circuit configues correctly under the software control so that the scan based tests are truly supported.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

We have studied general aspects of on–chip ECCs, and the testability of parallel implementation of on–chip Hamming (n,k) SEC–DED codes for embedded memories. Based on the development and implementation of the built–in testable architecture and design method for on–chip ECCs, the following conclusions are drawn:

The test matrix compaction scheme, dominated by the Hamming (n,k) ECC codes, constitutes a major component in a cost–effective built–in test architecture for the on–chip ECC circuitry. The row/column linear compaction, implemented through the test matrix mappings, resulted in a 50% overhead reduction in the built–in test vector generation circuit than the customary BIST implementation methods.

The impacts of the built–in test on the normal system performance is a consequence of any BIST architectures. The built–in test implemented in this study impacted the memory system speed performance by only one gate delay, due to the need for multiplexing normal data inputs and test stimulus inputs to the ECC. It is imperative to note that this side effect is independent of the size of the on–chip ECC.

Testability specification is a measure of the built–in testable VLSI/ASIC design. We have proposed a natural circuit partitioning, and investigated the coverage of the set of tests based on the BIST scheme, and found that the test vector space generated and expanded on–the–chip covers all test stimulus required for testing the PG/SG circuitry, and the syndrome space created from the output of testing the PG/SG covers the test vectors re-

quired for testing SD/DC portion of the on–chip ECC circuit. Therefore, we anticipated a >99% fault coverage of the stuck–at faults in the on–chip ECC circuitry.

Aliasing is an error/fault masking problem inherit with test response compression for signature analyze. We extended the number of input bits to the MISR to 22 bits by adding the 6 syndrome bits; We anticipated, by this approach, the aliasing rate can be of the order of 2^{-22}.

Dynamically reconfigurable built–in test circuitry design embodied in our design implementation produced a compact on–chip test fixture. Associated with an integrated built–in test architecture, the same on–chip test circuitry can be easily reconfigured under software control to support multiple self test methods (on–demand pseudorandom test, the pseudoconcurrent (interleaved) test, the scan test and the scan by–pass test). Resetting the test circuitry when exiting from any test modes except for the psudoconcurrent mode, reduces the steps in self test control procedures, and consequently reduce the size of the test controller.

In summary, this research work has the following contributions:

1) A built–in testable architecture is developed for on–chip ECCs. It provides an alternative BIST solution to self testing embedded memories with on–chip ECC based on a divide–and–conquer strategies.
2) Identified a test vector compaction scheme, which resulted in a 50% reduction in on–chip test generation hardware.
3) Developed a design–for–testability method for on–chip ECCs of Hamming (n,k) SEC–DED codes. It takes advantages of multiple built–in test methodologies and provides > 99% fault coverage of stuck–at faults in the on–chip ECC circuitry.

4) The built–in testable on–chip ECC design method is implemented and verified for a Hamming (22,16) SEC–DED circuit, and prototyped using a Xilinx FPGA.

5) The BIST solution supports multiprocessor based system solutions with embedded private memory configurations. Testing an array of homogeneous on–chip ECCs of embedded memories can be effectively done in parallel by using one on–chip test generation circuit.

6) Provided considerable experience and insights into issues facing system designers in light of increasing levels of integration.

Further development of this research work is recommended as follows:

1) The design methods may be expanded to high level HDL models with testability. A modular VHDL model of the built–in testable ECC can be developed.

2) The BIST architecture and implementation methods may be converted into design aids in VLSI/ASIC EDA environments. Studies on EDA methodologies such as logic and test synthesis (compile) methods and techniques are needed.

3) The algorithms which generates best testable on–chip ECC codes may enhance the design–for–testability methods directly. This work may also contribute to practical aspects of applying error coding theory.

REFERENCES

[ABRA90] M. Abramovici M.A. Breuer, and A.D. Friedman, "Digital Systems Testing and Testable Design," *Computer Science Press*, New York, 1990

[AgSe88] V. D. Agrawal and S. C. Seth, "Test generation for VLSI Chips," *Computer Society Press,* Washington, D.C., 1988

[BARD87] P. H. Bardell, "Built–In Test for VLSI: Pseudorandom Techniques," John Wiley & Sons, Inc., 1987

[BOOS70] D. Bossen, "Optimum test patterns for parity networks," in *Proc. AFIPS Fall Joint Computer Conf.* pp. 63–67, Nov. 1970

[CHEN84] C. L. Chen and M. Y. Hsiao, "Error–Correcting Codes for Semiconductor Memory Applications: A State of the Art Review," *IBM Journal of Research Development* 28, pp 124–34, March 1984

[CLIF 74] R. Clif *and T.R.N. Rao*, "Improving the Yield of LSI Memory Chips by Application of Coding," Proc. of 8th Annual Princeton Conf. Infor. Sci. Syst., 1974

[DAVI85] H. L. Davis, "A Word–Wide 1 Mb ROM with Error Correction." *Dig., 1985 IEEE Int. Solid–State Circuits Conf,* WAM3.2, pp 40–41, Feb. 1985

[DONN91] J. Donnell, "Boundary Scan Puts Tomorrow's Devices To Test," *Electronic Design*, pp. 75–86, June 1991

[DOSI92] B. Nadeau–Dostie, "Scan design for Integrated Circuits," Training Course at the *Workshop on New Directions for Testing,* BNR, May 1992

[FASA90] P. P. Fasang, " Testing Embedded Rams in ASIC Chips," *IEEE Custom Integrated Circuits Conference*, 1990

[FrSa91] M. Franklin, K. Saluja, "Pattern Sensitive Fault Testing of RAMs with Built–in ECC", IEEE Test reference, pp.385–392. 1991

[FuAr89] K. Furutani, K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, and K. Mashiko, "A Built–in Hamming Code ECC Circuit for DRAMs," *IEEE J. Solid–State Circuits,* vol24, pp.50–55, Feb.1989

[FuHe88] T. Fuja, C. Heegard, and R.M. Goodman, "Linear Sum Code for Random Access Memories," *IEEE Trans. Computer, vol.37* pp. 1030–1042, 1988

[FuPr90] E. Fujiwara and D.K. Pradhan, "Error–Control Coding in Computers," *IEEE Computer*, pp.63–72, July 1990

[GAIT88] N. Gaitanis, " The design of TSC Error C/D Circuits for SEC/DED Codes," *IEEE Trans. on Computers* C–37, pp. 258–65, March 1988

[GlBr89] C. S. Gloster and F. Brglez, "Boundary Scan with Built–in Self–Test," *IEEE Design & Test of Computers,* pp.266–274, Feb.1989

[GoHl83] P. Golan, J. Hlavicka, "A Method For Parallel Decoding of Double–Error Correcting Group Codes," *IEEE Test Conference*, pp. 338–341, 1983

[HoMc90] P.D. Hortensius, R. D. McLeod, B. W. Podaima, "Cellular Automata Circuits For Built–In Self–Test," *IBM Jour. of R & D,* vol. 34 May 1990

[HSIA70] M.Y. Hsiao. "A Class of Optimal Minimum Odd–Weight–Column SEC–DED Codes," *IBM Journal of Research Development* 14. July 1970

[JaSt86] S. Jain, C. Stroud, "Built–in Self testing of Embedded Memories," *IEEE Design & Test*, pp 27–37, 1986

[KALT90] H. Kalter, "DRAM, 16 Mbit Device, 50ns, on–chip ECC, pipeline layout and Redundancy," J–SC, pp.1118–1128, Oct. 1990

[LoHu88] F. Lombardi, "Approaches for the Repair of VLSI/WSI DRAMs by Row/Column Deletion," in Proc. 18th Int. FTCS. pp.342–347, 1988

[MANO83] T. Mano, "Circuit Technologies for a VLSI Memory," *IEEE Jour. of Solid–State Circuits* SC–18, pp. 463–70, Oct. 1983

[MaTu90] C. M. Maunder and R. E. Tulloss, "The Test Access Port and Boundary Scan Architecture," IEEE Computer Society Press, Los Alamitos, California, 1990

