A HIGH-LEVEL LANGUAGE AND CAD ENVIRONMENT FOR BIST EMBEDDING

by
RODRIGUE BYRNE
B.Sc(Hons), B. Eng, Memorial University of Newfoundland, 1984
M.Sc, University of Victoria, 1988

A Dissertation Submitted In Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. D.M. Miller, Supervisor (Department of Computer Science)

Dr. J. Ellis, Departmental Member (Department of Computer Science)

Dr. J.M. Myzio, Departmental Member (Department of Computer Science)

Dr. N.J. Dimopoulos, Outside Member (Department of Electrical & Computer Engineering)

Dr. M. Soma, External Examiner (University of Washington, U.S.A.)

Name Rodrigue Byrne

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.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

UMI

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

Supervisor: Dr. D. M. Miller

Abstract

The reliable construction of VLSI integrated circuits (ICs) requires that the ICs be tested after fabrication. An alternative to performing external testing is to create ICs that can test themselves with a built-in self-test (BIST) mode. Unfortunately the problem of embedding a self-test operating mode to the functional design is difficult for two reasons. 1) The creation of test sets that effectively test digital circuits requires the solution of several intractable problems. 2) The hardware resources dedicated to self-test are usually constrained.

Modifications to the Logic III hardware description language and a new computer-aided design (CAD) tool, **lg3**, are presented in this dissertation as an environment that allows BIST embedding to be created and evaluated. The major premise behind this work is that BIST design can be treated in a similar fashion as functional design, and that the designer can address the constraints of a BIST mode at the same time as the functional constraints.

The modified language, called Logic III(UVic), allows BIST embeddings to be specified by an *embedding* module which describes how the circuit's memory elements are realized. This dissertation presents a library of embedding modules that realize several of the most common BIST architectures.

Case studies using this environment are presented for an ALU, CORDIC, GCD, and string matching circuits. A BIST mode with almost 100% single stuck-at fault coverage is realized for each circuit. This shows that the CAD environment can be used to create self-testing circuits.

In addition to aiding users in embedding BIST functionality, the **lg3** tool can be used to evaluate specific BIST architectures. Properties of BIST test pattern generators are presented that are used in analyzing the effectiveness of the generators for delay-fault testing. A novel approach based on creating a deterministic finite automaton that recognizes the fault-free responses is presented.

Examiners:

Dr. D.M. Miller, Supervisor (Department of Computer Science)

Dr. J. Ellis, Departmental Member (Department of Computer Science)

Dr. J.M. Muzio, Departmental Member (Department of Computer Science)

Dr. N.J. Dimopoulos, Outside Member (Department of Electrical & Computer Engineering)

Dr. M. Soma, External Examiner (University of Washington, U.S.A.)

Contents

List of Figures

List of Tables

Acknowledgment

I wish to thank my supervisor, Dr. D.M. Miller, for his help and guidance during the course of this work.

The work on the analysis of transition properties for linear finite state machines was a cooperative effort with Shujian Zhang. I would like to thank Bill Gardner for the use of his string matching circuit.

I acknowledge the financial support from a University of Victoria Fellowship and I wish to thank the Department of Computer Science for the use of its facilities.

Chapter 1

Introduction

Built-In Self-Test Design

1.1 The Problem and the Approach

Testing is required to ensure that a digital system has been fabricated such that it performs its intended function. Testing is performed by applying inputs to the system and observing the system's response. Designing digital systems that can test themselves is the topic addressed in this dissertation. Systems that can test themselves are said to contain a built-in self-test (BIST) operating mode. The problem of including a BIST mode in a digital system is called the BIST embedding problem. A hardware design language (HDL) based on Logic III HDL, and a CAD tool that supports the design and evaluation BIST embeddings are described.

One approach to testing consists of completely exercising the circuit's functional behaviour. Unfortunately, this method requires time that is exponential in the number of inputs and state variables. A more practical testing method checks for the presence of physical defects that are modeled as faults. Faults model the physical defects of a system as effects on the behaviour of primitive components in a digital system. Thus this testing

approach is based on looking for ways in which the manufacturing process can fail. The assumption made in this approach is that if no faults are found, then the circuit is functionally correct. In this dissertation, the testing approach is based on finding faults.

The approach taken to *embed* BIST functionality into a circuit is to augment the Logic III hardware description language with features that allow the easy specification of BIST architectures and to provide a computer aided design (CAD) tool, **lg3**, that evaluates the BIST embedding's test effectiveness. This approach is based on the following premises:

1. The diverse and stringent design requirements for some designs can only be productively satisfied by allowing the designers to implement the BIST embedding directly. In other words, the same design constraints that require the implementation of functional designs by a designer also apply to BIST designs.

2. Test effectiveness is measured by the number of faults the BIST mode detects. For many BIST embeddings, fault simulation provides a practical way of determining the number of faults detected. Indeed, for some BIST architectures, fault simulation is the only general way of determining the specific faults covered.

When designing a circuit, a designer is faced with many constraints, such as: time-to-market, speed, power, area, testability, interfacing constraints, *etc.* These constraints and combinations of these constraints make it difficult to use only one architecture to realize some function. For example, architectures for the ubiquitous adder come in many flavors: ripple-carry, carry-look-ahead, carry-save, *etc.* Each architecture fills a particular niche in satisfying the requirements of the design. In a similar fashion, many of the BIST architectures fit in different niches, where each architecture satisfies some combinations of constraints.

The purpose of functional design is to create a circuit that complies with some specified input/output relation. Functional simulation of the circuit is used to verify that the actual input/output relation matches the expected input/output relation. On the other hand, the

purpose of BIST design is to create a circuit that performs an operation to determine if the circuit contains any faults from a given fault set. Since fault simulation computes the effect of these faults, and a BIST mode is designed to detect these faults, the BIST mode's detection ability can be validated during the fault simulation.

This dissertation uses the single stuck-at fault model[1]. A stuck-at fault occurs when an interconnecting signal in a circuit is stuck at a logical 1 or 0 value. The *single* stuck-at model [SK90, page 4] assumes that only *one* interconnection is stuck. The percentage of faults detected over total faults (i.e., the fault coverage) measures the success of a BIST embedding.

In summary, the major premise of this work is that the design of BIST functionality is similar to the design of any functionality, and the augmentation of a HDL to include BIST specifying features elevates the BIST embedding problem to the same level as other design problems. Fault simulation is used to determine the BIST mode's ability to detect the modeled faults.

1.2 Outline

BIST architectures [ABF90, page 458] can be classified depending on when the test occurs. Testing that occurs as part of the normal operation of the circuit is called *on-line* testing. On-line testing can be further divided into *concurrent* or *nonconcurrent* testing, where concurrent testing occurs simultaneously with normal operation, and nonconcurrent testing occurs during idle periods. If the testing occurs as a separate operational mode, then it is called *off-line* testing. This dissertation is primarily concerned with the design of off-line BIST architectures.

A review and classification of off-line BIST architectures is presented in Chapter 2. Off-line BIST architectures are designed to detect the modeled faults by generating test stimuli

[1] Extending the fault simulator of lg3 to handle other fault models is discussed in Section 4.3.1.

and observing the circuit's response. Test stimulus generation and response analysis are the major components of off-line BIST architectures.

A stimulus generator's purpose in a BIST embedding is to apply test patterns to the circuit that allow the effect of the faults to be seen by the response analysis structure. In the case where the stimulus generator is incorporated into the functional circuit, the purpose is still to make the fault effects visible to the response analysis structure. A fault that is made visible is *exposed*. The design goal of stimulus generation in BIST is to *expose* all the modeled faults.

The stimulus generators can be categorized according to the approach used to expose the fault effects to the response analysis structure. The four categories are: exhaustive, pseudoexhaustive, random, and deterministic. The exhaustive approach applies all possible input patterns to the circuit. The pseudoexhaustive approach partitions the circuit into subcircuits that can be exhaustively stimulated. These two approaches guarantee that all the faults that can be detected by single test patterns are detected. Test sequences purposely developed to expose a given fault set are used to design the deterministic stimulus sources. In contrast, the random stimulus source is based on circuits that are easy to build and that generate pseudorandom patterns. Thus these patterns are not designed to detect a particular fault set.

The purpose of the response analysis component of the off-line BIST architectures is to capture errors, specifically the errors caused by the modeled faults in the circuit. Since this component is built into the design, the response analysis structure is usually restricted to a design that *characterizes* the circuit's response. A circuit's behaviour is characterized by attaching the circuit's observation points to the inputs of the response analysis structure. In general, the characterization process cannot uniquely identify all test responses. Aliasing occurs when the characterization process yields the same result for a good circuit and for a circuit with errors. If the response analysis structure is modeled as a finite state machine[2]

[2]Of course, since it is implemented as part of the circuit it can be modeled as a FSM.

(FSM), then after the test stimuli are applied the characterization is based on the last state of the FSM. Aliasing occurs when the final state for an error response and a good response is identical. If the errors in the output response are modeled, then the effectiveness of a BIST response analysis component can be measured by how it partitions the space of error responses. The partitions can be used to determine the probability of aliasing (i.e., the number of error responses that have the same characterization as the fault-free response divided by the total number of error responses).

The method used to compute the characterization can be used to categorize the different response analysis components. These methods are: ones counting, transition counting, parity, syndrome (a type of ones counting), polynomial division, the state of a Multi-Input Shift Register (MISR), and the state of an arbitrary Finite State Machine (FSM).

An important point to note is that the response analysis component can only decrease the test effectiveness of the BIST mode. This can be expressed as: the number of faults detected equals the number of faults exposed multiplied by one minus the aliasing probability.

The approaches reported in the literature for adding BIST functionality are reviewed in Chapter 2. The digital circuit design process can be considered as consisting of four separate phases. These phases are: requirement, behaviour, register-transfer level, and gate level. Note that the design process is iterative. The addition of a BIST mode is considered at all stages.

BIST embedding is considered at the requirement level as a set of constraints that the design must satisfy. For example, restricting the design to synchronous logic is the major constraint assumed by many BIST architectures. This restriction is also assumed in this dissertation. Other restrictions made to improve a circuit's testability are: no redundant logic, no dynamic logic, no wired logic and the memory elements of the system must be initializable.

The behavioural level also considers BIST embedding as a set of constraints. These constraints affect how the required behaviour is realized at the register-transfer level. BIST

embedding at the register-transfer level is mainly concerned with configuring the registers to act as pattern generators and response analyzers. Extra BIST structures may also be added. The control of the BIST mode is also addressed at this level.

BIST embedding at the gate level is very similar to the register-transfer level, except that instead of configuring registers as BIST structures, individual memory elements are grouped and configured to act as BIST structures.

In the top two levels, no structural description of the design exists, thus the fault models (based on the physical defects) used to measure the test effectiveness cannot be applied. Both the register-transfer level and the gate level represent the design with a structural description. The register-transfer level partitions the design into registers and operational blocks. The gate level represents the design as a collection of gates wired together. The stuck-at fault model can be applied to these interconnections.

In determining the test effectiveness of a BIST embedding, the closer the design description is to the description used for manufacturing, the closer the modeled faults are to the physical defects. Thus embedding BIST at the gate level means the gate level fault simulation results used to measure test effectiveness are the most realistic.

Chapter 3 describes the modifications and additions to the HDL, Logic III, that support BIST embedding. The augmented language is called Logic III(UVic). The Logic III language is chosen for three reasons. First, since the language describes hardware at the gate and register-transfer level, fault simulation could be used to evaluate the test effectiveness of a BIST embedding. Specifically, each variable in Logic III(UVic) represents an interconnection of structural components. Thus the stuck-at fault model is represented as variables stuck-at 0 or 1. Another advantage of using a language at the gate and register-transfer level is that most BIST architectures are described at this level. The third reason is more pragmatic; it is easier to experiment with language features if the initial language is simple. For example, since Logic III is a simpler language than the VHSIC[3] Hardware Description

[3]VHSIC stands for Very High Speed Integrated Circuits.

Language (VHDL), there are fewer details for the implementor to be concerned about.

The modifications to Logic III are based on two premises:

1. Since off-line BIST architectures are based on stimulus and response analysis structures, these architectures can be realized by modifying the memory elements used by the functional design.

2. Since functional design focuses on realizing the design by decomposing the required function into simpler functions that can be implemented in hardware, and BIST embedding design focuses on constructing an operating mode that tests all the components, the language constructs used to realize the functional and BIST design should be separate. Another way to express this difference is that functional design is concerned with individual components that realize a subfunction, and BIST design is concerned with an operating mode of all the components. Also, separating the test circuitry specification from the functional circuitry specification allows these two traditionally separate activities to remain separate.

Using these two premises Logic III is modified by introducing a new data type, called **reg**, that represents a 1-bit memory element, and by providing a method of specifying what circuitry is used to realize the memory element. Thus in Logic III(UVic) the only allowable way to introduce memory elements into the design is with variables of this new data type. An additional advantage of this restriction is the elimination of most of the design-for-test rules, since most of these rules deal with how memory elements cannot be configured (e.g., the clock inputs for a memory element cannot be connected to any logic). A circuit module, called an *embedding* module, with an interface consisting of an array of inputs and an array of outputs, specifies how the variables of type **reg** are implemented. It is important to note that there are no other restrictions on the embedding module. Thus the user has the full power of the language to describe how the memory elements are configured.

This dissertation shows how these two features are sufficient to specify many of the

most useful BIST architectures. Discussion on how to implement the embedding modules for scan, STUMPS, BEST, BILBO, and CSTP BIST architectures is presented at the end of Chapter 3. These embedding modules form a library of BIST architectures that enables a `lg3` user to easily embed one or a combination of the above BIST architectures.

The other additions to Logic III discussed in Chapter 3 increase the expressive power of the language. These additions are useful for both BIST and functional design. The additions are:

- functions,
- recursive functions and modules,
- flexible arrays,
- array properties of `first`, `last`, and `size`,
- array slices, and
- collections of scalars and arrays.

Since CAD tools are ultimately implemented as software systems, the design and implementation of the CAD systems are valid concerns. Object-oriented design and programming is the latest popular paradigm in the construction of software systems, and several researchers [Wol91, GCG+89, Che87] suggest that CAD systems can be effectively designed and implemented using this approach. Chapter 4 discusses the implementation of the `lg3` CAD tool using object-oriented design and the object-oriented programming language C++. The chapter identifies the major classes and objects used in the design. The experience of using this approach is also discussed.

To demonstrate the ability of Logic III(UVic) and the `lg3` CAD tool in embedding BIST, four case studies are discussed in Chapter 5. In the first case, BIST circuitry is added to a portion of the TMS32010 Data Path consisting of an arithmetic logic unit (ALU) and a

multiplier. The second case study uses a circuit that implements the CORDIC algorithm. The third case study is based on a circuit that realizes Euclid's Greatest Common Divisor (GCD) algorithm. The fourth and final case study evaluates the testability of a string matching circuit.

Chapter 6 discusses an analysis of a stimulus source for delay faults and introduces a novel BIST response analysis structure. The delay faults considered require two test patterns, and the analysis introduces a measure to indicate the effectiveness of BIST generators based on Linear Finite State Machines (LFSM). Since delay faults require two test patterns, the number of unique pairs of test patterns (transitions) produced by the generator is an important metric in determining the generator's effectiveness.

The metric introduced in Chapter 6 counts the number of unique transitions for a subset of the generator's outputs. These subsets are called windows. The number of transitions for the windows of internal linear feedback shift registers (LFSRs), external LFSRs, and linear hybrid cellular automata (LHCAs) are derived. A permutation of the generator outputs, called the cross-over permutation, is shown to produce the maximum number of transitions.

A strategy is discussed for capturing errors in the response of a test stimulation based on creating a deterministic finite automaton (DFA) that accepts a language that includes the correct response. An analysis is presented of the theoretical aliasing probabilities as well as experiments using the lg3 CAD tool to evaluate the actual aliasing using the ISCAS85 [BF85] benchmark circuits.

The following languages are used as the basis of the DFAs:

1. The difference between the number of ones and the number of zeros in a string in the language should not exceed a fixed lower or upper bound.
2. The number of adjacent ones/zeros is bounded by a maximum value.
3. The number of adjacent ones and adjacent zeros is bounded by a maximum value.

Each of the above languages can be recognized by a counter and simple controller.

The conclusions of the work presented in this dissertation are found in the last chapter. This dissertation is based on the premises that designers should deal with the BIST embedding problem directly and that fault simulation can be used to create and optimize BIST embeddings. Hopefully, the goal of this dissertation to show that Logi. III(UVic), the BIST embedding library and the lg3 CAD tool provide an environment that enables designers to create BIST embeddings will be successful.

Chapter 2

Built-In Self-Test Review

2.1 Off-line BIST Architectures

Incorporating off-line BIST into a circuit, C, provides an operational mode that, when enabled, results in a self-test. The purpose of the self-test is to determine if C contains any faults from a particular fault set. If no faults are found, then the circuit is deemed to be functioning correctly.

A fault f is *exposed*, if for at least one s, $\text{obs}(C, s) \neq \text{obs}(C_f, s)$, where $\text{obs}(C, s)$ is the response from the observation points for the fault-free C with stimulus s and $\text{obs}(C_f, s)$ is the response from the observation points for C containing f with stimulus s. The stimuli are the input patterns to C. The two main objectives in the design of an off-line BIST mode are:

- exposing every fault from the given fault set, and
- capturing any fault effect, so that the presence of the fault is detected.

The success of exposing faults is measured by the achieved fault coverage, FC. FC is the ratio of exposed faults to total faults.

In off-line BIST architectures, the first objective is the accomplished by the stimulus structure. Given a circuit C, and fault set F, the stimulus source must provide at least one stimulus s, such that

$$\text{obs}(C, s) \neq \text{obs}(C_f, s)$$

for each $f \in F$. The major design difficulty is to create a stimulus structure that is as simple as possible and yet can expose all the faults. Since the test length depends on the number of stimuli applied, another design goal is to minimize the number of stimuli applied.

The second objective is the responsibility of the response analysis structure. Since the size of the stimulus set and thus the output response can be quite large, the response analysis structure usually compacts the response. Given an output response sequence, R, for a k output circuit, the output response analyzer (ORA) computes a *characterization*,

$$\text{ORA}(R)$$

of this sequence. The sequence, R, consists of k-tuples, r_i, where k is the number of observation points. Since information is lost by the compaction process, it is possible for the fault effects to also be lost. Aliasing occurs when the faulty and fault-free characterizations are equivalent. That is

$$\text{ORA}(R) = \text{ORA}(R_f),$$

where R_f is the response sequence for the faulty circuit. A fault is *detected* when $\text{ORA}(R) \neq \text{ORA}(R_f)$. The success of capturing the exposed faults is measured by the degree of aliasing, FD. FD is the ratio of detected faults to exposed faults. The major design problem for response analysis is to create a response analysis structure that is as simple as possible and yet can capture all of the fault effects. An example ORA function for a single output circuit is:

$$\sum_{i=1}^{N} r_i.$$

where N is the number of elements in the sequence and r_i is either 1 or 0.

Figure 2.1 shows the BIST architecture where th non-BIST mode circuitry, the *kernel*, and the BIST mode circuitry are realized by separate components. Although this is the simplest architecture in terms of the interaction of the kernel and the testing circuitry, other architectures are possible. Figure 2.2 shows other architectures where the stimulus and response analysis functions are merged with the kernel (i.e., the same components are used to implement self-test functions and normal functions). If the circuitry that is already fulfilling some functional role can be used for a testing purpose as well, then a savings in the amount of BIST specific circuitry can be realized.

Figure 2.1 Separated BIST Architecture

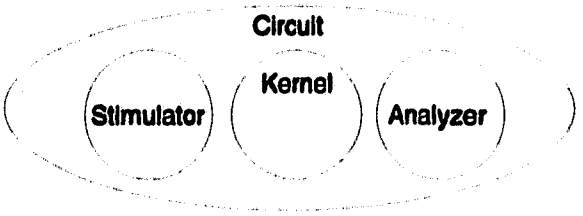
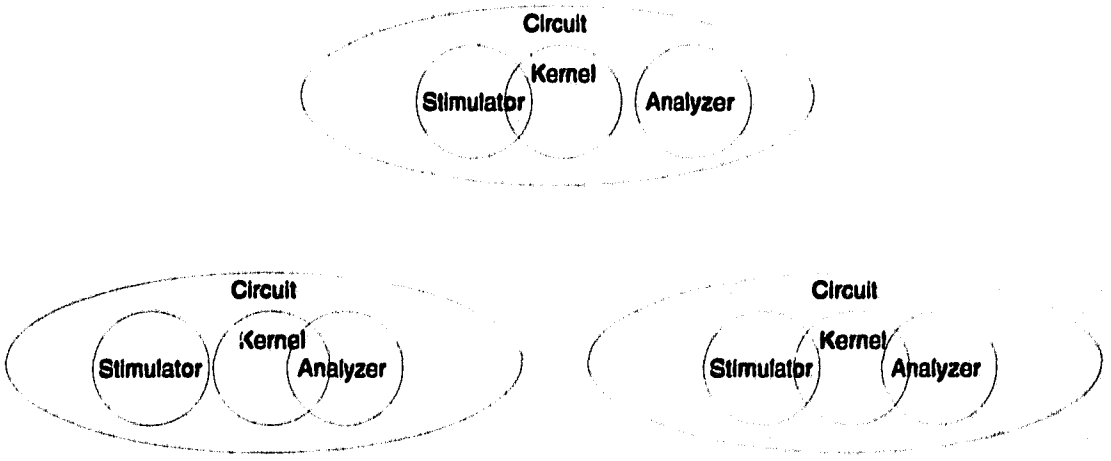


Figure 2.2 Alternative BIST Architecture



Circuits containing BIST functionality are usually restricted to circuits based on syn-

chronous logic. Some of the reasons [ABF90, Tur90, RS89, FNH89] for avoiding asynchronous logic are:

- the design difficulty of asynchronous logic because of hazards and races;
- the computational intractability of breaking feedback paths [GJ78] by converting an asynchronous design into a combinational one as required by some BIST architectures (e.g., scan paths [ELWW91]);
- the common stimulus sources do not follow the fundamental mode assumption (i.e., only one input is changed at a time) required by certain asynchronous logics;
- asynchronous logic often requires redundant (untestabie) logic; and
- zero-delay simulation algorithms cannot be used, and therefore more computationally expensive algorithms are required to simulate the behaviour of asynchronous logic.

The circuit's memory elements must be settable to a known state to guarantee a repeatable test sequence. Initialization of memory elements can be accomplished by either an initialization sequence applied to the logic containing the memory elements (i.e., by an internal controller), or by adding a set/reset feature to the individual memory elements.

The requirement to clock memory elements from a common signal is also a restriction of several BIST architectures. This means that no logic can be placed between the common clocking signal(s) and the clock input(s) of a memory element. Several BIST architectures need to modify the clocking signals and the restriction of a common clock simplifies these modifications.

2.2 Stimulus Structures

In off-line BIST architectures, the goal of the stimulus structure is to generate stimuli that expose the faults in the selected fault set. Four approaches to stimulus generation design are the following:

exhaustive, all possible input stimuli are applied to the kernel. Since the amount of time spent on testing is limited, the number of circuit inputs that can be exhaustively stimulated is usually limited to $\lfloor \log_2(\text{maximum testing cycles}) \rfloor$. The maximum number of inputs is N_{ex}.

pseudoexhaustive, where the circuit can be partitioned into subcircuits, each dependent on a subset of the inputs, and thus can be stimulated exhaustively. The circuit is tested by testing the partitions.

random, a stimulus source with a fixed sequence is used to excite the kernel. The major advantage of this approach is the low cost of the stimulus structures. Its major disadvantage is the difficulty of designing a stimulus structure to produce all the stimuli required to expose the faults in the given fault set. The stimulus structure can be modeled by a finite state machine. Thus the designer is usually restricted to choosing a starting state and the number of test cycles. Fault simulation is required to determine which faults are exposed.

deterministic, the stimulus source is required to generate a specific set of stimuli. Analysis of the kernel determines the stimulus set. The structures used for the deterministic approach are usually based on random structures that have been extended to generate specific stimuli, in addition to their pseudorandom patterns.

These approaches can be distinguished by the way each one analyzes the circuit to be tested. The exhaustive approach depends on the number of primary inputs, n, of the circuit to be tested. If $n \leq N_{ex}$, then the exhaustive approach can be used. The pseudoexhaustive approach requires structural and logical analysis of the circuit to group the inputs. The number of inputs in each group must be less than or equal to N_{ex}. In the random approach, a stimulus source that only applies a subset of all stimuli is used. Fault simulation is necessary to determine the test effectiveness, therefore the circuit must be analyzed at the structural/logical level. Structural analysis is also used to determine the type of stimulus

Table 2.1 Stimulus Design Approaches

Approach	Exhaustive	Pseudoexhaustive	Random	Deterministic
Kernel Type	n-input CL	CL where the inputs can be grouped	CL, SL	CL, SL
Kernel Analysis	none	structural analysis to assign groupings	structural analysis for some methods	structural and ATPG
Stimulus Structure	separate	separate	separate, merged	separate, merged
Test Effectiveness	100% for combinational faults	100% for combinational faults	requires fault simulation	may require fault simulation
Test Length	2^n	$2^w \leq T \leq T_u$ [a]	depends on fault simulation results	depends on method

[a] see Section 2.2.2.

structure. Deterministic testing requires an analysis of the circuit to determine what test stimuli are required.

Design approach considerations are shown in Table 2.1. The type of circuitry that each approach works with is given in the Kernel Type row. The circuitry types are combinational logic, CL, and sequential logic, SL. Any other restrictions are also indicated. The type of circuit analysis required, if any, is found in Kernel Analysis. Whether the stimulus structure is merged or separate from the kernel is indicated in the Stimulus Structure row. The Test Effectiveness row gives the expected fault coverage for each approach. Although the amount of design effort is not easily quantified, the amount of circuit analysis, the complexity of the stimulus structures and response analysis structures can be used as an indication of the design effort. Finally, the test stimuli set size for each approach is found under Test Length. The following subsections discuss each approach in more detail.

2.2.1 Exhaustive Stimulation

If the kernel is a combinational logic block with n inputs, then an exhaustive stimulus source must provide all 2^n input patterns. For a sequential circuit with k feedback lines and n inputs, at least $2^n \cdot 2^k$ stimulus cycles must be applied. More cycles are required if the feedback lines cannot be directly controlled.

All the faults that do not convert a combinational circuit into a sequential circuit are covered when a combinational logic block is exhaustively stimulated. Thus a major advantage of exhaustive stimulation is the guaranteed best possible purely combinational fault coverage. Unfortunately, the number of stimulus patterns grows exponentially with the number of inputs, n, and becomes uneconomical when $n > N_{ex}$. N_{ex} inputs require $2^{N_{ex}}$ input stimuli, and therefore an exhaustive test needs $2^{N_{ex}}$ testing cycles. The value of N_{ex} depends on the particular manufacturer, a usual value is $N_{ex} = 22$ [ABF90, page 460]. The analysis cost for exhaustive testing of CL is minimal, since only the number of primary inputs for the kernel need be considered.

The difficulty of designing a circuit to exhaustively stimulate sequential circuits, usually limits exhaustive stimulation to testing of CL. Sequential circuits that possess memory elements but no feedback paths can be modeled as CL. Thus exhaustive stimulation can be used [WH92]. Most exhaustive stimulus structures are not merged with a kernel. Typically exhaustive stimulus sources are linear feedback shift registers (LFSRs), complete LFSRs, linear hybrid cellular automata (LHCA[1]), and binary counters. These specific structures are discussed in Section 2.2.5.

2.2.2 Pseudoexhaustive

When the number of inputs is greater than N_{ex}, then an exhaustive stimulation approach is not feasible. Exhaustive stimulation can still be used if the circuit can be partitioned into

[1] Depending on the context the A in LHCA stands for either automata or automaton

subcircuits that depend only on a subset of the inputs. These input subsets, or groupings, can then be exhaustively stimulated. Grouping of the inputs requires analysis of the circuit structure, thus pseudoexhaustive stimulation needs more design effort than exhaustive stimulation.

Grouping the input set is based on two approaches to partitioning the kernel:

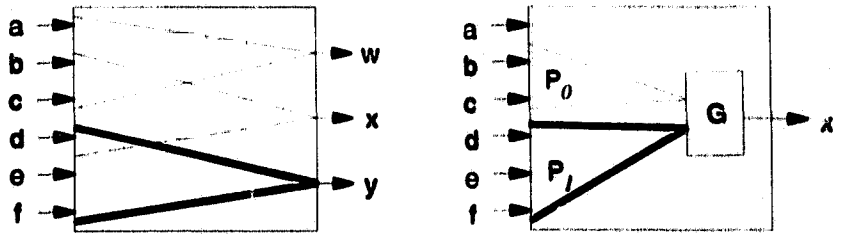
cone segmenting The group of inputs that control each output of a multiple output CL are identified. All the logic driven by the group of inputs that affects an output is called a cone. If the size of each input group is less than N_{ex}, then a generator can be designed to exhaustively stimulate each grouping. Figure 2.3(a) shows a combinational logic block with three outputs and six inputs. The cone for each output is drawn. Output w depends only on inputs a, b, and c. Therefore, w's logic cone can be exhaustively tested by applying all 2^3 input patterns to the inputs a, b, and c.

sensitized path segmenting The circuit is partitioned into cones of internal nodes, path from each node to an output is created and the cone is tested by exhaustively stimulating its inputs. The circuit is partitioned into a cover set of these cones, and by testing the cones the entire circuit is tested. Figure 2.3(b) shows two internal cones, P_0 and P_1. These cones drive a gate, G, connected to the output x. P_0 can be tested exhaustively, if the output of the P_1 cone is set to a value that allows the output of P_0 to propagate through G. P_1 can be tested in turn, by setting P_0's output to a value that allows P_1's output to propagate to x. This approach is not usually applicable to BIST since the stimulus source can be quite complicated.

For cone segmenting, the kernel is assumed to be a CL block with n inputs and m outputs. Let X_i correspond to an input group. M is the number of groups. The maximum time to test the kernel is:

$$T_u = \sum_{i=1}^{M} 2^{|X_i|}$$

Figure 2.3 Segmenting Approaches



(a) Cone Segmenting (b) Path Segmenting

cycles, where $|X_i|$ is the size of the X_i group and if each partition is tested in sequence. For some circuits, a stimulus source can be designed to test in:

$$2^w \text{ where } w = \max\{|X_1|, |X_2|, ..., |X_M|\}$$

cycles. Note that w must be less than or equal to n. The biggest grouping requires 2^w and the remaining subdivisions are tested concurrently. The number of testing cycles, T, for pseudoexhaustive testing is bounded by

$$2^w \leq T \leq T_u.$$

In cone segmenting, one possible strategy is to find the cone with the largest number of inputs, w, and to generate an exhaustive test set for each of the subsets of these inputs. In [TW83], Tang and Woo show that constant weight codes can be used as a stimuli set. Wang and McCluskey [WM86] present a stimulus source based on a LFSR that generates such a pseudoexhaustive test set. A generator based on a LFSR and shift register defined in [BCR83] can also be used.

2.2.3 Random

The design of the stimulus structure in the random approach is based on creating the cheapest structure and not on generating the specific stimulus required by the kernel. This is based on the observation that many faults are exposed by any set of *random* stimuli [BMS87, ELWW91, LBDG87, MZ91]. Unlike the exhaustive approach where by applying all possible stimuli, the fault coverage of combinational faults is guaranteed, the random approach requires analysis to determine its test effectiveness. The major advantage of this approach is that a stimulus structure can be chosen that is cheap to build. In practice, simple stimulus structures have been empirically shown to generate stimuli that are effective and the required test length is less than the exhaustive approach. Unfortunately, there exist circuits [BMS87, Chapter 7] where this approach to stimulus generation is not adequate (i.e., circuits with faults that are only exposed with a few stimuli).

The sequence produced by the stimulus structure can be considered as a set of *random* pattern. This allows statistical analysis to be used in determining test effectiveness and the test length [BMS87, ABF90]. The sequences should be more properly referred to as *pseudorandom*, since the same sequence can be reproduced. Although statistical analysis can be used to give estimates of test effectiveness and test length required, fault simulation is necessary to determine the exact fault coverage and test length for a particular circuit.

The memory elements used to contain the state for the test generator can be shared with the functional design, or be solely used for testing purposes. This stimulus structure design approach can be further subdivided by the roles the logic that implements the *next state* function performs. The next state logic can be used solely for testing, T, testing and normal, TN, and solely functional, N roles. The greater the use of functional logic, the cheaper the cost of the overall design.

Stimulus sources based on separate or functional memory elements and testing next state logic allow the designer the most control over the testing stimuli. LFSMs (Linear Finite State Machines) are the most commonly used stimulus structure because of the low

hardware cost. Section 2.2.5 describes the LFSMs in greater detail. LFSRs and LHCA are the two types of LFSMs used. Many BIST architectures have been proposed using LFSMs as stimulus sources. In [KMZ80], an architecture called BILBO (Build-In Logic Block Observation) is described where the memory elements are grouped into registers that have a stimulus generation mode. In this mode the register is turned into a LFSR.

When LFSRs, or any other LFSMs, are used as stimulus sources, the designer must choose the next state function F, the initial state I, and the number of states N to cycle through. Let the stimuli set required for a particular circuit to expose all of its faults be denoted by S. Let the stimuli generated by the LFSM be denoted by G. The problem of choosing a particular F, I, and N such that $S \subseteq G$ and $|G|$ is minimal is intractable for several reasons:

- The generation of the a single $s \in S$ is equivalent to the satisfiability problem [Fuj85, page 113], and is therefore known to be NP-complete.
- There is an exponential number of initial states and feedback networks.

Except by a random selection of F, I and C and using fault simulation to determine the fault coverage, no method has been reported to allow a designer to find a selection that results in a minimal $|G|$. As mentioned previously, in practice, LFSMs have been empirically shown to be effective. Obviously, minimizing the next state function logic, and the number of test cycles are BIST embedding design goals.

For some circuits, the number of patterns required to achieve a reasonable fault coverage can be excessive. The probability of an individual output of a maximum cycle LFSMs generating a 1 is 0.5. In [SLC75], the number of patterns required to achieve a reasonable fault coverage is reduced if the probability of an individual output generating a 1 is biased towards 1.0 or 0.0. These biased patterns are called *weighted random patterns.* The weight is the probability that an output produces a 1. Procedures for determining what weights to use for a circuit are given in [ELWW91, chapter 14] and [LBDG87]. Circuits with fanouts

can require more than one weight distributions. [Wun88] shows how to compute multiple weight distributions. Descriptions of hardware implementations of weighted random pattern generators are found in [BCK89], [Wun87] and [ELWW91, chapter 14].

The HILDO BIST architecture presented in [BM84] suggests using functional circuitry and EXOR gates to compute the next state function for the stimulus source. This architecture is shown to be effective for several designs. Merging the test and functional logic (TN) reduces the required hardware, at the cost of decreasing the designer's control. The main problem with this approach is the possibility of creating a next state function with a small cycle length and thus poor test effectiveness.

Figure 2.4(a) and Figure 2.4(b) show the two configurations of the TN stimulus structure. The register blocks can contain memory elements, or they can contain testing logic that is part of the next state function. In 2.4(a), the logic that is used to compute the next state function, is also the kernel under test. Proposed BIST architectures that follow the 2.4(a) configuration are found in [KP89, BAV90, Rud90, Win89]. In 2.4(b), a separate kernel is being tested. Examples of 2.4(b) can be found in [BM84, KHT88].

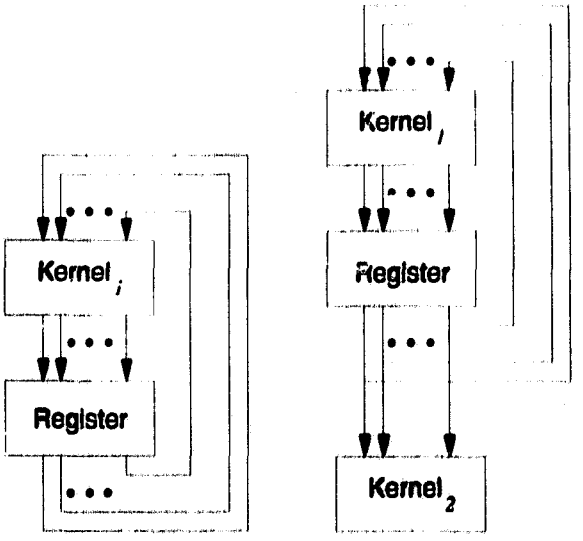
The last type of stimulus sources are those in which some part of the functional circuitry can be used as a stimulus source directly. Binary counters are an example.

2.2.4 Deterministic

Complete fault coverage can be guaranteed if the stimulus source generates the stimuli required to exposed the faults in the kernel. The design problem is then, given a stimuli set, create a stimulus structure to generate that set. An automatic test pattern generation program [ABF90, Chapter 6] can be used to find the stimuli set. Two design goals for this stimulus structure are:

- the structure should be as small and as simple as possible, and

Figure 2.4 Configurations of the TN stimulus structure



(a) Merged (b) Separate

- the size of the stimuli generated should be as close as possible to the size of the minimum stimuli set.

Unfortunately, these two goals conflict. To avoid storing all the necessary stimuli requires a state machine that generates a stimulus sequence. Creating a simple next state function that only produces the necessary stimulus is difficult.

The above two design goals lead to two types of stimulus structures:

1. A structure that **stores** all of the stimuli (e.g., a ROM).
2. A structure that **stores** some state machine configuration information and **generates** stimuli that includes the necessary stimuli.

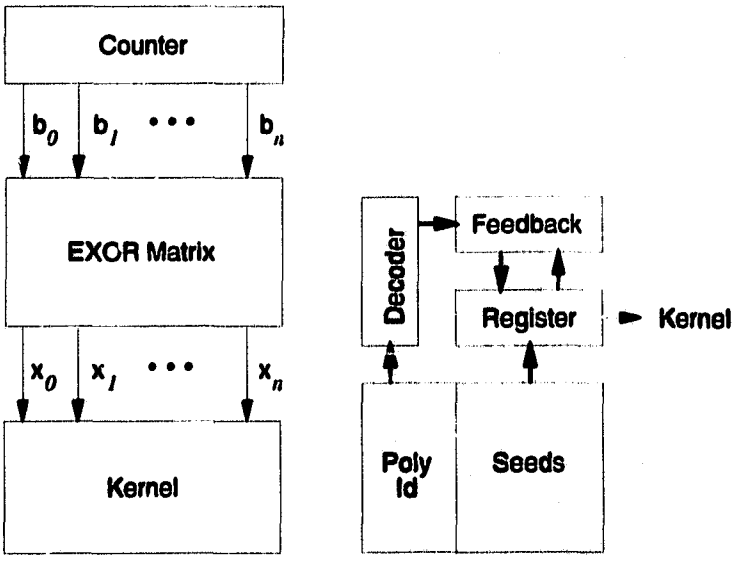
Storing all of the test stimuli on a single chip is usually impractical because of the quantity. If the BIST architecture is for board level test, then a ROM external to the kernel under test can be used.

The store and generate architecture is introduced in [AC81]. In [AJ89], a test set is embedded into the structure shown in Figure 2.5(a). An EXOR matrix maps a binary counter's outputs into the test stimuli. Configuring the type and initial state for an LFSR is suggested in [HTRC92]. Figure 2.5(b) shows a diagram of their structure. Another architecture based on configuring CAs is discussed in [vCD92].

2.2.5 Counters

The counters discussed in this section are used as stimulus sources in many BIST architecture. Descriptions, characterizations, construction details and cost for the LFSM counters are presented. Although other types of counters can be used, they require more logic and thus cost more. If the functional design contains a non-LFSM counter, then that counter can be used as the stimulus source.

Figure 2.5 Specific Architectures for Deterministic Stimulus Structures



(a) EXOR Matrix (b) LFSR

External LFSRs, internal LFSRs, and LHCA are the most common types of LFSMs used in BIST architectures. LFSMs are constructed with memory elements and EXOR gates.

An external LFSR structure is shown in Figure 2.6. The circles labeled with c_i are scalar multipliers. Assignment of 0 or 1 to the entries in the vector $(c_1, c_2, \cdots, c_{n-2}, c_{n-1})$ describes a particular LFSR.

Figure 2.7 shows the structure of an internal LFSR. The internal LFSR is also customized by assigning 0 or 1 to the entries of the vector $(c_1, c_2, \cdots, c_{n-2}, c_{n-1})$.

The structure of a LHCA is depicted in Figure 2.8. For a LHCA, each memory element is updated by a 90-rule or a 150-rule. The assignment of $r_i = 1$, creates a 150-rule, otherwise the memory element is updated by a 90-rule.

LFSMs can also be described by a transition matrix T. The next state, s^+ can be computed from the current state, s by:

$$s^+ = s \cdot T$$

with matrix multiplication over GF(2). For the LFSRs, note that the c_i entries in the transition matrix correspond to the customization vector entries.

Figure 2.6 External LFSR

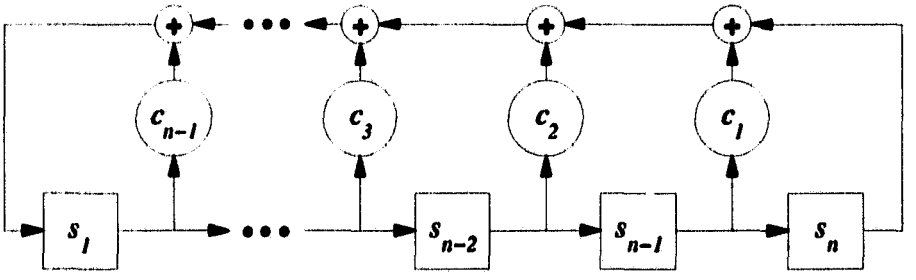


Figure 2.7 Internal LFSR

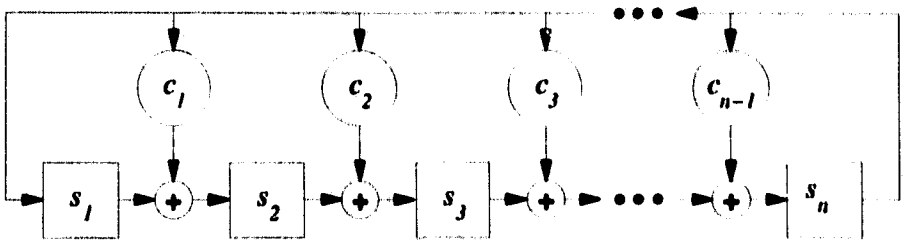
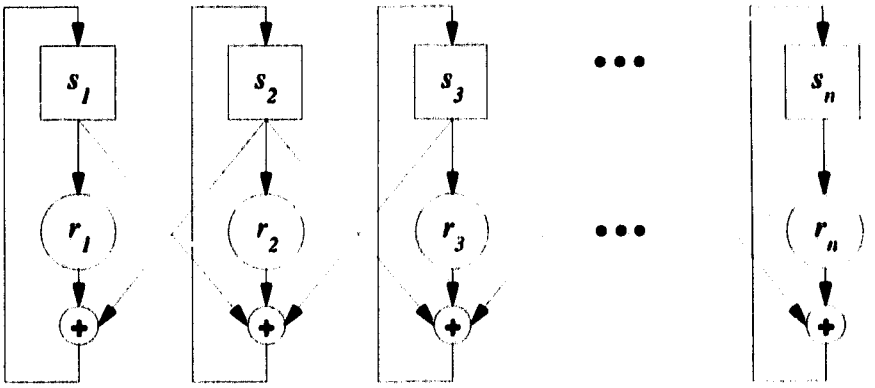


Figure 2.8 LHCA



An external LFSR transition matrix has the following form:

$$T = \begin{bmatrix} c_{n-1} & 1 & 0 & \cdots & 0 \\ c_{n-2} & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & \\ c_1 & 0 & 0 & & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

An internal LFSR transition matrix form is:

$$T = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & 0 & & 1 \\ 1 & c_1 & c_2 & \cdots & c_{n-1} \end{bmatrix}$$

The LHCA matrix is:

$$T = \begin{bmatrix} r_1 & 1 & 0 & \cdots & 0 \\ 1 & r_2 & 1 & & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & & 1 & r_{n-1} & 1 \\ 0 & \cdots & 0 & 1 & r_n \end{bmatrix}$$

When LFSMs are used as stimulus sources, the generation of all output patterns is an important property. An n-cell LFSM that generates all patterns, excepting the all zeros pattern, generates $2^n - 1$ patterns. If a linear machine is placed in the all zeros state, it remains in the all zeros state since $\mathbf{0} \cdot T = \mathbf{0}$. LFSRs can be modified to generate the all zeros state [BMS87, page 147]. The modified machine is a de Bruijn counter.

The *characteristic polynomial* of the matrix, T, is the polynomial $P(\lambda) = \det(\lambda I - T)$. The characteristic polynomial for the internal and external LFSR is

$$P(x) = x^n + r_{n-1}x^{n-1} + \cdots + r_2x^2 + r_1x^1 + 1,$$

where r_i is an entry in the customization vector and $\lambda = x$. The characteristic polynomial for the n-cell LHCA [SSMM90] in Figure 2.8 is given by:

$$\begin{aligned} P_n(x) &= (x + c_n)P_{n-1}(x) + P_{n-2}(x) \\ P_1(x) &= x + c_1 \\ P_0(x) &= 1. \end{aligned}$$

If the characteristic polynomial is *primitive* [BMS87, page 75], then an n-cell LFSM has maximum cycle $(2^n - 1)$ length. The number of EXOR gates required to implement the LFSM can be used as a cost measure. A table of minimum cost primitive polynomials for polynomials of degrees 1 to 300 is found in [BMS87, page 336]. A table of the first 150 minimum cost LHCA is found in [ZMM91]. The cost is measured by the number of coefficients.

Table 2.2 Minimum Cost LFSRs and LHCA to $n = 16$

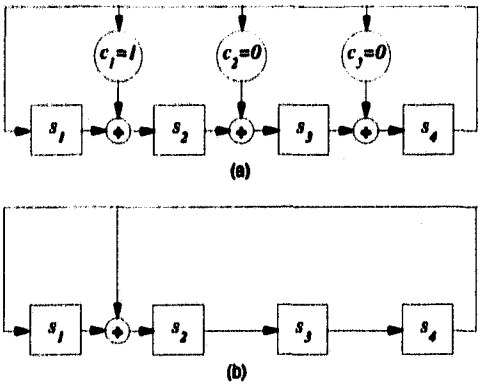
n	LFSR Polynomial Coefficients	LHCA 150-rule Positions
1		1
2	1	1
3	1	1
4	1	1 3
5	2	1
6	1	1
7	1	3
8	6 5 1	2 3
9	4	1
10	3	2 7
11	2	1
12	4 3	3 7
13	4 3 1	5
14	12 11 1	1
15	1	3
16	5 3 2	1 15

Table 2.2 lists some of the minimum cost LFSRs and LHCA with primitive polynomials for n up to 16. The LFSR column lists the coefficients, c_i, where $c_i = 1$. Since $c_n = 1$ and $c_0 = 1$, these nonzero coefficients are not listed. For $n = 4$, there is only one coefficient, $c_1 = 1$, thus the polynomial is

$$x^4 + x + 1.$$

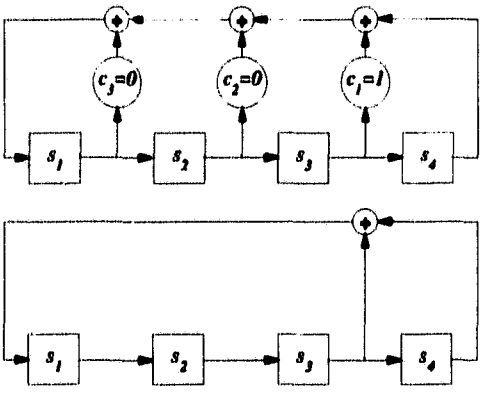
The structural realization of this polynomial for an internal LFSR is shown in Figure 2.9. Note that $(c_1, c_2, c_3) = (1, 0, 0)$. Figure 2.9(a) shows the coefficient assignment and Figure 2.9(b) shows a simplified version.

Figure 2.9 Realization of $x^4 + x + 1$ Internal LFSR



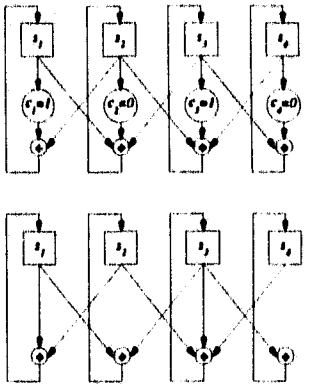
The external LFSR realization of $x^4 + x + 1$ is shown in Figure 2.10. If k is the number of nonzero coefficients, then $k - 2$ two-input EXOR gates can be used to implement the feedback EXOR function.

Figure 2.10 Realization of $x^4 + x + 1$ External LFSR



From Table 2.2 the minimum cost 4-cell LHCA has 150-rules in positions 1 and 3. A diagram of this LHCA is shown in Figure 2.11.

Figure 2.11 Realization of 4-cell LHCA



The number of 2-input **EXOR** gates as a function of the customization vector for the each type of LFSM is given in Table 2.3.

Table 2.3 LFSM Cost

LFSM Type	Cost
n-cell Internal LFSR	wt [a](customization vector)
n-cell External LFSR	wt (customization vector)
n-cell LHCA	n + wt (customization vector) - 2

[a] wt weight, number of 1's

2.3 Response Analysis Structures

The response analysis structure implements the ORA function. This function characterizes the circuit response to the test stimuli. Chapter 10 of [ABF90] and chapter 4 of [BMS87] discuss the most common approaches to response analysis. The design goals of the response analysis structure are:

- the structure is as simple as possible, and
- aliasing is minimized.

Table 2.4 summarizes the common approaches reported in [ABF90, Chapter 10] and [BMS87, Chapter 4] to designing a response analysis structure. The parameter N is the test length. R is the response sequence. $P(M|r)$ is the probability of masking (aliasing) given the parameter r. r is the count of either the number of ones or the number of transitions. $P(M)$ is the probability of masking. The ORA function, the aliasing probability and the typical hardware components for each approach are given in the table. The ORA function describes how the characterization is computed.

2.3.1 Ones Counting

Ones counting compacts the response of a single output circuit by counting the number of ones present in the output sequence, R. The response analysis structure can be implemented by an m-bit binary counter. If the number of ones expected for the fault-free circuit is r, then $m = \lceil \log_2 r \rceil$, thus an m-bit counter is big enough to record the maximum count. The counter can be designed to remain at $2^m - 1$ to prevent aliasing by wrap around. No restriction is placed on the circuit's stimulus in this approach. Circuit simulation or a known good circuit can be used to determine the correct ones counting. Merging the outputs of a multiple output circuits with a shift register can be used to apply this technique to multioutput circuits. Also each output can be assigned a weight, and the sum of the weighted outputs computed. This approach is discussed in [Los78, Par76, Hay76].

2.3.2 Transition Counting

The transition counting approach [Par76, Hay76] compacts the circuit response by counting the number of $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions in the output sequence of a single output kernel. Figure 2.12 shows the structure of a transition counter using a memory element, an EXOR

Table 2.4 Output Response Analysis Approaches

Approach	Function[a]	Aliasing Probability	Structure
Ones Counting	$r = 1C(R) = \sum_{i=1}^{N} r_i$	$P(M\|r) = \frac{\binom{N}{r}-1}{2^N-1}$	counter
Transition Counting	$r = TC(R) = \sum_{i=1}^{N-1} r_i \oplus r_{i+1}$	$P(M\|r) = \frac{2\binom{N-1}{r}-1}{2^N-1}$	counter
Syndrome [b]	$S(R) = \sum_{i=1}^{2^k} r_i / 2^k$	0 for syndrome testable circuits	counter
Parity	$P(R) = \left(\sum_{i=1}^{N} r_i\right) \bmod 2$	$P(M) = \frac{(2^N/2)-1}{2^N-1} \simeq 1/2$	an EXOR and a ME
Signature Analyzer [c]	$SA(R) = \bigoplus_{i=0}^{N-1} r_{i+1}x^i \bmod G(x)$	$P(M) = \frac{2^{N-k}-1}{2^N-1} \simeq 2^{-k}$	LFSR
MISR [d e]	$\vec{s}_i = \vec{s}_{i-1} \cdot T \oplus \vec{r}_i,\ \vec{s}_0 = \vec{0}$	2^{-k}	k-bit MISR
AFSM [f]	$\vec{s}_i = N_{st}(\vec{s}_{i-1}, r_i)$		Kernel

[a] i is the sequence's index, k is the number of inputs

[b] Only applicable to k-input CL and all 2^k patterns are applied.

[c] $G(x)$ is the characteristic polynomial of the LFSR.

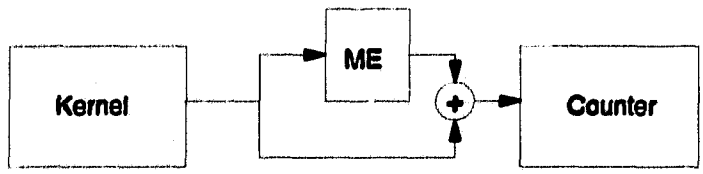
[d] The final state, $\vec{s}_n$ is the characterization. T is the transition matrix for the feedback logic.

[e] Handles a k output circuit.

[f] Handles multiple output circuits.

gate, and a binary counter. Transitions are detected by the memory element and the EXOR gate which compute $r_i \oplus r_{i-1}$ and recorded by the counter. The counter can be designed to remain at its maximum value to prevent aliasing by wrap around. Unlike the ones counting approach, the order of the stimuli affects the transition count. Circuit simulation or using a known good circuit is necessary to determine the correct transition count. Transition counting for multiple output circuits can be done by merging the outputs into one sequence with a shift register.

Figure 2.12 Transition Counter Structure



2.3.3 Parity

Using the parity of the response sequence is one of the simplest approaches to compacting the response. An EXOR gate and a memory element can be used to compute the parity of a response. This technique can be extended to multiple output kernels by either EXORing all the outputs together and computing the parity of the combined outputs or computing a separate parity for each output. If the outputs are EXORed together, then the errors affecting an even number of outputs are lost.

Although this technique is very cheap, it also has the poorest aliasing probability of any of the other approaches. The probability approaches 1/2 as the sequence size increases. No restriction is placed on the stimulus source for this method. The order of the stimuli has no affect on the parity. Circuit simulation or using a known good circuit is required to compute the correct parity.

2.3.4 Syndrome

The syndrome approach [BMS87, page 102] uses ones counting to compact the circuits output response with the added constraint that all 2^k input patterns for a k input CL circuit are required. Syndrome compaction has two major advantages over ones counting:

- A circuit's syndrome can be calculated by simple analysis of the circuit structure. Thus circuit simulation is not required to determine the syndrome. This advantage is restricted to circuits that can be exhaustively stimulated.
- Some circuits are syndrome testable, in that the presence of any single stuck-at fault causes the circuit to have a different syndrome. This means that aliasing is not a problem for syndrome testable circuits. If a circuit is not syndrome testable, then it can be modified to be syndrome testable.

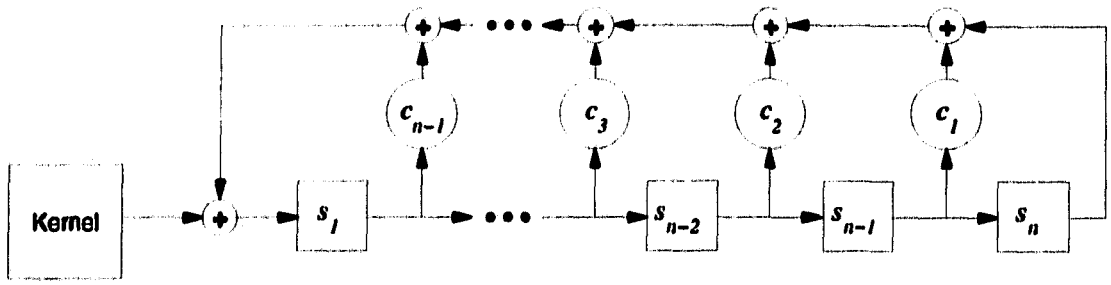
The same hardware structure for ones counting can be used for the syndrome approach. Multioutput circuits can be handled by computing the syndrome for each output. In [BSMS81], multioutput circuits are handled by computing a weighted sum of the syndromes. Unfortunately, unless the weights are chosen carefully, aliasing can occur.

2.3.5 Signature Analysis

Signature analysis [ABF90, BMS87, ELWW91], based on LFSRs, is the most popular response analysis technique. It also has one of the simplest hardware structures. Assuming an error model where all erroneous responses are equally likely, the aliasing probability for a k-bit signature analysis register is 2^{-k}. This result follows from the fact that remainder of the polynomial division partitions the response sequences into 2^k classes. This means that in theory the aliasing probability can be made arbitrarily small, irrespective of the response sequence.

Figure 2.13 shows the typical structure for a LFSR used as a signature analysis register. The selection of which feedback taps to use is still an open problem, but tap positions that correspond to primitive polynomials appear to give good results[ABF90, page 445].

Figure 2.13 Signature Analysis Register



Circuit simulation or a known good circuit can be used to determine the correct signature. This approach is extended to multioutput circuits (See the MISR section). If BIST design is based on the scan approach, then an LFSR can be used to compact the response of the scan chain.

2.3.6 Multiple-Input Shift Register (MISR)

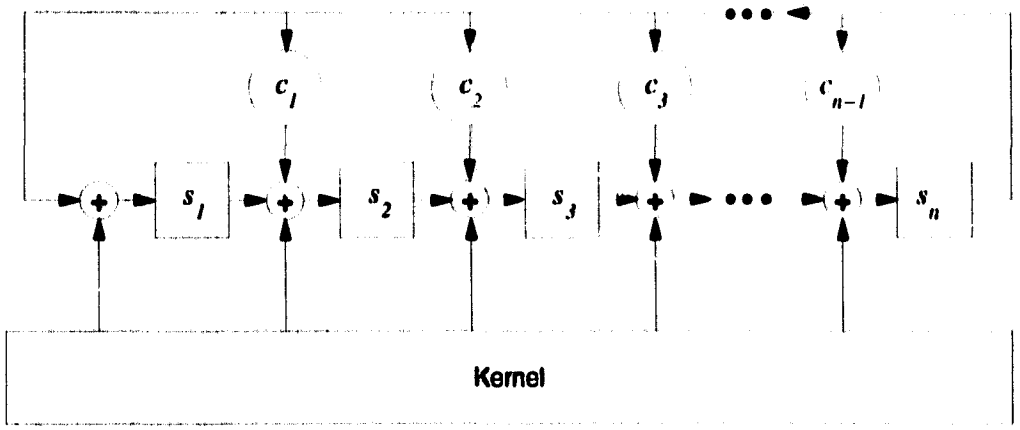
MISRs are the most common structure used to capture and compact the response of a multioutput circuit in parallel. Using different analysis techniques and assumptions the aliasing probability of a k-bit MISR has been shown to be 2^{-k} [DOF+89, DOR91, Hla92].

Figure 2.14 shows a MISR based on an internal LFSR. Circuit simulation or a known good circuit can be used to determine the correct signature.

2.3.7 Autonomous Finite State Machine (AFSM)

In all of the response analysis structures, a FSM driven by the observed circuit response is used to compact the response. If the kernel that is being tested is a FSM, then a possible

Figure 2.14 Multiple-Input Shift Register used for response compaction.



response observation strategy uses the final state of the FSM as the test characterization. The aliasing probabilities cannot in general be analyzed, thus fault simulation must be used to determine if the final state does capture all the exposed faults. This approach has the advantage of being very cheap (i.e., no extra hardware) but only applicable to a certain class of circuits. Accumulators seem to give good results[RT92].

2.4 Built-in Self-Test Embedding

2.4.1 Embedding Goals

The BIST embedding goals are:

- All of the faults in the fault set should be exposed and detected. The success of this goal can be measured by the test effectiveness (TE). TE is the the ratio(expressed as a percentage) of the number of detected faults to the total number of faults.
- The circuitry required to implement the BIST function should be minimal. The amount of BIST specific circuitry (BC) can be measured by the area it occupies, or the number of primitive gates it requires.

- The time required to perform the BIST mode should be minimal. The testing time, test length (TL), can be measured by the number of clock cycles required by the test.
- There should be no impact on the functional timing performance. The number of gate delays (GD) required by the BIST specific circuit in critical timing paths can be used to measure the effect on timing performance.
- The design effort (DE) required to add the BIST functionality should be minimal. The DE includes designer and computer time.

As usual, the design goals are not compatible and a BIST embedding approach performs tradeoffs between these goals. Although no direct relationships exist between the design goals, the following observations can be used to indicate how changing one goal can affect the other goals.

- As the TL increases, the number of exposed faults does not decrease. Assuming no aliasing in the response analysis structure, the TE does not decrease with increased TL. Thus increasing the TL might increase the TE.
- For a given circuit, if the circuit's observability and controllability is improved, then the TL can decrease. Thus if extra BIST specific circuitry (BC) is added to improve the observability and controlability, then the TL can decrease.
- Increasing the BC usually decreases the circuit's performance measured in GD, since the increased BC usually increases the delay through the circuit.
- The amount of DE depends on the methodology used to embed the BIST mode. In general, as the design effort increases, the remaining goals of test effectiveness, test length, required BIST circuitry, and timing performance should be minimized.

2.4.2 Embedding Levels

Digital systems are described at four levels:

requirement, the description of the capabilities of the system,

behavioural, the algorithmic description defining the input, output, and timing relationships,

register-transfer structural, the interconnection of registers and functional blocks.

gate structural, the gates and their interconnections.

The process of adding BIST functionality to a design is called BIST embedding. BIST functionality enables a design to stimulate itself and characterize the response. BIST embeddings can be considered at each digital design level. Since the type of description varies for each level, the type and method of embedding BIST also varies.

BIST performs structural testing. The purpose of the test is to detect faults from a particular fault model. The most common fault models are based on structure (e.g., single stuck-at). Since the requirement and behaviour levels do not deal with *structural* details, BIST embedding can only be considered as constraints on the design process. The architecture and technology decisions made at the requirement and behavioural levels, must enable the design to be structurally testable. The types of stimulus and output analysis structures and their control can be considered at the structural level. Obviously, all the structural information is available at the gate level, thus detail analysis of the fault coverage achieved by a particular BIST embedding can be performed.

A review of the BIST embedding approaches for each level is discussed in the following subsections.

2.4.2.1 Requirement

Natural languages or formal languages are used in the requirement description level to specify the capabilities of the system. For BIST, the description includes a statement that requires the system to be self-testing. At this level the structure of the system has not been decided, but BIST embedding can be considered by constraining how the system is built.

To enable self-test, the follow restrictions are usually adopted. These restrictions (rules) are general guidelines on what structures should be used or avoided to create testable designs. Although some of the guidelines can be ignored, they generally make the task of BIST embedding much easier. The following list of design-for-test (DFT) rules and their rationale is derived from [ABF90, BMS87, FNH89, Lal85, Tur90].

- Using synchronous logic avoids the design and testing difficulties of asynchronous logic. Some problems with asynchronous logic are the design difficulty of handling hazards and races, and the difficulty of simulating the effect of faults in asynchronous logic.
- All memory elements should be initializable to enable a repeatable test mode.
- The complicated fault models and difficulty of board level testing of wired logic, implies that this technology should be avoided.
- Since dynamic logic imposes timing restriction on test application, it should be avoided.
- No one-shots, since these circuits are only externally testable.
- Since redundant logic is not testable, avoid it.
- Separate analog and digital circuitry to allow separate testing.

2.4.2.2 Behavioural

Hardware description languages (HDLs) like VHDL and Verilog can specify the behaviour of a design. Synthesis systems [Gaj88, WC91, GDWL92] are used to create a description that can be manufactured from the behavioural description.

The hardware synthesis problem is usually broken up into the following steps [MPC88]:

- Parse the HDL and generate an internal representation. An internal representation usually consists of a combination of control and data flow graphs.
- Schedule the operations to "control steps."
- Allocate the functional units, registers, and interconnections.

The scheduling and allocating steps are closely related and dependent on each other.

These steps transform a behavioural description into a structural description, usually at the register-transfer level (RTL). In general, BIST embedding is considered at this level by restricting the architecture of the generated RTL designs.

In a paper by Papachristou [Pap89], the generated RTL architecture by the synthesis system is restricted to *testable functional blocks* (TFBs). A TFB consists of three layers: a layer composed of multiplexors, the next layer contains the combinational logic, and the last layer contains a BILBO register. Interconnected TFBs comprise the synthesized circuit. After an initial scheduling phase by one of the well known scheduling algorithms, rescheduling transformations are used if Papachristou's allocation algorithm fails to generate TFBs.

The SYNTEST synthesis system [HPCN92] also uses TFBs in its incorporation of BIST embeddings. The system performs BIST embedding after scheduling and allocation phases. Each register in the RTL description is initially assigned pattern generation and response analysis roles. An iterative procedure is used to merge these roles. The SYNTEST system allows the user to relax the requirement that each component block is a TFB.

In the paper by Avra [Avr91], the register and interconnection allocation phases of their synthesis system are restricted to producing designs in which pseudorandom BIST architectures can be easily embedded. Specifically, the paper assumes that all the registers in the data path can be configured to operate as normal registers, shift registers, pseudorandom pattern generator registers, and multiple input signature registers (MISRs). Their system minimizes *self-adjacent* registers. A register is self-adjacent when its inputs and outputs are connected to the outputs and inputs of the *same* combinational block.

In [LWJA92], the register allocation and interconnection allocations phases of the PHITS synthesis system considers testability by using heuristics based on improving the controllability and observability of the registers and reducing the sequential depth between registers. They propose the following rules to improve controllability and observability:

- Allocate registers to primary input and output variables whenever possible.
- Minimize the sequential depth between controliable and observable registers.

This system does not embed BIST, but it enhances the testability of the system.

Gebotys and Elmasry incorporate BIST embedding in their [GE92, GE89] CATREE and CATREE2 synthesis systems. In their CATREE system, BIST is embedded after a complete functional design has been synthesized. Their approach in selecting a BIST architecture is similar to the expert system used in [ZB88]. They record the test length and test coverage for each functional unit used in their synthesis library. The cost of the extra hardware, testing length and coverage costs are used to select the embedding that satisfies the user's input constraints. The CATREE2 synthesis system incorporates BIST after functional unit allocation, but before register allocation and interconnect allocation. The test embedding steps are: functional unit partitioning, test operation assignment and test scheduling. The CATREE and CATREE2 system are restricted to BIST architectures that test combinational logic.

BIST embedding of scan and BILBO BIST architecture has been incorporated in the Silc silicon compiler[FH86]. The system enforces a set of testability rules in the user's designs. The TESTPERT expert system analyzes the user's design and the test requirements and generates the testing structures and test control logic. The BIST structures are customized for the logic generated by the silicon compiler.

Another approach for embedding BIST at the behavioural level has been proposed by Beenker, Dekker, and Stans in [BDS89, BDSS90]. They call their approach "macro testing." This approach has been implemented in the PIRAMID hardware synthesis (silicon compiler) system. The compiler generates circuits using only testable macro blocks. That is, each macro block has been designed to contain a self-test mode. The silicon compiler has to connect the macro blocks together such that they remain testable and it has to generate test control circuitry.

2.4.2.3 Register-Transfer – Structural

At the register-transfer structural description level a design is described by a collection of interconnected components. These components can be classified as functional units or registers. The functional units usually consist of combinational logic blocks.

BIST embedding is considered at the register-transfer level (RTL) by binding stimulus sources, response analyzers to the registers and then determining how the tests are controlled. The main design concerns for BIST embedding at this level are:

- identifying the subblocks to be tested (partitioning might be required);
- determining to which registers to add stimulus functionality;
- determining to which registers to add response analysis functionality; and
- determining how to control the BIST mode.

Expert systems have been used to accomplish all of the above embedding tasks.

The testable design expert system (TDES)[ZB88, AB85] is one of the first systems to use an expert systems approach to solving the BIST embedding problem at the register-transfer level. These papers introduce the ideas of an I-path. An I-path can transfer data between the circuit's components. For example, multiplexor can be used as part of an I-Path. I-Paths are used to transfer test stimuli and test responses from the testing resources. This system captures design knowledge as testable design methodology (TDM) templates. The template describes the method's structural architecture, the type of subcircuit that can be tested (ROM, PLA), the testing resources, and testing time.

The "DFT Expert System" [BP89] embeds various BIST architectures in a circuit described at the register-transfer level. Some of the BIST structures used in this system are: scan paths, LFSRs, and counters. The system first classifies the circuit's components into data transport (DT) components and data processor (DP) components. The typical DT

components are registers and multiplexors. After DT and DP identification, the system then selects the test methods, configures the global DFT, and generates the test schedules. The user specifies the design constraints by giving weights between 0 and 1 to the following motivation factors: maximize fault coverage, minimize testing time, minimize logic area overhead, minimize test generation overhead, minimize performance degradation overheard, and minimize overhead from extra I/O lines. A simple graph model used to minimize BIST area overhead is presented in [CBAP89], by the group that developed the "DFT Expert System."

The BIDES expert system [KTH91, KTH88] incorporates a BILBO based BIST architecture into a design described in VHDL. The system creates a structural description from the VHDL design, and then examines a search space of different BIST embeddings looking for one that satisfies the user's constraints. Test length and fault coverage are determined by fault simulation. The system does not use this knowledge to modify the embedding. A BIST embedding consists of choosing testing roles for a set of registers, and determining the logic to be tested by these registers. The BIDES system is restricted to analyzing designs that are composed of "Structured/Synchronous Building Blocks (SBB)." A SBB is a combinational logic block followed by a register.

The SHADE[PBDB87] and CAST[KA88] design systems are examples of systems that perform BIST embedding for a fixed architecture. The SHADE system uses an approach similar to one used in the PIRAMID hardware synthesis system of building the circuit with testable components. The CAST system provides a graphical user interface to aid the user in embedding two specific BIST architectures into a register-transfer level description.

Su and Kime also present a CAD tool [SK90] that embeds a pseudoexhaustive BIST architecture at the register-transfer level. Their tool is restricted to circuits that have a "sensitizable hierarchy."

2.4.2.4 Gate Structural

Since gates are the primitive components of many IC technologies, test effective analysis at this level allows the most accepted standard for test effectiveness measures to be performed. Standard cells and gate arrays are examples of two of the more common technologies. The stuck-at fault model, assumes that physical defects in the components can be modeled by fixing the inputs or outputs of gates to one logical value. The stuck-at fault model [SK90, page 4] based on having the gate's inputs and outputs stuck is used in this dissertation.

The test effectiveness of a BIST embedding is measured by the fault coverage achieved. Since the fault models are based on wires and gates, the effectiveness of any BIST embedding can only be measured at the gate level.

In the design of a BIST embedding, fault simulation can be used not only to measure the embedding test effectiveness but also to:

- aid in the selection of stimulus sources and configurations;
- aid in the selection of observation points; and
- find redundant logic.

BIST embedding using fault simulation at the gate level is discussed in [Dun89, TS88, McL92, IB89].

Duncan [Dun89] uses the following methodology to embed BIST at the gate level:

1. Add BIST logic.
2. Use fault simulation to determine controllability problems.
3. Add logic to improve controllability.
4. Repeat 2 and 3 until there are no more controllability problems.
5. Use fault simulation to determine observability problems.

6. Add logic to improve observability.
7. Repeat 5 and 6 until there are no more observability problems.
8. Ensure timing constraints are still met.

The placement of the stimulus and response analysis structures is not discussed. Applying this methodology on three chips: a write data path chip, an instruction decode chip, and a cache master arbitration chip results in single stuck-at fault coverage of 100%, 100% and 98% respectively.

In [TS88], intelligent knowledge-based system (IKBS) techniques for a macrocell based design are used to embed the stimulus and response analysis structures. Fault simulation using the parallel pattern single fault propagation (PPSFP) algorithm [ELWW91] is used to measure fault coverage, to aid in the modification of the logic to improve controllability and observability, and to remove redundant logic. One hundred percent fault coverage is achieved for most of their sample circuits.

Logic and fault simulation are also used by McLeod [McL92] to aid in BIST embedding at the gate level. Specific circuit structures that are difficult to test with a pseudorandom approach are given. Methods for improving the oberservability and controllability of these structures are also presented.

Fault simulation is used by Iyengar and Brand [IB89] in selecting stimulus and observation points. They uses a scan based BIST architecture.

The BIST embedding approaches used at the RTL level in Subsection 2.4.2.3 can also be applied to the gate level. Gate level descriptions are grouped into *clouds* and *registers* by the Crete system[GRB91]. A cloud is a possible empty group of combinational logic gates. A register is a group of flip-flops. The gate level description is clustered into clouds and registers so that a register-transfer level BIST embedding analysis can be used.

Two gates are placed in the same cloud if at least one of the following conditions is met:

1. The output of one gate is connected to the input of the other gate.
2. The same input signal feeds the two gates.
3. A flip-flop's output is connected to one gate, and the flip-flop's complemented output is connected to the other gate.

Two flip-flops are placed in the same register if both of the following conditions are true.

1. The data inputs of the flip-flops are connected to the same cloud, and the data outputs of the flip-flops are connected to the same cloud.
2. The clocking and control logic of the flip-flops are identical.

Methods for embedding the pseudoexhaustive BIST architecture in a gate level description by grouping and partitioning the groups so that the groups can be pseudoexhaustively tested is discussed in [CYL92]. Results of grouping and partitioning the ISCAS benchmarks are also presented [CYL92, DB91, GRB91].

If the circuit contains no redundant logic, then the test embedding process does not have to check for redundant/nontestable logic. Minimization of redundant logic by logic partitioning and resynthesis is discussed in [DB91]. Their method does not guarantee the complete elimination of redundant logic.

2.5 Discussion of Embedding Approaches

The BIST embedding approaches reviewed in Section 2.4.2 can be divided into two general approaches:

1. Approaches that consider the interconnection of the register-transfer design, the data transfers, and the operating modes of the registers.
2. Approaches that use fault simulation at the gate level to direct the BIST embedding process.

BIST embedding approaches at the behavioural level restrict their scheduling and allocation phases to produce RTL architectures that can be tested with a fixed set of BIST architectures. The most common BIST architectures are scan based ones and BILBO ones. Thus the behavioural level approaches can be classified as using the first approach.

The register-transfer level approaches usually examine the design at the register and their interconnection to determine where to place the BIST structures. Thus this method can also be classified as using the first general approach.

Several common traits of the first general approach are:

- the circuit is partitioned into combinational logic and then the combinational logic is tested,
- only a few fixed architectures are chosen (e.g., BILBO),
- fault simulation is used to determine fault coverage and test length of library components,
- fault simulation is not used to modify the embedding, and
- redundant logic is not considered.

The gate level embedding approaches follow the second general approach. Fault simulation is used to determine stimulus points and observation points. Hard to test logic is identified and modified to improve its testability.

The gate level approaches are less structured than the behavioural or register-transfer level approaches and thus less amenable to automation.

The approach taken in this dissertation and in the design of Logic III(UVic) and the `lg3` CAD tool is to try and strike a balance between these two approaches. The architecture is specified at the register-transfer level and fault simulation of the gate level description is used to evaluate the embedding. The evaluation can then be used as feedback to modify the embedding.

This dissertation's approach fits in reasonably well with the "macro test" [BDS89, BDSS90] approach in that the macros are designed to be testable. The `lg3` environment provides a set of facilities to created circuits that are self-testable.

Chapter 3

Logic III(UVic)

3.1 The Approach

Two of the major concerns of incorporating BIST functionality into a design are the following:

- The measurement of the resulting test effectiveness to ensure that it is sufficient.
- Specifying BIST embeddings easily and concisely so that the *original* design description is not disturbed.

The digital design process can be considered to pass through four stages: requirement, behavioural, register-transfer structural, and gate structural (see Section 2.4.2). Each of these stages uses a different design description. Some designs skip the behavioural stage. Since test effectiveness is measured by structural fault models, the test effectiveness concerns can only be considered at the structural and physical level. Concise design descriptions are more easily expressed at abstract design levels. For example, it is easier to state $a+b$, then to describe how to build an adder with transistors. This implies that BIST embedding should be described at the requirement and behavioural levels. The testing of a design is concerned

with the *global* design, while functional design attempts to isolate the different functional units. Thus test embedding descriptions differ from functional descriptions, because of the global/local differences. One of the goals for the Logic III(UVic) language is to allow both BIST embedding and functional designs to be easily specified.

BIST embedding is addressed by augmenting a hardware description language (HDL) to contain features that can be used to specify BIST embeddings. To simplify implementation issues, the Logic III language [MCN90] is used. This augmented language is called Logic III(UVic). Logic III(UVic) allows designs to be captured at the structural level. The structural level provided by Logic III(UVic) allows concise BIST specifications, but still preserves a close mapping between the language constructs and the physical description level. Logic III(UVic) uses variables of type **node**, **input**, and **output** to represent circuit nets. Since most fault models use the circuit nets to specify the fault's location and Logic III(UVic) represents nets as variables, the fault effects are identified by Logic III(UVic) variables. In the single stuck-at fault model, a net's value is stuck-at 1 or 0. Thus if the BIST embedding does not detect this fault, it is reported to the user by the Logic III(UVic) variable name.

The rationale for BIST incorporation by addition of BIST embedding constructs in a HDL are the following:

- The BIST embedding constructs added to the HDL allow the BIST functionality to be captured explicitly as part of the design process.
- If the constructs are *expressive enough*, then different BIST embeddings can be explored.

The compiler and functional/fault simulation tool developed to determine the test effectiveness is also discussed in this chapter. Test effectiveness is addressed by fault simulating the netlist of primitive gates generated by the Logic III(UVic) compiler. This netlist is created by expanding the hierarchical description in to a flat description composed of onl,

primitive gates. This expansion process is called flattening. Fault simulation at the gate level provides a reasonable measure of test effectiveness (most widely accepted). The fault coverage results must be expressed at the BIST specification level to allow designers to modify their designs if the achieved fault coverage is not sufficient. Thus the compiler and simulator are closely coupled.

3.2 The Built-In Self-Test Embedding Problem

The BIST embedding problem can be partitioned into the following subproblems:

- choice of the stimulus structures,
- choice of the response analysis structures,
- placement of the stimulus structure,
- placement of the response analysis structure, and
- control of the BIST mode.

From Section 2.4.1, the goals of a BIST embedding are:

- maximize the test effectiveness (i.e., maximize fault coverage (FC)),
- minimize the BIST specific circuitry (BC),
- minimize the test length (TL),
- minimize the timing performance impact, measured in gate delays (GD), and
- minimize the design effort (DE).

In other words, the primary task of the BIST embedding is to ensure that the design detects all faults from a specified fault set. The BIST functionality should be implemented with the fewest resources, no impact on the timing performance, and the fewest number of testing cycles. The effort required to do BIST design should be minimal.

3.2.1 Stimulus Structure Design

The stimulus structure determines the maximum achievable fault coverage. A fault is exposed only if the necessary stimulus is generated by the stimulus structure.

The functional design process consists of two phases: synthesis, where an entity is created to satisfy a requirement, and analysis, where the created entity's behaviour is verified against the expected behaviour. The design process is repeated if the created entity's behaviour does not match its expected behaviour.

In digital circuit design, the entity created is the circuit. The circuit's input/output mapping, and the input/output timing relationship are analyzed to see if it complies with the desired behaviour. If the expected behaviour differs from the realized behaviour, then the circuit must be modified. Normally, it is *easy* to determine which part of the design needs modification (i.e., the designer can trace the logic from the error output).

A requirement of embedding BIST into a digital design is to detect the presence of faults from a specific fault set. The design goal is to detect all the faults. Once a BIST mode is added to a circuit, the BIST mode is analyzed to determine the fault coverage.

For random BIST architectures, one of the major design problems is determining how to modify the embedding in order to improve the achieved fault coverage. The difficulty can be attributed to the fact that the patterns produced by these stimulus structures are not explicitly designed to expose the faults from the specified fault set. Unlike functional digital design, the modifications required to improve the fault coverage are not obvious.

If the patterns are not effective (i.e. the stimulus source does not generate the necessary patterns), then the designer's options are:

- use another stimulus source, and
- modify the kernel by adding stimulus and observation points (See 3.2.2).

3.2.2 Stimulus and Observation Points Design

The placement of the stimulus and observation points define the subcircuit which is to be tested (i.e., the kernel). The addition of stimulus and observation points directly affects the timing performance and amount of BIST specific circuitry. The choice of the stimulus points, where the stimulus is injected, contributes to how effective a stimulus structure is in exposing faults. The observation points placement determines which faults are detectable. Any fault located between the stimulus source and the observation points can be detected. In general, the greater the number of stimulus and observation points, the smaller the number of patterns necessary to expose the faults in the given subcircuit. Thus the TL can be reduced by increasing the number of stimulus and observation points.