[McCL86] E. J. McCluskey, "Fault–Tolerant Computing, theory and Techniques", Pretice–Hall Inc., vol. 1, pp.95–173, 1986

[MEHR84] S. Mehrotra, "A 64 Kb CMOS EEROM with On–Chip ECC," *Dig., 18th Annual Int. Symp. Fault–Tolerant Comput,* June 1988

[MiZh90] D. M. Miller, S. Zhang, W. Pries, and R. D. Mcleod, "Estimating Aliasing in CA and LFSR Based Signature Registers," *Proc. of 1990 IEEE*

Int. Conf. on Computer Design, pp. 157–159, 1990

[MILL88] D. M. Miller, "Review of Built–In Self–Test Methodologies," *Canadian Conf. on ECE.* pp. 375–378, Nov. 1988

[MoTh86] P. Moritz, L. Thorsen, "CMOS Circuit Testability," *IEEE Jour. of Solid–State Circuits*, vol. sc–21, No. 2, pp. 306–309, April 1986

[NOOR90] A. Noore, "DRAMs, Fault–Tolerant Memory Designs for Improved Yield and Reliability," *ISCAS 90*, vol. 4, pp. 2744–2747, 1990

[OHNI81] N. Ohnishi, T. Ishikawa, and N. Mutoh, "System Configuration on Full Wafer LSI," *Proc. of Int. Symp. Mini And Microcomputers,* Jan. 1981

[PETE61] W. W. Peterson, *Error–Correcting Codes.* New York: The M.I.T. Press and John Wiley & Sons, Inc., 1961

[RaFu89] T. Rao, E. Fujiwara, "Error–Control Coding For Computer Systems," New Jersey: Prentice–Hall, Inc., 1989

[SHIN83] T.Shinoda, *Y. Ohnishi, H. Kawamoto*, and K. Narita, "A 1Mb ROM with on–chip ECC for Yield Improvement," *dig., 1983 IEEE Int. Solid–State Circuits Conference*, pp 158–59, Feb. 1983

[SuWa84] Z. Sun, L. Wang, "Self–Testing of Embedded RAMs," *1984 International Test Conference*, pp. 148–156, 1984

[TrAg93] R. Treuer, V. Agarwal, "Built–in Self–Diagnosis for Repairable Em bedded RAMs," *IEEE Design & Test of Computers,* pp.24–33, June 1993

[WALT90] C. J. Walter, "Distributed Fault–Tolerant Computing System, Error Monitoring Observations," *FTCS* 90, pp 48–55, 1990

[WIER93] R. W. Wieler, Z. Zhang, and R. D. McLeod, "Using an FPGA Based Computer as a Hardware Emulator for Built–in Self–Test Structures," *Fifth Int. Workshop on Rapid System prototyping*, pp.7–18, France, June/94

[WiPa83] T. Williams, K. Parker, "Design for Testability — A Survey," *Proc. of IEEE,* vol.71, No. 1, pp. 98–112, Jan. 1983

[WILL89] T. W. Williams, W. Daehn " Aliasing Errors In Multiple Input signature Analysis Registers," *IEEE ETC89*, pp 338–345, 1989

[WILS93] R. Wilson, "Megacells A Challenge For ASIC Designers," Electronic Engineering Times, Sept. 27, 1993

[YAMA88] T. Yamada, H. Kotani, J. Matsushima and M. Inoue, "A 4–Mbit DRAM with 16–bit Concurrent ECC," *IEEE J. Solid–State Circuits,* vol.23, pp.20–25, Feb.1988

[ZHAN94] Zaifu Zhang, "Delay Fault Testing and Statistical Detectability Analysis in Scan–based Design Logic Circuits," Ph.D. Candidacy Report, Dept ECE, University of Manitoba, 1994

APPENDIX A:

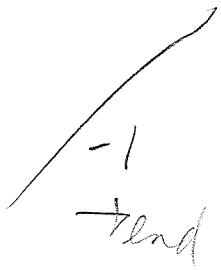
SIMULATION STIMULUS (.stim Files)

(in Verilog HDL)

The following .stim files record the test patterns developed for design verifications over the six (6) modular designs in the hierarchy of the on–chip BIT ECC design. The .sim files were written in Verilog HDL and served as simulation stimulus in the design verification process using Verilog–XL. The schematics of the design units are cited for quick references.

i) **ecc.stim** is for the ECC_CORE unit (ecc.sch is on p.66, Chapter 5).
ii) **prtg.stim** is for the on–chip PRTG unit (prtg.sch is on p. 69, Chapter 5).
iii) **mux.stim** is for the multiplexer unit (mux.sch is on p.72, Chapter 5).
iv) **misr_core.stim, sig_sim.stim,** and **misr.stim** are for the on–chip MISR unit (misr.sch is on p.75, Chapter 5).
v) **t_ctrller.stim** is for the built–in test controller unit (t_ctrller.sch is on p.80, Chapter 5).
vi) **ecc_t.stim** is for the top level on–chip BIT ECC (ecc_t.sch is on p.85, Chapter 5).

```
// ECCF.STIM
-----------------------------
'timescale 1 ns/100 ps

module test;

reg D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16,D17,D18,D19,D20,D21;
wire CORRECTABLE_ERROR;
wire DOUBLE_ERROR;
wire O0;
wire O1;
wire O2;
wire O3;
wire O4;
wire O5;
wire O6;
wire O7;
wire O8;
wire O9;
wire O10;
wire O11;
wire O12;
wire O13;
wire O14;
wire O15;
wire S0;
wire S1;
wire S2;
wire S3;
wire S4;
wire S5;

supply1 XVDD;
supply0 XGND;

ECCF t1
(.CORRECTABLE_ERROR(CORRECTABLE_ERROR),
.DOUBLE_ERROR(DOUBLE_ERROR),.D0(D0),.D1(D1),.D2(D2),.D3(D3),.D4(D4),.D5(D5),
.D6(D6),.D7(D7),.D8(D8),.D9(D9),.D10(D10),.D11(D11),.D12(D12),
.D13(D13),.D14(D14),.D15(D15),.D16(D16),.D17(D17),.D18(D18),
.D19(D19),.D20(D20),.D21(D21),.O0(O0),.O1(O1),.O2(O2),.O3(O3),
.O4(O4),.O5(O5),.O6(O6),.O7(O7),.O8(O8),.O9(O9),.O10(O10),
.O11(O11),.O12(O12),.O13(O13),.O14(O14),.O15(O15),.S0(S0),
.S1(S1),.S2(S2),.S3(S3),.S4(S4),.S5(S5));

initial
  begin
    #10       // coded word (0000000000000000)
    D0=0;     //   with check bits(000000)
    D1=0;
```



```
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=0;
D15=0;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10       // coded word (1111111111111111)
D0=1;     //   with check bits(000000)
D1=1;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10       // coded word (0111111111111111)
D0=0;     //   with check bits(110100)
D1=1;
D2=1;
```

```
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=1;
D18=0;
D19=1;
D20=0;
D21=0;

#10        // coded word (1011111111111111)
D0=1;      //   with check bits (101100)
D1=0;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=0;
D18=1;
D19=1;
D20=0;
D21=0;

#10       // coded word (1101111111111111)
D0=1;     //   with check bits (011100)
D1=1;
D2=0;
D3=1;
```

```
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=0;
D17=1;
D18=1;
D19=1;
D20=0;
D21=0;

#10       // coded word (1110111111111111)
D0=1;     //   with check bits (110010)
D1=1;
D2=1;
D3=0;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=1;
D18=0;
D19=0;
D20=1;
D21=0;

#10       // coded word (0101010101010101)
D0=0;     //   with check bits (011110)
D1=1;
D2=0;
D3=1;
D4=0;
```

```
D5=1;
D6=0;
D7=1;
D8=0;
D9=1;
D10=0;
D11=1;
D12=0;
D13=1;
D14=0;
D15=1;
D16=0;
D17=1;
D18=1;
D19=1;
D20=1;
D21=0;

#10      // coded word (1010101010101010)
D0=1;    //   with check bits (011110)
D1=0;
D2=1;
D3=0;
D4=1;
D5=0;
D6=1;
D7=0;
D8=1;
D9=0;
D10=1;
D11=0;
D12=1;
D13=0;
D14=1;
D15=0;
D16=0;
D17=1;
D18=1;
D19=1;
D20=1;
D21=0;

#10      // coded word (0000000000000001)
D0=0;    //   with check bits (001011)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
```

```
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=0;
D15=1;
D16=0;
D17=0;
D18=1;
D19=0;
D20=1;
D21=1;

#10       // coded word (0000000000000010)
D0=0;     //   with check bits(010011)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=1;
D15=0;
D16=0;
D17=1;
D18=0;
D19=0;
D20=1;
D21=1;

#10       // coded word (0000000000000100)
D0=0;     //   with check bits (100011)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
```

```
D9=0;
D10=0;
D11=0;
D12=0;
D13=1;
D14=0;
D15=0;
D16=1;
D17=0;
D18=0;
D19=0;
D20=1;
D21=1;

#10       // coded word (0000000000001000)
D0=0;     //   with check bits (001101)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=1;
D13=0;
D14=0;
D15=0;
D16=0;
D17=0;
D18=1;
D19=1;
D20=0;
D21=1;

#10       // coded word (0000000000010000)
D0=0;     //   with check bits (010101)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
D9=0;
D10=0;
```

```
D11=1;
D12=0;
D13=0;
D14=0;
D15=0;
D16=0;
D17=1;
D18=0;
D19=1;
D20=0;
D21=1;

// inject errors
// sequence: no error, 1 error, 2 errors

#10        // coded word (0000000000000000)
D0=0;     //   with check bits(000000)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=0;
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=0;
D15=0;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10        // coded word (0000000000000000)
D0=0;     //   with check bits(000000)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=1;     // 1 bit error
D7=0;
D8=0;
```

```
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=0;
D15=0;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10      // coded word (0000000000000000)
D0=0;    //   with check bits(000000)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=1;    // 1 bit error
D7=0;
D8=0;
D9=0;
D10=0;
D11=0;
D12=0;
D13=0;
D14=0;
D15=0;
D16=0;
D17=0;
D18=0;
D19=1;   // 1 bit error
D20=0;
D21=0;

// inject errors
// sequence: no error, 1 error, 2 errors

#10      // coded word (1111111111111111)
D0=1;    //   with check bits(000000)
D1=1;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
```

```
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10      // coded word (1111111111111111)
D0=1;    //   with check bits(000000)
D1=0;    // 1 bit error
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;

#10      // coded word (1111111111111111)
D0=1;    //   with check bits(000000)
D1=0;    // 1 bit error
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
```

```
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=0;   // 1 bit error
D15=1;
D16=0;
D17=0;
D18=0;
D19=0;
D20=0;
D21=0;


// inject errors
// sequence: no error, 1 error, 2 errors

#10      // coded word (0111111111111111)
D0=0;    //   with check bits(110100)
D1=1;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=1;
D18=0;
D19=1;
D20=0;
D21=0;

#10      // coded word (0111111111111111)
D0=0;    //   with check bits(110100)
D1=1;
D2=1;
D3=1;
D4=1;
D5=1;
```

```
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=0;    // 1 bit error
D16=1;
D17=1;
D18=0;
D19=1;
D20=0;
D21=0;

#10      // coded word (0111111111111111)
D0=0;    //   with check bits(110100)
D1=1;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=0;     // 1 bit error
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=0;    // 1 bit error
D16=1;
D17=1;
D18=0;
D19=1;
D20=0;
D21=0;

#10       // coded word (1011111111111111)