In designing a BIST embedding, the kernel that is to be tested is determined by the placement of stimulus and observation points. The difficulty of exposing a particular fault depends on controllability and observability of the net associated with the fault. The controllability of a net is measured by the number of input patterns that are required to set the net to a particular value. If the stimulus points are attached directly to the net, then the controllability is perfect. As more logic is placed between the stimulus inputs and the net, the controllability is usually reduced. Multiplexors (MUXes) are examples of logic that does not reduce controllability. Observability measures how difficult it is to set up a path that propagates the fault's effect from the net to the observation points. The sensitized path propagates the fault effect to an observation point. Placing the observation point on the net gives perfect observability.

At the expense of adding stimulus and observation points to every net, perfect testing access could be embedded. To reduce the embedding cost the number of stimulus and observation points should be reduced. As the number of stimulus and observation points decreases, the number of patterns generated by the stimulus structure must increase to expose the same fault set. Thus the complexity of the stimulus structure can be traded off against the number of stimulus and observation points.

Faults that are difficult to test by pseudorandom sources, are called random pattern resistant. Changing the stimulus and observation points, allowing better controllability and observability, can reduce the number of random pattern resistant faults. Fault simulation can be used to aid in the placement of stimulus and observation points.

3.2.3 Response Analysis Design

Since the response analysis structure function is to capture any error effects, the choice of a response analysis structure is based on this ability, and its circuitry cost. Aliasing probabilities can be used as a measure of the response analysis structure's ability to capture error effects. From Section 2.3, MISRs provide the smallest aliasing probabilities at the lowest cost. If the state of the functional circuit is used, then a lower cost response analysis structure is possible (i.e., no cost), but more design effort is required. This extra effort is caused by extra fault simulations required to determine the fault coverage.

3.2.4 BIST Controller Design

A BIST controller is responsible for accepting external commands, controlling the individual tests, and reporting the test results.

In controlling the test, the following actions are usually performed:

1. Initializing the memory elements of the stimulus structures, the kernels, and the response analysis structures.
2. The following is repeated for each individual test:
   (a) Set any test mode lines to their test values.
   (b) Cycle the test for the required test length.
   (c) Collect the response and compare it with the expected response.

3. Report the test results.

The testing interface is concerned with reporting the test results and accepting commands. If the BIST embedding is on a single chip, then the interface could be as simple as two lines. The test is started with a TEST pin, and the result is reported by a PASS/FAIL pin. The BIST mode of JTAG[IEE90] can be used as a testing interface.

BIST controller design is similar to any other controller design.

3.3 Logic III(UVic)

The modifications to Logic III which add BIST embedding features, facilitate the interface between the language and the simulator, and increase the language expressiveness. Some of the expressive features are motivated by testing concerns. The language reference manual (see Appendix A) contains a complete description of the language. This section introduces the language and discusses in detail the modified features and their rationale.

The traditional way of describing circuit structure is with schematic diagrams. A carry circuit is shown in Figure 3.1(a). Figure 3.1(b) shows a sum circuit. Schematic diagrams are composed of symbols, lines, and labels. The symbols represent information transforming entities, referred to as *modules*. The lines, referred to as *nets*, represent interconnections between entities allowing information to be transferred between them. Lines with the same label are connected. For example, in Figure 3.1(a) the lines labeled b are connected.

Hierarchical circuit description is accomplished by grouping a set of modules and nets, and then labeling the nets that connect to the outside. The dashed lines in Figure 3.1 indicate the grouping. The labeled nets are referred to as *ports*. Using the carry and sum modules, an adder module is created (See Figure 3.2). Modules that are not composed of other modules are called *primitive*.

The Logic III(UVic) language is designed to represent the structure of a circuit by providing constructs for modules, ports, and nets. Figure 3.3 shows the Logic III(UVic)

Figure 3.1 Adder and Carry Circuits

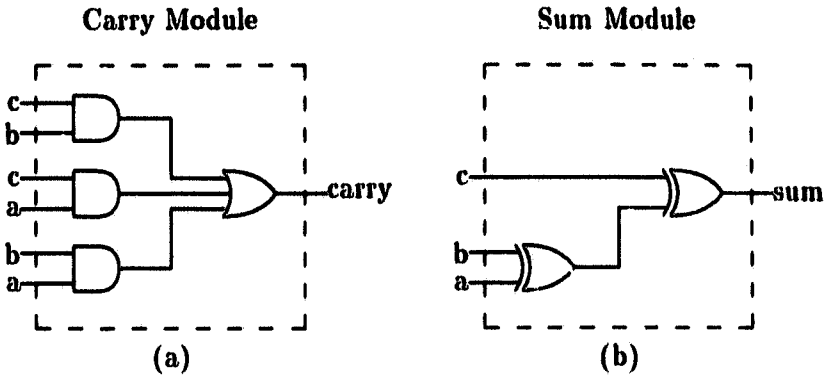
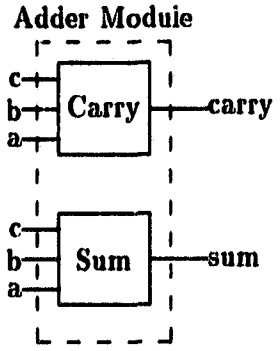


Figure 3.2 Hierarchical Modules



text for the carry and sum circuits. The syntax of a module is similar to that of a Pascal procedure. A module's parameters represent the ports. Variables are used to represent nets. The variable types, **input** and **output**, specify which variables are the module's inputs and outputs. A variable of type **node** specifies a net that is used locally to the module.

Figure 3.3 Logic III(UVic) code for sum and carry modules

```
net_module exor ( x,y : input; xor_xy : output ); external;
net_module a2 ( x,y : input; and_xy : output ); external;
net_module o3 ( x,y,z : input; or_xyz : output ); external;

net_module carry( a,b,c : input; carry : output );
var
    ab, ac, bc : node;
begin
    a2( a, b, ab);
    a2( a, c, ac);
    a2( b, c, bc);
    o3( ab, ac, bc, carry );
end.

net_module sum( a,b,c : input; sum : output);
var
    ab : node;
begin
    exor( a, b, ab);
    exor( ab, c, sum);
end.
```

The ports and name of the primitive modules **a2**, **o3**, and **exor** are introduced with a module definition where the body is defined as **external**. Module **a2** is defined by its **net_module** definition, and it is *instantiated* in the **carry** module. The **a2** module is a *component* of the **carry** module. A module instantiation means that a physical realization of the module is included in the construction of the module in which it is instantiated. Nets are similarly defined in a variable definition and are instantiated when they are used in a component module.

Although instantiating a component module uses the same syntax as Pascal's procedure calls, the semantics are completely different. The arguments to a component module and

the associated parameters in the component module are variables which represent the same net. Instantiating a module is equivalent to copying the component's body in place, with the necessary variable name changes to ensure that the different nets remain distinct.

The Logic III(UVic) program for an adder circuit depicted in Figure 3.2 is:

```
net_module adder( x,y,z : input; sum, carry : output );
begin
    sum( x,y,z, sum);
    carry( x,y,z, carry);
end.
```

The previous examples show modules where the structure is fixed, but Logic III and Logic III(UVic) have the ability to describe modules where the structure can vary. For example:

```
net_module add_sub( as_flag : integer;
    a,b,c : input; sum, carry : output );
var
    b_inv : node;
begin
    if as_flag = 1 then begin
        adder(a,b,c, sum, carry);
    end
    else begin
        ii(b, b_inv);
        adder(a,b_inv,c, sum, carry)
    end;
end.
```

Depending on the value of the integer parameter `as_flag` this module produces either a subtractor or an adder module. Customized modules are specified with non-net parameters and/or array parameters. A Logic III(UVic) module definition is therefore a template for a set of possible modules. A customizable module can also be viewed as a function returning the structural description of the circuit. In summary, Logic III(UVic) describes circuits with nets, modules, and ports. Nets are represented by variables of type **node**, **input**, or **output**. Modules are represented by the **net_module** construct. The parameters of a **net_module** represent ports.

3.3.1 Types, Constants, Arrays, and Globals

The variable types in Logic III(UVic) can be separated into those that are part of the structural description and those that are used to customize the structural description. The string and integer types are used to parameterize the structural description. Variables of these types are directly interpreted by a Logic III(UVic) compiler, by themselves they do not represent any structure. The **node** type represents the interconnection element in the structural description (i.e., the wire). The **input** and **output** types are special forms of the **node** type. They only can occur in the parameter list of a module or in the **var** section of the **circuit** module. The types **input** and **output** in the **circuit** module specify the variables that are the primary inputs and outputs for the circuit. The term, net is used when a variable can be either a **node**, **input**, or **output** type. A **reg** type variable represents a 1-bit memory element. The **reg** data type is an addition to Logic III. Variables of type **reg** can act as BIST observation and stimulus points.

The following design fragment declares an integer, **i**, two nets, **a** and **b**, and a memory element, **r**.

```
var
    i : integer; // an integer
    a,b : node ; // two local nets
    r : reg; // a 1 bit memory element
```

Variables are also declared in parameter lists.

```
net_module min_term0( a,b,c : input; abc : output);
net_module mux( x,y : input; sel : input; mux_out : output);
```

The two operations defined for net variables are instantiating and aliasing. A net variable is instantiated when it occurs as an argument in a module instantiation. A net variable is aliased to the associated parameter variable of the component module in which it appears as an argument.

In a module's parameter list an **input** type specifies that the signal on the net is generated outside of the module. It is illegal for an input variable to become an argument of a

module where the matching parameter's type is **output**. For an output variable the signal on the net is either generated by a module instantiated inside the module, or by one of the input nets. Thus it must appear as an argument to a component module where the matching parameter's type is **output**, or it must be aliased with an input variable.

Variables of type **reg** are instantiated by placing the variable as an argument to a module and as the target of an update statement. The update statement specifies which net drives the input to the memory element. The output of the **reg** variable memory elements is accessed by using the variable as an ordinary net variable. In a module instantiation the associated parameter's type must be **input**. Instantiation of a **reg** variable produces the components that implement the memory element. The syntax of an update statement is:

$$x \leftarrow y$$

where **x** is **reg** and **y** is a net or a **reg** variable. The **reg** variable is also used in BIST embeddings.

Integer variables have the usual set of arithmetic and comparison operations defined. These arithmetic operations are: addition, multiplication, subtraction, division, and remainder (modulus). The comparison operations are: equal to, less than, less than or equal to, greater than, and greater than or equal to. The value of an integer variable can be modified with an assignment statement. Built-in integer functions that calculate the minimum, maximum, and ceiling of the logarithm base 2 of their arguments are provided.

Strings can be assigned, compared for equality and substrings can be extracted. The syntax for substring extraction is identical to that for array slicing. The syntax for array slices is defined later in this section.

Another addition to Logic III(UVic) is the ability to define symbolic integer constants. The constant section allows the user to give constants more meaningful names and localizes the constant definition, enabling easy modification. A constant can be used immediately after its definition. Some examples are:

```
const
    word_size = 16;
    mid_bit_index = word_size/2;
```

Array types are also supported in Logic III(UVic). One dimensional arrays of the base types can be created. In addition to fixed-size arrays, Logic III(UVic) adds variable-size arrays. Variable-size arrays can only occur in parameter lists and local to modules. The array declaration syntax is similar to Pascal's syntax.

```
var
    word : array [3..18] of node;
    register : array [1..8] of reg;
    sort_list : array [-5..4] of integer;
```

The variable **word** is a 16 element array of type **node**. Its lower bound is 3 and its upper bound is 18. As shown in the declaration of **sort_list**, array bounds can be negative.

For example, given the following array definition and module header:

```
var counter : array [3..10] of node;
net_module bin_counter( cnt : array [] of output );
```

and the instantiation statement:

```
bin_counter( counter );
```

the parameter **cnt** size is (10-3)+1= 8, its first index is 0, and its last index is 7.

The array elements can be referenced as individual elements or as slices. The operation of array slicing allows a subarray, a continuous segment of the array, to be referenced. Array elements are accessed with the syntax:

array_var [*expression*]

The array access syntax for a slice is:

array_var [e_1 .. e_2]

The e_1 expression must be less than or equal to the e_2 expression. An array slice of size one is still an array. Example array definitions for the variables **arr** and **fred** are:

```
var
    arr : array [1..8] of node;
    fred : array [1..16] of integer;
```

A slice containing the elements 3, 4 and 5 of arr is `arr[3..5]`. The slice `fred[5..5]` is a size one integer array, while `fred[5]` is an integer. The range of all array accesses must be between the lower and upper bounds of the array.

The following properties are associated with arrays:

Property	Meaning
first	the array's lower bound
last	the array's upper bound
size	the array's size

For the `first`, `last`, and `size` property, the property expression evaluates to an integer constant. Using the definition of `fred`, the following expression values are:

```
fred'first = 1
fred'last  = 16
fred'size  = 16
```

In addition to array variables, one dimensional arrays can be created by concatenating scalar variables, array variables, and array slices. For the following variable definitions:

```
var
    a,b,c : node;
    e     : array [5..15] of node;
```

A collection of these variables forming an array can be specified with:

```
[ a,b,c, e[15], e[6..7] ]
```

The type of this collection is `array [0..5] of node`. The lower bound of the collection is 0. Its upper bound is the sum of the sizes of all the items in the collection minus one. An example of using a collection is:

```
var a,b,c,d,e : node;

net_module And( x : array [] of input; y : output); external;

And( [a,b,c,d], e );
```

The collection of scalar elements **a**, **b**, **c**, and **d** form a four element array, that is aliased to the **x** array of **And**.

Unlike Logic III, the global variable definition section can occur multiple times in a Logic III(UVic) design. Allowing more than one occurrence of the global section is convenient in the construction of libraries. Global variables are declared with the syntax:

global

variable declarations

end.

Ideally, the multiple global variables declaration should be replaced by a means to define packages (à la ADA).

3.3.2 Recursion and Functions

The ability for modules to include a recursive instantiation is a major change to Logic III. Recursive descriptions can be used to construct tree like circuits. For example, the text for an arbitrarily large AND gate is:

```
net_module And( x : array [1..] of input; y : output );
const
    n = x'size; mid = n/2 + x'first - 1;
var
    t1, t2 : node;
begin
    if n > 2 then begin
       And( x[1..mid], t1);      // handle half of the inputs
       And( x[mid+1..n], t2);  // handle the other half
       a2( t1, t2, y);
    end
    else if n = 2 then a2( x[1], x[2], y)
    else connect( x[1], y); // n = 1
end.
```

This module is customized by the size of the argument x. Slices are very useful for partitioning arrays. Note that constants can be initialized with parameter values.

Instantiating And with the variable acc of type array [1..4] of node yields the following structure:

```
a2( acc[1], acc[2], t1);
a2( acc[3], acc[4], t2);
a2( t1, t2, and_out);
```

A more comprehensive example of recursion is a multiplier implemented as an adder array [WE85]. The recursive part of the multiplier is:

```
net_module multiplier( x,y : array [1..] of input;
    z : array [1..x'size+y'size] of output;);
const
    z_sz = z'size; y_sz = y'size; x_sz = x'size;
    pp_sz = z_sz - 1; pp_ind = (pp_sz - x_sz) + 1;
var
    part_prod : array [1..pp_sz] of node;
    part_sum, mpy_by_1, mpy_1_a : array [1..x_sz] of node;
begin
    if y'size > 2 then begin
        mpy_by_one ( x, y[y_sz], mpy_by_1 );
        multiplier ( x, y[1..y_sz-1], part_prod);
        mpy_add(mpy_by_1,part_prod[pp_ind..pp_sz],z[pp_ind..z_sz]);
        connect( part_prod[1..pp_ind-1], z[1..pp_ind-1] );
    end
    else begin
        mpy_by_one ( x, y[1], mpy_by_1 );
        connect( mpy_by_1[1], z[1] );
        mpy_by_one ( x, y[2], mpy_by_1_a );
        mpy_add(mpy_by_1[2..x_sz],mpy_by_1_a[1..x_sz-1],part_sum);
        connect(part_sum[1..x_sz-1], z[2..x_sz]);
        ha(part_sum[x_sz],mpy_by_1_a[x_sz],z[z_sz-1],z[z_sz]);
    end
end.
```

Array slicing is used to partition the multiplier, y, into the most significant bit and the remaining bits. The remaining bits and the multiplicand, x, are arguments to the recursive instantiation of the multiplier. y is multiplied by the most significant bit and the partial product is added in the shifted position to the result of the recursive multiplier instantiation.

Using a schematic capture system to create the multiplier structure can be tedious and error prone.

The function module is another major change. All of the basic types and arrays of the types can be returned by functions. The standard example function is:

```
function fac( n : integer ) : integer ;
begin
    if n > 0 then fac :=  n * fac( n - 1 ); else fac := 1;
end.
```

The function's name appears as a local variable in the scope of the function.

A structural function that computes the bitwise AND of two arrays is:

```
function bit_and( x : array [] of input;
    y : array [1..x'size] of input ) : array [1..x'size] of output;
var
    i : integer;
begin
    for i := x'first to x'last do a2(x[i], y[i], bit_and[i]);
end.
```

Note that `bit_and[i]` is an access to the element in the array returned by the function and not a recursive call. Permuting the order of observation or generation points into the observation or generation structure is a useful capability when performing BIST embedding. User functions can be written to perform common permutations. For example, the cross-over permutation is useful for generating the stimulus for delay faults [ZBM92]. A cross over function is:

```
function cross( x : array [1..] of input )
          : array [x'first..x'last] of output;
const
    mid = x'size / 2;
var
    permute : array [x'first..x'last] of integer;
    i : integer;
begin
    for i := 1 to mid do permute[2*i-1] := i;
    for i := mid+1 to x'last do permute[2*(i-mid)] := i;
    for i := 1 to x'size do
        connect(x[i], cross[permute[i]]);
end.
```

3.3.3 Build-In Self Test Embedding Features

The BIST embedding features are based on the following observations:

- Testing of combinational logic is easier than testing of sequential logic
- Registers can be easily modified to include stimulus and response observation abilities (see Section 2.4.2.3).
- Most scan based design for testability approaches (full-scan) convert all the memory elements into scannable memory elements.
- BIST embedding is often considered at the RTL level (also see Section 2.4.2.3).

What these observations have in common is that the input and output of memory elements are used as the stimulus and response observation test points. Thus the structures for stimulus generation and response observation are part of the memory element's functionality.

In Logic III(UVic), memory elements are described with **reg** variables. The **use** statement specifies the structure to be instantiated for the set of **reg** variables instantiated in the body of the use statement. The syntax for the **use** statement is:

use *embedding-mod* **in** *body*

The *embedding-mod* must be the name of a **net_module** with the following parameter list:

```
net_module embedding( d:array[]of input; q:array[]of output);
```

An embedding module which instantiates each **reg** variable as a D flip-flop is:

```
net_module register (d:array [] of input; q:array [] of output);
var
    i : integer;
begin
    for i := 1 to d'size do dff(d,q);
end.
```

For all the variables of type **reg**, the nets driving the inputs to the **reg** defined in the **update** statement, and the **reg** outputs are collected in **d** and **q** arrays respectively. The embedding module is passed these two arrays when it is instantiated. Since the user defines the embedding module, any structure can be instantiated for the memory elements.

Thus the following two program fragments are structurally equivalent.

With reg variables	reg variables instantiated

var
  a : array [1..2] of reg;
  x : array [1..4] of node;
  y : array [1..2] of node;
begin
  use register in begin
  a2(x[1], x[2], y[1]);
  a[1] <- y[1];
  a2(x[2], x[3], y[2]);
  a[2] <- y[2];
  end
end.
	var
	  a : array [1..2] of reg;
	  x : array [1..4] of node;
	  y : array [1..2] of node;
	begin
	  a2(x[1], x[2], y[1]);
	  dff(y[1], a[1]);
	  a2(x[2], x[3], y[2]);
	  dff(y[2], a[2]);
	end.

An example of embedding a LFSR and a MISR into an adder circuit is:

```
circuit reg_adder(
    x, y : array [1..8] of input;
    add : array [1..9] of output );
var
    z : array [1..9] of node;
    x_reg, y_reg : array [1..8] of reg;
    z_reg : array [1..9] of reg;
    i : integer;
begin
    use gen_embed in // insert the stimulus at the inputs
        for i := x'first to x'last do begin
            x_reg[i] <- x[i];
            y_reg[i] <- y[i];
        end;
    adder( x_reg, y_reg, add );
    use misr_embed in // analyzes the response at the outputs
        for i := add'first to add'last do begin
            connect( z_reg[i], z[i] );
            z_reg[i] <- add[i];
        end;
end.
```

One of the main advantages of this specification approach is the decoupling of the *functional* design from the *BIST* design. Different embeddings can be realized by simply

changing the embedding module name. Replacing the input and output registers with BIST mode registers realizes a BIST embedding without the explicit specification. Specifying the BIST mode in the same way as the functional mode increases the difficulties of trying different BIST embeddings.

The above example shows a close association between the register definitions and their BIST embedding. In general, the use statement can be applied to any Logic III(UVic) statement. For example, the following program embeds a scan based BIST approach into an entire chip.

```
net_module top_chip( x,y,z : input; a,b,c : output ); external;
net_module scan_bist( d:array[] of input; q :array[] of output);
external;

circuit top;
var
    x, y, z : input; a, b, c : output;
begin
    use scan_bist in top_chip(x,y,z, a,b,c);
end.
```

Assuming there are no other use statements, all of the inputs and outputs of **reg** variables are grouped into arrays and passed to **scan_bist**.

3.3.4 Proposed new BIST features in Logic III(UVic)

The testability problems of circuits with redundant logic are partially addressed by two features proposed but not implemented in Logic III(UVic). The purpose of these constructs is to allow the user to identify the logic that is unnecessary.

The first feature allows a user to mark outputs of a module that are not used. This feature is called capping and an example of its syntax is:

module (a, b, c, _, d, _, e);

The fourth and sixth arguments of *module* is a "_". The "_" specifies that the matching output of the module is unused. Note that the matching parameter's type must be of type **output**. The purpose of this construct is to enable a user to utilize a more general module, but still to inform the CAD system that the marked outputs are unused. The CAD system can at worst remove the faults associated with the output logic, and at best remove the unused output logic. This feature can be simply implemented by a procedure that starts at the output net and removes the logic that only drives this net.

The second feature allows the specification of inputs to modules that are either constant 1 or constant 0. This feature complements the capping feature. An example of its syntax is:

module (TRUE, FALSE, c, _, d, _, e);

This construct informs the CAD system that the logic attached to the *constant* inputs can be simplified. Assuming the CAD system knows the boolean function implemented by the primitive gates, adding this feature to the language consists of applying boolean algebra rules to all the primitive gates driven by the constant inputs.

These two features can be used to specialize a library module so that it can still be used, instead of having the designer develop a new module with almost identical behaviour. An example of using these features is to customize an adder module with a carry-in input and carry-out output to an adder module with no carry in or carry out. For example:

```
adder(TRUE, x, y, z, _);
```

3.4 Language and CAD Environment Issues

The premise for the design of the interface features is that in order for the fault simulation results to be useful to the designer, any testing problems must be reported at the Logic III(UVic) language level.

The purpose of the **lg3** CAD tool is to enable its users to:

- verify that the circuit is behaving as expected,
- evaluate the test effectiveness of a BIST embedding,
- experiment with different BIST stimulus structures and response analysis configurations as a means of optimizing the BIST embedding goals.

To accomplish the first goal, the users of the **lg3** must be able to specify the input stimulus sources and to display the responses. Specifying the input stimulus sources raises the following issues:

- how the signals are identified,
- how the simulator is controlled, and
- how the stimulus sources are specified.

The most natural way of identifying the signals/nets is by using the same *names* as the ones that appear in the Logic III(UVic) design description. Since global and primary input nets are only instantiated once, their names uniquely identify them. Unfortunately the names of nets defined in component modules are not sufficient since those nets can be instantiated more than once. Every occurrence of a component module instantiates all of the nets in its definition.

The way to solve the duplicate name problem is to use a *hierarchical* name that uniquely identifies a net. The construction of hierarchical names is not easy in Logic III(UVic), since the component modules are not named. The problem of uniquely naming the nets is addressed later in this section. In the current implementation of **lg3**, the user can only specify how to stimulate the primary inputs and any global inputs.

The input stimulus is specified by the following set of pattern generato

file, a file containing test input prepared by the user,

seq, a fixed sequence,

counter, a binary counter, and

LFSM, the following set of LFSMs: LHCA, LFSR, TWODCA (a two-dimensional cellular automaton), and versions that also contain the all zeros pattern.

The full details of these generators are found in Appendix B. These pattern generators are attached to a Logic III(UVic) net with the following syntax. Assuming the primary inputs to a circuit are:

```
x,y : array [1..8] of input;
```

then the input stimulus is specified by:

```
x :- counter(1);
y :- file("y_input.data");
```

Note that the size of the counter is not explicitly given, but instead the size is inferred from the size of the `x` array. Thus the binary counter's size is set to eight. The argument of 1 to `counter(1)` specifies that the counter's initial value is 1. The least significant bit of the counter is attached to `x[1]`, the net in the array with the lowest index. The file generator reads the stimulus from the file `"y_input.data"`.

A complete testing script example is:

```
testing
  // variables to control the testing script are located here
begin
    x :- counter(1);
    y :- file("y_input.data");
    simulate(256);
end.
```

All of the variables declared in the circuit section and in the global sections of a Logic III(UVic) design are visible in the testing script. The `simulate` command directs `lg3` to perform 256 simulation cycles. This testing script does not specify which nets are to be displayed. The following subsection discusses how this is done by adding simulation specific features to Logic III(UVic).

Evaluation of a BIST embedding is performed by setting up the BIST mode of operation and performing a fault simulation. The normal stimulus input features of `lg3` can be used to setup the BIST mode. The important questions to address in determining BIST test effectiveness are "what faults" and "how do they correspond to the Logic III(UVic) description."

The structural single stuck-at fault model is used in `lg3`. This fault model is based on inserting stuck-at faults at the interconnections of the functional modules. The interconnections are represented in Logic III(UVic) as net variables. Thus the faults are associated with Logic III(UVic) net variables.

Therefore a method of uniquely naming each net in the design is required. This reintroduces the problem of how to uniquely identify a net in a module where the module can be instantiated more than once. The net can be identified by describing the module instance that contains the net. A module instance is described by listing the hierarchy of module instances in which it is created.

For example, the hierarchical name for the net `fred` in the following example is `top.one.two.fred` where `top.one.two` identifies the module instance containing `fred`.

```
net_module two( ... );
var
   fred : node;
begin
...
end.

net_module one( ... );
begin
...
   two( ... ); // line 36
...
end.

circuit top;
begin
...
   one( ... ); // line 47
...
end.
```

Unfortunately, this scheme is insufficient for Logic III(UVic) since the name of the component module can appear more than once in the enclosing module. This can be solved by appending the line number in which the module appears. Thus the unique name for `fred` becomes `top.one@47.two@36.fred`.

The remaining problem of uniquely identifying a module instantiation is illustrated by:

```
for i := 1 to 10 do
    two( . . );
```

In this case the line number is not sufficient to uniquely identify the module. The solution is to include the value of the loop's counter in the hierarchical name. The hierarchical name for `fred` is now `top.one@47.two@36<` i `>.fred`.

In summary, faults are associated with the instantiated nets and Logic III(UVic) uses the net's hierarchical name to reference the faults. These hierarchical names are the **primary** way that the `lg3` reports testing difficulties to the user. Specifically, the faults that cannot be tested by a BIST embedding are reported with the hierarchical name.

The designer of an off-line BIST embedding is concerned with the effectiveness of the stimulus generation structure and the response analysis structure. These separate concerns are addressed in lg3 by recording which faults are *exposed* and which faults are *detected*. An *exposed* fault is a fault that causes the output values at the observation points to differ from the fault-free case at least once during the test mode. A *detected* fault is a fault that causes the final state of the BIST response analysis structure to differ from the fault-free state.

In the preceding discussion, the method of defining the "observation points" is not given. Section 3.4.1 describes the addition to Logic III(UVic) that specifies the location of the observation points.

In evaluating a BIST embedding, the designer usually requires the following:

- the fault coverage of exposed faults,
- the fault coverage of detected faults,
- names of the unexposed and undetected faults, and
- the fault coverage for each test cycle.

The fault coverage of exposed faults informs the user about the effectiveness of the stimulus generation structure. The detected and exposed fault coverage inform the user about the effectiveness of the response analysis structure. When the respective fault coverage is not sufficient then, by examining the unexposed/undetected faults, the user can modify the embedding to improve the coverage. Chapter 5 illustrates how Logic III(UVic) and the lg3 CAD tool are used to embed a BIST mode of operation into a set of example circuits.

The lg3 user's manual (see Appendix B) lists the commands and syntax of lg3's simulation command language. The following information is provided to aid the BIST embedding designers:

- the fault coverage for the exposed and detected faults,

- the names of the unexposed and undetected faults,
- the fault coverage for every test cycle, and
- the number of gates (used to determine BIST gate overhead).

In addition, the `lg3` tool provides the following:

- direct simulation support for standard BIST stimulus generators, and
- direct simulation support for standard response characterization.

The last two features are useful in helping the designer to experiment with different BIST stimulus and response analysis approaches, so that the BIST embedding goals can be optimized.

LFSMs and binary counters are the most common pattern generators used in BIST embeddings. The `lg3` CAD tool provides direct support for these pattern generators. The user can select the starting state for the generator. The next state function of the LFSMs is also selectable. The next state function with minimum cost maximum cycle is provided by default for the LFSRs and LHCA. By providing the set of pattern generators, the `lg3` user can select the pattern generator and an initial state and then uses the fault simulator to determine its test effectiveness.

The final state of the response analysis circuits using MISRs and (multiple input cellular automaton) MICAs can be calculated by `lg3`. The tool can also perform fault simulation to determine if for a particular MISR or MICA any aliasing occurs. Simulation studies[ZM90] have shown that the aliasing is usually negligible.

3.4.1 Logic III(UVic) Simulation Interface Features

Although the `use` and `update` statements are sufficient to specify a BIST embedding, the user still has to inform the simulation system about the following:

- The location of the BIST observation points.
- The external stimulus used to activate the BIST mode.
- The faults the system should consider.
- The nets to display during functional simulation.

Section 3.4 and Appendix B describe how the external stimulus is specified. Specification of the other information raises the issue of where to put the specification: in Logic III(UVic) or in the simulation control language. For the observation and display nets, adding the specification to Logic III(UVic) allows the nets to be referred to directly. If the specification of the nets being observed or displayed is done in the simulation control language, then the only way to distinguish a net is by its hierarchical name.

Logic III(UVic) is augmented to include the specification of the observation nets and the display nets. This simplifies the specification of the nets, at the cost of sometimes referencing more than one net. An example Logic III(UVic) description specifying observer nets and display nets is:

```
net_module carry( a,b,c : input; carry : output );
var
    ab, ac, bc : node;
observer // the observers go after the declarations
    ab,a;
display
    a,b,c ;
begin
    a2( a, b, ab);
    a2( a, c, ac);
    a2( b, c, bc);
    o3( ab, ac, bc, carry );
end.
```

This syntax has the advantage of being able to easily specify the observation or display nets. Unfortunately, every time a **carry** module is instantiated, the **ab** and **a** nets are also added to the simulator's observation list. This is also true for the display list.

In practice, this has not been a problem for the display nets, since most of the *interesting* nets have had only one instance. Also the nets chosen for observer nets are usually the outputs of the flip-flops used in the BIST embedding and thus every instance of the nets should be an observer.

If the net name chosen to be displayed is an array, then there is a choice of display formats. The formats are shown in Table 3.1.

Table 3.1 Display Formats

bin	binary (largest index first)
binstr	binary (smallest index first)
oct	octal (index 0 is 2^0)
hex	hexadecimal (index 0 is 2^0)
sgndec	two's complement decimal
fix:*i*	decimal fix point with radix in *i*th position
sgnfix:*i*	two's complement decimal fix point with radix in *i*th position

An example of displaying an array as a two's complement fixed point number is:

```
var
     mult : array [1..8] of input;
display
     mult:sgnfix:7
```

Control of which nets that have faults inserted is important for two reasons: 1) The net where the faults are known to be untestable should be marked, so that they can be removed from the fault simulation list. 2) The fault simulation can be made more efficient if only relevant faults are simulated.

Untestable faults are caused by redundant logic. Redundant logic is sometimes required in a design to eliminate glitches. Since the user is normally aware of this untestable logic, providing a method of capturing this information is useful in eliminating the relevant faults from the fault list. Also if during the BIST embedding process a user determines that some logic is redundant, this should be captured. If the testing of a circuit is broken into several

separate test sessions, then only the faults that are considered by the individual test sessions should be included.

The addition of the fault disabling specification to the Logic III(UVic) can be justified by: 1) documenting of redundant logic should be done in the design description, and 2) the specification is easier in the design description than in the simulation control description.

Faults are removed from the fault list with the `faults_off` statement. An example of removing all the faults associated with the x array is:

```
function bit_and( x : array [] of input;
   y : array [1..x'size] of input): array [1..x'size] of output ;
var
    i : integer;
faults_off
    x;
begin
    for i := x'first to x'last do a2( x[i], y[i], bit_and[i] );
end.
```

There are two forms of the `faults_off` statement, the first form, which contains a list of net variables, and the second form which contains no list. The first form removes the faults from the nets listed. The second form removes all the faults from the nets defined inside the module, including any nets defined inside component modules. A typical place that the second form is used is in an embedding module. This `faults_off` statement removes all the faults from the BIST specific circuitry, thus only the faults from the circuitry to be tested are considered. This is important when different BIST approaches are being evaluated.

3.5 BIST Architectures and Logic III(UVic)

The goal of this section is to illustrate the ability of Logic III(UVic) to specify many of the common BIST architectures. The use statement and the associated embedding module are the mechanism in Logic III(UVic) to add a particular BIST architecture to a design. The embedding module receives an array of register inputs and an array of matched register

outputs for the register variables contained in statements in the body of the use statement. Different BIST architectures are realized with different embedding modules.

The circuitry that implements a BIST mode must provide a means of controlling the BIST mode, stimulating the circuit (i.e., applying test patterns), observing the response, and determining the test result. With the exception of the control logic, the Logic III(UVic) code that provides this functionality for several common BIST architectures is presented in the following subsections.

In addition to the embedding module, other modules are required to implement the complete BIST mode. These modules usually implement the controller, provide stimulus and response analysis circuitry and connect the BIST components together.

3.5.1 Scan Based BIST Architectures

One of the most common DFT techniques is to modify the memory elements to incorporate a scan path. The scan path transforms the circuit into a combinational circuit for the purposes of testing. Several BIST architectures are based on scan paths. Examples are LOCST (LSSD On-Chip Self-Test) [ABF90, section 11.4.4], CEBS (Centralized and Embedded BIST Architecture) [ABF90, section 11.4.7] and STUMPS (Self-Testing Using MISR and Parallel SRSG) [ABF90, section 11.4.5](see Section 3.5.2 for STUMPS).

If there is only one scan path in the circuit, then only one global signal is required to switch between normal mode and BIST mode. The Logic III(UVic) definitions for the signals required to implement a scan path are:

```
global
var
        scan_on, scan_in, scan_out    : input;
        scan_chain : array [1..16] of node;
        scan_chain_index : integer;
end
```

The signal **scan_on** converts the memory elements into a scan path when set to 1. The signals **scan_in** and **scan_out** provide external access to the scan path. The elements of

the array **scan_chain** are used to link the scan path register together. The integer variable **scan_chain_index** is an artifact of the implementation used to link up the scan path. Since Logic III(UVic) cannot specify the initial value for a variable, an initialization function is required to set up the global variables necessary for this embedding. This function is called **init_scan**.

The embedding module, that converts all the **reg** variables into elements of the scan chain is:

```
net_module scan_embed (
    obs     : array [1..] of input;
    gen     : array [obs'first..obs'last] of output );
const
    size = obs'size;
var
    i       : integer;
    d_in    : array [obs'first..obs'last] of node;
observer
    gen;
faults_off;
begin
    mux2(scan_on, obs[obs'first], scan_chain[scan_chain_index],
           d_in[obs'first]);
    cdff(d_in[obs'first], gen[obs'first]);
    scan_chain_index := scan_chain_index + 1;
    for i := obs'first+1 to obs'last do begin
           mux2(scan_on, obs[i], gen[i-1], d_in[i]);
           cdff( d_in[i],  gen[i]);
    end;
    connect( gen[obs'last], scan_chain[scan_chain_index] );
end.
```

Note that the **faults_off** directive is used to disable all the faults for this module. In order to allow different BIST embeddings for a circuit to be evaluated, the fault set for the circuit should remain the same. Removing all the faults associated with the BIST specific circuitry results in using only the circuit's faults. This makes the comparison of different embeddings easier.

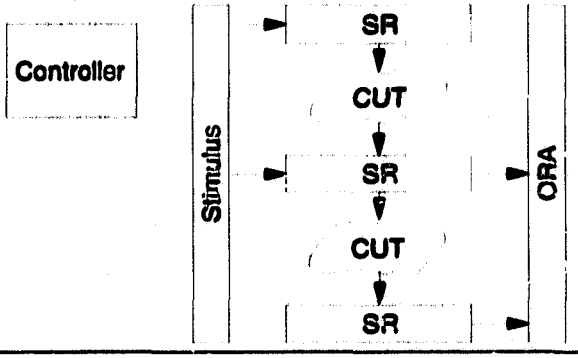
To complete the scan embedding, the signals **scan_in** and **scan_out** must be attached to the scan chain. This is accomplished with the following code:

```
net_module scan_connect_chain( dummy : integer);
begin
    connect( scan_chain[scan_chain_index], scan_out);
    connect( scan_chain[scan_chain'first], scan_in);
end.
```

Once the scan chain is embedded, the LOCST architecture can be easily realized by connecting the `scan_in` signal to a LFSR, and the `scan_out` signal to a LFSR.

3.5.2 Self-testing Using MISR and Parallel SRSG (STUMPS) Architecture

Figure 3.4 STUMPS Architecture



Another popular BIST architecture, based on scan chains, is the STUMPS (Self-testing Using MISR and Parallel SRSG) [ABF90, section 11.4.5] architecture. Figure 3.4 shows the block diagram for this architecture. Unlike the following architectures the circuit is not tested every clock cycle, but instead a test consists of a scan phase followed by an application of one test vector. This process is repeated for every test vector. The stimulus is generated by a LFSR and the response is compacted by a MISR with only the necessary number of data taps.

The control signals required by the STUMPS embedding are:

```
global
var
    stumps_gen : array [1..16] of node;
    stumps_obs : array [1..16] of node;
    stumps_gen_index, stumps_obs_index, stumps_max_size : integer;
    stumps_shift : input;
end.
```

The pattern generator is attached to the stumps_gen signals. The response analyzer is attached to the stumps_obs signals.

The STUMPS embedding module is:

```
net_module stumps_embed(
    obs     : array [] of input;
    gen     : array [obs'first..obs'last] of output );
faults_off;
begin
    stumps_max_size := max(stumps_max_size, obs'size);

    n_sh_register( stumps_shift, stumps_gen[stumps_gen_index],
            obs, stumps_obs[stumps_obs_index], gen );
    stumps_gen_index := stumps_gen_index + 1;
    stumps_obs_index := stumps_obs_index + 1;
end.
```

The component module, n_sh_register, realizes a n-bit shift register.

Two auxiliary modules provide the pattern generator and the response analyzer. They are:

```
net_module stumps_generator_ctrl(gsize : integer );
var
    pat : array[1..gsize] of node;
    i,j : integer;
faults_off;
begin
        lfsr(pat);
        j := pat'first;
        for i := stumps_gen'first to stumps_gen_index-1 do begin
                connect( pat[j], stumps_gen[i] );
                j := j + 1;
        end;
end.
```

```
net_module stumps_observer( osize : integer );
var
        sig : array[1..osize] of node;
display
        sig, stumps_obs[1..stumps_obs_index-1];
observer
        sig;
begin
        part_misr(stumps_obs[1..stumps_obs_index-1], sig);
end.
```

The `stumps_generator_ctrl` module instantiates a `lfsr` module and attaches the `stumps_gen` signals to the LFSR outputs. The `part_misr` module is a MISR where only some of the memory elements receive external inputs.