D0=1;     //   with check bits (101100)
D1=0;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
```

```
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=0;
D18=1;
D19=1;
D20=1;   // 1 bit error
D21=0;

#10       // coded word (1011111111111111)
D0=1;     //   with check bits (101100)
D1=0;
D2=1;
D3=1;
D4=1;
D5=1;
D6=1;
D7=1;
D8=1;
D9=1;
D10=1;
D11=1;
D12=1;
D13=1;
D14=1;
D15=1;
D16=1;
D17=0;
D18=1;
D19=0;   // 1 bit error
D20=1;   // 1 bit error
D21=0;

// triple and quatple errors

#10      // coded word (0000000000000000)
D0=0;    //   with check bits(000000)
D1=0;
D2=0;
D3=0;
D4=0;
D5=0;
D6=1;    // 1 bit error
```

```
    D7=1;    // 1 bit errot
    D8=0;
    D9=0;
    D10=0;
    D11=0;
    D12=0;
    D13=0;
    D14=0;
    D15=0;
    D16=0;
    D17=0;
    D18=0;
    D19=1;   // 1 bit error
    D20=0;
    D21=0;

    #10      // coded word (0000000000000000)
    D0=0;    //   with check bits(000000)
    D1=0;
    D2=0;
    D3=0;
    D4=0;
    D5=0;
    D6=1;    // 1 bit error
    D7=1;    // 1 bit errot
    D8=0;
    D9=1;    // 1 bit error
    D10=0;
    D11=0;
    D12=0;
    D13=0;
    D14=0;
    D15=0;
    D16=0;
    D17=0;
    D18=0;
    D19=1;   // 1 bit error
    D20=0;
    D21=0;

    #10000 $stop; // Change data every 1 MHz
  end

initial
  begin
    $display("                    DDDDDDDDDDDDDDDDDDDDDDOOOOOOOOOOOOOOOOOOSSSSSSCD");
    $display("                    01234567891111111111220123456789111111012345OO");
    $display("                              012345678901          012345      RU");
    $display("                                                                RB");
    $display("                                                                EL");
```

```
    $display("                                                    CE");
    $display("                                                    T_");
    $display("                                                    AE");
    $display("                                                    BR");
    $display("                                                    LR");
    $display("                                                    EO");
    $display("                                                    _R");
    $display("                                                    E ");
    $display("                                                    R ");
    $display("                                                    R ");
    $display("                                                    O ");
    $display("                                                    R ");
    $monitor($time,,
            D0, D1, D2, D3, D4, D5,
            D6, D7, D8, D9, D10, D11,
            D12, D13, D14, D15, D16, D17,
            D18, D19, D20, D21, O0, O1,
            O2, O3, O4, O5, O6, O7, O8,
            O9, O10, O11, O12, O13, O14,
            O15, S0, S1, S2, S3, S4, S5,
        CORRECTABLE_ERROR,DOUBLE_ERROR);
  end
initial
  $gr_waves(
            "D0",D0,"D1",D1,"D2",D2,"D3",D3,"D4",D4,"D5",D5,
            "D6",D6,"D7",D7,"D8",D8,"D9",D9,"D10",D10,"D11",D11,
            "D12",D12,"D13",D13,"D14",D14,"D15",D15,"D16",D16,"D17",D17,
            "D18",D18,"D19",D19,"D20",D20,"D21",D21,"O0",O0,"O1",O1,
            "O2",O2,"O3",O3,"O4",O4,"O5",O5,"O6",O6,"O7",O7,"O8",O8,
            "O9",O9,"O10",O10,"O11",O11,"O12",O12,"O13",O13,"O14",O14,
            "O15",O15,"S0",S0,"S1",S1,"S2",S2,"S3",S3,"S4",S4,"S5",S5,
        "CORRECTABLE_ERROR",CORRECTABLE_ERROR,
        "DOUBLE_ERROR",DOUBLE_ERROR);

initial
  #10000
  $ps_waves("gr_wave.ps", "ECC_Core");
endmodule
```

```
--------------------------
//   PRTG.STIM
--------------------------


'timescale 1 ns/100 ps

module test;

reg CLK,GLOBALSTRT,HOLD,RESET,SCAN_IN,SE1;
wire T0;
wire T1;
wire T2;
wire T3;
wire T4;
wire T5;
wire T6;
wire T7;
wire T8;
wire T9;
wire T10;
integer i;

supply1 XVDD;
supply0 XGND;

PRTG_COREF t1
(.CLK(CLK),.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),
.RESET(RESET),.SCAN_IN(SCAN_IN),.SE1(SE1),.T0(T0),.T1(T1),.T2(T2),
.T3(T3),.T4(T4),.T5(T5),.T6(T6),.T7(T7),.T8(T8),.T9(T9),
.T10(T10));

initial
  begin
    #5 CLK=1;
    #5 CLK=~CLK;
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=0;
    SE1=0;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    for( i=0; i < 4096; i=i+1)
      begin
        #5 CLK=~CLK;
      end
```

```
    #10000 $stop; // Change data every 1 MHz
  end

initial
  begin
    $display("                    CGHRSSTTTTTTTTTT");
    $display("                    LLOECE01234567891");
    $display("                    KOLSA1          0");
    $display("                     BDEN            ");
    $display("                     A T_            ");
    $display("                     L  I            ");
    $display("                     S  N            ");
    $display("                     T               ");
    $display("                     R               ");
    $display("                     T               ");
    $monitor($time,,  CLK, GLOBALSTRT, HOLD,
           RESET, SCAN_IN, SE1, T0, T1,
           T2, T3, T4, T5, T6, T7, T8,
           T9, T10);

  end
initial
  $gr_waves("CLK",CLK,"GLOBALSTRT",GLOBALSTRT,"HOLD",HOLD,
           "RESET",RESET,"SCAN_IN",SCAN_IN,"SE1",SE1,"T0",T0,"T1",T1,
           "T2",T2,"T3",T3,"T4",T4,"T5",T5,"T6",T6,"T7",T7,"T8",T8,
           "T9",T9,"T10",T10);

endmodule
```

```
------------------
//  MUX.STIM
------------------


'timescale 1 ns/100 ps

module test;

reg CLK,GLOBALSTRT,HOLD,RESET,SCAN_IN,SE1;
wire END;
wire SCAN_OUT;
wire T0;
wire T1;
wire T2;
wire T3;
wire T4;
wire T5;
wire T6;
wire T7;
wire T8;
wire T9;
wire T10;
wire T11;
wire T12;
wire T13;
wire T14;
wire T15;
wire T16;
wire T17;
wire T18;
wire T19;
wire T20;
wire T21;
integer i;

supply1 XVDD;
supply0 XGND;

PRTGF t1
(.CLK(CLK),.END(END),.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),
.RESET(RESET),.SCAN_IN(SCAN_IN),.SCAN_OUT(SCAN_OUT),.SE1(SE1),.T0(T0),
.T1(T1),.T2(T2),.T3(T3),.T4(T4),.T5(T5),.T6(T6),.T7(T7),
.T8(T8),.T9(T9),.T10(T10),.T11(T11),.T12(T12),.T13(T13),
.T14(T14),.T15(T15),.T16(T16),.T17(T17),.T18(T18),.T19(T19),
.T20(T20),.T21(T21));

initial
  begin
    #5 CLK=1;
    #5 CLK=~CLK;
```

```
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=0;
    SE1=0;        //MUX selects feedback
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    for( i=0; i < 4096; i=i+1)
      begin
        #5 CLK=~CLK;
      end

    #5 CLK=~CLK;
    SE1=1;        // MUX selects Scan_In input
    SCAN_IN=1;
    for(i=0; i<50; i=i+1)
      begin
       #5 CLK=~CLK;
      end