3.5.3 Built-In Evaluation and Self-Test (BEST) Architecture

Figure 3.5 BEST Architecture

The built-in evaluation and self-test (BEST) [ABF90, section 11.4.2] BIST architecture shown in Figure 3.5 applies the test stimulus to the inputs of the kernel in parallel and observes the response in parallel. For this architecture to be directly embedded the kernel must be surrounded by input and output registers. An example of a functional circuit that can use this architecture is:

```
net_module fred( x : array [] of input ; y : array of output );
var
    rx, ry : array [x'first..x'last] of node;
begin
    n_reg(x,rx); kernel(rx,ry); n_reg(ry,y);
end.
```

In this architecture, the stimulus and response analysis structures are distinct, therefore two different embedding modules are required. To allow the modules to be switched from BIST mode to normal mode the following global signals are required:

```
global
var
    gen_on : array [1..16] of input;
    gen_on_index : integer;
    misr_on :  array [1..16] of input;
    misr_on_index : integer;
end.
```

The elements of the array of primary input signals, gen_on control the BIST/normal mode of the stimulus embedding modules. Elements in misr_on control the response modules. The stimulus and response observation modules are:

```
// stimulus structure
net_module gen_embed(
    obs     : array [1..] of input,
    gen     : array [obs'first..obs'last] of output );
const
    size = gen'size;
var
    d_s,merge: array [gen'first..gen'last] of node;
    i        : integer;
faults_off;
begin
    mux2(gen_on[gen_on_index],obs[gen'first],
         gen[gen'last],d_s[gen'first]);
    dff(d_s[gen'first], gen[gen'first]);
    for i := gen'first+1 to gen'last do begin
        if lfsr_tap(size,i) = 0 then
            connect(gen[i-1], merge[i]);
        else
            exor(gen[i-1], gen[gen'last], merge[i]);
        mux2(gen_on[gen_on_index], obs[i], merge[i], d_s[i]);
        dff(merge[i], gen[i]);
    end;
    gen_on_index := gen_on_index + 1;
end.
```

```
// response analysis structure
net_module misr_embed(
    data    : array [0..] of input;
    sig     : array [data'first..data'last] of output );
const
    size = data'size;
var
    i       : integer;
observer
    sig;
faults_off;
begin
    on_misr_me(misr_on[misr_on_index],
sig[sig'last],data[0],sig[0]);
    for i := 1 to data'last do
        if lfsr_tap(size,i) = 0 then
            on_misr_me(misr_on[misr_on_index],
sig[i-1],data[i],sig[i])
        else
            on_misr_tap_me(misr_on[misr_on_index],
                sig[i-1], data[i], sig[sig'last], sig[i]);
    misr_on_index := misr_on_index + 1;
end.
```

The function `lfsr_tap(size,i)` returns 1 if a tap that implements a minimum cost LFSR is required in that position of the LFSR. To embed this architecture in the module `fred`, the following modifications are performed:

```
net_module fred( x : array [] of input ; y : array of output );
var
    rx, ry : array [x'first..x'last] of node;
begin
     use gen_embed in n_reg(x,rx);
     comb_part(rx,ry);
     use misr_embed in n_reg(ry,y);
end.
```

To complete the BIST embedding the BIST controller and the circuit to check the MISR's final state are required.

Since a circuit is usually composed of several kernels, individual embedding modules are required to allow each kernel to be tested in parallel. Figure 3.6 shows an example of a circuit with two kernels. In this architecture both kernels are tested simultaneously.

The embedding module for both generation and observation is:

Figure 3.6 Multiple Parallel Pattern Generation and Response Analysis Architecture


```
net_module gen_misr_embed(
    obs     : array [0..] of input;
    gen     : array [obs'first..obs'last] of output );
var
    sig     : array [obs'first..obs'last] of node;
observer
    sig;
begin
    gen_embed(obs, gen);
    misr(gen, sig);
end.
```

Note that the MISR outputs are declared as observers, thus informing the simulation system to use these outputs to detect faults. Also note that the `faults_off` declaration is used to remove the faults from the BIST specific circuitry. Since part of the logic of the BIST mode is untestable, turning off the faults in this logic allows `lg3` to remove these untestable faults from consideration.

If the `embed_misr` module is used instead of the `gen_misr_embed`, then the BIST specific circuitry is reduced. This style of BIST architecture is called HILDO [BM84]. Since the patterns produced by the MISR depend on its internal state and the circuit driving its input, its test effectiveness can be worse than a LFSR (i.e., output patterns can repeat with the MISR, while each output pattern is unique for a LFSR).

3.5.4 BILBO BIST Architecture

This BIST architecture is introduced by Konemann [KMZ80] in 1980. A BILBO register provides four modes of operations:

- parallel load mode,

- shift mode,
- pattern generation/response analysis (a MISR) mode, and
- a clear mode.

During BIST mode a BILBO register is placed in one of these modes, depending on its placement and the type of tests performed.

The control signals for the embedded BILBO registers are:

```
global
var
    bilbo_sin : array [1..16] of input;
    bilbo_sin_index : integer;
    bilbo_sout: array [1..16] of output;
    bilbo_sout_index : integer;

    bilbo_b1, bilbo_b2 : array[1..16] of input;
    bilbo_index : integer;
end.
```

The pair of elements in the **bilbo_b1** and **bilbo_b2** control the mode for one of the embedded BILBO registers. The signals **bilbo_sin** and **bilbo_sout** are the shift-in and shift-out ports for a BILBO register. The embedding module for a BILBO register is:

```
net_module bilbo_embed(
    obs     : array [1..] of input;
    gen     : array [obs'first..obs'last] of output );
const
    size = obs'size;
var
    taps,s  : array [1..size] of node;
    feedback: node;
    i, no_taps : integer;
observer
    gen;
faults_off;
begin
    no_taps := 1;
    for i := 1 to obs'last do begin
        bilbo_me(obs[i], s[i-1], gen[i], s[i]);
        if lfsr_tap(size,size-i) = 1 then begin
            connect(s[i], taps[no_taps] );
            no_taps := no_taps + 1;
        end;
    end;
    Xor(taps[1..no_taps-1], feedback);
    mux2(bilbo_bi[bilbo_index], bilbo_sin[bilbo_sin_index],
         feedback, s[0]);
    bilbo_sin_index := bilbo_sin_index + 1;
    ii( s[obs'last], bilbo_sout[bilbo_sout_index]);
    bilbo_sout_index := bilbo_sout_index + 1;
    bilbo_index := bilbo_index + 1;
end.
```

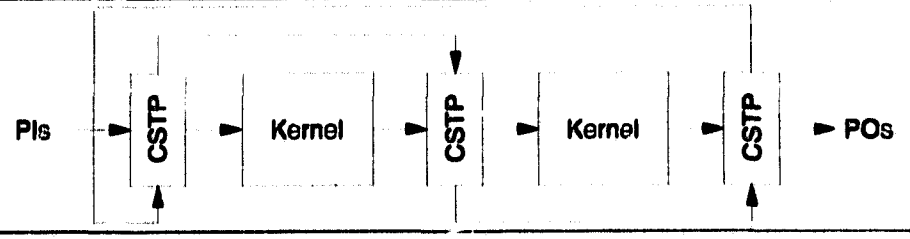
This embedding module illustrates that quite complex "testing structures" can be embedded with the Logic III(UVic) use statement. The embedding modules used in the BEST architectures can be considered optimized versions of a BILBO register. Thus in a particular test, if only one of the BILBO modes is necessary, then one of the embedding modules for the BEST architecture can be used.

The shifting mode of the BIST registers can be used to scan out the final state of a test, and check it with the known good final state. Since many different test schedules are possible, different BIST test controller need to be designed for each circuit.

3.5.5 Circular Self-Test Path Architecture

The CSTP (Circular Self-Test Path) BIST architecture combines the circuitry that performs the stimulus and response analysis functions. In CSTP, the circuit is converted into an AFSM and a test is performed by initializing the machine's state, and cycling the machine for a fixed number of cycles. The final state is used to determine the outcome of the test. Figure 3.7 shows an example of this architecture, where the normal registers have been converted into CSTP registers.

Figure 3.7 CSTP Architecture



The only global signal required by this architecture is the signal that switches the registers from normal mode to CSTP mode. When the signal **cstp_on** is set to 1 the test mode is activated. The signals in array **cstp_chain** are used to link the CSTP registers together.

```
global
var
    cstp_on     : input;
    cstp_chain : array [1..16] of node;
    cstp_chain_index : integer;
end.
```

The CSTP embedding module is:

```
net_module cstp_embed (
    obs    : array [1..] of input;
    gen    : array [obs'first..obs'last] of output );
const
    size = obs'size;
var
    i      : integer;
observer
    gen;
faults_off;
begin
    // link up the previous CSTP register
    cstp_me(obs[obs'first], cstp_chain[cstp_chain_index],
            gen[obs'first]);
    cstp_chain_index := cstp_chain_index + 1;
    for i := obs'first+1 to obs'last do
        cstp_me(gen[i-1], obs[i], gen[i]);
    connect( gen[obs'last], cstp_chain[cstp_chain_index] );
end.
```

The memory element for this architecture is:

```
net_module cstp_me(ch, d: input; q : output);
var
    d_in, t1 : node;
faults_off;
begin
    a2(cstp_on, ch, t1);
    exor(t1, d, d_in);
    cdff(d_in, q);
end.
```

Notice that the **cstp_on** signal connects the D flip-flop to the EXOR of its normal input and the previous cell in the circular test path.

To complete the CSTP embedding, the following module connects the last memory element to the first, thus completing the circle.

```
net_module cstp_connect_chain;
begin
    connect(cstp_chain[cstp_chain_index],
            cstp_chain[cstp_chain'first]);
end.
```

3.6 BIST Library Organization

The Logic III(UVic) modules in the preceding sections implement a variety of BIST architectures. These modules and modules that describe specific BIST structures and general logic structures have been grouped together into a library that supports BIST embedding. Appendix C gives the complete Logic III(UVic) source for the library.

The utility modules provide the following components:

`And(x:array [] of input; a:output)`, creates a `x'size` input circuit that performs an AND.

`Or(x:array [] of input; o:output)`, creates a `x'size` input circuit that performs an OR.

`Xor(x:array [] of input; o:output)`, creates a `x'size` input circuit that performs an XOR.

`bitwise_and(x,y:array [] of input; xy: [] of output)`, creates a circuit that performs a bitwise AND on the arrays x and y to produce xy (i.e., `x[i] · y[i] = xy[i]`).

`bitwise_or(x,y:array [] of input; xy: [] of output)`, creates a circuit that performs a bitwise OR on the arrays x and y to produce xy.

`bitwise_xor(x,y:array [] of input; xy: [] of output)`, creates a circuit that performs a bitwise XOR on the arrays x and y to produce xy.

`n_mux2(sel:input; x,y:array [] of input; z:array [] of output)` creates an n-bit multiplexor.

`tc_adder(a,b:array [] of input; cin:input; s:array [] of output; c:output)` creates an n-bit adder.

`n_add_sub(as_ctrl:input;a,b:array [] of input; s:array [] of output)` if `as_ctrl = 1` then `s = a - b` else `s = a + b`.

The following functions and modules are provided to directly support BIST response analysis and stimulus generation structures. These modules are used extensively in the realization of the BIST architecture modules.

`function lfsr_tap( size, index : integer ) : integer;` returns 1 if a tap must be added at position `index` of a `size`-bit internal LFSR. This resulting maximum cycle LFSR has a minimum number of taps.

`function embed_state( p, i : integer) : integer;` returns 1 if position `p` of a register's initial state should be 1. Argument `i` specifies the type of assignment (see Section 5.2.2).

`lfsr( pat : array [] of output )` creates a minimum cost maximum cycle internal LFSR.

`z_lfsr( pat : array [] of output )` creates a minimum cost maximum cycle internal LFSR that also generates the all zero state.

`init_lfsr( init:integer; pat:array [] of output )` creates a minimum cost maximum cycle internal LFSR where the initial state is specified by `init`.

`misr( data:array [] of input; sig: array [] of output )` creates a MISR using the `lfsr_tap` function to determine the feedback function. The input and output sizes have to agree.

`part_misr( data:array [] of input; sig: array [] of output )` creates a MISR using the `lfsr_tap` function to determine the feedback function. The MISR's size is given by the size of the `sig` array. Only `data'size` input taps are added.

`n_sh_register(sh_ctl,sin:input; din: array [] of input;`
`sout:output; dout: array [] of output)` creates a scannable register.

The previous library modules are all used in the construction of the user visible embedding modules. There are four types of modules or functions required to specify the

embedding of a particular architecture. The types are: 1) initialization functions, 2) embedding modules, 3) auxiliary modules, and 3) cleanup modules.

Initialize functions are more an artifact of the implementation approach. These functions are required to setup the global variables used in the creation of embeddings. The embedding modules specify how the **reg** variables are instantiated via the **use** statement. The cleanup module is used to finish off the embedding (e.g., connect the first and last elements of a scan chain to the external nets). Some BIST architectures require extra modules to complete the embedding, these are referred to as auxiliary modules.

Table 3.2 lists the types and names of the modules that support BIST embedding. An example of embedding the STUMPS architecture is:

```
begin
    assert( init_stumps(1) ); // init function
    ...
    use stumps_embed in
        // user's design containing reg variables
    ...
    stumps_generator_ctrl( 16 ); // aux
    stumps_observer( 16 );       // aux
    finish_stumps(1);            // finish
end.
```

The case studies chapter presents more examples of using this library. Note that any combination of these embedding can be used together.

Table 3.2 BIST Embedding Library

Architecture	init	embed	aux and finish
combinational	init_comb	comb_embed	finish_comb
scan	init_scan	scan_embed	finish_scan
STUMPS	init_stumps	stumps_embed	finish_stumps stumps_generator_ctrl stumps_observers
CSTP	cstp_scan	cstp_embed cstp_scan_embed	finish_cstp
BEST	init_best	gen_embed misr_embed gen_misr_embed	finish_best
BILBO	init_bilbo	best_bilbo	finish_bilbo

Chapter 4

Implementation of Lg3

4.1 Why Object-Oriented Design?

The success or failure of a CAD system depends directly on the services provided by the system. CAD systems are realized by software systems running on computers. The ability of a CAD system to change to satisfy its users and its designers depends directly on the software design methodology and programming language used to realize the system. This is especially true if the functionality of the CAD systems is still being developed.

Use of the object-oriented methodology for CAD systems has been suggested by Wolf in [Wol91, Wol89, Wol87]. [Wol91] is a tutorial article on object-oriented programming and it illustrates several possible applications. A more concrete example of using object-oriented programming for building hardware description and measurement systems is described in [Wol89] and [Wol87]. In [GCG+89], Gupta, Cheng, Gupta, Hardonag, and Breuer discuss the effectiveness of using an object-oriented database management system to support rapid prototyping. In [Che87], Cherry reports the experience of using the object-oriented language *flavors* in developing a CAD tool that supports: schematic capture, electric level simulation, switch level simulation, virtual grid symbolic layout with compaction and pitch-matching, automatic standard cell layout generation, floorplanning, and comparison of layout and

schematic designs. Cherry states that: "Object-oriented programming, which allows a diverse collection of objects to exhibit a generic behaviour made the integration of a large, diverse collection of tools easier."

The requirement to support rapid prototyping and to create a large CAD program suggested the use of object-oriented design [Boo91, Cop92, Bud92] and programming in the development of the lg3 CAD tool. The C++ programming language support for object-oriented design and efficient imperative procedures made it the logical choice for the implementation language. This choice allowed rapid changes to be made to the lg3 CAD tool as new approaches for solving the BIST embedding problem developed.

Object-oriented programming is defined by Booch [Boo91, page 36] as:

> "Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

Objects can be considered as entities that have a well defined behaviour. A class is a set of objects that share the same behaviour and structure. An object is an instance of a class.

Identifying which parts of the problem area correspond to objects and classes is a crucial step in object-oriented design. Booch's overview of the object-oriented design process [Boo91, page 190] is:

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects.

Coplien[Cop92, page 204] also states that:

> "For good object-oriented design, we want application domain *entities* to map onto objects, and types to map onto classes of the solution domain, ... Application domain entities are notions that we care about in our applications: transactions, money, accounts, and tellers in a banking system; phones, calls, lines, and switches in a telecommunication system."

The purpose of this chapter is to briefly outline the design and implementation of **lg3** by identifying the object and class organization of the program that implements the CAD tool and by describing the most important behaviour realized by each class. The term, *method*, refers to the program code that realizes the specific behaviour for an object.

The secondary motivation for this chapter is to discuss the implementation issues of the language features introduced into Logic III to produce Logic III(UVic).

The design is composed of two major components: the Logic III(UVic) parser and netlist translator and the functional and fault simulator. The first component is responsible for parsing and interpreting Logic III(UVic). It produces a flattened netlist, simulation setup and control commands for the simulation system. The simulation system produces functional and fault simulation results to enable **lg3** users to evaluate their designs.

4.2 Logic III(UVic) Parser and Netlist Translator

4.2.1 Parser

In order to identify the classes and objects in the parser design, the application domain entities must be examined. Some of the application domain entities of the Logic III(UVic) language are:

- **net_module**, **function**, and **circuit** modules,

- the customizing statements: if-then-else, while, for, assignment,
- the binding and instantiating statements,
- the data types of `string`, `integer`, `input`, `output`, `node`, and `reg`,
- one dimensional arrays of the base data types, and
- expressions that act on the data types.

In the implementation of `lg3` the following entities of the Logic III(UVic) language (i.e., the application domain) are represented by C++ classes:

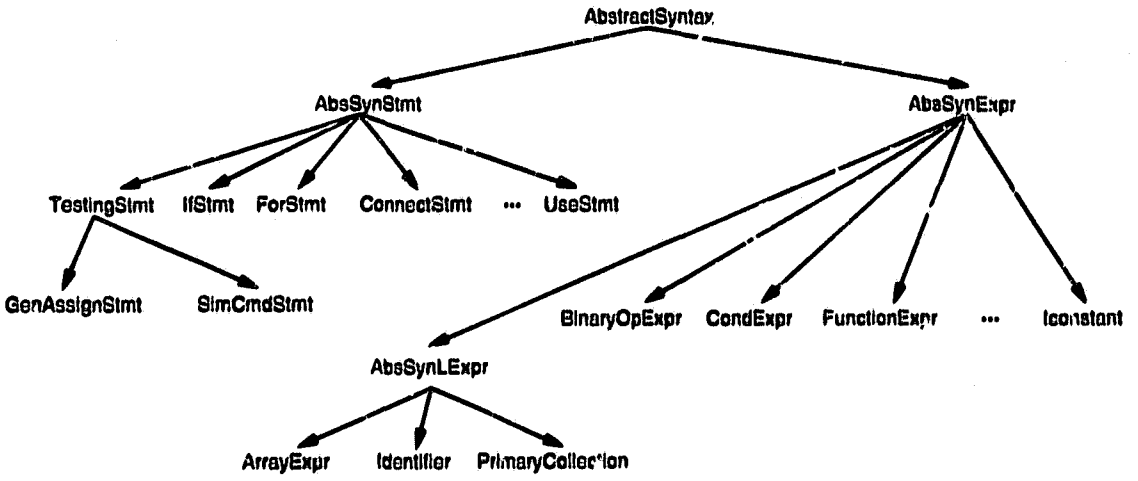
- the syntax of statements and expressions,
- the data types,
- the "values" of the variables, and
- the interface and implementation of modules and functions.

In parsing and interpreting a Logic III(UVic) description, `lg3` follows a common practice in compiler design of creating an abstract syntax description of the design text. The parser and lexical analyzer functions are generated by the compiler tools `bison`[DS91] and `flex`[Pax90]. `bison` accepts a context free grammar with actions associated with each grammar rule and produces a C-language function that parses the input and executes the actions when the associated grammar rule is recognized. `flex` accepts a set of regular expressions and associated C-program statements as input and produces a function that executes the actions whenever the associated regular expression matches with the character input. The function produced by `bison` uses the function produced by `flex` to recognize tokens in the language.

Most of the actions specified in the bison grammar file create the abstract syntax objects from the parsed text. The class `AbstractSyntax` is the base class for the abstract syntax

objects. Objects of this class only know how to print a representation of themselves. The **print** method is used primarily to ensure that the grammar used to generate the parser is correct. Figure 4.1 shows the class hierarchy used to represent Logic III(UVic) constructs.

Figure 4.1 Abstract Syntax Class Hierarchy



The bodies of modules and functions in Logic III(UVic) consist of statements and expressions. Statements and expressions are represented by classes **AbsSynStmt** and **AbsSynExpr**, respectively. A specialization of **AbsSynExpr** to form **AbsSynLExpr** is used to distinguish between expressions that refer to values and expressions that can be modified or bound (i.e., nets are bound to gates).

Every statement in Logic III(UVic) is represented by a class derived from the class **AbsSynStmt**. The mappings from the classes to the particular abstract syntax statements are presented in Table 4.1. In the grammar rule column, all the upper case words beginning with a **T** represent a token, and all the lower case words are nonterminals.

In addition to the **print** member function, each class has an **evaluate** member function. The evaluate function implements the semantic action associated with that statement. For example, the **evaluate** member function of an object of class **WhileStmt** evaluates its condition and then evaluates its body as long as the condition remains true.

Table 4.1 Class for Abstract Syntax Statements

Class	Grammar rule
StmtList	T_BEGIN stmt_list T_END
ConnectStmt	T_CONNECT '(' primary ',' primary ')'
BindStmt	T_IDENT '(' expr_list ')'
UseStmt	T_USE T_IDENT T_IN stmt
WriteStmt	T_WRITE '(' expr_list ')'
AssertStmt	T_ASSERT '(' cond_expr ')'
IfStmt	T_IF cond_expr T_THEN stmt
IfElseStmt	T_IF cond_expr T_THEN stmt T_ELSE stmt
AssignStmt	primary T_ASIGNOP expr
RegUpdateStmt	primary T_REG_UPDATE expr
WhileStmt	T_WHILE cond_expr T_DO stmt
ForStmt	T_FOR primary T_ASIGNOP expr T_TO expr T_DO stmt

The advantages of creating the above class hierarchy are demonstrated by describing how a new statement can be added to Logic III(UVic). A new statement, *NS*, can be incorporated into the language by adding the grammar rule to the bison description that parses the concrete syntax and creates a *NS* object. The final step is the definition of the the **print** and **evaluate** methods. The method **print** should print a representation of the statement, and the method **evaluate** should interpret the semantic action associated with the statement. For example, adding a Pascal like *repeat-until* statement requires the following grammar rule:

```
stmt: T_REPEAT stmt T_UNTIL cond_expr
    {
        $$ = new RepeatStmt( $2, $4 );
    }
    ;
```

where **T_REPEAT** and **T_UNTIL** are tokens returned by the lexical analyzer, **stmt** is a nonterminal representing a language statement, and **cond_expr** is a nonterminal representing a boolean condition. The action creates a **RepeatStmt** object. The objects representing the loop's body and condition are passed to the **RepeatStmt** constructor [1].

[1] A constructor in C++ creates and initializes an object of the given type.

Every expression in Logic III(UVic) is represented by a class derived from **AbsSynExpr**. The expressions derived from **AbsSynExpr** are given in Table 4.2.

Table 4.2 Classes for Abstract Syntax Expressions

Class	Grammar rule
NegateExpr	'-' expr
BinaryOpExpr	expr bin_op expr
Log2Function	T_LOG2 '(' expr ')
MinFunction	T_MIN '(' expr ')
MaxFunction	T_MAX '(' expr ')
FunctionExpr	T_IDENT '(' expr_list ')
PropertyExpr	expr ''' T_IDENT
Iconstant	T_ICONSTANT
Sconstant	T_SCONSTANT
Identifier	T_IDENT
ArrayExpr	T_IDENT '[' expr ']'
PrimaryCollection	'[' primary_list ']'
CondExpr	T_NOT cond_expr
	cond_expr cond_op cond_expr
RelBinaryOpExpr	expr rel_op expr

The class **AbsSynExpr** provides the following methods, in addition to the **print** method inherited from the base class **AbstractSyntax**.

evaluate(Variable &val) evaluates the expression and returns its result in **val**. The **Variable** class is ultimately responsible for implementing the appropriate action. A discussion of the class **Variable** follows later in this section.

type() returns a pointer to an object of class **SymbolInfo**. This method is used to check the compatibility of the expression's type with the operation.

Classes **Identifier**, **PrimaryCollection**, and **ArrayExpr** are all derived from **AbsSynLExpr**. The **L** in the name **AbsSynLExpr** refers to the C programming term *lvalue*. An *lvalue* is an expression that can appear on the left hand side of an assignment operator

(i.e., it refers to something that can be modified). A class derived from **AbsSynLExpr** has to implement the method **modify(const Variable &v)**. The value of the object **v** is used to modify the recipient of the method.

Logic III(UVic) supports the basic type of integer, string and net. It also support one-dimensional arrays of these types. The class **SymbolInfo** is used to represent this type information. All of the operations on the base types are implemented by the class **Variable**.

The choice of isolating all the "data" manipulation operations to the **Variable** class allows easy addition of some data types. Logic III(UVic) uses the integer and string data types to control the customization of the **net_modules** and **function** modules. The advantage of this organization is demonstrated by describing how a bit vector data type can be added to the **Variable** class. A bit vector type improves the language's ability to specify customizations.

To add this data type, the classes **Variable** and **SymInfo** need to be modified. Since the Logic III(UVic) syntax is not changed, the bit vector operations are expressed in terms of the current syntax (e.g., a * b means bitwise AND). If the new data type's operations cannot use the current syntax, then additions to the bison grammar file and deriving a new class from the **AbsSynExpr** class are necessary.

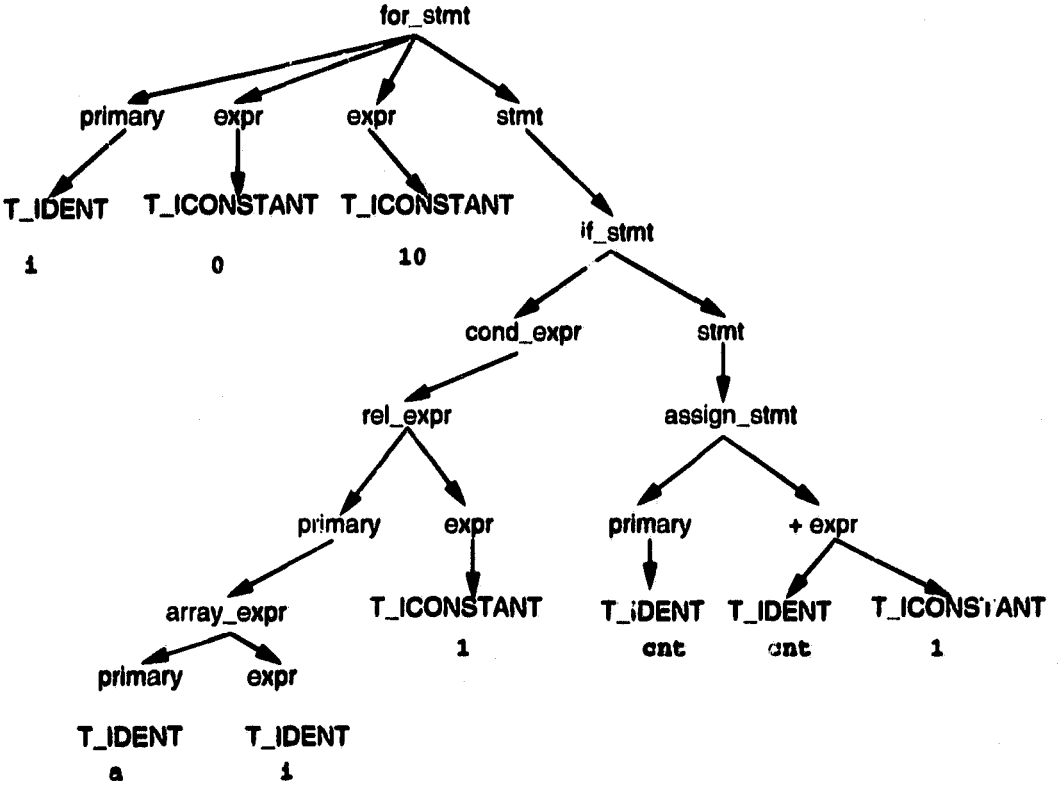
The following example illustrates the evaluation of a Logic III(UVic) program that counts the number of occurrence of 1 in the integer array a.

```
for i := 0 to 10 do
    if a[i] = 1 then
        cnt := cnt + 1;
```

Figure 4.2 shows the abstract syntax diagram for this example. Figure 4.3 shows the classes of the objects created by the parser. Invoking the **evaluate()** method on the *for-stmt* object causes the initialization expression to be evaluated and assigned to the variable **i**, the termination expression is then evaluated and the current value of **i** is compared to the termination expression. If **i** is less than the termination expression then the **stmt**

is evaluated. Next, 1 is added to `i` and the loop is repeated. The actions of the other statements and expressions are invoked with their `evaluate` method.

Figure 4.2 Abstract Syntax for Logic III(UVic) Example

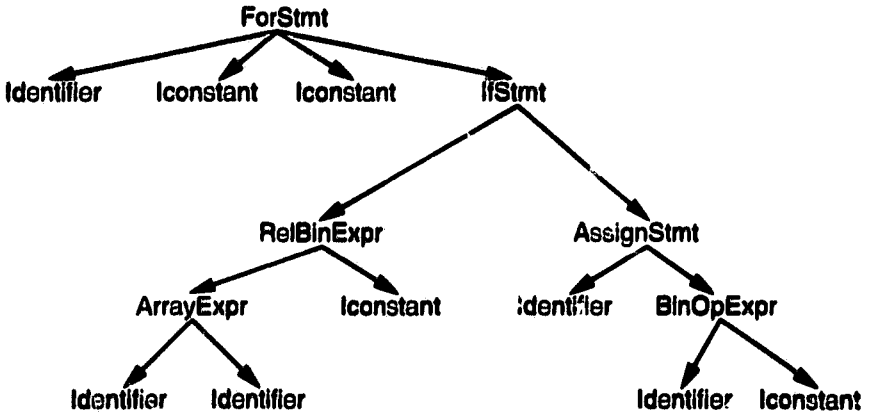


4.2.2 Netlist Translator

Logic III(UVic) describes the structure of digital circuits in terms of a hierarchy of modules and customizable modules. Ultimately this hierarchical description must be flattened into a netlist of primitive modules. In Logic III(UVic), a primitive module is a module for which its body is defined as `external`.

The requirement to flatten the hierarchical description comes from two sources:

Figure 4.3 Classes for Logic III(UVic) Example



- The requirement to build the design. Standard cell ICs and field programmable gate arrays (FPGAs) implementation technologies are constructured using primitive *gates.* For example, the primitive modules `a2`, `ai2`, ..., `exor` used in Chapter 5 are cells from a CMOS cell library.

- The requirement to model the functional and fault behaviour at an appropriate level. The single stuck-at fault model is based on modeling faults at the interconnections of primitive gates. The ISCAS85 and ISCAS89 benchmarks used in testing research are netlists of primitive gates.

4.2.2.1 Netlist Objects

The three easily recognizable *objects* in a netlist are:

gates representing entities that map input signals to output signals,

ports representing the interface between a gate's inputs and outputs and the external world, and

nets representing the entities that allow information to be transferred between gates (i.e., wires).

The ports and gates can be considered as one object. That is, the input and output ports of a gate are simply attributes of a gate.

A hierarchical netlist allows gates to be defined in terms of a netlist in which its external nets are attached to the ports of the gate. A flattened netlist contains gates where only the ports of the gates are defined.

In order for a Logic III(UVic) design to be realized as an application specific integrated circuit (ASIC) or an FPGA, or it's functional and test effectiveness to be analyzed it must be translated into a netlist[2].

The flattened netlist is created by invoking the `evaluate` method on the topmost node of the abstract syntax tree. The netlist is represented by a `Lg3Gate` class and a `Lg3Signal` class. Each `Lg3Gate` object references the `Lg3Signal` signal objects that are attached. Each `Lg3Signal` object also references the attached `Lg3Gate` objects.

In addition to the connectivity information, an object of the `Lg3Gate` class contains the following information:

gate_type, the name of the primitive gate (e.g., `exor`),

instance_name, the name that uniquely identifies this particular gate, and

attributes, a table of attribute names and associated parameters. The attribute table is the method used to communicate information to the simulation component of the `lg3` CAD tool. The only defined attribute is "`gt`", which specifies the behaviour of this gate.

[2]Actually, the function and test effectiveness of circuits can be analyzed from a hierarchical netlist, but at the price of distancing the fault or functional analysis further from the circuit's physical realization.

Lg3Signal object contains the connectivity information, the instance name and a flag indicating if there are any faults associated with this net. The instance name provides the link between the functional and fault simulation results and the text of the Logic III(UVic) design.

4.2.2.2 Evaluation and Flattening

The process of evaluating the Logic III(UVic) statements to produce the flattened netlist can be divided into statement evaluation that results in customizing the design and statement evaluation that create the **Lg3Signal** and **Lg3Gate** objects.

Evaluation of the following objects performs customization: **IfStmt**, **IfElseStmt**, **WhileStmt**, **ForStmt**, and **AssignStmt**. Expressions that evaluate to integers or strings are also part of the customization process.

Evaluation of objects of the following classes create the flattened netlist: **BindStmt**, **ConnectStmt**, **UseStmt**, and **RegUpdateStmt**. An important point to note is that each object of these classes corresponds directly to a Logic III(UVic) construct in the design description. Thus there is a one-to-one mapping between each construct and a set of **Lg3Signal** and **Lg3Gate** objects.

Evaluation of a **ConnectStmt** associates two net variables or an array of net variables with the same wire or array of wires. When a **RegUpdateStmt** object is evaluated, an association between a net and a port of an embedding module is created.

The actual creation of **Lg3Gate** objects occurs during evaluation of **BindStmt** objects. If the binding statement's module (see Section A) is primitive, then a **Lg3Gate** object corresponding to that module is created. The assigned instance name of the **Lg3Gate** depends on the *binding context*. The binding context identifies where in the module hierarchy this gate occurs.

The creation of new **Lg3Signal** objects is not reasonable at this point because of the semantics of the **connect** statement. Instead, **Lg3Signal** objects can only be created when

the Logic III(UVic) variables that reference them have disappeared out of scope. If a **Lg3Signal** object is created whenever the associated net variable first appeared in the binding statement, then the problem of merging two **Lg3Signal** objects needs to be handled. The problem can be illustrated with the following Logic III(UVic) fragment:

```
a2(x,y,z);      // [1]
o2(a,b,c);      // [2]
connect(x,a);   // [3]
```

If **Lg3Signal** objects are created for **x** at statement **[1]** and **a** at statement **[2]**, then when the **connect** statement is evaluated one of the objects has to be destroyed. Matters are made even more complicated when the two binding statements occur in different modules.

The semantics of the **connect** statement that allow Logic III(UVic) variables to be aliased at any point of the evaluation process considerably complicates the **Lg3Signal** object generation. A modification of the **connect** statement's semantics to one similar to the aliasing declaration of VHDL [IEE88, page 4-13] would simplify the implementation.

If the module in the binding statement is not primitive, then evaluation of this module consists of the following steps:

1. Evaluate all the integer and string expression in the argument list.
2. Create the activation record for the local variables.
3. Bind the local nets in the parameter list with their local names.
4. Update the binding context (e.g., add this module to the hierarchical name).
5. Evaluate the statements in the body.
6. Create any **Lg3Signal** objects that are referenced by local net variables.

Note that it is the last step of evaluating a nonprimitive module that creates the **Lg3Signal** objects. Thus only variables of type **node** are created when evaluation of the binding statement is finished.

4.3 Simulator

The requirements of the simulator component of the Logic III(UVic) tool are as follows:

- functional and fault simulation of designs specified by the Logic III(UVic) description;
- using the Logic III(UVic) net names in reporting the response of functional and fault simulations; and
- supporting the common BIST pattern generators and response analyzers.

In addition, the simulator should be as fast as possible to allow the user to experiment with different embeddings.

A possible set of classes for objects of the simulator domain include the following:

gates, boolean functions or memory elements,

signals, the interconnection of gates,

faults, models of the types of incorrect behaviour,

fault_info, statistics gather during fault simulation,

signal_values, the physical signal values (i.e., 0V=0, 5V=1),

generators, sources of stimulus for the circuit,

observers, techniques to analyzing the output of circuit,

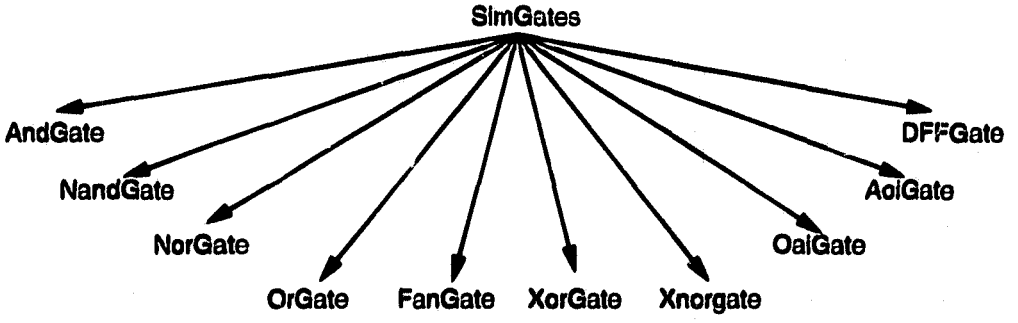
simulator, an entity that given a netlist, a set of generators and observers simulates the behaviour of system, and

fault simulator, an entity that given a netlist, a set generator and observers, and a fault list simulates the effects of the faults on the circuit.

The gates in the **lg3** simulator are required to model the behaviour of the primitive set of gates used to realize digital designs. These gates usually consist of the following: AND, OR, NOT, EXOR, AOI and OAI. The modeled gates produce output signals as a function of their input signals. The class **SimGate** is used as a base class to capture the common behaviour of mapping inputs into outputs. The method **evaluate** directs the object to recalculate its output as a function of its input.

Since all gates are derived from the **SimGate** base class, the class responsible for performing the actual simulation need only be aware of this method. It is the responsibility of the developer to create a derived gate for each primitive gate. The inheritance tree of simulation gates is shown in Figure 4.4. The main advantage of this approach is that different gate types can be easily added by simply deriving the new gate from the **SimGate** base class and implementing the **evaluate** method.

Figure 4.4 Simulator Gates



The interconnection of **SimGate** objects is represented by the **Signal** class objects. Thus **Signal** class objects represent the physical wire interconnecting the gates. A **Signal** class object contains the simulated value of the wire. In addition to storing the wire's value, the **Signal** class objects are used by the simulator to determine the evaluation order of the **SimGate** objects and to determine where the faults are inserted. The flattened netlist consisting of **Lg3Gate** and **Lg3Signal** objects are used to create matching **SimGate** and **Signal** objects.

The `SignalValue` class is responsible for representing and performing operations on the signal values. The current implementation uses the binary values of 0 or 1 to represent the signal. A multivalued representation could be used by simply redefining this class.

The `SimGate` class illustrates the effectiveness of using inheritance of a base class with a given set of methods to provide flexibility in adding functionality to the simulator. In terms of C++ , this base class is called an *Abstract Base Class* [Str91, page 192]. The added functionality in this case is the creation of new primitive gates.

The classes for the pattern generators and observers are also implemented as base classes with a set of methods that define their interface. Since stimulus generation and response observation are the two main components of any BIST embedding and BIST research deals with creating these components, providing an abstract base class for these two classes allows the necessary flexibility. Table 4.3 lists the currently supported generator and observer types.

Table 4.3 Derived Classes for Pattern Generators and Response Analyzers

Class	Pattern Generator	Class	Output Observer
CaGen	LHCA	Comp (MISR, MICA)	Signature analyzer
CountGen	Binary counter	Run	DFA
LfsrGen	LFSR	UpDn (up/down counter)	DFA
TwoDCaGen	Two-Dimension LHCA		

The `PatGen` class for pattern generators defines the methods: `pattern`, `next_pattern`, `first_pattern`, and `more_pattern`. The method `pattern` returns the current pattern. The pattern generator is reset to the first pattern by the method `first_pattern`. The next pattern in the sequence is obtained with the `next_pattern` method. The method `more_pattern` returns true if there are more patterns in the generator's sequence. The derived classes realize: binary counters, LFSRs, LHCA, and two-dimensional CAs.

The class for observers is called `Observer`. It defines the methods: `input`, `reset_state`, and `check`. These methods respectively, accept input, reset the observer's state, and com-

pare the observer's state against another observer's state. The supported observers compute the state of a MISR, a MICA, an up/down counter, and a run counter. The MISR and MICA perform their typical output analysis role. The response analysis architectures of Section 6.4 require the up/down and run counter.