    #5 CLK=~CLK;
    SE1=1;        // MUX selects Scan_In input
    SCAN_IN=0;
    for(i=0; i<50; i=i+1)
      begin
       #5 CLK=~CLK;
      end

    #10000 $stop; // Change data every 1 MHz
  end

initial
  begin
    $display("                    CEGHRSSSTTTTTTTTTTTTTTTTTTTTT");
    $display("                    LNLOECCE0123456789111111111122");
    $display("                    KDOLSAA1          012345678901");
    $display("                      BDENN                      ");
    $display("                      A T__                      ");
    $display("                      L  IO                      ");
    $display("                      S  NU                      ");
    $display("                      T   T                      ");
    $display("                      R                          ");
    $display("                      T                          ");
    $monitor($time,,  CLK, END, GLOBALSTRT,
           HOLD, RESET, SCAN_IN,
           SCAN_OUT, SE1, T0, T1, T2, T3, T4,
```

```
            T5, T6, T7, T8, T9, T10,
            T11, T12, T13, T14, T15, T16,
            T17, T18, T19, T20, T21);

  end
initial
  $gr_waves("CLK",CLK,"END",END,"GLOBALSTRT",GLOBALSTRT,
            "HOLD",HOLD,"RESET",RESET,"SCAN_IN",SCAN_IN,
            "SCAN_OUT",SCAN_OUT,"SE1",SE1,"T0",T0,"T1",T1,"T2",T2,"T3",T3,"T4",T4,
            "T5",T5,"T6",T6,"T7",T7,"T8",T8,"T9",T9,"T10",T10,
            "T11",T11,"T12",T12,"T13",T13,"T14",T14,"T15",T15,"T16",T16,
            "T17",T17,"T18",T18,"T19",T19,"T20",T20,"T21",T21);

endmodule
```

```
---------------------------
//  MISR_CORE.STIM
---------------------------


'timescale 1 ns/100 ps

module test;

reg
A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,A21,B0,B1,B
2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16,B17,B18,B19,B20,B21,SE0;
wire X0;
wire X1;
wire X2;
wire X3;
wire X4;
wire X5;
wire X6;
wire X7;
wire X8;
wire X9;
wire X10;
wire X11;
wire X12;
wire X13;
wire X14;
wire X15;
wire X16;
wire X17;
wire X18;
wire X19;
wire X20;
wire X21;

supply1 XVDD;
supply0 XGND;

MUXF t1
(.A0(A0),.A1(A1),.A2(A2),.A3(A3),.A4(A4),.A5(A5),
.A6(A6),.A7(A7),.A8(A8),.A9(A9),.A10(A10),.A11(A11),.A12(A12),
.A13(A13),.A14(A14),.A15(A15),.A16(A16),.A17(A17),.A18(A18),
.A19(A19),.A20(A20),.A21(A21),.B0(B0),.B1(B1),.B2(B2),.B3(B3),
.B4(B4),.B5(B5),.B6(B6),.B7(B7),.B8(B8),.B9(B9),.B10(B10),
.B11(B11),.B12(B12),.B13(B13),.B14(B14),.B15(B15),.B16(B16),
.B17(B17),.B18(B18),.B19(B19),.B20(B20),.B21(B21),.SE0(SE0),
.X0(X0),.X1(X1),.X2(X2),.X3(X3),.X4(X4),.X5(X5),.X6(X6),
.X7(X7),.X8(X8),.X9(X9),.X10(X10),.X11(X11),.X12(X12),
.X13(X13),.X14(X14),.X15(X15),.X16(X16),.X17(X17),.X18(X18),
.X19(X19),.X20(X20),.X21(X21));
```

```
initial
  begin
    A0=1;
    A1=1;
    A2=1;
    A3=1;
    A4=1;
    A5=0;
    A6=0;
    A7=0;
    A8=0;
    A9=0;
    A10=1;
    A11=1;
    A12=1;
    A13=1;
    A14=1;
    A15=0;
    A16=0;
    A17=0;
    A18=0;
    A19=0;
    A20=0;
    A21=0;
    B0=0;
    B1=0;
    B2=0;
    B3=0;
    B4=0;
    B5=1;
    B6=1;
    B7=1;
    B8=1;
    B9=1;
    B10=0;
    B11=0;
    B12=0;
    B13=0;
    B14=0;
    B15=1;
    B16=1;
    B17=1;
    B18=1;
    B19=1;
    B20=1;
    B21=1;
    SE0=0;
    #10000 $stop; // Change data every 1 MHz
  end
```

```
initial
  begin



   $display("   AAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBSXXXXXXXXXXXXXXXXXXXXX");
   $display("  012345678911111111111220123456789111111111122E012345678911111111111122");
   $display("            012345678901          0123456789010          012345678901");
    $monitor($time,,  A0, A1, A2, A3, A4, A5,
           A6, A7, A8, A9, A10, A11,
           A12, A13, A14, A15, A16, A17,
           A18, A19, A20, A21, B0, B1,
           B2, B3, B4, B5, B6, B7, B8,
           B9, B10, B11, B12, B13, B14,
           B15, B16, B17, B18, B19, B20,
           B21, SE0, X0, X1, X2, X3, X4,
           X5, X6, X7, X8, X9, X10,
           X11, X12, X13, X14, X15, X16,
           X17, X18, X19, X20, X21);

  end
initial
  $gr_waves("A0",A0,"A1",A1,"A2",A2,"A3",A3,"A4",A4,"A5",A5,
           "A6",A6,"A7",A7,"A8",A8,"A9",A9,"A10",A10,"A11",A11,
           "A12",A12,"A13",A13,"A14",A14,"A15",A15,"A16",A16,"A17",A17,
           "A18",A18,"A19",A19,"A20",A20,"A21",A21,"B0",B0,"B1",B1,
           "B2",B2,"B3",B3,"B4",B4,"B5",B5,"B6",B6,"B7",B7,"B8",B8,
           "B9",B9,"B10",B10,"B11",B11,"B12",B12,"B13",B13,"B14",B14,
           "B15",B15,"B16",B16,"B17",B17,"B18",B18,"B19",B19,"B20",B20,
           "B21",B21,"SE0",SE0,"X0",X0,"X1",X1,"X2",X2,"X3",X3,"X4",X4,
           "X5",X5,"X6",X6,"X7",X7,"X8",X8,"X9",X9,"X10",X10,
           "X11",X11,"X12",X12,"X13",X13,"X14",X14,"X15",X15,"X16",X16,
           "X17",X17,"X18",X18,"X19",X19,"X20",X20,"X21",X21);

endmodule
```

```
--------------------
//  SIG_SIM.STIM
--------------------


'timescale 1 ns/100 ps

module test;

reg CLK,GLOBALSTRT,HOLD,RESET,SCAN_IN,SE1,SE2;
wire COR_ERROR;
wire C0;
wire C1;
wire C2;
wire C3;
wire C4;
wire C5;
wire C6;
wire C7;
wire C8;
wire C9;
wire C10;
wire C11;
wire C12;
wire C13;
wire C14;
wire C15;
wire C16;
wire C17;
wire C18;
wire C19;
wire C20;
wire C21;
wire DOUBLE_ERROR;
wire D0;
wire D1;
wire D2;
wire D3;
wire D4;
wire D5;
wire D6;
wire D7;
wire D8;
wire D9;
wire D10;
wire D11;
wire D12;
wire D13;
wire D14;
wire D15;
```

```
wire D16;
wire D17;
wire D18;
wire D19;
wire D20;
wire D21;
wire SG0;
wire SG1;
wire SG2;
wire SG3;
wire SG4;
wire SG5;
wire SG6;
wire SG7;
wire SG8;
wire SG9;
wire SG10;
wire SG11;
wire SG12;
wire SG13;
wire SG14;
wire SG15;
wire SG16;
wire SG17;
wire SG18;
wire SG19;
wire SG20;
wire SG21;
integer i;


supply1 XVDD;
supply0 XGND;


SIG_SIMF t1
(.CLK(CLK),.COR_ERROR(COR_ERROR),.C0(C0),.C1(C1),
.C2(C2),.C3(C3),.C4(C4),.C5(C5),.C6(C6),.C7(C7),.C8(C8),
.C9(C9),.C10(C10),.C11(C11),.C12(C12),.C13(C13),.C14(C14),
.C15(C15),.C16(C16),.C17(C17),.C18(C18),.C19(C19),.C20(C20),
.C21(C21),.DOUBLE_ERROR(DOUBLE_ERROR),.D0(D0),.D1(D1),.D2(D2),
.D3(D3),.D4(D4),.D5(D5),.D6(D6),.D7(D7),.D8(D8),.D9(D9),
.D10(D10),.D11(D11),.D12(D12),.D13(D13),.D14(D14),.D15(D15),
.D16(D16),.D17(D17),.D18(D18),.D19(D19),.D20(D20),.D21(D21),
.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),.RESET(RESET),.SCAN_IN(SCAN_IN),.SE1(SE1),
.SE2(SE2),.SG0(SG0),.SG1(SG1),.SG2(SG2),.SG3(SG3),.SG4(SG4),
.SG5(SG5),.SG6(SG6),.SG7(SG7),.SG8(SG8),.SG9(SG9),.SG10(SG10),
.SG11(SG11),.SG12(SG12),.SG13(SG13),.SG14(SG14),.SG15(SG15),
.SG16(SG16),.SG17(SG17),.SG18(SG18),.SG19(SG19),.SG20(SG20),
.SG21(SG21));
```

```
initial
  begin
    #0 RESET=1;  //to set seed and clear MISR