The previous classes provide the objects needed to build the functional and fault free simulators. The class `Simulator` implements an event driven functional simulator. The class `FaultSimulator` implements the parallel pattern single fault propagation (PPSFP) [ELWW91] and parallel fault [ABF90, page 135] fault simulators as two separate methods.

The rationale for supporting two simulators[3] is:

- While the PPSFP fault simulation algorithm is much faster than the parallel fault algorithm, the PPSFP algorithm only works for combinational logic. Since many BIST architectures do not partition the circuit into combinational logic blocks for testing, a fault simulator for sequential logic is necessary. The CSTP architecture is an example of a BIST architecture in which the circuit is tested as sequential logic.

Thus the PPSFP algorithm can be used to measure the test effectiveness of BIST architecture that partition the circuit into combinational blocks, while the parallel fault algorithm can be used for the sequential circuits. Using the parallel fault algorithm has the advantages of also determining if any aliasing is present in the BIST embedding. The parallel fault algorithm also records the fault coverage for every simulation cycle.

Table 4.4 presents the simulation times required to achieve 100% single stuck-at fault coverage for some of the non-redundant ISCAS85 benchmarks for the two algorithms. The speed ratio varies from 1 to 1214 times for the two algorithms.

[3] An additional advantage of using two algorithms is that their results can be compared to gain confidence in the simulator's correctness.

Table 4.4 Timings in CPU seconds for PPSFP and Parallel Fault.

Circuit	Gates	Cycles	PPSFP	Parallel Fault
c17	6	13	0.03	0.04
c432	157	1631	0.78	24.98
c499	202	976	1.59	15.51
c880	409	29035	5.47	801.77
c1355	578	1995	7.65	242.00
c1908	1045	14161	9.43	5041.73
c2670	1351	750716	331.55	-
c3540	1848	23744	19.56	21965.70
c6288	2399	304	628.76	11215.50

4.3.1 Adding the Delay Fault Model

The extension of the fault simulators from the single stuck-at fault model to the slow-to-rise/slow-to-fall gate delay model is reasonably simple. For the PPSFP algorithm, only the fault insertion step of the algorithm needs to be modified. In the single stuck-at fault simulation, either an all zeros or all ones pattern replaces the correct signal pattern. In the delay fault simulation, the functional pattern is examined and modified to reflect the fault effect of slow-to-rise or slow-to-fall.

The addition of a simulation gate that realizes an 1 cycle inertia delay can be used to model the slow-to-rise and slow-to-fall faults in the parallel fault algorithm. This gate is inserted at the fault location.

4.4 Experience with the lg3 Design

In [Wol91], Wolf makes the claim that:

> "CAD programs are particularly well suited to object-oriented programming because they lead us to think about objects we can naturally express as ab-

> stract data types and because the applications use common features that are appropriate for inheritance and run-time function determination."

The experience of developing the `lg3` did match the claim by Wolf. For example, the representation of simulation gate entities by the `Simgate` class created an abstract data type that encapsulated all the behaviour of a simulation gate. Inheritance is used to create a specific simulation gate type (e.g., an EXOR gate). The ability of C++ to perform run-time function determination is a convenient way of determining which gate to evaluate during simulation.

One of the promises of the object-oriented approach is fast prototyping. This promise is realized, especially by the ability to add new BIST response observer types. Chapter 6 introduces a novel BIST response analysis structure based on deterministic finite automata (DFAs). It only required a day to add the simulation behaviour of these DFAs to the `lg3` CAD. Another example is the and-or-invert and or-and-invert simulation gates. These gates are added simply by deriving the classes `AoiGate` and `OaiGate` from `SimGate` and then implementing the required methods. The easy addition of the `use` and `update` Logic III(UVic) statements is the third example of the fast prototyping ability.

The advantages of fast prototyping is also observed in the flexibility of the classes used to implement the functional and fault simulator. Three different simulators are built from these classes. These simulators are: a functional simulator, a PPSFP (Parallel Pattern Single Fault Propagation) fault simulator, and a parallel fault simulator.

Although the fault simulator is not designed for *extreme* speed[4], the fault simulators are fast enough to allow a reasonable number of BIST embedding experiments. For example, using the PPSFP fault simulation, simulation of 10240000 vectors for the ISCAS85 c7552 requires 7364.91 CPU seconds on a HP 9000 Model 735 workstation. There are 3868 gates in the c7552 circuit. This is certainly quite fast enough to allow experimenting with the

[4]Replacing the use of virtual functions in simulation gate evaluation and replacing the event driven schedule with a static schedule would increase the simulation speed.

type and seed of the generator. Also the times for the PPSFP algorithm for the remaining ISCAS85 benchmark circuits are presented in Table 4.4. The times varied from 0.03 to 628.76 seconds.

Chapter 5

Case Studies

5.1 TMS32010 Data Path Case Study

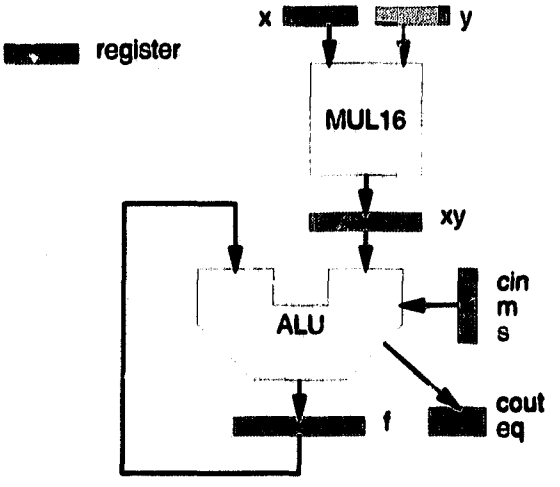
This section uses the case study described by Kim, Ha, and Tront [KHT88] as the first case study to illustrate the capabilities of the BIST embedding features of Logic III(UVic) and evaluation features of the `lg3` CAD tool.

5.1.1 ALU and Multiplier

In [KHT88], four different BIST embeddings for a 16-bit multiplier and ALU, with the input registers **x**, **y**, the multiplier output register **xy**, and ALU output registers **f**, **eq** and **cout** circuit, depicted in Figure 5.1, are presented to show the effectiveness of using MISRs as pattern generators in BIST. The goal of this section is to show the ease of specifying different BIST embeddings for this circuit with Logic III(UVic).

The 16-bit ALU is constructed with circuits that are functionally equivalent to the SN74181 and the SN74182 circuits. SN74181 is a 4-bit ALU and SN74182 is a look-ahead-carry circuit. The module for the ALU16 is:

Figure 5.1 Part of a TMS32010 Data Path




```
net_module alu16(
    m : input; s : array [1..4] of input;
    a,b : array [1..16] of input; cin : input;
    f : array [1..16] of output; cout, eq : output;);
var
    e : array [0..3] of node; co : array [3..3] of node;
    g, p : array [0..2] of node; c : array [1..3] of node;
    t_cin : node;
begin
    sn74182(cin, g[0..2], p[0..2], c);
    sn74181(m,s,a[1..4],b[1..4],cin,f[1..4],_,p[0],g[0],e[0]);
    sn74181(m,s,a[5..8],b[5..8],c[1],f[5..8],_,p[1],g[1],e[1]);
    sn74181(m,s,a[9..12],b[9..12],c[2],f[9..12],_,p[2],g[2],e[2]);
    sn74181(m,s,a[13..16],b[13..16],c[3],f[13..16],co[3],_,_,e[3]);
    connect(co[3], cout);
    a4(e[0], e[1], e[2], e[3], eq);
end.
```

Note that the unused outputs of the **sn74181** module are *capped* with "_"[1] Capping the outputs can allow the CAD tool to eliminate any unnecessary logic. If the logic is not eliminated, then the internal lines which are not necessary can be removed from consideration as possible fault locations.

[1]Although the current CAD tool does not support this construct, for these expirements the unused outputs are disabled manually.

The recursive multiplier in Section 3.3.2 realizes the multiplier in the TMS32010 data path. An exhaustive application of all input patterns to this multiplier shows that some of the logic is redundant. The **faults_off** statement can be used to document these redundancies and to remove them from the fault list. The redundancies are located in the **mpy_add** routine. The routine with the relevant **faults_off** statement is:

```
net_module mpy_add( x : array [1..] of input;
    y:array [1..x'size] of input;
    s:array [1..x'size+1] of output);
const
    x_sz = x'size;
var
    x1, coutb : node; carry : array [1..x_sz] of node;
    i : integer;
faults_off
        x[2], y[2], carry[1], y[x_sz], carry[x_sz-1];
begin
    half_adder2( x[1], y[1], s[1], carry[1]);
    for i := 2 to x_sz do begin
     adder2( x[i], y[i], carry[i-1], s[i], carry[i]);
    end;
    connect(carry[x_sz], s[x_sz+1]);
end.
```

Similarly, an exhaustive application of all input patterns to the 16-bit ALU shows that the ALU contains no redundancies.

The elimination or documenting of the redundant logic at the individual module level, simplifies the BIST embedding processes since the user now knows that all the remaining faults should be detectable, assuming that the BIST embedding tests the individual modules. Thus the only possibility of redundant logic occurring is if the possible output values of a module connected to another module are not sufficient to require the full function in the second module.

The Logic III(UVic) code for the data path is:

```
net_module main_alu16(
    m : input; s : array [1..4] of input;
    x, y : array [1..8] of input; cin : input;
    F : array [1..16] of output; Cout, Eq : output; );
var
    r_m : reg; r_s : array [1..4] of reg;
    r_x,r_y : array [1..8] of reg; r_cin : reg;
    xy  : array [1..16] of node; r_xy : array [1..16] of reg;
    f : array [1..16] of node; r_f : array [1..16] of reg;
    cout,eq : node; r_cout,r_eq : reg;
begin
    multiplier(r_x,r_y, xy);
    alu16(r_m,r_s,r_xy,r_f,r_cin, f,cout, eq);
    connect(r_f, F); connect([r_cout, r_eq], [Cout, Eq]);

    r_x <- x; r_y <- y; r_m <- m; r_s <- s; r_cin <- cin;
    r_xy <- xy;
    r_f <- f; r_cout <- cout; r_eq <- eq;
end.
```

The different BIST embeddings are realized by configuring the registers `r_x`, `r_y`, `r_m`, `r_s`, `r_cin`, `r_xy`, `r_f`, `r_cout`, and `r_eq` with the `use` statement.

5.1.2 BIST Embeddings for the TMS32010

One of the first steps in a BIST embedding is to see if the complete collection of modules is random-pattern testable. Alternatively if the number of stimulus points is less than N_{ex} (see Section 2.2.2), then an exhaustive stimulus approach could be used. The combinational embedding transforms every register input into an observation point and every register output into a stimulus point. Registers that are not in loops are transformed into direct connections with the `Fan` module. The resulting circuit is strictly combinational and thus the fast fault simulator of `lg3` can be used. The embedding is realized by bracketing the `update` statements with the BIST embedding *init* and *finish* modules. The embedding is:

```
assert( init_comb(1) = 1 );
use Fan in begin
    r_x <- x; r_y <- y; r_m <- m; r_s <- s; r_cin <- cin;
    r_xy <- xy;
    r_cout <- cout; r_eq <- eq;
end;
use comb_embed in r_f <- f;
finish_comb(1);
```

The associated testing script is:

```
testing
begin
    [m,s,cin,x,y,comb_generator[1..comb_index-1]] :-lfsr(1);// [1]
    fault_simulate(1024*100);                               // [2]
    write("gates = ", number_of_gates, " ");                // [3]
    write("cycles = ", number_of_cycles, " ");              // [4]
    write("faults = ", number_of_faults, " ");              // [5]
    write("exposed= ", number_of_exposed, "\n");            // [6]
    print_non_exposed;                                      // [7]
end.
```

Line 1 of the testing script attaches a minimum cost maximum cycle LFSR pattern generator to the collection of nets (see Section 3.3.1) on the left hand side of the binding operator ":-". The LFSR is minimum cost in the sense that the characteristic polynomial contains the minimum number of nonzero coefficients. The table in appendix A of [SK90] is used to determine the coefficients. The second line directs the simulator to perform a fault simulation with 102400 patterns. The simulator continues for 102400 patterns or until all the faults are exposed. Lines 3 through 6 print out statistics of the simulation and the circuit. The variables: `number_of_gates`, `number_of_cycles`, `number_of_faults`, and `number_of_exposed` are predefined by the simulation system. The last line, line 7, prints the faults, if any, that are not exposed during the simulation.

The output of `lg3` for this embedding and test script is:

```
gates = 610 cycles = 7469 faults = 2164 exposed= 2164
Execution Resources: 9.92 user 0.13 sys
```

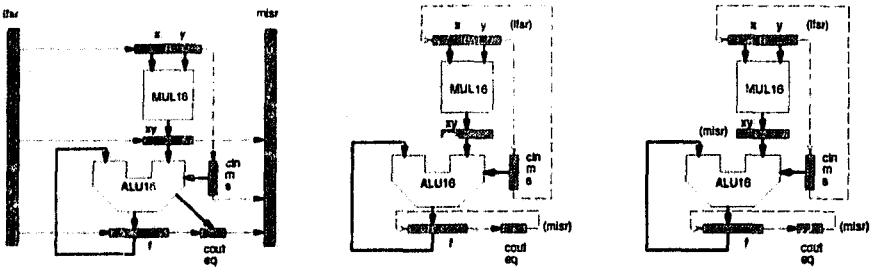
This result shows that the circuit is 100% random-pattern testable after 7469 cycles. Also important to note is the run time of 10 seconds[2]. Since the runtime is quite small, the

[2]All CPU times report are for a HP 9000 Model 735.

user could experiment with different stimulus sources to try and reduce the number of test cycles.

Figure 5.2 shows the architectures of the six different BIST embeddings for the TMS32010 data path. All of the embedding stimulus sources are *random* (see Section 2.2), since this circuit is pseudorandom testable. The next state functions of the stimulus sources depend on both testing and normal (TN) circuitry and solely testing (T) circuitry.

Figure 5.2 TMS32010 Data Path BIST Embeddings



(a) E2 (b) E3 (c) E4

(d) E5 (e) E6 (f) E7

The E2 embedding realizes the STUMPS (Section 3.5.2) architecture. The code for its embedding is:

```
assert( init_stumps(1) = 1 );
use stumps_embed in begin  // scan chain 1
    r_x <- x; r_y <- y; r_m <- m; r_s <- s; r_cin <- cin;
end;
use stumps_embed in r_xy <- xy;  // scan chain 2
use stumps_embed in begin   // scan chain 3
    r_f <- f; r_cout <- cout; r_eq <- eq;
end;
stumps_generator_ctrl( 16 );
stumps_observer( 16 );
finish_stumps(1);
```

This example follows a style that is common in all the embeddings that use the library of embedding modules. The **init_stumps** function sets up the global variables used by the embedding. If the initialization is successful a 1 is returned. The **finish_stumps** module performs any required global connections. Modules **stumps_generator_ctrl** and **stumps_observer** realize the STUMPS stimulus structure and response analysis structure. The parameter of 16 gives the length of the LFSR and MISR for these structures. The embedding modules for the other architectures in the library all have an **init** function and a **finish** module.

This embedding has three separate scan chains, where each **use** statement contain . scan chain. The ordering of the scan elements depends on the **update** statement ordering. If only one scan chain is specified, then the LOCST [ABF90, section 11.4.4] BIST architecture is realized. Each scan chain should partition the kernel into two separate subcircuits. Increasing the number of scan chains can reduce the maximum scan chain length and therefore decreases the testing time since more test patterns can be applied for a fixed number of clock cycles. Each additional scan chain only requires an extra EXOR gate in the response analysis structure.

Instances of the BEST architecture (Section 3.5.3) are realized by the embeddings E3, E4 and E5. The embedding code for E3 is:

```
assert( init_best(1) = 1);
use gen_embed in begin // stimulus source
    r_x <- x; r_y <- y;
    r_m <- m; r_s <- s; r_cin <- cin;
end;
use dff_array in r_xy <- xy; // pass through
use misr_embed in begin
    r_f <- f; r_cout <- cout; r_eq <- eq; // response analysis
end;
finish_best(1);
```

The **gen_embed** embedding module transforms the registers **r_x**, **r_y**, **r_m**, **r_s**, and **r_cin** into a minimum cost LFSR and a set of normal mode registers. The registers operate as a LFSR when the global signal **gen_on[i]** is asserted, where **i** is the i-th **use** statement with **gen_embed** as the embedding module. A separate **gen_on** signal is produced for every **use** statement. The **r_xy** register is not modified. The response analysis structure is embedded in the registers **r_f**, **r_cout**, and **r_eq**. Since the response analysis structure is a MISR, care should be taken to ensure that it is large enough to reduce aliasing. In this case the MISR's length is 18, yielding an aliasing probability of 2^{-18}. Similar to the stimulus source, a global signal **misr_on** is created for every **use** statement with the **misr_embed** embedding module.

The following testing script is used to measure this embedding's effectiveness.

```
testing
var
    i, cycles : integer;
begin
    cycles := 1024*20;
    [x,y,m,s,cin] :- seq("0 1");
    for i := 1 to gen_on_index-1 do gen_on[i] :- seq("1 1");
    for i := 1 to misr_on_index-1 do misr_on[i] :- seq("1 1");
    fault_simulate(cycles);
    write("gates = ", number_of_gates, " ");
    write("cycles = ", number_of_cycles, " ");
    write("faults = ", number_of_faults, " ");
    write("exposed= ", number_of_exposed, " ");
    write("detected= ", number_of_detected, "\n");
    print_non_detected;
end.
```

Note that the primary inputs are driven to all zeros, they could be driven to any other value since the BIST mode ignores the primary inputs. The BIST mode control signals **gen_on** and **misr_on** are asserted for the simulation.

Simulating this embedding with **lg3** results in:

```
gates = 770 cycles = 20480 faults = 2164 exposed= 2164 detected= 2164
Execution Resources: 7208.06 user 0.1 sys
```

The last effective vector is at 16309. Thus this embedding achieves 100% fault coverage after the application of 16309 vectors.

The E4 embedding is the same as the E3 embedding expect the register **r_xy** is converted into a MISR during BIST mode. Its embedding code is:

```
assert( init_best(1) = 1);
use gen_embed in begin // stimulus source
    r_x <- x; r_y <- y;
    r_m <- m; r_s <- s; r_cin <- cin;
end;
use misr_embed in r_xy <- xy; // xy is now a misr
use misr_embed in begin
    r_f <- f; r_cout <- cout; r_eq <- eq; // response analysis
end;
finish_best(1),
end.
```

Note that the only difference is in the change of the embedding module from **dff_array** to **misr_embed** in the second **use** statement.

The simulation result for the E4 embedding is:

```
gates = 805 cycles = 10240 faults = 2164 exposed= 2164 detected= 2161
S@0:T.alu16@44.sn74182@24.oi4@35i.fan[2]
S@1:T.alu16@44.sn74181_nc@26.a4@323.fan[0]
S@1:T.alu16@44.sn74181_nc@27.ai4@322.fan[3]
Execution Resources: 3741.61 user 0.92 sys
```

In this simulation, the list of faults that are not detected at the 10240 cycle are printed. This means that state of the MISRs, at the 10240th cycle, for these faults is identical to

the fault free case and aliasing is present. Aliasing did not occur at the 7809th cycle where all the faults are detected. In the current version of the **lg3** CAD tool, the parallel fault simulator does not stop after all the faults are detected. The simulation times would be smaller if the simulator stopped after all the faults are detected.

It is interesting to see that embedding of the MISR in register **xy** results in 100% fault coverage in 7809 cycles as compared to 16309 cycles when the register is not modified. Two possible explanations for this effect are:

1. The MISR at register **xy** increases the number of nets observed during BIST mode.
2. The input to the ALU contains fewer repeated patterns, therefore allowing more faults in the ALU to be exposed. Specifically, after 10240 cycles, there are 86% distinct patterns from the **xy** register for E4, while E3 only has 48% distinct patterns.

E5 is the final example of a BEST BIST embedding. It replaces the register **xy** with the **gen_misr_embed** embedding module. This embedding separates the data path into two distinct testing units, one consisting of the ALU and another containing the multiplier. This embedding is the most expensive. Its test effectiveness reported by **lg3** is:

```
gates = 856 cycles = 10240 faults = 2164 exposed= 2164 detected= 2161
S00:T.alu16044.sn74181_npg028.L[55]
S01:T.alu16044.sn74181_npg028.O[79]
S01:T.alu16044.sn74181_nc025.a40323.fan[3]
Execution Resources: 3379.07 user 3.21 sys
```

As with the other BEST architectures, 100% fault coverage is achieved. All the faults are detected at the 6517th test cycle.

The last two embeddings, E6 and E7, implement the CSTP BIST (Section 3.5.5) architecture. The only difference between the two embeddings is the inclusion or exclusion of the register **xy** in the circular test path. The embedding code for E6, which includes the register **xy** is:

```
assert( init_cstp(1) = 1);
use cstp_scan_embed in begin // only a shifter register
    r_x <- x; r_y <- y;
    r_m <- m; r_s <- s; r_cin <- cin;
end;
use cstp_embed in begin
    r_xy <- xy; r_f <- f; r_cout <- cout; r_eq <- eq;
end;
finish_cstp(1);
```

The E7 code is:

```
assert( init_cstp(1) = 1);
use cstp_scan_embed in begin // only a shifter register
    r_x <- x; r_y <- y;
    r_m <- m; r_s <- s; r_cin <- cin;
end;
use dff_array in r_xy <- xy; // pass through
use cstp_embed in begin
    r_f <- f; r_cout <- cout; r_eq <- eq; // proper cstp register
end;
finish_cstp(1);
```

Again note that a different embedding is realized simply by changing the location of the use statement[3]. For both of these embeddings two types of CSTP embedding modules are used, one for registers that are attached to primary inputs and one for the remaining registers. The primary input registers are simple scan registers.

The testing script for both E6 and E7 is:

[3]A feature not present in the current lg3 tool, but useful to the designer, is a mechanism to list the embedding order of the memory elements. This feature allows the designers to verify their embedding.

```
testing
var
    i, cycles : integer;
begin
    cycles := 1024*20;
    [x,y,m,s,cin] :- seq("0 1");
    cstp_on :- seq("1 1");
    fault_simulate(cycles);
    write("gates = ", number_of_gates, " ");
    write("cycles = ", number_of_cycles, " ");
    write("faults = ", number_of_faults, " ");
    write("exposed= ", number_of_exposed, " ");
    write("detected= ", number_of_detected, "\n");
    print_non_detected;
end.
```

The simulation result for E6 is:

```
gates=800 cycles=20480 faults=2164 exposed=2164 detected=2164
Execution Resources: 22519.1 user 0.17 sys
```

One hundred percent fault coverage occurs at the 5606th test cycle.

The E7 result is:

```
gates=768 cycles=20480 faults=2164 exposed=2164 detected=2164
Execution Resources: 32774.1 user 4.45 sys
```

This embedding only requires 3847 test cycles to achieve 100% fault detection. This is the best test length.

5.1.3 Embedding Results

Table 5.1 summarizes the embedding simulation test results and each embedding overhead in gate count and area. The table reports only the faults associated with the kernel (i.e., the faults associated with the BIST circuitry are not considered). Reporting only the faults from the kernel enables the different BIST embeddings to be directly compared.

The OASIS CAD system [MCN90] is used to generate the standard cell layout for each embedding. The overheads do not include the BIST mode controller, or the circuitry

to compare the characterized output response with the fault free characterization. The controller for a STUMPS embedding is more complicated than either CSTP or BEST. A STUMPS controller must provide a scan phase and a test application phase, in addition to any initialization and comparison phases. The CSTP and BEST architectures only require: initialization, test application, and comparison phases.

Table 5.1 TMS32010 Embedding Results

TMS32010 Data Path faults=2164 gates=666 area=3185800					
Architecture	Embedding	Unexposed/ Undetected	E/D Cycles	Gate overhead %	Area overhead %
Combinational	E1	0/0	7469/-	-	-
STUMPS	E2	1/1	460000/460000	31.4	38.9
BEST	E3	0/0	16309/16309	15.6	22.0
BEST	E4	0/0	7809/7977	20.8	27.3
BEST	E5	0/0	6517/6517	28.5	26.8
CSTP	E6	0/0	5606/5606	20.1	22.7
CSTP	E7	0/0	3847/3847	15.3	19.0

The E1 embedding shows that the circuit is random-pattern testable. Since the combinational embedding is used soleiy to check if the circuit is random-pattern testable, no BIST circuitry is generated and thus no overheads are given. Embedding E5 partitions the circuit into combinational logic blocks and attaches both pattern generators and output response analyzers to the inputs and outputs of the kernel. Thus E5 directly implements E1, with the only difference being that E1 uses one LFSR as the pattern generator, while E5 uses two LFSRs.

The lowest overhead and shortest test sequence is achieved with the CSTP based E7 embedding. E7 having the lowest gate and area overhead is not surprising, since CSTP's modifications to the registers are the least, but it having the shortest test length is surprising. Even more surprising is the fact the E7's test length is smaller than E6's test length, since E6 contains one more CSTP register than E7. Since the CSTP BIST architecture converts a circuit into an AFSM, the number of states in its test cycle is difficult to predict. Thus

a particular CSTP embedding could yield an AFSM with only a small number of distinct states and therefore generate an insufficient number of test patterns to effectively test the circuit. A multiplier is a circuit whose cycle length can be small. For example, if the initial state of the circular path contains only one nonzero value, then the initial state repeats after n cycles, where n equals the circular path length. The counter-intuitive result where E7's test length is smallest indicates the need to use simulation to determine effective BIST embeddings.

The E2 embedding, based on the STUMPS architecture, has the worst overhead and test time. One fault remained undetected after 460000 cycles. The last effective test cycles occurred at cycle 154578. In principle, this fault should also be detectable, if the number of simulation cycles is increased. The time can be directly attributed to the scan phase of the BIST mode. The area overhead comes from the fact that in scan each memory element contains a MUX to switch between normal, and scan mode, while memory elements in CSTP and BEST only require an EXOR gate and an AND gate (see Section 3.5). Also separate pattern generators and response analyzers are required.

Ranking the embedding according to increasing test length gives: E7, E6, E5, E4, E3 and E2. The ranking based on decreasing gate overhead is: E7, E3, E6, E4, E5 and E2. Using decreasing area overhead yields: E7, E3, E6, E5, E4 and E2. Another interesting result from Table 5.1 is the variation of gate overhead with area overhead between E4 and E5. While there is a good correlation between gate overhead and area overhead, some CAD systems place and route subsystems can produce results that invalidate this correlation.

The number of test cycles required for a particular embedding can depend on the mapping of registers to BIST structures. For example, the embedding order of the `x`, `y`, `m`, `s`, and `cin` registers in the E5 embedding results in different test lengths. The embedding order depends on the textual occurrence of the binding or use statements. The following embedding statement from E5:

```
use gen_embed in begin
    r_x <- x; r_y <- y;
    r_m <- m; r_s <- s; r_cin <- cin;
end;
```

is equivalent to:

```
gen_embed([x,y,m,s,cin], [r_x,r_y,r_m,r_s,r_cin]);
```

The register variables are grouped according to their associated **use** statement occurrence. Thus a different ordering can be achieved by reordering the **use** statements. For example, the following reordering:

```
use gen_embed in begin
    r_m <- m; r_s <- s; r_cin <- cin;
    r_x <- x; r_y <- y;
end;
```

results in:

```
gen_embed([m,s,cin,x,y], [r_m,r_s,r_cin,r_x,r_y]);
```

Since the memory elements for the LFSR created by the module **gen_embed** are generated in order of their occurrence in the input array, reordering the input array can result in different test patterns being applied to the circuit and therefore different faults are exposed.

Table 5.2 shows the effect of varying the **use** statement order for the **x**, **y**, **m**, **s**, and **cin** registers in E5 on the number of cycles necessary to detect all of the faults. Note that the number of test cycles varies significantly.

In trial 2 of Table 5.2, the simulation limit of 20480 cycles is reached without detecting all the faults. Trails 3 and 4 require exactly the same number of test cycles. This behaviour is expected, since the circuitry attached to **x** and **y** is identical, and the LFSR produced by **gen_embed** contains no internal taps for the **x** and **y** partitions, therefore the identical test patterns are applied to the **x** and **y** circuitry.

Table 5.2 E5 Ordering Trials

Trial	Ordering	Cycles
1	`x, y, m, s, cin`	6517
2	`y, m, s, cin, x`	**20480** 1 undetected
3	`m, s, cin, x, y`	12239
4	`m, s, cin, y, z`	12239

5.1.4 Summary of the TMS32010 Data Path Case Study

The following points summarize the results for the TMS32010 data path case study:

- Only simple changes in the Logic III(UVic) design sources are required to generate many different BIST architectures. In Logic III(UVic), a BIST embedding is specified by the embedding modules and the arrangement of use statements. Thus with part of the library of embedding modules, six different embeddings are generated for the TMS32010 data path by choosing embedding modules and selecting the statements enclosed by the use statements.

- As indicated by the execution times, particular BIST embeddings are evaluated with the **lg3** tool in 1 to 5 hours of CPU time on a RISC based workstation. The speed of the evaluation easily allows many different BIST architectures and variations of those architectures to be examined.

- All of the BIST architecture examined for the TMS32010 data path achieve a fault coverage of 100%. It is difficult to predict the number of test cycles required by any one embedding. This difficulty is illustrated by the CSTP BIST architecture embedding E7 which requires the fewest test cycles. A general analysis of CSTP indicates that it should require more test cycles than a BEST BIST architecture. This difficulty indicates the need to perform fault simulations.

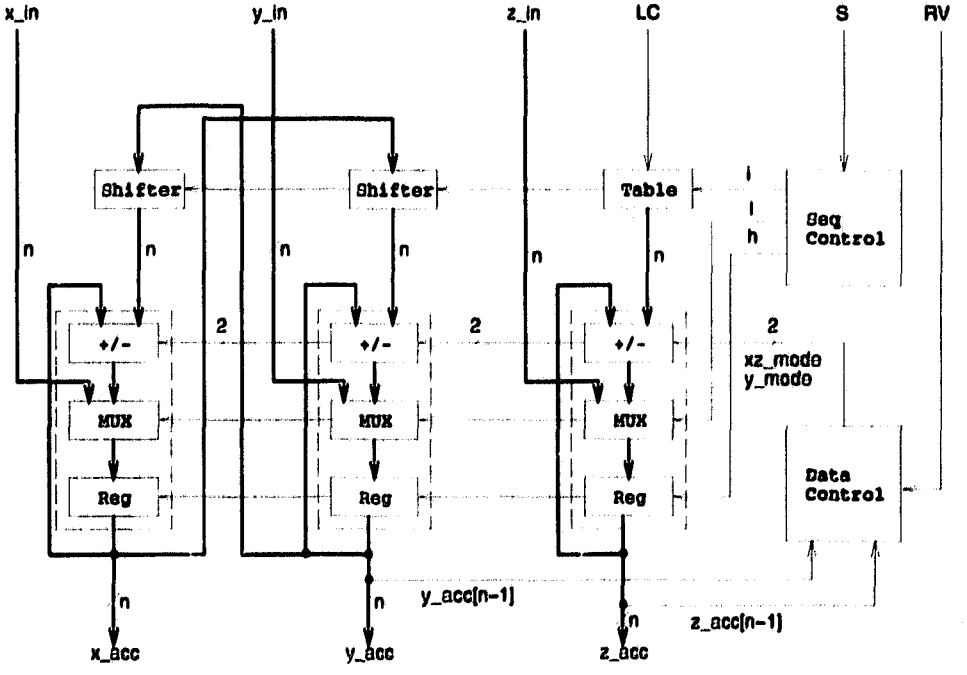
- The architecture with the smallest overhead and fewest number of test cycles is embedding E7.

- For the TMS32010 data path, Logic III(UVic) and the **lg3** tool allow the design space of BIST embeddings to be easily and effectively explored.

5.2 The CORDIC Case Study

In this section, a circuit that implements the CORDIC [Hwa70] technique is examined as the second case study. This case study also considers how a more intrusive embedding approach can realize greater savings. Unlike the TMS32010 data path, this circuit is a complete and independent unit. The CORDIC design contains both a data path and a controller. The Logic III(UVic) code for the processor is parameterized to generate a processor where the word length can be set to any value between 4 and 29 bits. See Appendix D for the complete listing of the CORDIC design.

Figure 5.3 CORDIC Modules



The CORDIC technique calculates the following mathematical functions : sin, cos, tan, square root, multiplication, and division using an iterative algorithm. It is implemented using only shifters, adder/subtractors, and registers. Figure 5.3 shows the high level structure of the CORDIC implementation.

Pseudo code with C-Language syntax for the CORDIC algorithm is shown in Figure 5.4. In the pseudo code, the *X*, *Y*, and *Z* registers (see line 2) are loaded via the primary inputs *x_in*, *y_in*, and *z_in* (see line 5). Strobing the *S (Start)* primary input initiates a computation (see line 4 - *while* loop). The *LC (Linear/Circular)* and *RV (Rotate/Vector)* inputs control the type of computations performed.

Figure 5.4 Pseudo-code for CORDIC Algorithm.

```
(1)  inputs x_in, y_in, z_in, LC, RV, S;
(2)  register x[0:7],y[0:7],z[0:7];
(3)  outputs x,y,z;

(4)  while( S != 1 )
       ;
(5)  x = x_in; y = y_in; z = z_in; // load the registers

(6)  for(i=0; i <= 6; i++){
(7)    op = RV&y[7] | (~RV)&(~z[7]); // data_controller
(8)    tx0 = y>>i;
(9)    ty0 = x>>i;
(10)   tz0 = table(i, LC);
(11)   negate(op,tx0,tx1);
(12)   negate(~op,ty0,ty1);
(13)   negate(op,tz0,tz1);
(14)   if (!LC ) x += tx1;
(15)   y += ty1;
(16)   z += tz1;
     }
```

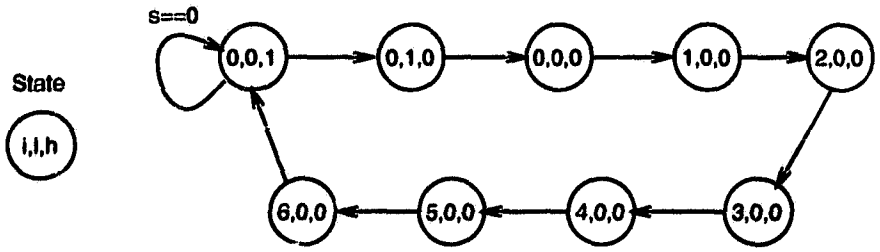
The Logic III(UVic) description of the CORDIC processor is given in Figure 5.5. The word size of the processor depends on the size of the **x_load** parameter. The comments in the module's body are used as tags to refer to the binding statement for the BIST embeddings.

Figure 5.5 Logic III(UVic) Code for CORDIC Algorithm

```
net_module cordic(
    start,rv_mode,lc_mode: input;
    x_ld                 : array [] of input;
    y_load, z_load       : array [x_ld'first..x_ld'last] of input;
    x_acc,y_acc,z_acc    : array [x_ld'first..x_ld'last] of output);
const
    left_sh = 1;
    right_sh = x_ld'size-1;
var
    shift_sel          : array [ -left_sh..right_sh ] of node;
    x_sh,y_sh,z_table  : array [x_ld'first..x_ld'last] of node;
    x_md,x_hd,x_ld     : node ;
    y_md,y_hd,y_ld     : node ;
    z_md,z_hd,z_ld     : node ;
    x_r,y_r,z_r        : array [x_ld'first..x_ld'last] of reg;
    st_r,rv_r,lc_r     : reg;
begin
    accumulator( x_md, x_hd, x_ld, x_r, x_sh, x_acc); // mod_x
    arithmetic_shifter( left_sh, right_sh, y_acc, shift_sel, x_sh);

    accumulator( y_md, y_hd, y_ld, y_r, y_sh, y_acc); // mod_y
    arithmetic_shifter( left_sh, right_sh, x_acc, shift_sel, y_sh);

    accumulator( z_md, z_hd, z_ld, z_r, z_table, z_acc); // mod_z
    cordic_table (left_sh, lc_r, shift_sel, z_table );

    cordic_controller( st_r, rv_r, lc_r,
        y_acc[y_acc'last], z_acc[z_acc'last], shift_sel,
        x_md, x_hd, x_ld, y_md, y_hd, y_ld, z_md, z_hd, z_ld ); // mod_c

    st_r <- start; rv_r <- rv_mode; lc_r <- lc_mode;
    x_r <- x_ld; y_r <- y_load; z_r <- z_load;
end.
```

The sequential controller, sc, waits for the start strobe, controls the iterations and maintains the index value i. Sh_x and Sh_y perform the arithmetic right shift operations. The sequential controller is described by the Finite State Machine (FSM) depicted in Figure 5.6. The negations and additions are done by an adder/subtractor (+/−) unit. The data controller determines whether the operation performed is an add or a subtract.

Figure 5.6 CORDIC Sequential Controller FSM.



The arithmetic right shifter, called `arith shifter`, uses a gate-level barrel shifter in this construction. The `n_add_sub` module (for the additions and subtractions) is created with a linear array of full bit adders. Subtraction is implemented by an array of XOR gates attached to one of the adder's inputs. This realizes a one's complement. The two's complement is obtained by setting the carry in of the least significant bit to 1 (i.e., complement and increment).

5.2.1 BIST Embeddings for the CORDIC processor

5.2.1.1 Standard Architectures

The strategy to embed BIST into the CORDIC processor is similar to the TMS32010 data path. First, the individual modules are stimulated with pseudorandom sequences to check the testability of each module.

A simulation of the `cordic_table` module where the word size equals 8 yields the following redundancies:

```
S@1:T.cordic_table@131.aoi22@113<0>.fan[3]
S@1:T.cordic_table@131.aoi22@113<1>.fan[3]
S@1:T.cordic_table@131.aoi22@113<2>.fan[3]
S@1:T.cordic_table@131.aoi22@113<3>.fan[3]
S@1:T.cordic_table@131.aoi22@113<4>.fan[3]
S@1:T.cordic_table@131.aoi22@113<5>.fan[3]
S@1:T.cordic_table@131.aoi22@113<6>.fan[3]
```

These nets appear in the following fragment:

```
i1 ( lc_md, lc_md_bar );
for i := cir'first to cir'last do begin
    aoi22( cir[i], lc_md_bar, lin[i], lc_md, z_table_bar[i] );
//                                      ^ s@1
    i1( z_table_bar[i], z_table[i])
end;
```

The logic driving the `cir[i]` and `lin[i]` produces the same values, therefore a stuck-at-one fault at `lc_md` is not detected.

Once any redundancies are removed or marked with the `faults_off` statement from the component modules the entire CORDIC processor is transformed into a combinational circuit with the `comb_embed` and tested with a pseudorandom sequence.

The combinational embedding is accomplished with the addition of the following code to the code in Figure 5.5. The word size is set to three for this simulation to enable exhaustive simulation of the control circuitry.

```
// cordic net_module header
begin
    assert( init_comb(1) = 1 );
    use comb_embed in begin
       // body of the cordic net_module
    end;
    finish_comb(1);
end.
```

The testing script is:

```
testing
var
    cycles : integer;
begin
    cycles := 67108864; //2^26
    [st, rv, lc, x, y, z] :- seq("0 1");
    comb_generator[1..comb_index-1] :- zlfsr(1);
    fault_simulate(cycles);
    write("gates= ", number_of_gates, " ");
    write("cycles= ", number_of_cycles, " ");
    write("faults= ", number_of_faults, " ");
    write("exposed= ", number_of_exposed, " ");
    write("detected= ", number_of_detected, "\n");
    print_non_exposed;
end.
```

This testing script is almost identical to the one used in the TMS32010 data path combinational embedding. This similarity suggests that a macro facility for common testing scripts sl ·id be added to the testing language. The primary inputs are again bypassed by the en. .edding, and therefore can be driven to any value. The simulation result is:

```
gates=137 cycles=67108864 faults=435 exposed=434 detected=0
S@1:T.cordic@273.cordic_controller@245.cor_seq_controller@87.a2@68.fan[1]
Execution Resources: 4895.57 user 1.42 sys
```

Note that the PPSFP algorithm only report the exposed faults, thus the number of detected faults is 0. The code for the unexposed fault is:

```
a2( S0_, state[1], load);
//          ^ s@1
```

The inability to expose this fault can be attributed to the poor observability of the `load` line. The above simulation showed an exhaustive simulation, careful analysis of the circuit could also be used to prove the fault's undetectablity. Another technique is to use an ATPG program to see if a test could be generated for the fault. The user can get the system to ignore the problem by removing that fault from the fault list by adding:

```
faults_off state[1];
```

to the module's header. Alternatively, the user could improve the observability of **load** line. The strategy chosen for this case study is to remove the fault.