    #10 CLK=1;
    #10 CLK=~CLK;
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=0;
    SE1=0;
    SE2=1;
    #10 CLK=~CLK;
    #10 CLK=~CLK;
    #10 CLK=~CLK;
    for( i=0; i < 4096; i=i+1)
      begin
        #10 CLK=~CLK;
      end
    #10000 $stop; // Change data every 1 MHz
  end


initial
  begin
    $display("                  CCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDDDDDGHRSSSSSSSSSSSSSSSSSSSSSSSSS");
   $display("                  LO01234567891111111111122O01234567891111111111122LOECEEGGGGGGGGGGGGGGGGGGGGG");
   $display("                  KR          012345678901U          012345678901OLSA12012345678911111111111122");
   $display("                   _                    B                    BDEN            012345678901");
   $display("                   E                    L                    A T_                        ");
   $display("                   R                    E                    L  I                        ");
   $display("                   R                    _                    S  N                        ");
   $display("                   O                    E                    T                           ");
   $display("                   R                    R                    R                           ");
   $display("                                        R                    T                           ");
   $display("                                        O                                                ");
   $display("                                        R                                                ");

    $monitor($time,,  CLK, COR_ERROR, C0, C1,
            C2, C3, C4, C5, C6, C7, C8,
            C9, C10, C11, C12, C13, C14,
            C15, C16, C17, C18, C19, C20,
            C21, DOUBLE_ERROR, D0, D1,
            D2, D3, D4, D5, D6, D7, D8,
            D9, D10, D11, D12, D13, D14,
            D15, D16, D17, D18, D19, D20,
            D21, GLOBALSTRT, HOLD, RESET,
            SCAN_IN, SE1, SE2, SG0, SG1, SG2,
            SG3, SG4, SG5, SG6, SG7, SG8,
            SG9, SG10, SG11, SG12, SG13,
```

```
            SG14, SG15, SG16, SG17, SG18,
            SG19, SG20, SG21);

  end
initial
  $gr_waves("CLK",CLK,"COR_ERROR",COR_ERROR,"C0",C0,"C1",C1,
            "C2",C2,"C3",C3,"C4",C4,"C5",C5,"C6",C6,"C7",C7,"C8",C8,
            "C9",C9,"C10",C10,"C11",C11,"C12",C12,"C13",C13,"C14",C14,
            "C15",C15,"C16",C16,"C17",C17,"C18",C18,"C19",C19,"C20",C20,
            "C21",C21,"DOUBLE_ERROR",DOUBLE_ERROR,"D0",D0,"D1",D1,
            "D2",D2,"D3",D3,"D4",D4,"D5",D5,"D6",D6,"D7",D7,"D8",D8,
            "D9",D9,"D10",D10,"D11",D11,"D12",D12,"D13",D13,"D14",D14,
            "D15",D15,"D16",D16,"D17",D17,"D18",D18,"D19",D19,"D20",D20,
            "D21",D21,"GLOBALSTRT",GLOBALSTRT,"HOLD",HOLD,"RESET",RESET,
            "SCAN_IN",SCAN_IN,"SE1",SE1,"SE2",SE2,"SG0",SG0,"SG1",SG1,"SG2",SG2,
            "SG3",SG3,"SG4",SG4,"SG5",SG5,"SG6",SG6,"SG7",SG7,"SG8",SG8,
            "SG9",SG9,"SG10",SG10,"SG11",SG11,"SG12",SG12,"SG13",SG13,
            "SG14",SG14,"SG15",SG15,"SG16",SG16,"SG17",SG17,"SG18",SG18,
            "SG19",SG19,"SG20",SG20,"SG21",SG21);

endmodule
```

```
------------------------
//  MISR_CORE.STIM
------------------------



'timescale 1 ns/100 ps

module test;

reg
CLK,GLOBALSTRT,HOLD,RESET,SCAN_IN,SE2,S0,S1,S2,S3,S4,S5,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9
,Y10,Y11,Y12,Y13,Y14,Y15;
wire W0;
wire W1;
wire W2;
wire W3;
wire W4;
wire W5;
wire W6;
wire W7;
wire W8;
wire W9;
wire W10;
wire W11;
wire W12;
wire W13;
wire W14;
wire W15;
wire W16;
wire W17;
wire W18;
wire W19;
wire W20;
wire W21;
integer i;

supply1 XVDD;
supply0 XGND;

MISR_COREF t1
(.CLK(CLK),.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),
.RESET(RESET),.SCAN_IN(SCAN_IN),.SE2(SE2),.S0(S0),.S1(S1),.S2(S2),
.S3(S3),.S4(S4),.S5(S5),.W0(W0),.W1(W1),.W2(W2),.W3(W3),
.W4(W4),.W5(W5),.W6(W6),.W7(W7),.W8(W8),.W9(W9),.W10(W10),
.W11(W11),.W12(W12),.W13(W13),.W14(W14),.W15(W15),.W16(W16),
.W17(W17),.W18(W18),.W19(W19),.W20(W20),.W21(W21),.Y0(Y0),
.Y1(Y1),.Y2(Y2),.Y3(Y3),.Y4(Y4),.Y5(Y5),.Y6(Y6),.Y7(Y7),
.Y8(Y8),.Y9(Y9),.Y10(Y10),.Y11(Y11),.Y12(Y12),.Y13(Y13),
.Y14(Y14),.Y15(Y15));
```

```
initial
  begin
    #0 CLK=0;
    RESET=1;
    #20 CLK=~CLK;

     // add to try final signature
    #20 CLK=~CLK;
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=0;
    SE2=1;    // select feedback
    Y0=0;
    Y1=0;
    Y2=0;
    Y3=0;
    Y4=1;
    Y5=0;
    Y6=1;
    Y7=1;
    Y8=1;
    Y9=0;
    Y10=0;
    Y11=1;
    Y12=0;
    Y13=1;
    Y14=1;
    Y15=1;
    S0=1;
    S1=0;
    S2=0;
    S3=0;
    S4=0;
    S5=0;
    #20 CLK=~CLK;
    #20 CLK=~CLK;

    #20 CLK=~CLK;
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=0;
    SE2=1;    // select feedback
    Y0=1;
    Y1=1;
    Y2=1;
    Y3=1;
    Y4=1;
```

```
Y5=1;
Y6=1;
Y7=1;
Y8=1;
Y9=1;
Y10=1;
Y11=1;
Y12=1;
Y13=1;
Y14=1;
Y15=1;
S0=1;
S1=1;
S2=1;
S3=1;
S4=1;
S5=1;

#10 CLK=~CLK;
GLOBALSTRT=0;
HOLD=0;
RESET=0;
SCAN_IN=1;
SE2=1;    // select feedback
Y0=0;
Y1=0;
Y2=0;
Y3=0;
Y4=0;
Y5=0;
Y6=0;
Y7=0;
Y8=0;
Y9=0;
Y10=0;
Y11=0;
Y12=0;
Y13=0;
Y14=0;
Y15=0;
S0=0;
S1=0;
S2=0;
S3=0;
S4=0;
S5=0;

#10 CLK=~CLK;
GLOBALSTRT=0;
HOLD=0;
```

```
RESET=0;
SCAN_IN=0;
SE2=1;    // select feedback
Y0=1;
Y1=1;
Y2=1;
Y3=1;
Y4=1;
Y5=1;
Y6=1;
Y7=1;
Y8=1;
Y9=1;
Y10=1;
Y11=1;
Y12=1;
Y13=1;
Y14=1;
Y15=1;
S0=1;
S1=1;
S2=1;
S3=1;
S4=1;
S5=1;

for(i=0; i<6; i=i+1)
  begin
   #10 CLK=~CLK;
  end

GLOBALSTRT=0;
HOLD=0;
RESET=0;
SCAN_IN=1;
SE2=1;    // select feedback
Y0=0;
Y1=0;
Y2=0;
Y3=0;
Y4=0;
Y5=0;
Y6=0;
Y7=0;
Y8=0;
Y9=0;
Y10=0;
Y11=0;
Y12=0;
Y13=0;
```

```
Y14=0;
Y15=0;
S0=0;
S1=0;
S2=0;
S3=0;
S4=0;
S5=0;

for(i=0; i<6; i=i+1)
  begin
   #10 CLK=~CLK;
  end

#10 CLK=~CLK;
GLOBALSTRT=0;
HOLD=0;
RESET=0;
SCAN_IN=0;
SE2=0;    // select scan-in
Y0=1;
Y1=1;
Y2=1;
Y3=1;
Y4=1;
Y5=1;
Y6=1;
Y7=1;
Y8=1;
Y9=1;
Y10=1;
Y11=1;
Y12=1;
Y13=1;
Y14=1;
Y15=1;
S0=1;
S1=1;
S2=1;
S3=1;
S4=1;
S5=1;


for(i=0; i<35; i=i+1)
  begin
   #10 CLK=~CLK;
  end

#10 CLK=~CLK;
```

```
    GLOBALSTRT=0;
    HOLD=0;
    RESET=0;
    SCAN_IN=1;
    SE2=0;    // select scan-in
   Y0=0;
    Y1=0;
    Y2=0;
    Y3=0;
    Y4=0;
    Y5=0;
    Y6=0;
    Y7=0;
    Y8=0;
    Y9=0;
    Y10=0;
    Y11=0;
    Y12=0;
    Y13=0;
    Y14=0;
    Y15=0;
    S0=0;
    S1=0;
    S2=0;
    S3=0;
    S4=0;
    S5=0;

    for(i=0; i< 45; i=i+1)
      begin
       #10 CLK=~CLK;
      end


#10000 $stop; // Change data every 1 MHz
  end

initial
  begin