Since the CORDIC processor is pseudorandom testable, the BEST and CSTP BIST architectures are examined in the following embeddings. A CSTP based BIST embedding, C2, is specified with:

```
begin
    use cstp_embed in begin
      accumulator( x_md, x_hd, x_ld, x_r, x_sh, x_acc); // mod-x
      arithmetic_shifter( left_sh, right_sh, y_acc, shift_sel, x_sh);
      accumulator( y_md, y_hd, y_ld, y_r, y_sh, y_acc); // mod-y
      arithmetic_shifter( left_sh, right_sh, x_acc, shift_sel, y_sh);
      accumulator( z_md, z_hd, z_ld, z_r, z_table, z_acc); // mod-z
      cordic_table (left_sh, lc_r, shift_sel, z_table );

      cordic_controller( st_r, rv_r, lc_r,
        y_acc[y_acc'last], z_acc[z_acc'last], shift_sel,
        x_md,x_hd,x_ld,y_md,y_hd, y_ld, z_md, z_hd, z_ld ); // mod-c
    end;

    use cstp_scan_embed in begin
        st_r <- start; rv_r <- rv_mode; lc_r <- lc_mode; // upd-m
        x_r <- x_load; y_r <- y_load; z_r <- z_load;     // upd-d
    end;
end.
```

An equivalent way of describing this embedding is to specify which modules are contained in the body of the use statement with the embedding module, *MODULE*. This description is given in Table 5.3.

Table 5.3 C2 CSTP BIST Embedding

C2 CORDIC Processor Embedding			
Embedding	Architecture	Embed. Modules	Modules
C2	CSTP	**cstp_embed**	mod-x, mod-y, mod-z, mod-c
		cstp_scan_embed	mod-c, upd-m, upd-d

The simulation results for this embedding is:

```
gates=636 cycles=4096 faults=1394 exposed=1394 detected=1393
S01:T.cordic017.cordic_controller041.cor_seq_controller089.a2064.fan[1]
Execution Resources: 1989.78 user 1 sys
```

The other embedding trials are described in Table 5.4. Embedding C3 modifies C2 by removing the y accumulator register from the test path (i.e., the y accumulator register operates in normal mode during the test). Embeddings C4, C5, and C6 all realize versions of the BEST BIST architecture. In C4, all the combinational logic is partitioned and tested by separate pattern generators and response analyzers. Embedding C5 also uses the response analyzer MISRs as pattern generators for the accumulators. C5 is modified to produce C6 by using the y accumulator register as a normal mode register.

Table 5.4 CSTP and BEST BIST Embeddings

More CORDIC Processor Embeddings			
Embedding	Architecture	Embed Modules	Modules
C3	CSTP	cstp_embed cstp_scan_embed	mod-x, mod-z, mod-c upd-m, upd-d
C4	BEST	gen_misr_embed gen_embed	mod-x, mod-y, mod-z, mod-c upd-m, upd-d
C5	BEST	misr_embed gen_embed	mod-x, mod-y, mod-z, mod-c upd-m, upd-d
C6	BEST	misr_embed gen_embed	mod-x, mod-z, mod-c upd-m, upd-d

Table 5.5 CORDIC Embedding Results

CORDIC Processor faults=1336 gates=489				
Architecture	Embedding	UnE/UnD	E/D Cycles	Gate Overhead %
Combinational	C1	0/0	4881/-	-
CSTP	C2	0/0	974/974	28.9
CSTP	C3	0/0	3442/3442	25.6
BEST	C4	0/0	769/769	49.0
BEST	C5	0/0	630/630	30.0
BEST	C6	0/0	18136/18136	26.4

Table 5.5 gives the simulation results of the CORDIC embeddings. Embeddings C2 through C6 for the CORDIC processor achieved 100% single stuck-at fault coverage. The number of test cycles required for the embeddings varies from 630 to 18136 cycles. Unlike the TMS32010 data path embeddings the number of test cycles required for each architecture more closely matched their expected ranking. That is, the BEST architecture should require fewer test cycles, since the pattern generator is independent of the functional circuitry, and always produces distinct test patterns. This effect of longer test cycles is also present in embedding C5 and C6 where a MISR acts as a pattern generator. It is interesting to note that the overhead of the CSTP embedding C3 is very close to the overhead of C6. This shows that while CSTP is a cheap embedding, the BEST architecture can be competitive in overhead. Since BEST embedding partition the design into combination sections, the PPSFP fault simulation algorithm can be used to determine the fault coverage.

5.2.2 Effect of the BIST Generator's Seed

The default initial state for the memory elements used in the CSTP and BEST embedding modules is the all ones state. The test length required to achieve a certain fault coverage depends on the state sequence of the BIST stimulus source and the state sequence depends on the initial state of the BIST stimulus source. The initial state for the BEST and CSTP embedding modules is controlled by the argument to the respective initialization functions, `init_best` and `init_cstp`.

The argument affects the initial state in the following way:

$$\text{state} = \begin{cases} \text{all zeros} & n = 0 \\ \text{all ones} & n = 1 \\ \text{1 in first memory element} & n = 2 \\ \text{1 in every } n\text{th memory element} & n > 2 \end{cases}$$

Thus an argument of 4 causes every 4th memory element to be initialized to 1, the rest are initialized to 0. The results for an argument of $n = 0$ to $n = 16$ for the CSTP

embedding, C2 and the BEST embedding, C4 are given in Table 5.6.

Table 5.6 Effect of the Seed on Test Length

n	C2 - CSTP	C4 - BEST
1	974	769
2	619	596
3	935	563
4	1292	551
5	657	519
6	520	450
7	1873	924
8	1062	423
9	573	391
10	1679	748
11	634	636
12	448	800
13	646	685
14	495	923
15	689	591
16	698	582
	avg=862.12	avg=410.45
	std=410.45	std=157.63
	min=448	min=391
	max=1873	max=924

The average and standard deviation for the two embeddings show that the effect of the seed on the test length is more pronounced for the CSTP embedding, than the BEST embedding. The ratios of the maximum to minimum test length of 4.2 and 2.4 for CSTP and BEST embeddings shows that the initial seed is important in determining test length. Therefore having the ability to easily experiment with different initial seeds can result in a reasonable savings in test time.

5.2.3 Intrusive Embeddings

The preceding embeddings incorporate BIST by augmenting the registers present in the design. A more intrusive embedding can be used to reduce the area and performance overheads of a BIST embedding. In this context, intrusive means that the design is analyzed in greater detail to see if parts of the functional design can be *co-opted*. Also extra logic can be added to facilitate the embedding.

During the CORDIC computation only the state of x and y accumulators affect the next state, thus with the exception of the initial loading phase the processor is an AFSM (Autonomous Finite State Machine). By disabling the load phase, the following three embeddings transform the CORDIC processor into an AFSM for testing purposes. The code for embedding C7 is given in Figure 5.7.

The three additional AND gates disable the loading logic during test mode. These additions reduce the embedding cost since pattern generators do not have to be placed on the `x_load`, `y_load` and `z_load` data load lines. The price paid for a cheaper embedding is the inability to test the data load lines during BIST mode. Note that these lines can be easily tested by a system level test which loads data into the CORDIC processors.

The control lines `start`, `rv_mode`, and `lc_mode` still require a pattern generator to be attached. The `use` statement with the `gen_embed` embedding module creates this pattern generator. The `n_buf` embedding module simply creates buffers (i.e., no memory elements) for the `x_load`, `y_load`, and `z_load` lines. A CSTP architecture is still inserted into the x, y, and z accumulator registers.

The simulation results for embedding C7 are:

Figure 5.7 Intrusive BIST Embedding for CORDIC

```
begin
    assert( init_cstp(1) = 1 );
    assert( init_best(1) = 1 );
    use cstp_embed in begin
        cordic_controller( st_r, rv_r, lc_r,
            y_acc[y_acc'last], z_acc[z_acc'last], shift_sel,
            x_md, x_hd, x_ld0, y_md, y_hd, y_ld0, z_md, z_hd, z_ld0);
        a2(test, x_ld0, x_ld); // NOTE: the test specific logic
        a2(test, y_ld0, y_ld);
        a2(test, z_ld0, z_ld);

        accumulator( x_md, x_hd, x_ld, x_r, x_sh, x_acc);
        arithmetic_shifter( left_sh, right_sh, y_acc, shift_sel, x_sh);

        accumulator( y_md, y_hd, y_ld, y_r, y_sh, y_acc);
        arithmetic_shifter( left_sh, right_sh, x_acc, shift_sel, y_sh);

        accumulator( z_md, z_hd, z_ld, z_r, z_table, z_acc);
        cordic_table (left_sh, lc_r, shift_sel, z_table );
    end;
    use gen_embed in begin
        st_r <- start; rv_r <- rv_mode; lc_r <- lc_mode;
    end;
    use n_buf in begin
        x_r <- x_load; y_r <- y_load; z_r <- z_load;
    end;
    finish_cstp(1);
    finish_best(1);
end.
```

```
gates=586 cycles=512 faults=1297 exposed=1270 detected=1270
S01:T.cordic078.accumulator041.n_mux20397.sel_bar
S01:T.cordic078.accumulator044.n_mux20397.sel_bar
S01:T.cordic078.accumulator047.n_mux20397.sel_bar
S01:T.cordic078.accumulator041.n_mux20397.aoi220233<0>.fan[1]
S01:T.cordic078.accumulator041.n_mux20397.aoi220233<1>.fan[1]
S01:T.cordic078.accumulator041.n_mux20397.aoi220233<2>.fan[1]
S01:T.cordic078.accumulator047.n_mux20397.aoi220233<5>.fan[1]
                              .
                              .
                              .
S01:T.cordic078.accumulator047.n_mux20397.aoi220233<6>.fan[1]
S01:T.cordic078.accumulator047.n_mux20397.aoi220233<7>.fan[1]
Execution Resources: 140.71 user 0.17 sys
```

With the exception of the load logic, 100% single stuck-at fault coverage is achieved.

The CSTP BIST embedding is removed from the x and y accumulator registers of C7 to realize the C8 embedding. This is a good example of the idea that, "No BIST, Is the best BIST!" In order words, some circuits can be configured as AFSM without the addition of any testing circuitry, and still achieve a good self test. This embedding illustrates the ability of the `lg3` CAD tool to determine the test effectiveness of alternative BIST architectures. The C8 embedding also achieved 100% single stuck-at fault coverage. An alternate to C8 consisting of removing the CSTP mode from the z accumulator, but this results in reduced fault coverage.

Embedding C9 is similar to C8, where the only difference is that a BEST BIST architecture is embedded in the z accumulator and the control logic. Again, 100% fault coverage is achieved.

A summary of the test effectiveness results for C7, C8, and C9 is given in Table 5.7. The gate overhead is reduced to 19.1% for the intrusive approach, as compared to 25.6% for the previous embeddings. Curiously, the number of test cycles is also better for the intrusive approach in this case. This again stresses the importance of simulation in attempting to realize good BIST embeddings.

Table 5.7 Intrusive CORDIC Embedding Results

CORDIC Processor faults=1297 gates=465				
Architecture	Embedding	UnE/UnD	E/D Cycles	Gate Overhead %
CSTP,BEST	C7	27/27	431/431	22.6
CSTP,BEST	C8	27/27	480/480	19.1
BEST	C9	27/27	238/238	19.8

5.2.4 BIST Embedding Robustness

The ability of this CORDIC description to generate CORDIC processors with different word sizes can be used to test how effective a particular BIST embedding is when the circuit's logic is changed. The effect of modifying the logic on a BIST embedding's test effectiveness can be used as an indication of how robust a particular embedding is to change. Ideally, a small change in the logic should not affect the embeddings achieved fault coverage. Of course, some changes to digital circuits can result in huge losses in fault coverage.

BIST embeddings based on exhaustive and pseudoexhaustive architectures are not affected by changes in the circuit's logic. It is difficult to predict the test effectiveness of the random pattern generation architectures, since the test stimulus is based on pseudorandom sequences. The greatest effect should be observable on the deterministic architectures, since a test pattern set is created for a particular version of the circuit and any changes to the circuit can require a new test set.

Table 5.8 shows that all the CORDIC processors where the word size varies form 4 to 29 achieved 100% single stuck-at fault coverage with the combinational embedding (i.e., embedding C1). Since 100% fault coverage is achieved on all designs, this indicates, at least for the CORDIC processor, that any non-deterministic BIST architecture should be robust. The results can also be used as a data point to justify the claim that the functional design can be modified after a BIST embedding, without adversely affecting the BIST embedding test effectiveness. The number of test cycles required to achieve 100% fault coverage increases slightly with word size, but still follows no pattern.

Table 5.8 comb_embed for CORDIC where $n = 4 \cdots 29$

n	Gates	Faults	Cycles	n	Gates	Faults	Cycles
4	188	595	189	17	1295	3607	3547
5	237	758	410	18	1414	3906	33629
6	300	941	689	19	1543	4220	40488
7	363	1128	689	20	1682	4548	94802
8	432	1336	4881	21	1831	4890	1599
9	499	1539	9271	22	1982	5240	47503
10	580	1760	1208	23	2139	5602	44592
11	671	1995	1672	24	2294	5965	2560
12	760	2233	9825	25	2453	6336	8453
13	855	2482	9885	26	2616	6717	4596
14	952	2738	17348	27	2781	7104	82515
15	1051	2999	1613	28	2948	7497	59466
16	1190	3326	24030	29	3117	7898	5604

The test effectiveness results for the CORDIC processor with the word size set to 16 for embeddings C2 through C6 are given in Table 5.9. The success of detecting all the faults by the combinational embedding, again indicates that the random BIST architectures should also be successful. The results from Table 5.9 show that all the embeddings achieve 100% fault coverage. These results provide evidence for the robustness assertion.

The BIST embedding gate overhead for the 16-bit CORDIC processor of Table 5.9 is less than the overhead for the 8-bit CORDIC processor of Table 5.5. This decrease is due to the increase of the ratio of combinational logic to memory element logic. Since these embeddings only augment the memory elements, a proportional smaller amount of BIST specific logic is added to the circuit.

The conjecture that bigger circuits should require more test cycles to achieve the same level of fault coverage does not hold for the C6 embedding of the 8- and 16-bit CORDIC processors. The 16-bit C6 embedding requires fewer test cycles than the 8-bit C6 embedding.

Another discrepancy between the 8-bit and 16-bit embeddings is the relationship between the test cycles of C5 and C4. In the 8-bit case, C4 requires fewer cycles than C5,

while the converse is true for the 16-bit case.

Table 5.9 CORDIC Embedding Results

16-bit CORDIC Processor faults=3326 gates=1296				
Architecture	Embedding	UnE/UnD	E/D Cycles	Gate Overhead %
CSTP	C2-16	0/0	2109/2109	20.3
CSTP	C3-16	4/4	18612/18612	17.8
BEST	C4-16	0/0	6061/6061	33.4
BEST	C5-16	0/0	2511/2511	20.6
BEST	C6-16	0/0	8071/8071	18.1

The results of this section demonstrate that BIST embeddings can still be effective even after modifying the functional circuit.

5.2.5 Summary of the CORDIC Case Study

A summary of the results of the CORDIC case study are:

- The gate overhead of 25.6% for the C3 embedding and gate overhead of 26.4% for the C4 embeddings are very close. Since embedding C3 uses the CSTP architecture and embedding C4 uses the BEST architecture, the closeness of their gate overheads suggest that BEST architecture can sometimes be competitive with the CSTP architecture. The BEST architecture allows a faster fault simulation algorithm to be used, thus reducing the design costs.
- The CSTP embedding, C3, requires 5.27 times fewer test cycles to achieve 100% fault coverage than the BEST embedding, C5. This result again shows the advantage of using fault simulation to measure an embedding's test effectiveness.
- For the CORDIC circuit, Table 5.6 shows that changing the seed values affects the test length of the CSTP architecture more than the BEST architecture. A possible reason for this result is that fact the next state function of the CSTP architecture is nonlinear, while the BEST next state function is linear.

- The intrusive embeddings C7, C8, and C9 all achieve lower gate overheads than the embeddings C1 through C6 which only modify the registers. These embedding show that for some circuits a BIST mode can be realized by configuring the functional circuitry as an autonomous finite state machine. This approach has a nice maxim: "No BIST, Is the best BEST!"
- The simulation study of varying the word size for the CORDIC and using the same BIST architecture shows that random pattern generation BIST embeddings can be robust with respect to circuit changes.

5.3 Greatest Common Divisor Case Study

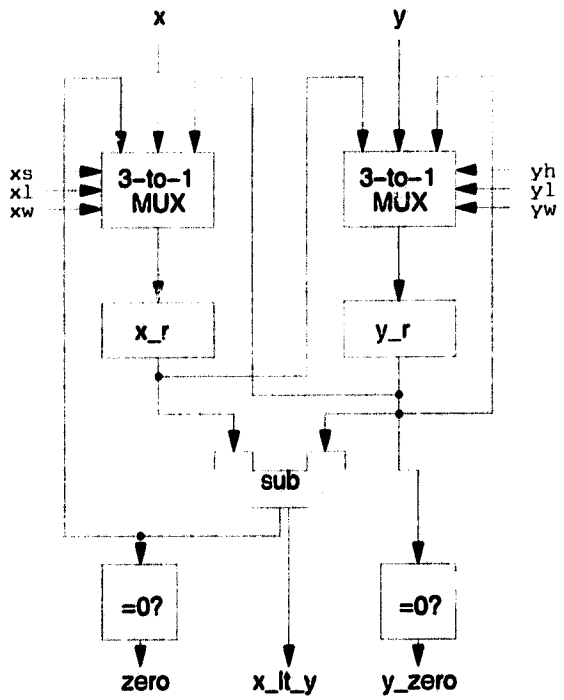
5.3.1 GCD Design

This case study is based on the common high level synthesis benchmark of computing the GCD (Greatest Common Divisor) of two positive integers. Euclid's algorithm is used. The remainder operation is implemented by repeated subtraction. The block diagram shown in Figure 5.8 gives the data path part of the GCD circuit. The circuit is composed of two n-bit 3-to-1 MUXes, two n-bit registers, two n-bit zero checkers (i.e., multi-input NOR circuit) and one n-bit subtractor.

The GCD circuit's operation is described by the following C code. `swap` exchanges the contents of the two registers.

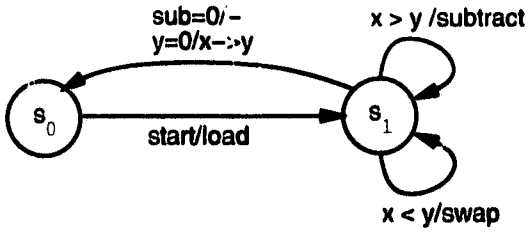
```
while( !start )
       ;
finish = 0;
x_r = x; y_r = y;

while( (x_r-y_r) != 0 && y_r !=0 ){
    if( x_r > y_r ) x_r = x_r - y_r;
    else swap(x_r,y_r);
}
if ( y_r == 0 )
    y_reg = x_reg;
finish = 1;
```

Figure 5.8 GCD Circuit Block Diagram



The state diagram that implements this algorithm is given in Figure 5.9. The most significant borrow of the subtractor is used to test if `x_r > y_r`. The values on the `x` and `y` lines are loaded into the respective registers when the `start` line goes high. The `y_r` contains the GCD of the values initially loaded into the `x_r` and `y_r` registers.

Figure 5.9 GCD Controller



The top level Logic iII(UVic) code is:

```
net_module gcd( start : input;
    x : array [] of input; y : array [x'first..x'last] of input;
    xy_gcd : array [x'first..x'last] of output; notbusy : output; );
var
    x_r, y_r : array [x'first..x'last] of reg;
    x_r_in, y_r_in : array [x'first..x'last] of node;
    xy_sub : array [x'first..x'last] of node;
    xs, xl, xw : node; // s - Subtractor, l - Load, w - sWap
    yh, yl, yw : node; // h - Hold, l - Load, w - sWap
    zero0, zero1, zero, x_lt_y : node;
    i : integer;
begin
    mux3_array(xy_sub, x, y_r, xs, xl, xw,  x_r_in);
    mux3_array(y_r, y, x_r,    yh, yl, yw,  y_r_in);
    for i := x'first to x'last do x_r[i] <- x_r_in[i];
    for i := x'first to x'last do y_r[i] <- y_r_in[i];
    subtractor( x_r, y_r, xy_sub, x_lt_y);
    Nor( xy_sub, zero0);
    Nor( y_r, zero1);
    o2(zero0, zero1, zero);
    gcd_controller(start, zero, zero1, x_lt_y,
        xs, xl, xw, yh, yl, yw, notbusy);
    connect( y_r, xy_gcd);
end.
```

5.3.2 BIST Embedding for GCD

The combinational embedding, **comb_embed**, is again used to determine how effective pseudorandom sequences are as test patterns for the GCD circuit. The embedding results for the GCD circuit where the word size varies from 4 to 32 bits is given in Table 5.10. The maximum number of simulation cycles is set to 1048576.

Table 5.10 GCD Combinational Embedding (G1) Results

Word Size	Gates	Faults	Exposed(un)	Cycles
4	52	283	283(0)	202
6	74	415	415(0)	376
8	92	543	543(0)	1495
10	118	679	679(0)	14354
12	136	807	807(0)	14202
14	154	935	935(0)	113695
16	172	1063	1063(0)	445376
18	198	1199	1193(6)	1048576
20	224	1335	1311(24)	1048576
22	242	1463	1426(37)	1048576
24	260	1591	1566(25)	1048576
26	278	1719	1682(37)	1048576
28	296	1847	1782(65)	1048576
30	314	1975	1925(50)	1048576
32	332	2103	2066(37)	1048576

For GCD circuits with word size greater than 18, the pseudorandom sequence with 2^{20} patterns is unable to expose all faults in the circuits. The unexposed faults all occur inside the zero detector (i.e, the NOR circuit). Note that all other faults are testable by a pseudorandom source. The faults in the NOR logic are not caused by redundant logic, but by the testing difficulties of wide NOR circuits.

Specifically, the stuck-at-0 faults on the NOR gate inputs can only be exposed with one pattern. Exposing the stuck-at-0 fault at input i of an n-input NOR gate requires the pattern $(0_1, 0_2, \cdots, 1_i, \cdots, 0_{n-1}, 0_n)$. Thus the stuck-at-0 fault for a n-input has a probability

of $1/2^n$ of being exposed (assuming a probability of .5 for any input being 1). Also, the stuck-at-1 fault on any input or output of the NOR circuit, can only be exposed with the all zeros pattern. For large n, the probability of the BIST stimulus source generated these pattern is negligible.

The designer of an embedding has several options to improve the testability of the large NOR circuits:

- Assuming the NOR circuit is implemented as a tree, add a BIST mode that converts some of the internal OR gates into EXOR gates.
- Use a weighted pseudorandom source as the stimulus source.
- Use a deterministic stimulus source.

A three-input circuit with an AND-OR-INVERT gate and two NAND gates can be used to realize both a NOR gate and an EXNOR gate. NOR and NAND gates can be used in construction of a wide NOR circuit. The Logic III(UVic) code is:

```
net_module ex_nor(x,y,c : input; o : output);
var
    x_,y_ : node;
begin
    ai2(c,x, x_);
    ai2(c,y, y_);
    aoi22(x,y_, x_,y, o);
end.
```

The NOR circuit in this design is composed of a tree of two-input internal gates, and two-input to four-input NOR gates connected to the NOR circuit inputs. One BIST approach is to convert the internal nodes into **ex_nor** and **ex_nand** gates. The result of changing the NOR circuit to a Bisted_NOR circuit and repeating the combinational embedding simulations is given in Table 5.11. The faults undetected during BIST mode are disabled (e.g., the **c** input of the **ex_nor** gate).

Table 5.11 GCD NOR with a BIST mode Embedding (G2) Results

Word Size	Gates	Faults	Exposed	Cycles
4	52	283	283	202
6	78	435	435	697
8	96	563	563	2027
10	130	739	739	406
12	148	867	867	515
14	166	995	995	2600
16	184	1123	1123	6696
18	218	1299	1299	923
20	252	1475	1475	810
22	270	1603	1603	1209
24	288	1731	1731	11540
26	306	1859	1859	10667
28	324	1987	1987	1701
30	342	2115	2115	1517
32	360	2243	2243	22035

Note that all of the faults are exposed in the GCD circuit with the modified NOR circuits. The test length is also considerably smaller for the GCD circuits with a word size greater than 10. The gate overhead varies from 5% to 8.5%.

The embedding of a weighted BIST stimulus source is identical to that of the combinational embedding with the exception that the stimulus source has a different output weight distribution. Table 5.12 presents the test effectiveness results for the CGD circuit with a weighted LFSR stimulus source. The outputs of the `x_r` and `y_r` registers are driven by a LFSR stimulus source where the probability of producing a 1 is 0.25 for all of the outputs except the output in position 2. Position 2's probability of producing a 1 is 0.5.

The stuck-at-0 fault associated with the third input to the NOR circuit connected to the `y_r` register is unexposed in this BIST embedding with 1048576 patterns. The results reported in Table 5.12 do not consider this fault. This embedding achieved 100% fault coverage for the remaining faults for all of the GCD circuits, excluding the GCD circuits where the word size is 28 and 30.

Table 5.12 GCD Weighted Pseudo Random Embedding (G3) Results

Word Size	Gates	Faults	Exposed(un)	Cycles
4	58	284	284	761
6	84	416	416	842
8	106	544	544	999
10	136	680	680	1848
12	158	808	808	1514
14	180	936	936	26263
16	202	1064	1064	17031
18	232	1200	1200	30921
20	262	1336	1336	41289
22	284	1464	1464	141623
24	306	1592	1592	105607
26	328	1720	1720	78087
28	350	1848	1840(8)	1048576
30	372	1976	1968(8)	1048576
32	394	2104	2104	368591

An embedding in which all of the BIST stimulus sources outputs are set to a probability of 0.25 results in a set of faults in the `subtractor` becoming not exposable. In general, more than one set of weights (see Section 2.2.4) is required to achieve 100% fault coverage. The weights are adjusted by observing the location of the faults in the `subtractor` and changing the associated stimulus output weights.

Using the weighted generator or modifying the OR circuitry can increase the testing effectiveness of the NOR circuitry. The weighted generator is less costly in terms of overhead, but it does require a longer test session. In this case, it seems that modifications to the NOR circuitry is simpler from the designers point of view and result in an embedding with better testability properties (i.e., shorter test length, testable by an unweighted pseudorandom generator).

Constructing a deterministic generator is the final option considered in this section for testing the NOR circuitry. Since the NOR circuitry can be tested with an all zeros pattern, and a set of patterns with only one input set to one and all others set to zero, a complete

LFSR with its initial state set to zero generates the required patterns. Setting the initial state for a normal LFSR to one in its first shift position and all zeros in the remaining positions generates all test pattern except the all zeros pattern. The normal LFSR is cheaper than the complete LFSR, and if the all zeros pattern appears in the pseudorandom sequence, then the normal LFSR can be used.

The Logic III(UVic) code that implements this *deterministic* BIST embedding (embedding G4) is:

```
net_module gcd(
     start : input; x : array [] of input;
     y : array [x'first..x'last] of input;
     xy_gcd : array [x'first..x'last] of output; notbusy : output; );
var
    x_r,y_r : array [x'first..x'last] of reg;
    x_r_in,y_r_in, xy_sub : array [x'first..x'last] of node;
    xs, xl, xw : node; // s - Subtractor, l - Load, w - sWap
    yh, yl, yw : node; // h - Hold, l - Load, w - sWap
    zero0, zero1, zero, x_lt_y : node;
    notbusy_in, zero_m : node; zero_r, notbusy_r:reg; // BIST obs
    notbusy_in : node;
    i : integer;
begin
    connect(notbusy, notbusy_r);
    mux3_array(xy_sub, x, y_r, xs, xl, xw,  x_r_in);
    mux3_array(y_r, y, x_r,    yh, yl, yw,  y_r_in);
    subtractor( x_r, y_r, xy_sub, x_lt_y);
    Nor( xy_sub, zero0);
    Nor( FFan(y_r), zero1);
    o2(zero0, zero1, zero);
    use gen_misr_embed in begin
        gcd_controller(start, zero_m, zero1, x_lt_y,
            xs, xl, xw, yh, yl, yw, notbusy_in);
        notbusy_r <- notbusy_in; zero_r <- zero; // BIST
        mux2(gen_on[gen_on_index], zero, zero_r, zero_m); //BIST

        for i := x'first to x'last do y_r[i] <- y_r_in[i];
        for i := x'first to x'last do x_r[i] <- x_r_in[i];
    end;
    connect( y_r, xy_gcd);
end.
```

Note that a gen_misr_embed embedding module is used. The generation and response observation roles are separated to allow the LFSR to generate the required sequence. The

initial evaluation of the deterministic embedding by **lg3** shows that some faults still escape detection. This is attributed to the poor observability of the **zero** and **notbusy** lines. Once observation points are added, the above embedding, G4, detects all the detectable faults in 2141 cycles. The simulation results of the **lg3** CAD tool are:

```
word_size= 16
gates=520 cycles=2200 faults=1005 exposed=1003 detected=1003
S@0:T.gcd0148.mux20127.aoi220269.fan[0]
S@1:T.gcd0148.mux20127.aoi220269.fan[0]
```

The two undetected faults are inputs of the MUX inserted to add the observation points, and thus cannot be tested in BIST mode.

The result of embedding the CSTP BIST architecture and using the NOR circuit that can be converted into a NOR-EXNOR in embedding G5 is:

```
gates=418 cycles=4096 faults=1059 exposed=1059 detected=1059
Execution Resources: 1052.63 user 0.31 sys
```

All of the considered faults are detected in 1655 cycles. The code for Embedding G5 is:

```
circuit g5;
var
    start : input; s_r : node;
    x, y : array [1..word_size] of input;
    x_r, y_r : array [1..word_size] of node;
    xy_gcd : array [1..word_size] of output;
    finished_in :  node; finished_r : reg; finished : output;
faults_off
    x,y,start, finished;
begin
    connect(finished_r[1], finished);
    assert( init_cstp(1) = 1);
    use cstp_scan_embed in begin
        n_reg([start], [s_r]); n_reg(x, x_r); n_reg(y, y_r);
    end;
    use cstp_embed in begin
        gcd(s_r, x_r,y_r, xy_gcd, finished_in[1]);
        finished_r[1] <- finished_in[1];
    end;
    finish_cstp(1);
end.
```

Table 5.13 GCD Embedding Results

16-bit GCD Processor gates=249				
Architecture	Embedding	Unexpose Undetected	E/D Cycles E/D Cycles	Gate Overhead %
Combinational	G1	0/0	445376/-	-
NORS-EXNORS Combinational	G2	0/0	6696/-	-
Weighted Combinational	G3	0/0	17031/-	-
Deterministic	G4	0/0	2141/2141	108.8
NORS-EXNORS CSTP	G5	0/0	1655/1655	67.8

Note that **gcd** module does not require any modifications.

Table 5.13 presents the test effectiveness results of the embeddings for the GCD circuit. Embedding G1 through G3 can be implemented with either a STUMPS or a BEST architecture. The unweighted pattern generator used in G1 requires the longest test time, but also uses the simplest pattern generator. The test time can be reduced by either using a weighted pattern generator (G2) or modifying the difficult to test NOR circuit (G3). G3 is slightly more expensive than G2 but requires fewer test cycles.

The results of embeddings G4 and G5 indicate that using the CSTP architecture and modifying the NOR circuit realize a cheaper embedding in terms of test time and gate overhead than the deterministic approach.

5.3.3 Summary of the GCD Case Study

The difficulty encountered in adding a BIST mode to the GCD is caused by the wide NOR circuit used to detect zeros. The three methods that solve the testing of NOR circuit are:

- Modify the NOR circuit by adding EXOR gates.
- Use a weighted random-pattern stimulus source.
- Construct a deterministic pattern stimulus source.

The modified NOR circuit can be added to the designers library, to be used the next time a wide NOR circuit is required.

5.4 Bertossi's String Matching Algorithm

The final case study consists of an implementation of a string matching system proposed by Bertossi[Ber90] as a directed studies project by Bill Gardner[Gar93]. Appendix D contains the Logic III(UVic) source code for the implementations. The ability of Logic III(UVic) to specify designs recursively is used extensively in this implementation.

Figure 5.10 2 × 4 String Comparison Circuit

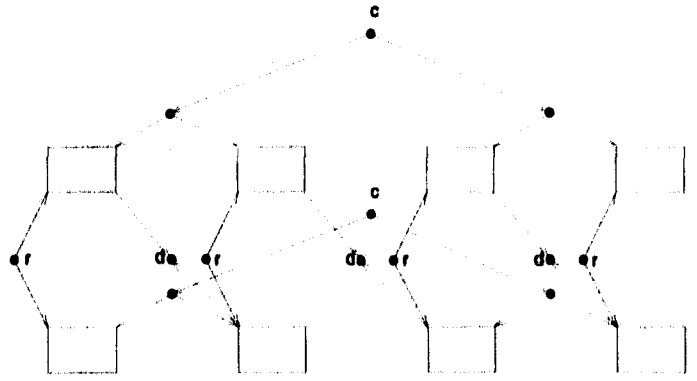


Figure 5.10 shows the interconnection of a 2×4 string comparison circuit. The pattern's size is 2 and the text's size is 4. The squares in Figure 5.10 represent comparison circuits. Binary trees are used to connect this circuit together. The inputs labeled, r, are the text input, and the inputs labeled, c, are the pattern inputs. The results of the comparison appears on the output labeled, d. Each input and output is the root of a binary tree. The data from the r and c input flows down towards the comparison, and the results flow towards the d outputs.

Examination of the interconnection structure of this design, yields the fact that the structure is an acyclic graph with the inputs as source nodes, and the outputs as sink

nodes. This structure suggests adding a BIST pattern generation structure at the primary inputs, and a BIST response analysis structure at the primary outputs.

The test script that evaluates this embedding is:

```
testing
begin
        [ P_init_, P_data_, T_init_, T_data_ ] :- ca(1);
        fault_simulate( cycles );
        write("gates= ", number_of_gates, " ");
        write("cycles= ", number_of_cycles, " ");
        write("faults= ", number_of_faults, " ");
        write("exposed= ", number_of_exposed, " ");
        write("detected= ", number_of_detected, "\n");
        print_non_exposed;
end.
```

The design is customized such that any $m \times n$ string matching circuit can be constructed. The fault simulation results of three different $m \times n$ circuits are presented in Table 5.14. For the 8×16 circuit, the following three pattern generator are used: a LFSR, a LFSR with its inputs permuted by the cross-over permutation, and a LHCA. Only the LHCA is used as the generator for the other circuits.

Table 5.14 Bertossi's String Matching Embedding Results

Generator	$m \times n$	Gates	Flip-flops	Faults	Exposed	Cycles
ca(1)	8×16	980	596	3196	3196	627
lfsr(1)	8×16	980	596	3196	3126	262144
x-lfsr(1)	8×16	980	596	3196	3196	836
ca(1)	8×32	2196	1316	7260	7260	1072
ca(1)	8×64	4628	2756	15388	15388	1329

In this case study no BIST circuit is embedded in the design, instead the `lg3` CAD tool is used to determine the external testability of the design. Since the design is easily testable by external stimulus, the simplest BIST embedding is to add a generator and response analyzer to the boundary of the circuit.

One interesting result is that the LHCA is a better pattern generator than the LFSR. This can be attributed to the correlation in adjacent lines in the sequence produced by the

LFSR. Note that the LFSR outputs are permuted so that the output lines are no longer adjacent (i.e., the cross-over permutation) the LFSR's test effectiveness is similar to the LHCA's test effectiveness.

Also listed in Table 5.14 is the number of flip-flops required for each circuit. The large number of flip-flops makes a full scan based design very expensive.

5.4.1 Summary of the String Matching Case Study

- For some circuits, simply adding generators and response analyzes to the primary inputs and primary outputs is sufficient to achieve very good fault coverage.
- For some very special circuits, the correlation between adjacent bits of a LFSR can result in test patterns that are not effective.

Chapter 6

Specific BIST Architectures

6.1 Two-Pattern BIST Generator Properties

The sequence of state encodings of an autonomous finite state machine (AFSM) is used as the source of test patterns in BIST applications. Test pattern generation for delay faults requires two successive patterns, one to setup the fault, the second to test the fault. Thus the fault coverage for delay faults depends on the AFSM state sequence. The class of CMOS stuck-open faults is an example of fault types that require two test patterns for detection.

Given a BIST generator, G, with n state variables, its effectiveness as a pattern generator for delay faults depends on the pair of successive states in its state sequence. Let S be a set, containing n state variables $\{s_1, s_2, s_3, \cdots, s_n\}$ and let X denote the state encoding sequence. The j-th element of the sequence is given by x_j, where $x_j = (s_{1_j}, s_{2_j}, \cdots, s_{n_j})$. s_{1_j} is the value of the state variable s_1 at the j-th position in the sequence. The state pair (x_j, x_{j+1}) is also referred to as transition t_j.

Example 6.1 *Let S contain the three state variables $\{s_1, s_2, s_3\}$ and let its state sequence be $X = (010, 001, 100, 110, 111, 011, 101, 010)$. An example of a transition is $t_2 = (001, 100)$.*

Note that $v_1 = 010$ and $x_8 = 010$ are the same, thus X repeats after 7 transitions (assuming that the FSM is autonomous).

The following theorems are useful in counting the number of transitions for an n state variable AFSM.

Theorem 6.1 *The maximum number of transitions for an AFSM with n state variables is 2^n.*

Proof:

Since only 2^n possible encodings exist for n state variables, and in the AFSM only one successor state is possible, there is a maximum of 2^n transitions. This follows directly from the fact the next state depends only on the current state of the AFSM.

Theorem 6.2 *If the AFSM's state sequence X contains k elements before repeating, then only k transitions are possible.*

Proof:

Obvious.

An n state variable AFSM can be used as the stimulus source for a n input l output digital circuit C. For the outputs of C, that depend on all the n inputs, a maximum of 2^n transitions (Theorem 6.1) can be applied. However, if one of C's outputs depends on only m where $m < n$ inputs, then an important question is: "how many distinct transitions can be applied to the m inputs?"

A general method to determine the number of transitions for a subset of the state variables of a LFSM is presented in [FM91]. A particularly interesting subset of the state variables for a LFSM are windows into an ordering of the state variables. Given the following

ordering for state variables of LFSRs and LHCA $(s_1, s_2, \cdots s_n)$, a window $w(i,k)$ contains the state variables from position i to $i+k-1$ (i.e., $(s_i, s_{i+1}, \cdots s_{i+k-1})$. In [ZBM92], Zhang, Byrne, and Miller derive expressions for the number of transitions for the windows of internal LFSRs, external LFSRs, and LHCA. The following sections reprove the results in [ZBM92] using the results in [FM91].

The next state function for LFSMs can be expressed by matrix multiplication over GF(2) (see Section 2.2.5). That is:

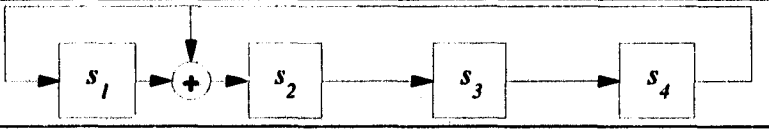
$$s^+ = s \cdot T.$$

The next state function of a window $w(i,k)$ is:

$$w(i,k)^+ = s \cdot T', \tag{6.1}$$

where T' is T with the j-th columns deleted where $1 \leq j < i$ and $(i+k)-1 < j \leq n$.

Figure 6.1 Internal LFSR for $x^4 + x + 1$



For example, the next state function for the internal LFSR depicted in Figure 6.1 is:

$$(s_1, s_2, s_3, s_4)^+ = (s_1, s_2, s_3, s_4) \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}.$$

The function for the window $w(2,2)$ is:

$$w(2,2)^+ = (s_2, s_3)^+ = (s_1, s_2, s_3, s_4) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}. \tag{6.2}$$

Note that the columns corresponding to the nonwindow state variables, s_1 and s_4 (i.e., 1 and 4), have been removed.

To use the result in [FM91], Equation 6.1 needs to be rewritten to separate the window variables from the nonwindow variables. These variables are called visible and nonvisible in [FM91]. If $\overline{w}$ represents the nonwindow variables, then equation (4) of [FM91] can be written as:

$$w(i,k)^+ = w(i,k) \cdot T_w + \overline{w}(i,k) \cdot T_{\overline{w}}.$$

Theorem 1 of [FM91] states that the number of transitions of $w(i,k)$ from every state encoding of $w(i,k)$ equals 2^r, where $r = \text{rank}(T_{\overline{w}})$.

Rewriting Equation 6.2 to separate the variables yields:

$$w(2,2)^+ = w(2,2) \cdot \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + \overline{w}(2,2) \cdot \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}.$$

Since the rank of $T_{\overline{w}}$ is 1, the number of transitions equals $2^r = 2^1$. The total number of transitions equals 2^3, since all 2^2 encodings occur for $w(2,2)$.

Table 6.1 shows the state sequence for the LFSR in Figure 6.1. Note that there are only 8 distinct transitions for the variables (s_2, s_3).

6.2 Counting the Transitions in a Window

This section analyzes the number of transitions for windows into the state variables of LFSRs and LHCA. Given an ordering for the state variables $\{s_1, s_2, \cdots, s_n\}$, represented by the tuple $(s_1, s_2, \cdots, s_n)$, a window of length k starting at the left position i is denoted by $w(i,k)$. For example, $w(2,2)$ of the tuple (s_1, s_2, s_3, s_4) is (s_2, s_3). The number of transitions for each window can be used as an indication of the effectiveness of a LFSR or a LHCA as a two-pattern generator.

Table 6.1 State Sequence for $x^4 + x + 1$.

Position	State Sequence (s_1, s_2, s_3, s_4)	(s_2, s_3) transitions (w, w^+)	First Occurrence
1	1000	(00,10)	1
2	0100	(10,01)	1
3	0010	(01,00)	1
4	0001	(00,10)	0
5	1100	(10,11)	1
6	0110	(11,01)	1
7	0011	(01,10)	1
8	1101	(10,01)	0
9	1010	(01,10)	0
10	0101	(10,11)	0
11	1110	(11,11)	1
12	0111	(11,11)	0
13	1111	(11,01)	0
14	1011	(01,00)	0
15	1001	(00,00)	1
16	1000	(00,00)	0

For an n state external LFSR depicted in Figure 2.6 and an ordering of the state variables according to their shift order, the number of transitions of a window $w(i,k)$ is:

$$w(i,k) = \begin{cases} 2^{k+1} & k < n-1 \\ 2^n - 1 & k \geq n-1 \end{cases} \tag{6.3}$$

The first condition of Equation 6.3 can be proved by showing that the rank of $T_{\overline{w}}$ is equal to 1. Since $k < n-1$, for $w(i,k)$, the state variable s_i depends on s_{i-1} or s_n. The only state variable in $T_{\overline{w}}$ is s_{i-1} or s_n, thus p_{j+1} next state equation is:

$$\begin{aligned} w(i,k)^+ &= w(i,k) \cdot T_w + s_{i-1} \cdot T_{\overline{w}}, \text{ or} \\ &= w(i,k) \cdot T_w + s_n \cdot T_{\overline{w}}. \end{aligned}$$

Since, $T_{\overline{w}}$, is a nonzero row vector, the entry corresponding to s_{i-1} (s_i is the left most state variable in $w(i,k)$) must be nonzero, the rank of $T_{\overline{w}}$ is 1.

The proof for $k = n$, follows directly from Theorem 6.2. For $k = n-1$, if the all zero state encoding is included in the LFSRs state sequence, then the proof for $k < n-1$ could be applied directly, yielding 2^n transitions. Since only the all zero encoding is missing, there are only $2^n - 1$ transitions.

Using the same state variable ordering as the external LFSR, the number of transitions for the window of an internal LFSR is:

$$w(i,k) = \begin{cases} 2^{k+2} & i > 1 \text{ and } k < n-2 \text{ and } \begin{array}{l}\text{at least one EXOR gate is} \\ \text{in the window}\end{array} \\ 2^{k+1} & k < n-1 \text{ and no EXOR gates are in the window,} \\ 2^n - 1 & k \geq n-1. \end{cases} \tag{6.4}$$

The last two conditions for Equation 6.4 correspond to an external LFSR, so the preceding proofs can be used. For the first condition:

$$w(i,k)^+ = w(i,k) \cdot T_w + (s_{i-1}, s_n) \cdot T_{\overline{w}}.$$

Since s_{i-1} and s_n determine the next state of different state variables, the two row vectors in T_w are not identical, thus the rank of T_w is 2. From Theorem 1 in [FM91] the number of transitions for $w(i,k)$ is $2^2 \cdot 2^k = 2^{k+2}$ assuming that all state encodings for $w(i,k)$ occur.

If the state variables of a LHCA are labeled in their linear order, (i.e., s_{i-1} and s_{i+1} determine the state of s_i), then the number of transitions in the window $w(i,k)$ for a LHCA is:

$$w(i,k) = \begin{cases} 2^{k+2} & k < n-1 \text{ and } i > 1 \text{ and } i < (n-k)+1 \\ 2^{k+1} & k < n-1 \text{ and } (i = 1 \text{ or } i = (n-k)+1) \\ 2^n - 1 & k \geq n-1 \end{cases} \quad (6.5)$$

The first condition of Equation 6.5 can be proved by noting that the next state of $w(i,k)$ depends on the two state variables, s_{i-1} and s_{i+k}, the variables to the immediate left and right of the window. Since the outside state variables determine the next state of different window state variables, the rank of T_n is 2. For the second condition, the window is in either the leftmost or rightmost position and only one state variable outside the window affects the window's next state. Thus the rank of T_n is 1. The last condition is similar to the last conditions of the internal and external LFSR, so an equivalent proof can be used.

6.3 Windows with Maximum Transitions

The maximum number of transitions for a window $w(i,k)$ is 2^{2k}. If all possible transitions are applied to a circuit, then all two-pattern detectable faults should be detected. To produce 2^{2k} transition with an n state variable BIST generator requires that $k < \lfloor n/2 \rfloor$. The generator must have a distinct state for each transition in a window.

If the rank of T_w in the next state equation of a LFSM BIST generator is k for a window of size k, then that window has 2^{2k} transitions. Thus if the state variables of a LFSM can be ordered so that the rank of $T_w = k$, then a window with the maximum number of transitions is possible.

For example, an internal LFSR using the polynomial $x^7 + x + 1$ with 7 state variables $(s_1, s_2, \cdots, s_7)$, and ordering of $(s_1, s_3, s_5, s_7, s_2, s_4, s_6)$, produces all $2^{3\cdot3}$ transitions for the window[1] (s_1, s_3, s_5). The next state equation is:

$$(s_1, s_3, s_5)^+ = (s_1, s_3, s_5) \cdot T_w + (s_2, s_4, s_6, s_7) \cdot T_{\overline{w}}.$$

and

$$T_{\overline{w}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Since the rank of $T_{\overline{w}}$ is 3 and the LFSR state sequence contains $2^7 - 1$ state encodings, the window (s_1, s_3, s_5) produces all possible transitions. Therefore the internal LFSR with the $x^7 + x + 1$ polynomial has at least one window with all possible transitions.

This ordering is called the *cross-over* ordering. Given the $(s_1, s_2, \cdots, s_n)$ state ordering, the cross-over ordering is:

$$(s_1, s_3, \cdots, s_n, s_2, s_4, \cdots, s_{n-1}) \quad n \text{ is odd}$$
$$(s_1, s_3, \cdots, s_{n-1}, s_2, s_4, \cdots, s_n) \quad n \text{ is even}$$

To prove that the cross-over permutation produces windows with maximum transition for all n state variable internal LFSRs, external LFSRs and LHCA where the characteristic polynomial is primitive, one must show that there exists at least one window where the associated $T_{\overline{w}}$'s rank is $k = \lfloor n/2 \rfloor$. n can be assumed to be odd without loss of generality. The matrix used to compute the next state for an arbitary external LFSR is:

$$T = \begin{bmatrix} c_{n-1} & 1 & 0 & \cdots & 0 \\ c_{n-2} & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & \\ c_1 & 0 & 0 & & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

[1]The meaning of a window is a bit stretched in this case.

The $T_{\overline{w}}$ matrix for the window $(s_2, s_4, \cdots, s_k)$ is created by choosing the even columns and then removing the even rows. This is illustrated by:

s_1	s_2	s_3	s_4	$\cdots$	s_n
c_{n-1}	1	0	0	$\cdots$	0
~~c_{n-2}~~	~~0~~	~~1~~	~~0~~	~~$\cdots$~~	~~0~~
c_{n-3}	0	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
~~c_1~~	~~0~~	~~0~~	~~0~~	~~$\cdots$~~	~~1~~
1	0	0	0	0	0

The even columns are outlined and the even rows are crossed out. The even columns in the transition matrix for an external LFSR contain a single one at row position $i - 1$ if the column position is i. The rows not crossed out from the $T_{\overline{w}}$ matrix. Since the single 1 in each column is in an odd row and each 1 is in a different row, the rank of $T_{\overline{w}}$ matrix is equal to the number of columns. Since there are k columns, the rank is equal to k. Assuming that all possible 2^k encodings occur for w and using Theorem 1 from [FM91], the number of transitions for w is equal to 2^{2k}. Thus the cross-over permutation produce a window with all possible transitions.

Similar arguments can be used for internal LFSRs and LHCA. The cross-over ordering can therefore be used to produce a two-pattern BIST generator where windows of size $k = \lfloor n/2 \rfloor$ produces all possible transitions.

6.4 DFA based Response Analysis

A new method for detecting errors in the test response of a circuit during BIST application is presented in this section. Existing methods of detecting errors in the test response are found in Section 2.3. Figure 6.2 shows a typical off-line BIST architecture, with the addition of a deterministic finite automaton (DFA) that monitors an output of the compactor. The DFA [LP81] is designed to accept a language that includes the string (i.e., the test sequence)

produced by a fault-free circuit under test. An error is detected if in the presence of a fault, the string produced is not in the language accepted by the DFA. Ideally, the number of strings accepted by the DFA should be as small as possible in order to improve error detection.

Figure 6.2 BIST Architecture

The output response of the fault-free circuit is used to customize one of the following four types of DFAs:

updn The DFA's input drives an up/down counter. 0 counts up and 1 counts down. The string is accepted as long as the counter's value remains within a given range.

run0 A DFA that counts the number of consecutive zeros. The string is accepted if the count is less than a maximum value.

run1 A DFA that counts the number of consecutive ones. The string is accepted if the count is less than a maximum value.

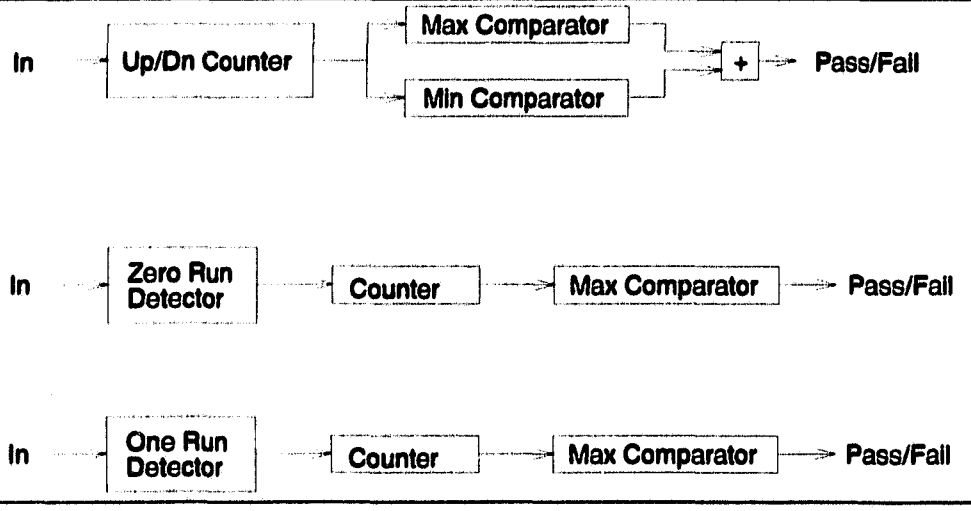
run2 A DFA that combines **run0** and **run1**.

These four DFA types are chosen for their low cost hardware implementations. The two customizing parameters for the *updn* DFA's are its maximum, C_{ub}, and minimum, C_{lb}, count values. The maximum number of consecutive zeros/ones customize the *run0*/*run1* DFAs. The *run0* and *run1* DFAs parameters are R_0, and R_1, respectively. The two parameters, R_0, and R_1, are used to customize the *run2* DFA. Section 6.4.1 discusses the implementation details for these DFAs.

6.4.1 DFA BIST Architecture

Customizing one of the DFA types consists of performing a fault-free simulation and measuring the associated parameter (e.g., C_{lb}, C_{ub}, R_0, and R_1). Given fixed values for the parameters, a particular DFA can easily be constructed. Figure 6.3 shows the block diagrams for the *updn*, *run0* and *run1* DFAs.

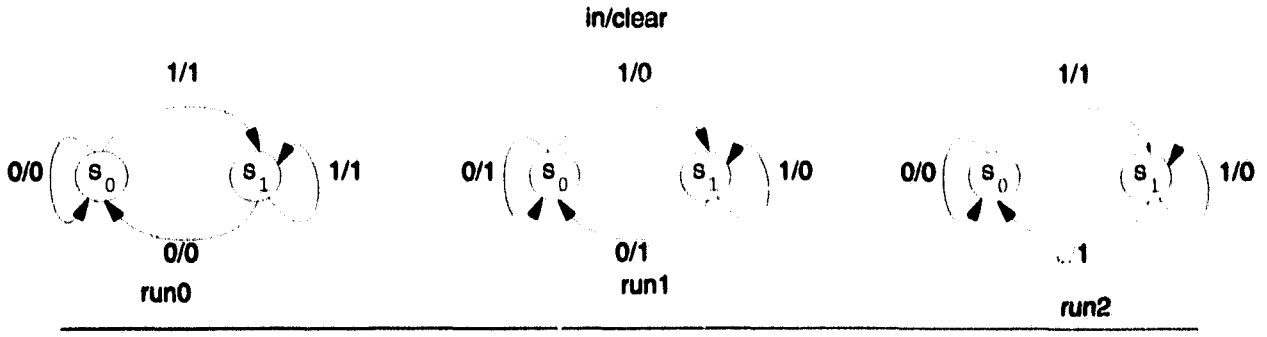
Figure 6.3 DFAs Block Diagrams



The *updn* DFA is composed of an up/down counter, and two comparators used to determine if the allowed range is exceeded. An up/down LFSR [Nic92] can be used rather than a binary up/down counter with significant area saving.

The *run0*, *run'* and *run2* DFAs can be implemented by a zero/one/zero-one sequence detector, a clearable counter and a comparator. State diagrams for these detectors are shown in Figure 6.4. The sequence detector is attached to one of the compactor's outputs and controls the counter by allowing the counter to advance during the run of ones and zeros. Again a LFSR can be used as the counter. Notice that the *run2* sequence detector only clears the counter when a run of zeros or ones finishes. A *run2* DFA's implementation requires a comparator for the maximum ones sequence, and one for the maximum zeros sequence.

Figure 6.4 Sequence Detector State Diagrams

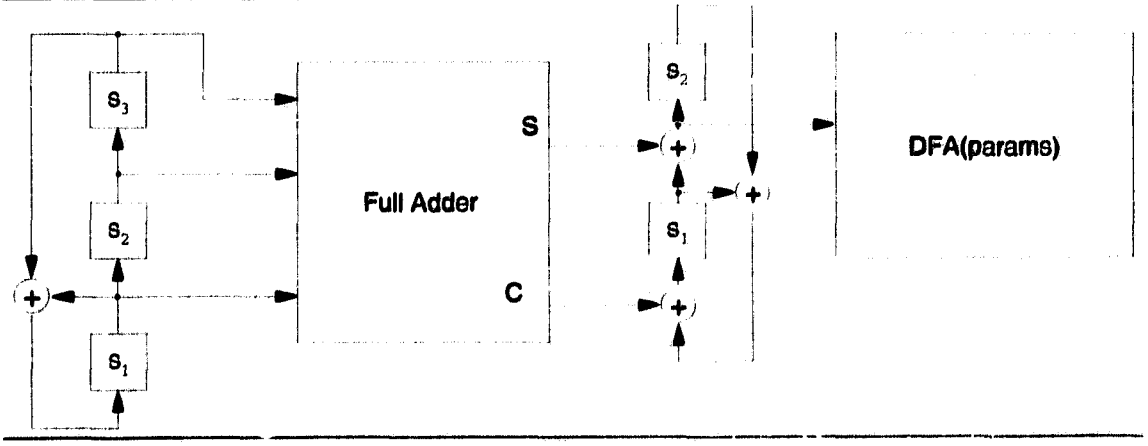


The hardware cost of the DFAs depend on their parameters. For all the DFAs, the value of their parameters determine the size of the counter. The following table gives the counter sizes.

DFA	Parameters	Counter Size
updn	C_{ub}, counters upper bound, C_{lb}, counters lower bound	$\lceil \log_2(C_{ub} - C_{lb}) \rceil$
run0	R_0, maximum number of consecutive zeros	$\lceil \log_2 R_0 \rceil$
run1	R_1, maximum number of consecutive ones	$\lceil \log_2 R_1 \rceil$
run2	R_1, R_0	$\max(\lceil \log_2 R_0 \rceil, \lceil \log_2 R_1 \rceil)$

The DFAs become more expensive as the counter size grows, and thus the larger the value of the DFAs parameters, the greater its cost.

Figure 6.5 Example of a DFA BIST Arrangement



Example 6.2 *Figure 6.5 shows a full adder circuit attached to a BIST generator, a MISR, and a DFA error detector. Table 6.2 gives, the full adder's response to the test inputs, the MISR's state after each input, and the different DFA's parameters for the s_1 MISR output at that point in the fault-free response.*

Table 6.2 Fault Free Simulation and DFA Parameters

(s_1, s_2, s_3)	C S	(s_1, s_2)	R_0	R_1	C
100	0 1	01	0	1	-1
110	1 0	00	1	0	0
111	1 1	11	0	1	-1
011	1 0	11	0	2	-2
101	1 0	11	0	3	-3
010	0 1	00	1	0	-2
001	0 1	01	0	1	-3
100	0 1	11	0	2	-4

The DFA parameters are $R_0 = 1$, $R_1 = 3$, $C_{lb} = -4$, and $C_{ub} = 0$.

The run0 *DFA detector finds any errors that cause s_1 MISR output to produce a run of more than one zeros. Similarly, the* run1 *DFA detects any faults that cause the number of consecutive ones to exceed three. The* updn *DFA finds any faults that cause the up/down counter to reach a count of -5 or +1.*

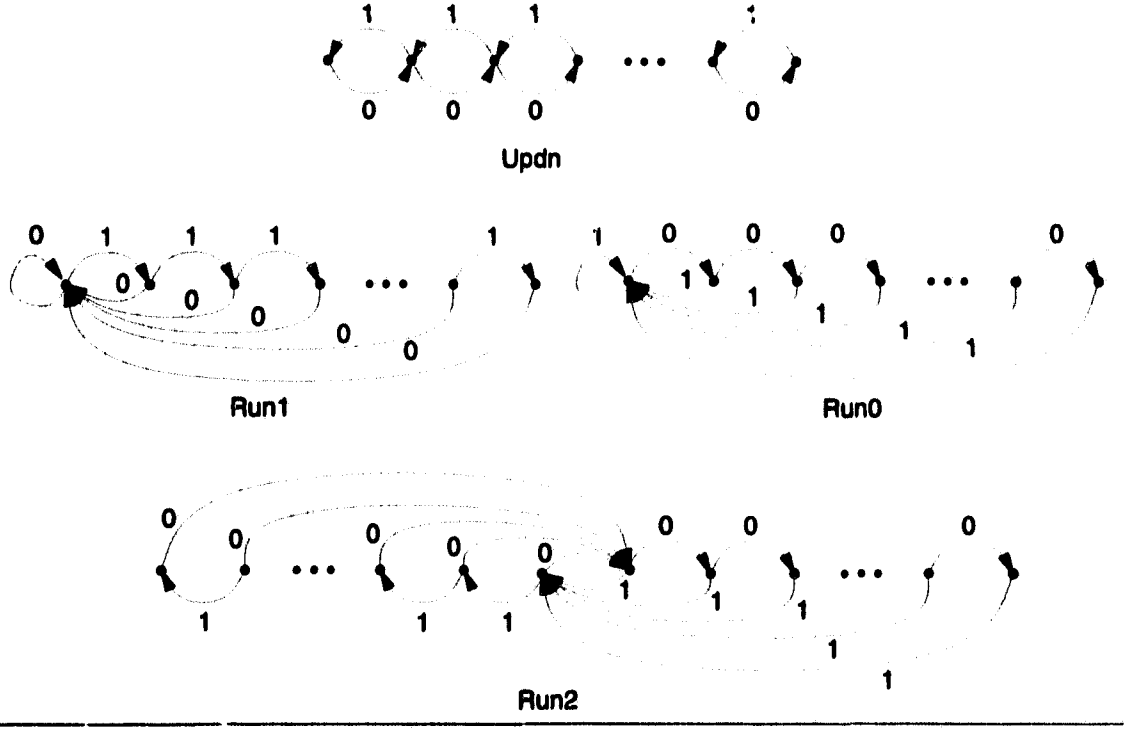
6.4.2 The DFA's Languages

The languages accepted by the DFAs are described by the following regular grammars.

Language	Grammar
updn	$L_1 \rightarrow 0L_2 \mid 0 \;\; L_i \rightarrow 0L_{i+1} \mid 1L_{i-1} \mid 0 \mid 1 \;\; L_n \rightarrow 1L_{n-1} \mid 1$
run0	$L_1 \rightarrow 0L_2 \mid 1L_1 \mid 0 \mid 1 \;\; L_i \rightarrow 0L_{i+1} \mid 1L_1 \mid 0 \mid 1 \;\; L_n \rightarrow 1L_0 \mid 1$
run1	$L_1 \rightarrow 1L_2 \mid 0L_0 \mid 0 \mid 1 \;\; L_i \rightarrow 1L_{i+1} \mid 0L_1 \mid 0 \mid 1 \;\; L_n \rightarrow 0L_1 \mid 0$
run2	$R_1 \rightarrow 1R_2 \mid 0L_1 \mid 0 \mid 1 \;\; R_i \rightarrow 1R_{i+1} \mid 0L_1 \;\; R_n \rightarrow 0L_1 \mid 0$ $L_1 \rightarrow 1R_1 \mid 0L_2 \mid 0 \mid 1 \;\; L_i \rightarrow 0L_{i+1} \mid 1R_0 \mid 1 \mid 0 \;\; L_n \rightarrow 1R_1 \mid 1$

Since each of these languages can be accepted by a DFA, the language can also be described by the state transition diagram of the DFA. Figure 6.6 shows the state transition diagrams for the four languages.

Figure 6.6 State diagrams for updn, run0, run1, and run2.



6.5 DFA Aliasing

The ability of a DFA language recognizer to distinguish between the correct test response and an error response depends on the DFA's aliasing probability. The aliasing probability is the probability that an incorrect test response is accepted as the correct response. Since the error response is detected if the DFA does not accept the response, assuming that all error responses are equally likely, the aliasing probability is given by:

$$\frac{\text{number of strings accepted by the DFA} - 1}{\text{number of all possible strings} - 1}. \tag{6.6}$$

One is subtracted from the numerator and the denominator to eliminate the counting of the correct response. Since the test response consists of only zeros and ones, assuming the string's length is k, Equation 6.6 can be rewritten as:

$$\frac{|\mathcal{L}(\text{DFA})| - 1}{2^k - 1}, \tag{6.7}$$

where $|\mathcal{L}(\text{DFA})|$ is cardinality of the language accepted by the DFA.

In order to determine the aliasing probability of a DFA, the number of strings accepted for a given length, k, must be counted. A method of counting the number of accepted strings is based on counting the number of paths in the DFA's state transition graph. Given a DFA with n states and its state transition graph G, the number of strings of length k is equal to the number of paths of length k from the start state to all accepting states. A path traces out the states the DFA goes through as it is recognizing a string. Given G, the number of strings accepted by the DFA of length k is equal to [Gib85, page 19]:

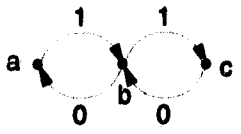
$$\sum_{j \in FS(G)} A^k(i,j), \tag{6.8}$$

where $A^k(i,j)$ is the entry at the i-th row and the j-th column of the adjacency matrix for G raised to the k-th power, $FS(G)$ is the set of final states, and i is the start state.

Using Equation 6.8, the aliasing probability for a particular DFA can be computed by:

$$\frac{\sum_{j \in FS(G)} A^k(i,j) - 1}{2^k - 1} \tag{6.9}$$

Figure 6.7 An UpDn DFA with 3 nodes



Another method of determining the number of paths for a state graph is to write and solve recurrence relations for the path length. For example, given the graph in Figure 6.7

the recurrence relations for the path length from each node are:

$$\begin{aligned} a(k) &= b(k-1) \qquad k > 1 \\ a(1) &= 1 \\ b(k) &= a(k-1) + c(k-1) \qquad k > 1 \\ b(1) &= 2 \\ c(k) &= b(k-1) \qquad k > 1 \\ c(1) &= 1 \end{aligned}$$

By symmetry, $a(k) = c(k)$, therefore $b(k) = a(k-1) + c(k-1)$ can be rewritten as:

$$b(k) = a(k-1) + a(k-1) = 2a(k-1).$$

Since $a(k) = b(k-1)$, the number of paths from **a** of length k is given by:

$$\begin{aligned} a(k) &= 2a(k-2) \quad k \geq 2, \\ a(1) &= 1, a(0) = 0. \end{aligned}$$

The solution for this recurrence is:

$$a(k) = 2^{\lfloor k/2 \rfloor}$$

Thus the aliasing probability of the DFA with the state transition graph shown in Figure 6.7 is:

$$\frac{2^{\lfloor k/2 \rfloor} - 1}{2^k - 1}.$$

The aliasing probability decreases exponentially with linear increases in the string length. In this case, if the response string's length is 32, then the aliasing probability is approximately equal to 2^{-16}. This is the same aliasing probability as a 16-bit LFSR. Unfortunately, the set of the recurrence relations becomes more complex as the size of the graph increases. Their solution also becomes more difficult.

6.6 Computing the Aliasing Probability

Equation 6.9 can be used to compute the aliasing probabilities for a given DFA, however the exponential growth in the number of paths for many DFAs requires arbitrary precision arithmetic. A direct implementation of Equation 6.9 requires arbitrary precision multiplication for the matrix multiplication.

Multiplication can be avoided, with a considerable savings in CPU time, by using the following algorithm. This algorithm is implemented in C++ and is used to count the number of paths of length 1 through k from node v.

```
foreach u ∈ G do
        paths_0(u) = 0
// initialize the paths value for the start node
paths_0(v) = 1
for i = 1 to k do
        foreach v ∈ G do
                sum = 0
                foreach u adjacent to v do
                        sum = sum + paths_{i-1}(v)
                paths_i(v) = sum
        totalpaths_i = 0
        foreach v ∈ G do
                totalpaths_i = totalpaths_i + paths_i(v)
```

totalpaths_i is the number of paths of length i from v.

6.7 Aliasing Probabilities

The following subsections present the aliasing probabilities for the four DFA types used by the DFA error detection technique. Each subsection presents plots of the aliasing probabilities as a function of test response length. The algorithm given in Section 6.6 is used to compute the aliasing probabilities for particular DFA instances. For some of the DFA types, bounds on the aliasing probabilities are also given.

6.7.1 UpDn DFAs

Figure 6.6 shows the state transition graph for the *updn* DFA. A plot of the aliasing probability for *updn* DFA's with parameters set to $C_{lb} = 1$ and C_{ub} to 20 through 180 in steps of 20 and with the DFA's counters initial value set to 1 is shown in Figure 6.8.

Figure 6.8 Aliasing Probabilities of the *updn* DFA, initial state = 1

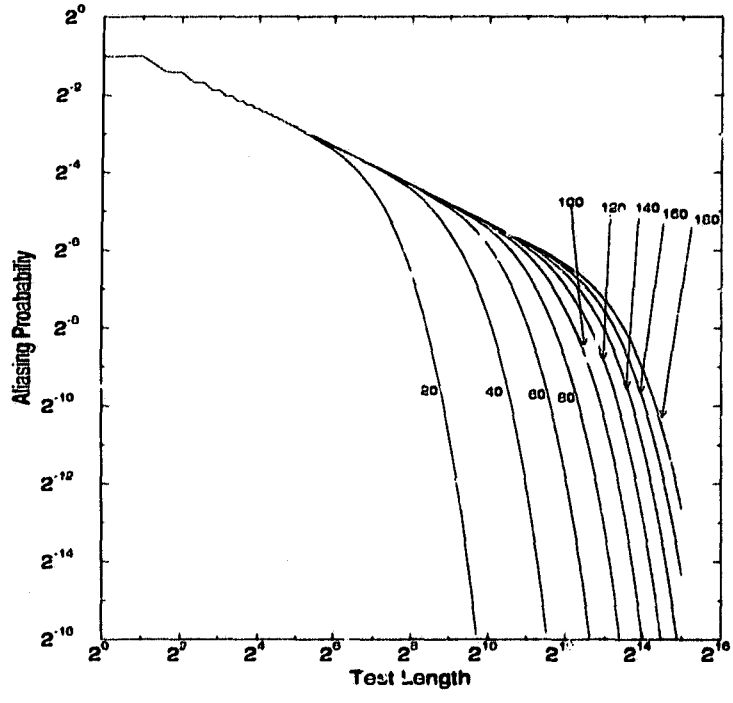
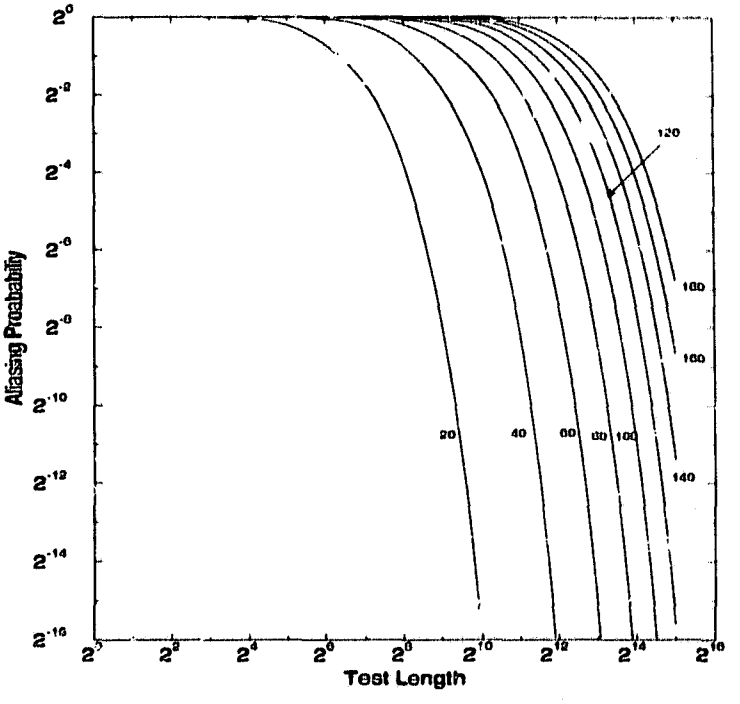


Figure 6.9 shows a plot of the aliasing probabilities for the same parameters as Figure 6.8, but the initial state is set to the DFA's middle state (e.g., for the DFA with parameters $C_{lb} = 1$ and $C_{ub} = 20$ the middle state is 10). Note that the aliasing probabilities are highest for DFAs when the initial state is the middle one. Examining the leftmost curve in Figure 6.9, the one corresponding to the DFA with parameters $C_{lb} = 1$ and $C_{ub} = 20$, one can see that the aliasing probability remains at 100% until the string length exceeds 10. This is due to the fact that this DFA accepts all possible strings of length less than 10. In general, if the initial state is 1 (i.e., the counter's initial value is 1), then all possible strings

of length $\min(|C_{lb}|, C_{ub})$ are accepted by the DFA. Thus for *updn* DFAs where the range of their parameter (i.e., $C_{ub} - C_{lb}$) are the same, lower aliasing probabilities are observed when the initial state is closer to one of the bounds.

Figure 6.9 Aliasing Probabilities of the *updn* DFA, middle initial state



Counting the paths in an *updn* state transition graph is equivalent to counting the number of paths in the grid depicted in Figure 6.10. A path in the grid must move between grid points either parallel or perpendicular to the coordinate axes. Moves parallel to the p axis correspond to receiving a zero, and increment the up/down counter. Moves parallel to the q axis correspond to receiving a one, and decrement the up/down counter. Only moves that increase the value of p or q are permitted. The difference between the p and q coordinate values corresponds to the up/down counter's value.

The number of paths from the origin to the grid points along the $p + q = k$ line without crossing the $ub = p - q$ line or the $lb = p - q$ line is equal to the number of paths of length k in the *updn* DFA in which the $C_{lb} = lb$ and $C_{ub} = ub$. This can be expressed by the

following equation:

$$\sum_{lb<(p+q=k)<ub} P(p,q) \tag{6.10}$$

where $P(p,q)$ is the number of paths from the origin to (p,q) without crossing the upper or lower bounding lines. The number of paths from a point (p_1,q_1) to a point (p_2,q_2) is:

$$\binom{(p_2-p_1)+(q_2-q_1)}{p_2-p_1} \text{or} \binom{(p_2-p_1)+(q_2-q_1)}{q_2-q_1}.$$

The reflection principle [Moh79] can be used to bound the number of paths for the *updn* DFAs. Referring to Figure 6.10 and using the reflection principle the number of paths form w to x, y, or z without crossing the line $ub = p - q$ is given by:

$$\binom{p+q}{p} - \binom{p+q}{p+ub}. \tag{6.11}$$

where p is the p coordinate value for the one of the end points, and q is the q coordinate value. Note that $\binom{p+q}{p}$ is the number of paths from w to one of x, y, or z, and $\binom{p+q}{p+ub}$ is the number of paths from w'' to x, y, or z.

Similarly, the number of paths to (p,q) without crossing the line $-lb = p - q$ is:

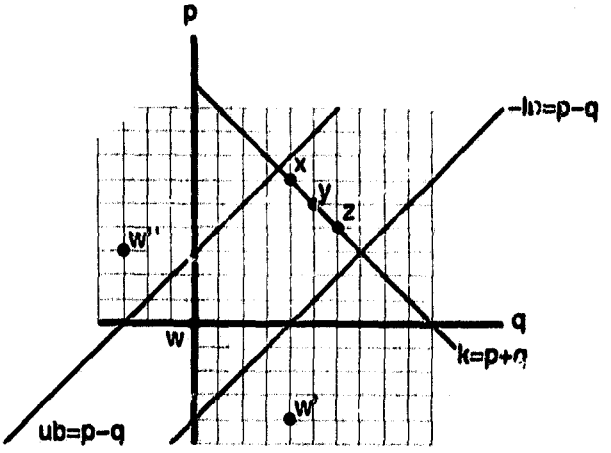
$$\binom{p+q}{p} - \binom{p+q}{p-lb}. \tag{6.12}$$

Unfortunately, there does not appear to be any general way of counting the number of paths from the origin to (p,q) that cross both bounding lines. Thus these expressions give the number of paths for an *updn* DFA where the upper or lower bound is equal to k can be derived. These expressions allow one to compute a bound on the number of paths but can not give an exact number.

6.7.2 Run0 and Run1 DFAs

The *run0* and *run1* DFAs have identical state transition graphs, when the transition labels are changed from 1s to 0s, and from 0s to 1s. Since their graphs are identical, the number of paths and thus the aliasing probability is also identical.

Figure 6.10 Bounding the number of paths for a *updn* DFA



A plot of aliasing probability for DFAs where the $R_{0/1}$ parameter is set to 5, 7, 8, 10, 12, 15, and 20 is given in Figure 6.11.

The number of paths of length k in a *run0/run1* DFA with the $R_{0/1}$ parameter is equal to $\text{Fib}_{R_{0/1}}(k)$, where $\text{Fib}_n(k) = \text{Fib}_n(k-1) + \text{Fib}_n(k-2) + \cdots + \text{Fib}_n(k-(n+1))$ (i.e., $\text{Fib}_n(k)$ is the n-th order Fibonacci sequence).

An example of a *run1* DFA with $R_1 = 3$ is shown in Figure 6.12(i). Figure 6.12(ii) shows a tree of the paths originating at a and terminating at a. A recurrence relation for the number of paths can be easily written from the path tree in Figure 6.12(ii) as follows

$$a(k) = a(k-1) + a(k-2) + a(k-3) + a(k-4). \tag{6.13}$$

Note that Equation 6.13 is equivalent to the definition of $\text{Fib}_3(k)$, and $R_1 = 3$, thus confirming the general statement for $n = 3$. The truth of the general statement can be seen by noting that incrementing the $R_{0/1}$ parameter adds one new node to the state transition graph, and one extra term to the recurrence relation.

Figure 6.11 Aliasing Probabilities of the *run0* and *run1* DFAs

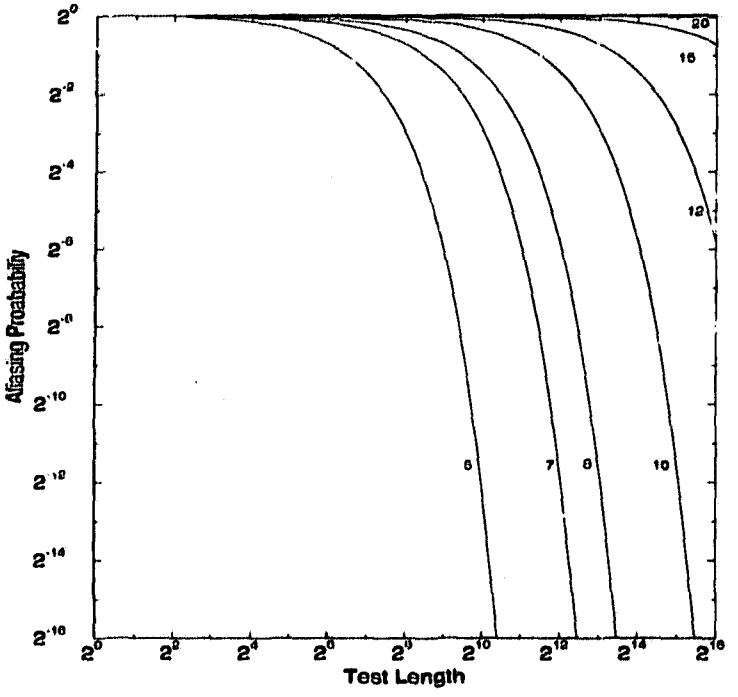
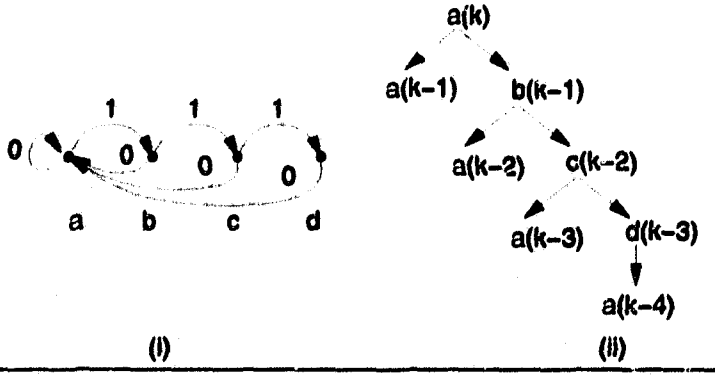


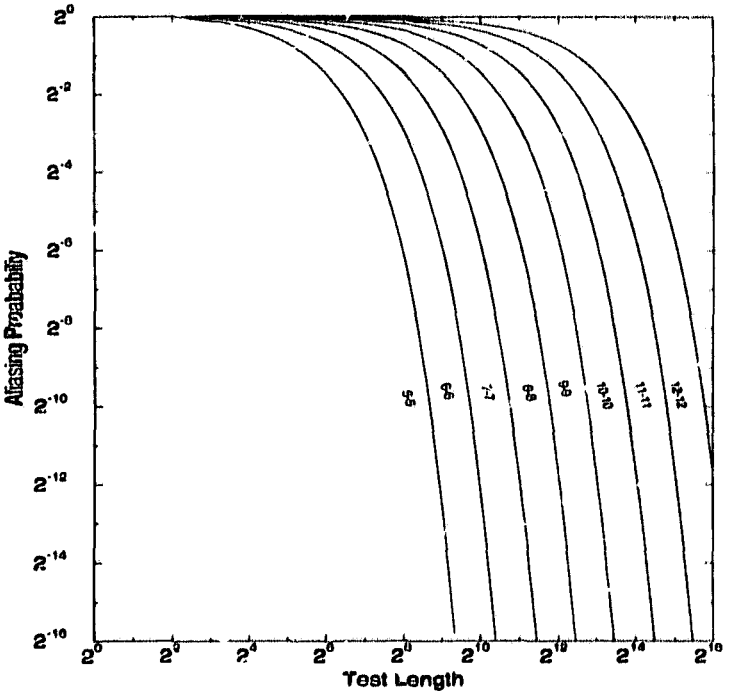
Figure 6.12 Recurrence Path Tree



6.7.3 Run2 DFAs

A counter based implementation of a *run0/run1* DFA, can be easily modified to count both the runs of zeros and runs of ones. Modification of the sequence detector and the addition of another compactor of a *run0/run1* DFA are the only changes required to realize a *run2* DFA. Moreover, at the cost of the extra hardware, the aliasing probability is substantially decreased. Figure 6.13 gives the aliasing probability for *run2* DFAs where the R_0 and R_1 parameters are both set to 5 through 12.