 $display("                    CGHRSSYYYYYYYYYYYYYYYYSSSSSSWWWWWWWWWWWWWWWWWWWWWW");
 $display("                    LLOECE012345678911111110123450123456789111111111122");
 $display("                    KOLSA2          012345                012345678901");
 $display("                     BDEN                                            ");
 $display("                     A T_                                            ");
 $display("                     L  I                                            ");
 $display("                     S  N                                            ");
 $display("                     T                                               ");
 $display("                     R                                               ");
 $display("                     T                                               ");
 $monitor($time,,  CLK, GLOBALSTRT, HOLD,
```

```
            RESET, SCAN_IN, SE2,
            Y0, Y1, Y2, Y3, Y4, Y5, Y6,
            Y7, Y8, Y9, Y10, Y11, Y12,
            Y13, Y14, Y15, S0, S1,
            S2, S3, S4, S5, W0, W1, W2,
            W3, W4, W5, W6, W7, W8, W9,
            W10, W11, W12, W13, W14, W15,
            W16, W17, W18, W19, W20, W21);


  end
initial
  $gr_waves("CLK",CLK,"GLOBALSTRT",GLOBALSTRT,"HOLD",HOLD,
            "RESET",RESET,"SCAN_IN",SCAN_IN,"SE2",SE2,
        "Y0",Y0,"Y1",Y1,"Y2",Y2,"Y3",Y3,"Y4",Y4,"Y5",Y5,"Y6",Y6,
        "Y7",Y7,"Y8",Y8,"Y9",Y9,"Y10",Y10,"Y11",Y11,"Y12",Y12,
        "Y13",Y13,"Y14",Y14,"Y15",Y15,
        "S0",S0,"S1",S1, "S2",S2,"S3",S3,"S4",S4,"S5",S5,
        "W0",W0,"W1",W1,"W2",W2,
            "W3",W3,"W4",W4,"W5",W5,"W6",W6,"W7",W7,"W8",W8,"W9",W9,
            "W10",W10,"W11",W11,"W12",W12,"W13",W13,"W14",W14,"W15",W15,
            "W16",W16,"W17",W17,"W18",W18,"W19",W19,"W20",W20,"W21",W21);

endmodule
```

```
----------------------
//  MISR.STIM
----------------------


'timescale 1 ns/100 ps

module test;

reg
CLK,END,GLOBALSTRT,HOLD,RESET,SCAN_IN,SE2,S0,S1,S2,S3,S4,S5,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y
8,Y9,Y10,Y11,Y12,Y13,Y14,Y15;
wire ECC_FAILURE;
wire SCAN_OUT;

supply1 XVDD;
supply0 XGND;

MISRF t1
(.CLK(CLK),.ECC_FAILURE(ECC_FAILURE),.END(END),
.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),.RESET(RESET),.SCAN_IN(SCAN_IN),
.SCAN_OUT(SCAN_OUT),.SE2(SE2),.S0(S0),.S1(S1),.S2(S2),.S3(S3),.S4(S4),
.S5(S5),.XGND(XGND),.XVDD(XVDD),.Y0(Y0),.Y1(Y1),.Y2(Y2),
.Y3(Y3),.Y4(Y4),.Y5(Y5),.Y6(Y6),.Y7(Y7),.Y8(Y8),.Y9(Y9),
.Y10(Y10),.Y11(Y11),.Y12(Y12),.Y13(Y13),.Y14(Y14),.Y15(Y15));

initial
  begin
    #0 CLK=1;
    END=0;
    GLOBALSTRT=0;
    HOLD=0;
    RESET=1;
    SCAN_IN=0;
    SE2=0;

    #10 CLK=~CLK;    //select multi-inputs
    RESET=0;
    SE2=1;
    Y0=0;            //set register output as correct signature
    Y1=0;
    Y2=0;
    Y3=0;
    Y4=1;
    Y5=0;
    Y6=1;
    Y7=1;
    Y8=1;
    Y9=0;
    Y10=0;
    Y11=1;
```

```
  Y12=0;
  Y13=1;
  Y14=1;
  Y15=1;
  S0=1;
  S1=0;
  S2=0;
  S3=0;
  S4=0;
  S5=0;
  END=1;          //to compare the signature
  #10 CLK=~CLK;
  #10 CLK=~CLK;
  END=0;
  #10 CLK=~CLK;   //read the flag ( that should be 0 )
  #10 CLK=~CLK;


  #10 CLK=~CLK;  //flip all input to be 0, and continue generating signatures
  #10 CLK=~CLK;
  Y0=0;
  Y1=0;
  Y2=0;
  Y3=0;
  Y4=0;
  Y5=0;
  Y6=0;
  Y7=0;
  Y8=0;
  Y9=0;
  Y10=0;
  Y11=0;
  Y12=0;
  Y13=0;
  Y14=0;
  Y15=0;
  S0=0;
  S1=0;
  S2=0;
  S3=0;
  S4=0;
  S5=0;
  #10 CLK=~CLK;
  #10 CLK=~CLK;
  #10 CLK=~CLK;
  #10 CLK=~CLK;

  #10000 $stop; // Change data every 1 MHz
end
```

```
initial
  begin



    $display("                    GCREHSSYYYYYYYYYYYYYYYYSSSSSSSE");
    $display("                    LLENOEC0123456789111111012345CC");
    $display("                    OKSDL2A          012345      AC");
    $display("                    B E D N                      N_");
    $display("                    A T   _                      _F");
    $display("                    L     I                      OA");
    $display("                    S     N                      UI");
    $display("                    T                            TL");
    $display("                    R                             U");
    $display("                    T                             R");
    $display("                                                  E");
    $monitor($time,, GLOBALSTRT, CLK, RESET, END,
           HOLD, SE2, SCAN_IN,
           Y0, Y1, Y2, Y3, Y4, Y5,
           Y6, Y7, Y8, Y9, Y10, Y11,
           Y12, Y13, Y14, Y15, S0, S1, S2, S3, S4,
        S5, SCAN_OUT, ECC_FAILURE );
  end
initial
  $gr_waves("GLOBALSTRT",GLOBALSTRT,"CLK",CLK,"RESET",RESET,"END",END,
      "HOLD",HOLD,"SE2",SE2,"SCAN_IN",SCAN_IN,"Y0",Y0,"Y1",Y1,
"Y2",Y2,"Y3",Y3,"Y4",Y4,"Y5",Y5, "Y6",Y6,"Y7",Y7,"Y8",Y8,
            "Y9",Y9,"Y10",Y10,"Y11",Y11,"Y12",Y12,"Y13",Y13,"Y14",Y14,
      "Y15",Y15, "S0",S0,"S1",S1,"S2",S2,"S3",S3,"S4",S4,"S5",S5,
           "SCAN_OUT",SCAN_OUT, "ECC_FAILURE",ECC_FAILURE);

endmodule
```

```
-------------------------
// TEST_CTRLLER.STIM
-------------------------


'timescale 1 ns/100 ps

module test;

reg CLK,END,GLOBALSTRT,RESET,SELF_TEST,TEST_MODE0,TEST_MODE1;
wire HOLD;
wire SET_RESET;
wire SE0;
wire SE1;
wire SE2;
integer i;

supply1 XVDD;
supply0 XGND;

TEST_CONTROLLERF t1
(.CLK(CLK),.END(END),.GLOBALSTRT(GLOBALSTRT),.HOLD(HOLD),
.RESET(RESET),.SELF_TEST(SELF_TEST),.SET_RESET(SET_RESET),.SE0(SE0),
.SE1(SE1),.SE2(SE2),.TEST_MODE0(TEST_MODE0),
.TEST_MODE1(TEST_MODE1),.XGND(XGND),.XVDD(XVDD));

initial
  begin
    #1 CLK=1;
    #5 CLK=~CLK;        // On-Demand-Test mode
    END=0;
    GLOBALSTRT=0;
    RESET=1;
    SELF_TEST=1;
    TEST_MODE0=1;
    TEST_MODE1=1;
    #5 CLK=~CLK;

    #5 CLK=~CLK;       // exit from ODT mode
    END=0;
    GLOBALSTRT=0;
    RESET=1;
    SELF_TEST=0;
    TEST_MODE0=1;
    TEST_MODE1=1;
    #5 CLK=~CLK;

    #5 CLK=~CLK;       // Psudo-Concurrent Test mode
    END=0;
    GLOBALSTRT=0;
    RESET=1;
```

```
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=0;
#5 CLK=~CLK;

#5 CLK=~CLK;       // exit from PCT mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=0;
TEST_MODE0=1;
TEST_MODE1=0;
#5 CLK=~CLK;

#5 CLK=~CLK;        // Scan-Test mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=0;
TEST_MODE1=1;
#5 CLK=~CLK;

#5 CLK=~CLK;       // exit from ST mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=0;
TEST_MODE0=0;
TEST_MODE1=1;
#5 CLK=~CLK;

#5 CLK=~CLK;       // Scan-Path ring
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=0;
TEST_MODE1=0;
#5 CLK=~CLK;

#5 CLK=~CLK;       // Reset
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=0;
TEST_MODE0=0;
TEST_MODE1=0;
#5 CLK=~CLK;
```

```
#5 CLK=~CLK;        // On-Demand-Test mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=1;
#5 CLK=~CLK;

#5 CLK=~CLK;        // ODT Test Cycle End
END=1;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=1;
#5 CLK=~CLK;

#5 CLK=~CLK;        // On-Demand-Test mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=1;
#5 CLK=~CLK;

#5 CLK=~CLK;       // Psudo-Concurrent Test mode
END=0;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=0;
#5 CLK=~CLK;

#5 CLK=~CLK;       // PCT Test Cycle  End
END=1;
GLOBALSTRT=0;
RESET=1;
SELF_TEST=1;
TEST_MODE0=1;
TEST_MODE1=0;
#5 CLK=~CLK;

#5 CLK=~CLK;       // Psudo-Concurrent Test mode
END=0;
GLOBALSTRT=0;
RESET=1;
```

```
    SELF_TEST=1;
    TEST_MODE0=1;
    TEST_MODE1=0;
    #5 CLK=~CLK;


    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #5 CLK=~CLK;
    #10000 $stop; // Change data every 1 MHz
  end

  initial
  begin
    $display("                    GCSTTERSSSHS");
    $display("                    LLEEENEEEEOE");
    $display("                    OkLSSDS012LT");
    $display("                    B FTT E   D_");
    $display("                    A ___ T    R");
    $display("                    L TMM      E");
    $display("                    S EOO      S");
    $display("                    T SDD      E");
    $display("                    R TEE      T");
    $display("                    T  01       ");
    $monitor($time,, GLOBALSTRT, CLK, SELF_TEST, TEST_MODE0, TEST_MODE1,
        END, RESET, SE0, SE1, SE2, HOLD, SET_RESET );
  end

initial
  $gr_waves(
        "GLOBALSTRT",GLOBALSTRT, "CLK",CLK,"SELF_TEST",SELF_TEST,
        "TEST_MODE0",TEST_MODE0,"TEST_MODE1", TEST_MODE1, "END",END,
        "RESET", RESET, "SE0", SE0, "SE1", SE1, "SE2",SE2, "HOLD",HOLD,
        "SET_RESET",SET_RESET );

endmodule
```

```
-----------------------
//  ECCT.STIM
-----------------------


'timescale 1 ns/100 ps

module test;

reg CLOCK,ECC_TEST,GLOBALSTRT,RESET,SCAN_IN,T_MODE0,T_MODE1;
wire COR_ERROR;
wire DOUBLE_ERROR;
wire ECC_FAILURE;
wire SCAN_OUT;
integer i;

supply1 XVDD;
supply0 XGND;

ECC_TF t1
(.CLOCK(CLOCK),.COR_ERROR(COR_ERROR),
.DOUBLE_ERROR(DOUBLE_ERROR),.ECC_FAILURE(ECC_FAILURE),.ECC_TEST(ECC_TEST),
.GLOBALSTRT(GLOBALSTRT),.RESET(RESET),.SCAN_IN(SCAN_IN),.SCAN_OUT(SCAN_OUT),
.T_MODE0(T_MODE0),.T_MODE1(T_MODE1),.XGND(XGND),.XVDD(XVDD));

initial
  begin
    #0 RESET=0;             // reset the whole system
    #35 CLOCK=1;
    #35 CLOCK=~CLOCK;        // On-Demand-Test mode
    GLOBALSTRT=0;
    RESET=1;
    SCAN_IN =1;
    ECC_TEST=1;
    T_MODE0=1;
    T_MODE1=1;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    for(i=0; i < 4096; i=i+1)
      #35 CLOCK=~CLOCK;

    #35 CLOCK=~CLOCK;        // exit from ODT mode
    GLOBALSTRT=0;
    RESET=1;
    SCAN_IN =1;
    ECC_TEST=0;
    T_MODE0=1;
    T_MODE1=1;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
```

```
#35 CLOCK=~CLOCK;
for(i=0; i < 4096; i=i+1)
  #35 CLOCK=~CLOCK;

#35 CLOCK=~CLOCK;       // Psudo-Concurrent Test mode
GLOBALSTRT=0;
RESET=1;
SCAN_IN =1;
ECC_TEST=1;
T_MODE0=1;
T_MODE1=0;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
for(i=0; i < 4096; i=i+1)
  #35 CLOCK=~CLOCK;

#35 CLOCK=~CLOCK;       // exit from PCT mode
GLOBALSTRT=0;
RESET=1;
SCAN_IN =1;
ECC_TEST=0;
T_MODE0=1;
T_MODE1=0;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
for(i=0; i < 4096; i=i+1)
  #35 CLOCK=~CLOCK;

#35 CLOCK=~CLOCK;       // Scan-Test mode
GLOBALSTRT=0;
RESET=1;
SCAN_IN =1;
ECC_TEST=1;
T_MODE0=0;
T_MODE1=1;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
#35 CLOCK=~CLOCK;
for(i=0; i < 4096; i=i+1)
  #35 CLOCK=~CLOCK;

#35 CLOCK=~CLOCK;       // exit from ST mode
GLOBALSTRT=0;
RESET=1;
SCAN_IN =1;
ECC_TEST=0;
T_MODE0=0;
```

```
    T_MODE1=1;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    for(i=0; i < 4096; i=i+1)
      #35 CLOCK=~CLOCK;

    #35 CLOCK=~CLOCK;        // Scan-Path ring
    GLOBALSTRT=0;
    RESET=1;
    SCAN_IN =1;
    ECC_TEST=1;
    T_MODE0=0;
    T_MODE1=0;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    for(i=0; i < 4096; i=i+1)
      #35 CLOCK=~CLOCK;

    #35 CLOCK=~CLOCK;        // Global Reset
    GLOBALSTRT=0;
    RESET=1;
    SCAN_IN =1;
    ECC_TEST=0;
    T_MODE0=0;
    T_MODE1=0;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    #35 CLOCK=~CLOCK;
    for(i=0; i < 4096; i=i+1)
      #35 CLOCK=~CLOCK;
  end

initial
  begin
    $display("                    CRGETTSSCDE");
    $display("                    LELC__CCOOC");
    $display("                    OSOCMMAARUC");
    $display("                    CEB_OONN_B_");
    $display("                    KTATDD__ELF");
    $display("                      LEEEIOREA");
    $display("                      SS01NUR_I");
    $display("                      TT   TOEL");
    $display("                      R     RRU");
    $display("                      T      RR");
    $display("                             OE");
    $display("                             R ");
    $monitor($time,,  CLOCK, RESET,GLOBALSTRT,
```

```
        ECC_TEST, T_MODE0, T_MODE1, SCAN_IN,
        SCAN_OUT, COR_ERROR, DOUBLE_ERROR, ECC_FAILURE);
  end


initial
  $gr_waves("CLOCK",CLOCK,"RESET",RESET,"GLOBALSTRT",GLOBALSTRT,
        "ECC_TEST",ECC_TEST,"T_MODE0",T_MODE0,"T_MODE1",T_MODE1,
        "SCAN_IN",SCAN_IN,
        "SCAN_OUT",SCAN_OUT,"COR_ERROR",COR_ERROR,
        "DOUBLE_ERROR",DOUBLE_ERROR,"ECC_FAILURE",ECC_FAILURE);

endmodule
```

APPENDIX B:

SIMULATION OUTPUT (.log Files)

(from Verilog–XL simulator)

The following .log files represent the design simulation output for the six (6) modular designs in the hierarchy of the on–chip BIT ECC design. The .log files were produced and logged by the Verilog–XL simulator during the simulation process, given the design netlist files (.v file) and the stimulus files (.stim file).

i) **ecc.log** is for the ECC_CORE unit.
ii) **prtg.log** is for the on–chip test generator PRTG unit.
iii) **mux.log** is for the ECC input multiplexer unit.
iv) **misr_core.log**, **sig_sim.log** and **misr.log** are for the on–chip signature analyzer MISR unit.
v) **t_ctrller.log** is for the built–in test controller unit.
vi) **ecc_t.log** is for the top–level design of the on–chip BIT ECC.

It is noted that these six (6) .log files will take about 500 pages of papers if printed. As such they are stored electronically in a floppy disk. The .log files as well as the thesis write–up are also FTP accessible through the internet network.