Figure 6.13 Aliasing Probabilities of the *run2* DFA



All possible strings of length less than $l = min(R_0, R_1)$ are accepted by a *run2* DFA. Thus 100% aliasing occurs for strings of length less than l. This effect is observerable in the plot.

6.7.4 Aliasing Probability Relationships

A summary of the common aliasing probability relationships of the four DFAs is presented in this subsection. A comparison between the aliasing probability of the four DFA types by machines with the same number of states is also presented.

The common relationships are:

- Smaller magnitude DFA parameters, R_0, R_1, and $C_{ub} - C_{lb}$, imply smaller aliasing probabilities. That is:

$$\begin{aligned} R_0 &\propto \text{aliasing probability} \\ R_1 &\propto \text{aliasing probability} \\ C_{ub} - C_{lb} &\propto \text{aliasing probability} \end{aligned}$$

  These relationships are apparent from the respective aliasing probability plots.

- For fixed DFA parameters, increasing the test length implies decreasing the aliasing probability. That is:

$$\text{test length} \propto (\text{aliasing probability})^{-1}$$

  This relationship is also apparent from the plots.

When selecting one of several generator sources, the first relationship can provide the discriminating criteria. For example, if the test sequence produced by a LHCA BIST generator results in a *run2* DFA parameter of 10, and the parameter of the test sequence of the same length produced by a LFSR is 15, then the LHCA test sequence results in a DFA with a lower aliasing probability.

The second relationship implies that as long as the DFA parameters remain fixed, a lower aliasing probability is realized with increased test length. This means that the test length should be equal to the largest length of a test sequence just before the DFA parameters increase. For example, if the DFA parameters increase after a test length of 1023 and sufficient faults are detected, then a test length of 1023 gives the lowest aliasing probability. Once the DFA parameters increase, the aliasing probability also increases.

Figure 6.14 shows the plots of the aliasing probabilities for the four DFA types when the number of states is equal to 10. Aliasing is the same for *run0* and *run1*. The *updn* DFA's aliasing probability is the smallest, next is *run2*, and the *run0/run1* DFAs aliasing probabilities is the worst. Thus given the choice of equal sized DFAs, the lowest aliasing probability is realized by the *updn* DFA. However, the *updn* DFA also requires more hardware than the other DFA approaches.

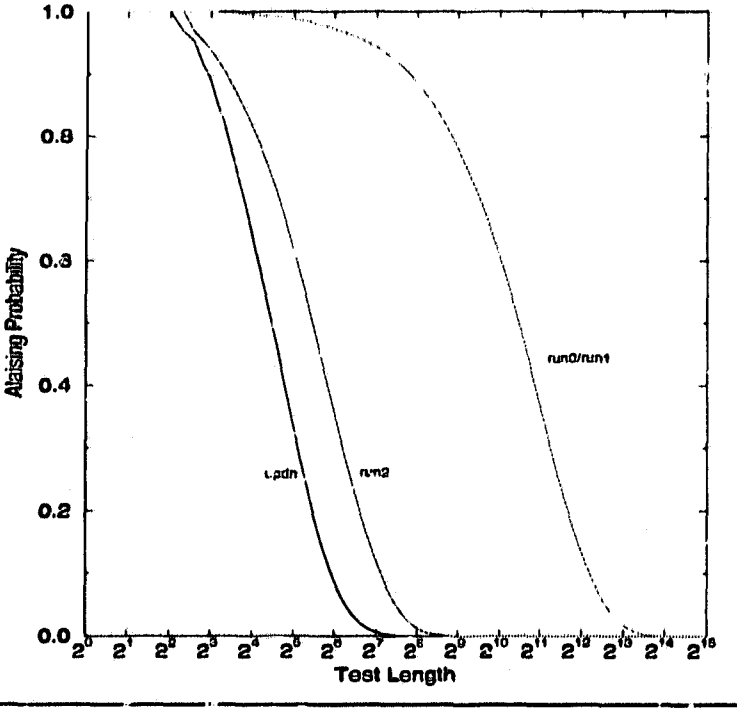
By examining the previous aliasing probability plots, one can see that the rate at which the aliasing probability increases with the DFA's parameters is different. When $R_{0/1}$ is greater than 20 for the *run0/run1* DFA, unrealistic test lengths are required to achieve a reasonable aliasing probability. The same is also true for the *run2* DFA. The *updn* DFA's parameter's range, $C_{ub} - C_{lb}$, gives reasonable aliasing probabilities for values up to 180.

6.7.5 Experiments and Results

The results of fault simulation experiments using the nonredundant versions of the ISCAS85 benchmarks [Tv91] are presented in this section. Converting the ISCAS85 benchmarks to Logic III(UVic) code allowed the use of the 1g3 CAD tool. Since, during fault simulation, a fault is detected whenever the output sequence is not accepted by the DFA, the detected fault can be dropped. The PPSFP [WEF+85] fault simulator in the 1g3 CAD tool simulates the given DFA and drops any fault whenever the input to the DFA results in an illegal state transition.

The results of the experiments for the four types of DFAs are found in Tables 6.3 to 6.6.

Figure 6.14 Aliasing Probabilities of the different DFA types with $n = 10$



The following two experiments are performed for each benchmark circuit with each DFA type.

Table 6.3 Updn DFA Results

gen is lfsr(1010..10), test length = 4096						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	13/494	.026	(-35,34) 70	.023
c499	750	750	2/748	.0027	(-15,44) 60	.0040
c880	942	912	3/909	.0033	(-28,20) 59	.0046
c1355	1566	1566	99/1467	.063	(-31,47) 79	.051
c1908	1858	1850	9/1841	.0049	(-36,17) 54	.0014
gen is lfsr(1010..10), test length = 32768						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	5/502	.010	(-155,23) 179	.0034
c499	750	750	0/750	0	(-59,85) 145	.00062
c880	942	942	1/941	.0010	(-5,153) 159	.00027
c1355	1566	1566	17/1549	.010	(-19,176) 196	.0069
c1908	1858	1857	4/1853	.0022	(-58,100) 159	0.0021

Table 6.4 Run3 DFA Results

gen is lfsr(1010..10), test length = 4096						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	89/418	.176	9	.134
c499	750	750	98/352	.130	9	.134
c880	942	912	360/552	.395	10	.368
c1355	1566	1566	208/1358	.133	9	.134
c1908	1858	1850	303/1547	.164	9	.134
gen is lfsr(1010..10), test length = 32768						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	74/433	.146	12	.135
c499	750	750	446/304	.594	14	.607
c830	942	942	155/787	.165	12	.135
c1355	1566	1566	246/1320	.157	12	.135
c1908	1858	1857	227/1630	.122	12	.135

1. The primary inputs are connected to a minimum cost LFSR [BMS87, Appendix 1] with an initial state of $1010\cdots10$. The circuit is simulated for 4096 test cycles, and

Table 6.5 Run1 DFA Results

gen is lfsr(1010..10), test length = 4096						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	12/495	.0236	8	.0178
c499	750	750	239/511	.319	10	.368
c880	942	912	128/784	.140	9	.134
c1355	1566	1566	932/634	.595	11	.610
c1908	1858	1850	294/1556	.159	9	.134
gen is lfsr(1010..10), test length = 32768						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	326/181	.643	14	.607
c499	750	750	268/482	.357	13	.368
c880	942	942	138/804	.146	12	.135
c1355	1566	1566	436/1130	.150	12	.135
c1908	1858	1857	255/1602	.137	12	.135

Table 6.6 Run2 DFA Results

gen is lfsr(1010..10), test length = 4096						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	1/506	.00197	9 – 8	.00231
c499	750	750	25/725	.0333	9 – 10	.0490
c880	942	912	82/831	.0888	10 – 9	.0490
c1355	1566	1566	119/1447	.0760	9 – 11	.0812
c1908	1858	1850	42/1808	.0227	9 – 10	.0490
gen is lfsr(1010..10), test length = 32768						
Circuit	Faults	Exposed	Un/Detected	Aliasing	Range	Th. Aliasing
c432	507	507	49/458	.0966	12 – 14	.0819
c499	750	750	161/589	.215	14 – 13	.223
c880	942	942	74/868	.0786	12 – 12	.0182
c1355	1566	1566	39/1527	.0249	12 – 12	.0182
c1908	1858	1857	29/1828	.0156	12 – 13	.0497

the relevant parameter is measured for all of the compactor's outputs. A MISR with a minimum cost feedback function is used as the compactor. Measuring the DFA's parameters for each output of the compactor allows the output with the smallest parameters to be chosen as the site to attach a DFA. Choosing the output with the smallest parameter(s) is based on the theoretical results that indicate that the DFA with the smallest parameters has the least aliasing. A fault simulation is then performed for 4096 cycles. All faults which cause the string to be illegal are detected.

2. The second experiment is identical to the first, except the number of simulation cycles is 32768.

Each row of the tables give: the circuit name, the maximum number of single stuck-at faults, the number of exposed faults, the number of detected faults, the actual aliasing, the parameter used by the attached DFA, and the theoretical aliasing probability. The actual aliasing is given by:

$$\frac{\text{exposed faults} - \text{detected faults}}{\text{exposed faults}}.$$

The theoretical aliasing probability is given by expression ... Section 6.4.2. The *Range* heading gives the particular DFA's parameters.

6.7.6 Discussion of the DFA Experiments and Design Issues

The error detection effectiveness of a DFA error detector depends on the customization parameters for that DFA (see Section 6.7.4). Smaller DFA parameters imply smaller aliasing probability, and thus better error detection ability. From the simulation results, one can see that larger test lengths yield larger DFA parameters and larger parameters result in worse aliasing probabilities.

On the other hand, as the test length increases, the aliasing probability for a DFA with fixed parameters decreases. Thus increasing the test length results in lower aliasing probabilities *if* the DFA parameters remain fixed.

The output sequence depends on the BIST generator, the circuit under test, and the data compactor. The BIST designer can only choose which generator and which compactor to use. Ideally the BIST generator should produce an output sequence for the circuit that exposes all faults, and yet results in small DFA parameters. The data compactor should also produce an output sequence that minimizes the DFA parameters.

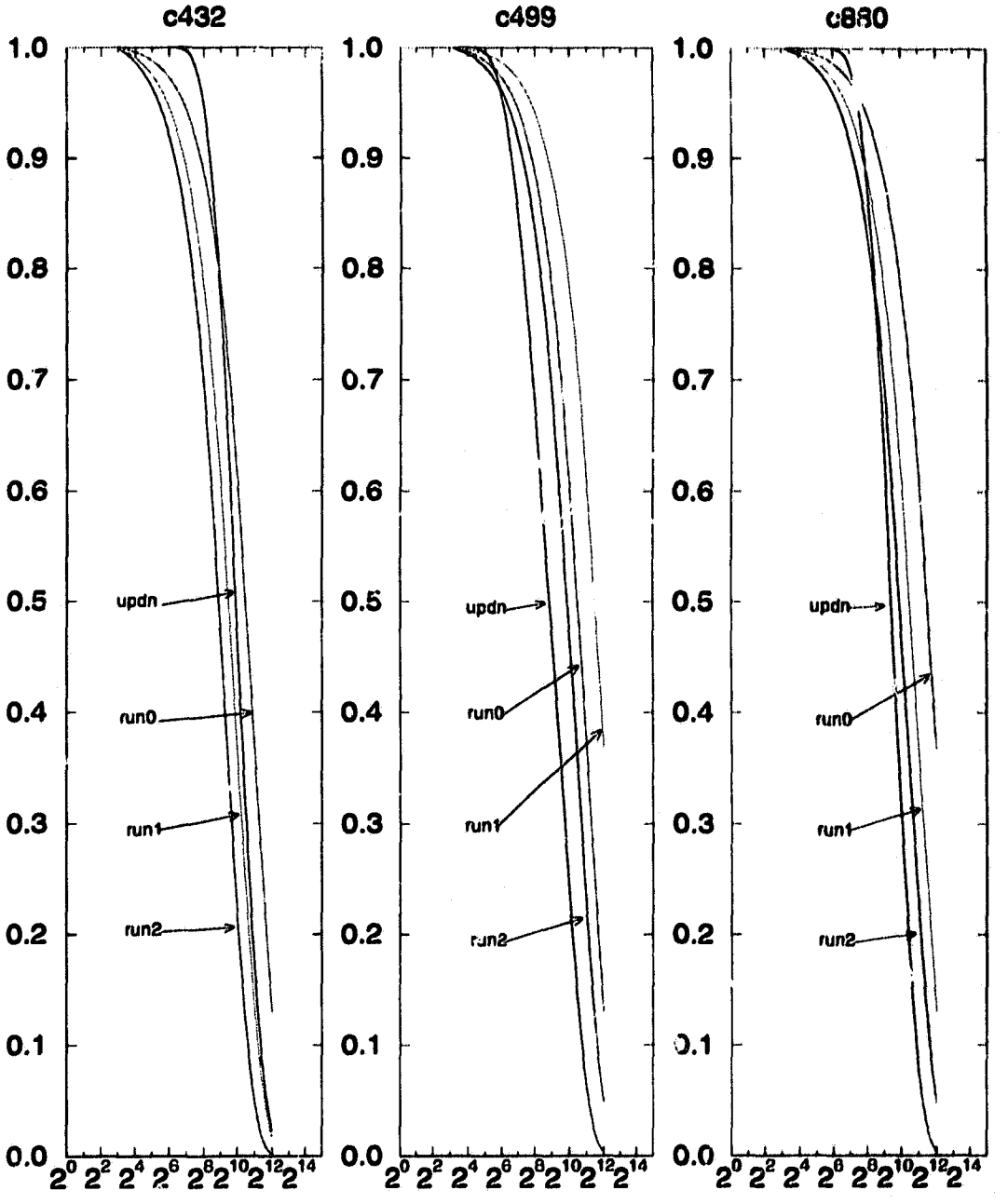
If the BIST generator and compactor are based on LFSMs, then there does not exist an easy way to satisfy the goal of minimizing the DFA's parameters. The only control the designer has is over the next state function. Designing a next state function to produce a particular sequence or an approximation of that sequence is difficult, if not impossible. However, simulation experiments can be performed to search for BIST generators and compactors.

During the fault-free simulation used to determine a DFA's parameters, if the system records the sequence length every time the DFA parameters increase, then the designer can choose a test length that is maximal with respect to a particular set of DFA parameters. If fault simulation experiments are used, then the designer can optimize the test length directly from the fault coverage results.

Of the four DFA types the *updn* DFA has the best fault detection ability. Only in one case did the *updn* DFA not have the best performance. Also the error detection ability improved with longer test lengths, whereas it is generally worse for the other DFA types. This means that for the *updn* DFA, the rate at which the DFA parameters increase is counteracted by the longer test lengths. The converse is true for the other DFA types.

Plots of the aliasing probabilities using the parameters computed from the fault free simulation of experiment 1 for the c432, c499, and c880 benchmark circuits are shown in Figure 6.15. These plots and the simulation result tables illustrate that the aliasing probabilities calculated from the DFA's parameters can be used as an indication of detection effectiveness of a particular DFA. Note that in all cases the ranking of the circuits by the aliasing probabilities is the same as the ranking by the number of undetected faults. Also

Figure 6.15 Aliasing Probabilities using the parameters computed for c432, c499, and c880 from experiment 1.



note that the aliasing probabilities of the *run2* DFAs are always less than the aliasing probabilities of *run0* and *run1*.

Since the *run2* DFA combines *run0* and *run1*, its error detection performance should be better than either *run0* or *run1*. The simulation results agree with this statement. There is no clear winner for the remaining two DFAs.

One possible explanation for the better performance of the *updn* DFA is that since the state of the up/down counter depends on the entire sequence, even a single bit error can result in one of the bounds being exceeded. Whereas the state of counter of the other DFAs depends on only the current run of zeros or ones (i.e., the counter is reset at the end of every run).

The major reason for still considering the *run0/1/2* DFAs is that they are cheaper to build than the *updn* DFA. Not only is the counter simple to build (i.e., it only needs to count in one direction), the measured DFA parameters are smaller. If the DFA parameters for the c432 circuit are compared, the *updn* DFA requires $\lceil \log_2(70) \rceil = 7$ bits, whereas the *run0/1/2* DFAs only require $\lceil \log_2 \max(8,9) \rceil = 4$ bits. The *run0/1/2* DFAs need a 1 bit sequence detector bringing the total number of memory elements up to 5. Thus a savings of 2 memory elements, and a simpler counter is realized.

6.8 Ways for Improving the DFA Error Detection Ability

This section presents five potential areas of study, that may/can improve the fault detection ability of the DFA error detectors. The areas are:

- Optimize the test length to achieve the best fault coverage.
- Try different BIST generators.
- Use another DFA type.

- Use the final state of the counters as an addition response characterization.
- Partition the test sequence into separate regions, and determine the DFA parameters for each region.

Optimizing the test length consists of finding the DFA parameters that result in the lowest aliasing probabilities with a test length long enough to expose all the given faults for a fixed BIST generator. Fault simulation can be used to determine the minimum test length required to expose the given faults. A fault-free simulation starting with the minimum test length and going to some maximum test length can be used to find the DFA parameters with the lowest aliasing probabilities. During the simulation, the test length is recorded whenever the DFA parameters increase. Given the DFA parameters and the test length the aliasing probabilities can be calculated and the lowest one chosen.

Using this approach, a test length of 5504 for the *run1* DFA is discovered which results in only 3 faults remaining undetected for the c432 circuit. This is an improvement in comparison to 12 undetected faults for a test length of 4096 and 326 faults for a test length of 32768.

A second method to improve the aliasing probabilities is to try different types of generators. There does not seem to be any systematic way of exploring the space of generator types.

Only four DFA types have been studied in this section, perhaps other DFA types could be found which have better aliasing characteristics. For example, another method of driving an up/down counter is to count down if the last two outputs are 00 and to count up if the last two outputs are 11.

In the proposed DFA types, an error is detected whenever a counter reaches a forbidden value, if after the test sequence is applied, the counter's final value is also checked, then further discrimination between the fault-free and faulty response is possible. The final state of the data compactor could also be checked.

At the cost of additional comparators, the test sequence could be divided into different segments. and DFA parameters for those segments could be used.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The goal of the work presented in this dissertation is to create a design environment that allows BIST embeddings to be easily specified and evaluated. Hopefully, the preceding chapters have demonstrated that the hardware description language, Logic III(UVic), the BIST embedding library and the **lg3** CAD tool provide a design environment that allows

- BIST embeddings to be performed easily and effectively, and
- new BIST architectures to be studied and developed.

A major premise of this dissertation is that if the BIST embedding problem is considered as part of the functional design problem and an environment is provided that supports BIST design, then higher quality BIST designs can result. The variety of the embeddings in Chapter 5 amply demonstrate that this premise has validity.

The conclusions from the major areas reported in this dissertation are discussed in the following subsections.

7.1.1 Language and Environment

The single idea of an embedding module provides the simplicity and the expressiveness to specify many useful BIST architectures. In particular, implementations of the scan, STUMPS, BEST, BILBO, and CSTP BIST architectures are easily specified. In general, any BIST architecture which modifies the memory elements in a design can be specified by an embedding module.

The introduction of the `reg` data type has the benefit of making the clocking of the memory elements the responsibility of the embedding module designer, and not the user. Since many DFT rules place restrictions on memory element clocking, the inclusion of the clocking design in the embedding module reduces the DFT rule checking problem to checking the embedding modules.

The `use` statement is the only method provided in Logic III(UVic) to specify a BIST embedding. This construct is sufficient to specify many diverse embeddings, as illustrated by the case study circuits. The embedding library enables many different embeddings to be realized by simply changing the name of the embedding module.

When a system is constructed by combining individual modules, the logic inside a module can become redundant. This logic can be tested by surrounding all the component modules with testing structures (i.e., testing the modules separately). The proposed constructs of capping and binding TRUE or FALSE to module inputs of Logic III(UVic) can eliminate some of this redundant logic.

In addition to specifying BIST embeddings, the language modifications to improve the structural expressiveness of Logic III provide for clear and economical descriptions. These additions include: the ability to define symbolic constants, the revamping of arrays and operations on arrays, collections, more than one global section, assertions, external bodies, functions, and recursion. Examples of using these constructs can be found in the BIST embedding library modules in Appendix C.

The introduction of constant definitions improved the readability and modifiability of Logic III(UVic) designs. A prime example is the `word_size` constant used in the CORDIC case study to specify the word size of the CORDIC processor. Note that to create a CORDIC processor with a different word size requires only a change to the `word_size` definition.

Variable size arrays allow the creation of library modules that are customizable on the array's size. The original Logic III supports variable sized arrays, but the user has to explicitly pass the array bounds as separate parameters. The arrays in Logic III(UVic) provide the array bound information implicitly. This also enables the compiler to perform stronger type checking.

Array slicing replaces the more tedious and error prone "loops of `connect` statements". For example, the expression

```
connect( a[1..8], lower );
```

replaces

```
for i := 1 to 8 do
    connect( a[i], lower[i] );
```

Or, more commonly,

```
for i := 1 to 8 do
    connect( a[i], lower[i] );
adder(lower, sum);
```

is replaced by:

```
adder( a[1..8], sum);
```

The addition of recursion proved to be very effective in describing many circuit structures. See the multiplier example in Section 3.3.2.

The ability to perform fault simulation is key to producing BIST embeddings with high fault coverage. Although fault simulation can be computationally expensive, the case studies

show that this cost may not be prohibitive. Moreover, fault simulation at the primitive gate level provides one of the most accepted measures of test effectiveness.

The association of faults with Logic III(UVic) variables and the reporting of hard-to-test faults by their Logic III(UVic) variable names allows a **lg3**'s user to localize the testing difficulty.

7.1.2 Object-oriented Design

The development of **lg3** demonstrates that the object-oriented design approach and the C++ programming language can be effective in the production of CAD tools. The ability of **lg3** to evolve rapidly demonstrates the effectiveness of the objects and classes presented in Chapter 4.

The abstract syntax classes and objects are effective in the implementation of the Logic III(UVic) parser, in that the addition of new language constructs usually does not affect the original constructs. The classes and objects designed for the simulation system also allow additions and modifications to be made to the simulators in a straightforward fashion. The classes developed for the **lg3** CAD tool should provide a reasonable starting point for similar CAD systems.

7.1.3 Case Studies

The combined approach of specifying BIST designs at the register-transfer level and using fault simulation at the gate level to measure the effectiveness does provide a successful environment for the creation of BIST operation modes in the case study circuits. The variety of the different BIST embedding shows the flexibility of allowing circuit designers to deal with the BIST embedding problem directly.

The common methodology of embedding BIST for the case studies is:

1. Check modules for redundant logic and remove or document any redundancies.

2. Check the complete circuit for redundant logic and remove or document any redundancies.

3. Embed and evaluate a BIST architecture.

4. Repeat step 3 until all the design goals are satisfied.

The following conclusions can be drawn from the BIST embedding case studies.

- The success of achieving almost 100% single stuck-at fault coverage for the four case study circuits indicates that the proposed BIST environment can be effective.
- Fault simulation results can be used to create and modify BIST embeddings at the RTL level.
- Experiments to determine if the circuits are random-pattern testable is the first step in the BIST embedding process. The results of these experiments show that the circuits are indeed random-pattern testable. Thus a reasonable conjecture is that data path and simple controller circuits are generally random-pattern testable.
- For the case studies, four causes can be identified for the faults that are not random-pattern testable:
  - The logic associated with the faults is redundant.
  - Some circuit structures are testable by only a few vectors (e.g., wide NOR circuits).
  - The logic's observability is poor (e.g., the load line of a register).
  - The correlation of the outputs stream of adjacent outputs for a LFSR results in a test pattern that is poor for some circuits (e.g , the string matching circuit).
- The variations in the test effectiveness of the different BIST architectures show that there is no definitive BIST architecture. For example, embedding C5 requires fewer

test cycles than C2 and both embeddings are of comparable cost. C5 uses the BEST architecture. C2 uses a CSTP architecture. Alternatively, C3 requires fewer test cycles than C6 and C3 uses a CSTP architecture, while C6 uses a BEST architecture. Again C3 and C6 are of comparable cost.

- Embeddings C8 and C9 give rise to a maxim: "The Best BIST is No BIST!" In these embeddings the CORDIC circuit is converted into an AFSM for testing purposes. These embedding achieved 100% single stuck-at fault coverage with little extra hardware. These embedding demonstrate the ability of the proposed CAD environment to enable the designer to develop novel embeddings.

- The wide NOR circuit in the GCD circuit is difficult to test. Replacing some of the NOR gates with EXNOR-NOR gates can dramatically improve the testability of the structure. This NOR circuit with the BIST mode can be added to the designer's library. Common circuit structures that are difficult to test could also have BIST versions designed.

7.1.4 Two-Pattern BIST Generators

The method introduced to measure the two-pattern generation test effectiveness for LFSMs enables different types of LFSMs to be compared. The number of distinct transitions from a window is generally larger for a LHCA than an internal LFSR. The number of transitions for a window in an internal LFSR is larger than an external LFSR if a tap is contained in the window. The cross-over permutation can be used to create windows that have the maximum number of transitions for both LHCAs and LFSRs.

7.1.5 DFA Response Analysis Architecture

The fault simulation experiments demonstrate the viability of using DFAs to detect the presence of single stuck-at faults. The *updn* DFAs have the best detection performance,

with either no or very few undetected faults. The *run2* DFAs aliasing ranges from .197% to 21.5%. In general, the *run0* and *run1* DFAs have poor detection performance, but in the *run1* experiment where the test length is 4096 all but 12 of the faults are detected for the c432 circuit. This indicates that for certain circuits, the *run0* and *run1* are reasonable candidates to consider.

In addition, the experiments also show a *good* correlation between the theoretical aliasing probabilities and the observed aliasing. From the experiments, it be can seen that smaller theoretical aliasing probability indicates smaller actual aliasing.

Since the theoretical aliasing probabilities of the DFAs approach zero as the test length approaches infinity, for a fixed parameter DFA negligible aliasing can be achieved by increasing the test length. The test length required to achieve negligible aliasing increases as the parameter(s) of the particular DFA increase. From the experimental results, larger test lengths also require larger DFAs and thus increasing the test length does not guarantee decreased aliasing.

An important point to note is that the results are reported for a fixed set of experiments and better detection performances are observed for different generators and test lengths. For example, with the *run1* DFA and a test length of 5504, only 3 faults remained undetected for the c432 circuit.

Several advantages of using this approach are:

- The DFA detection performance can be determined from the DFA's size, since the size can be used to determine the DFA's aliasing probability.
- Faults can be detected throughout the test, and thus this approach can be used in an online nonconcurrent BIST architecture.
- During fault simulation the faults that the DFA detects can be dropped. Fault dropping substantially decreases the fault simulation time, thus allowing more BIST embeddings to be tried.

7.2 Future Work

7.2.1 Language and Environment

Several of the additions and modifications to Logic III(UVic) that should be considered in future work are:

- There is no convenient way of adding observation points to a design. One possible approach would be to create an **observer** data type, so that a user could change the type of a net to **observer** if the observability needed to be increased. A third parameter could be added to the embedding module that would contain all the observers. It would be the responsibility of the embedding module designer to create the necessary observation structure.
- A similar method to add stimulus points, would be to also add a **stimulate** data type. Two additional parameters would be required for the embedding module. The first parameter would handle the connection to the net's source, the second parameter would handle the net's destination.
- The language should support a method of encapsulating any changes made to perform an intrusive BIST embedding.
- The customization constructs used to remove redundant logic can only deal with simple forms of unused logic. Is there any way that a user can specify more complex forms of redundancy?
- Logic III(UVic) does not implement the logic modules of Logic III. A possible technique to reintroduce logic modules is illustrated by the following example that computes the Boolean AND of its inputs. It uses functions to generate the Boolean equations. Notice that the idea of a LOGIC assignment is not necessary.

```
function And ( x : array [1..] of input ) : output;
const
    xs = x'size;
    mid = xs/2;
begin
    if xs = 1 then
        And :- x[1];
    else if xs = 2 then
        And :- x[1] AND x[2];
    else
        And :- And(x[1..mid]) AND And(x[mid+1..xs]);
end.
```

The operator ":-" binds the variable on the left with the Boolean expression on the right.

Possible additions to the lg3 CAD tool to improve the environment are:

- If Boolean expressions are reintroduced into Logic III(UVic), then the simulation system should support both functional and fault simulation of these expressions.
- The lg3 tool can only ensure a fault is redundant by exhaustive simulation. Better methods to detect redundant logic should be added.
- In the current version of lg3, a STUMPS BIST embedding has to be evaluated with the parallel fault simulation algorithm. Since the STUMPS architecture actually partitions a design into combinational blocks, the PPSFP algorithm could be used with a subsequent improvement in simulation speed.
- An essential feature in any production version of lg3 would be the ability to checkpoint the simulation. This is especially apparent for long fault simulations when the user wishes to increase the number of simulation cycles for a simulation in which the fault coverage is not adequate.
- Since fault simulation is an inherently parallel activity, a distributed version of the fault simulation system would be an asset.

- On a more speculative note, since designs in Logic III(UVic) can be described with recursive and iterative language constructs, perhaps these constructs can also be used in the creation of BIST structures.

An obvious extension to the BIST library is the addition of more BIST embedding architectures.

7.2.2 Object-oriented Design

Chapter 4 identifies the set of classes and objects used to build the **lg3** CAD tool. Creating and extending a stand alone library of the simulation classes would enhance the utility of this work.

7.2.3 Two-Pattern BIST Generators

The analysis of two-pattern generators is from the point of view of the generator. An important question to consider is how the structure of the circuit can be used to permute the generator's outputs to produce the best two-pattern generator for the circuit under test.

7.2.4 DFA Response Analysis Architecture

There are several open problems in the construction of DFA based response analysis structures:

- Although the aliasing probabilities for the DFAs can be calculated exactly and some bounds are presented, these bounds should be explored to see if they can be made tighter.
- The effect of different BIST generators on the parameters of the DFAs should be examined.

- Techniques to improve the error detection ability need to be examined. Some approaches are suggested in Section 6.8.
- In the DFA architecture presented, a MISR is used to compact the outputs of the circuit. Applying the DFA approach to the CSTP BIST architecture should be investigated.

In comparison to the design of a MISR based compactor, the designer of the DFA response analysis architecture has more freedom to create an analyzer that is customized to a particular test.

7.3 Concluding Remarks

This dissertation is based on the premises that designers should deal with the BIST embedding problem directly and that fault simulation can be used to create and optimize BIST embeddings. Hopefully, the goal of this dissertation, to show that Logic III(UVic), the BIST embedding library and the `lg3` CAD tool provide an environment that enables designers to create BIST embeddings, is successful.

The surprising results from the different case studies indicate that no one BIST architecture can fulfill the BIST embedding requirements. The varied results also indicate that in order for a BIST embedding system to be successful, it must allow flexibility in the types of BIST architectures that can be embedded.

In addition to the `lg3` CAD tool's role to aid BIST embedding designers, the tool can also be used to evaluate new BIST architectures. The development of the DFA based response analysis architecture provides support for this claim.

Bibliography

[ABB85] Magdy S. Abadir and Melvin A. Breuer. A knowledge-based system for designing test. *IEEE Design and Test of Computers*, pages 56–68, August 1985.

[ABF90] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Designs.* Computer Science Press, 1990.

[AC81] V. E. Agarwal and E. Cerny. Store and generate built-in-testing approach. In *International Symposium on Fault-Tolerant Computing*, pages 35–40, 1981.

[AJ89] Sheldon B. Akers and Winston Jansz. Test set embedding in a built-in self-test environment. In *Proceedings of the IEEE International Test Conference*, pages 257–263, 1989.

[Avr91] LaNae Avra. Allocation and assignment in high-level synthesis for self-testable data paths. In *Proceedings of the IEEE International Test Conference*, pages 463–472, 1991.

[BAV90] Ove Brynestad, Einar J. Aas, and Anne E. Vallestad. State transition graph analysis as a key to BIST fault coverage. In *Proceedings of the IEEE International Test Conference*, pages 537–543, 1990.

[BCK89] Franc Brglez, Clay Closter, and Gershon Kedem. Hardware-based weighted random pattern generation for boundary scan. In *Proceedings of the IEEE International Test Conference*, pages 264–274, 1989.

[BCR83] Zeev Barzilai, Don Coppersmith, and Arnold L. Rosenberg. Exhasutive generation of bit patterns with applications to VLSI self testing. *IEEE Transactions on Computers*, C-32(2):190–194, 1983.

[BDS89] Frans Beenker, Rob Dekker, and Rudi Stans. A testability strategy for silicon compilers. In *Proceedings of the IEEE International Test Conference*, pages 660–669, 1989.

[BDSS90] Frans Beenker, Rob Dekker, Rudi Stans, and Max Van Der Star. Implementing macro test in silicon compiler design. *IEEE Design and Test of Computers*, pages 41–51, April 1990.

[Ber90] Alan A. Bertossi. A VLSI system for string matching. *INTEGRATION, the VLSI journal*, 9:129–139, 1990.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and target traslator in FORTRAN. In *Proceedings of the IEEE International Test Conference*, pages 663–698, 1985.

[BM84] Frederick P. Beucler and Michael J Manner. HILDO: The highly integrated logic device observer. *VLSI Desgin*, 5:89–96, July 1984.

[BMS87] Paul H. Bardell, William H. McAnnay, and Jacob Savir. *Build-In Test for VLSI – Pseudorandom Techniques.* Wiley Inter-Science, 1987.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[BP89] Sudipta Bhawmik and P. Palchaudhuri. DFT expert: Designing testable VLSI circuits. *IEEE Design and Test of Computers*, pages 8–19, October 1989.

[BSMS81] Zeev Barzilai, Jacob Savir, George Markosky, and Merlen G. Smith. The weighted syndrome sums approach to VLSI testing. *IEEE Transactions on Computers*, C-30(12):996–1000, December 1981.

[Bud92] Tim Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, 1992.

[CBAP89] Prasaad R. Chalasani, Sudipta Bhawmik, Anurag Acharya, and P. Palchaudhuri. Design of testable VLSI circuits with minimum area overhead. *IEEE Transactions on Computers*, 38(10):1460–1462, 1989.

[Che87] James Cherry. 9 – CAD programming in an object-oriented programming environment. In *VLSI CAD Tools and Applications*, pages 265–294. Kluwer Academic Publishers, 1987.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison Wesley, 1992.

[CYL92] Chien-In Henry Chen, Joel Yuen, and Ji-Der Lee. Autonomous – tool for hardware partitioning in a build-in self-test environment. In *Proceedings of IEEE International Conference on Computer Design*, pages 264–267, 1992.

[DB91] Kaushik De and Prithviraj Banerjee. Logic partitioning and resynthesis for testability. In *Proceedings of the IEEE International Test Conference*, pages 906–915, 1991.

[DOF+89] M. Damiani, P. Olivo, M. Favalli, S. Favalli, S. Ercolani, and B. Ricco. Aliasing in signature analysis testing with multiple-input shift-registers. In *European Test Conference*, pages 346–353, 1989.

[DOR91] Maurizio Damiani, Piero Olivo, and Bruno Ricco. Analysis and design of linear finite state machines for signature analysis testing. *IEEE Transactions on Computers*, C-40(9):1034–1045, 1991.

[DS91] Charles Donnelly and Richard Stallman. *BISON, The YACC-compatible Parser Generator.* Free Software Foundation, 1.16 edition, December 1991.

[Dun89] Samuel H. Duncan. A BIST design methodology experiment. In *Proceedings of the IEEE International Test Conference*, pages 755–762, 1989.

[ELWW91] Edward B. Eichelberger, Erice Lindbloom, John A. Waicukauski, and Thomas W. Williams. *Structured Logic Testing.* Prentice Hall, 1991.

[FH86] H. S. Fung and S. Hirschhorn. An automatic DFT system for the Silc silicon compiler. *IEEE Design and Test of Computers*, pages 45–57, February 1986.

[FM91] Kiyoshi Furuya and Edward J. McCluskey. Two-pattern capabilities of autonomous TPG circuits. In *Proceedings of the IEEE International Test Conference*, pages 704–711, 1991.

[FNH89] Ronald R. Fritzmeier, H. Troy Nagle, and Charles F. Hawkins. Fundamentals of testability – a tutorial. *IEEE Transactions on Industrial Electronics*, 36(2):117–128, May 1989.

[Fuj85] Hideo Fujiwara. *Logic Testing and Design for Testability.* The MIT Press, 1985.

[Gaj88] Daniel D. Gajski, editor. *Silicon Compilation.* Addison-Wesley Publishing Company, 1988.

[Gar93] Bill Gardner. Design synthesis for string matching with FPGAs. Project Report, July 1993.

[GCG+89] Rajiv Gupta, Wesley H. Cheng, Rajesh Gupta, Ido Hardonag, and Melvin A. Breuer. An object-oriented VLSI CAD framework. *IEEE Computer*, 22(5):28–37, May 1989.

[GDWL92] Daniel Gajski, Nilil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[GE89] Catherine H. Gebotys and Mohamed I. Elmasry. Integration of algorithmic VLSI synthesis with testability incorporation. *IEEE Journal Solid State Circuits*, SC-24(2):409–416, April 1989.

[GE92] Catherine H. Gebotys and Mohamed I. Elmasry. *Optimal VLSI Architectural Synthesis: Area, Performance and Testability.* Kluwer Academic Publishers, 1992.

[Gib85] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[GJ78] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1978.

[GRB91] Rajiv Gupta, Rajagopalan, and Melvin A. Breuer. Reorganizing circuits to aid testability. *IEEE Design and Test of Computers*, pages 49–57, September 1991.

[Hay76] John P. Hayes. Check sum test methods. In *International Symposium on Fault-Tolerant Computing*, pages 114–119, 1976.

[Hla92] Andrzej Hlawiczka. Parallel signature analyzers using hybrid design of their linear feedback. *IEEE Transactions on Computers*, C-41(12):1562–1571, 1992.

[HPCN92] H. Harmanani, C. Papachristou, S. Chiu, and M. Nourani. SYNTEST: An environment for system-level design for test. In *EURO-DAC*, pages 402–407, 1992.

[HTRC92] S. Hellerbrand, S. Tarnicj, J. Rajski, and B. Courtois. Generation of vector patterns through reseeding of multiple-polynominal LFSRs. In *Proceedings of 6th Workshop on New Directions for Testing*, pages 218–228, 1992.

[Hwa70] Kai Hwang. *Computer Arithmetic Principles, Architectures, and Design*, pages 368–373. Wiley, 1970.

[IB89] Vijay S. Iyengar and Daniel Brand. Synthesis of pseudo-random pattern testable designs. In *Proceedings of the IEEE International Test Conference*, pages 501–508, 1989.

[IEE88] IEEE. *IEEE Standard VHDL, Language Reference Manual*, March 1988.

[IEE90] IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture*, May 1990.

[KA88] Jerzy Kalinowski and Alexander Albicki. Computer-aided design of self-testable VLSI circuits. In *CompEuro 88 - System Design: Concepts, Methods and Tools*, pages 324–328, 1988.

[KHT88] Kwanghyun Kim, Dong Sam Ha, and Joseph G. Tront. On using signature registers as pseudorandom pattern generators in built-in self-testing. *IEEE Transactions on Computer-Aided Design*, CAD-7(8):919–928, August 1988.

[KMZ80] B. Konemann, J. Mucha, and G. Zweihoff. Built-in logic block observation technique. *IEEE Journal Solid State Circuits*, SC-15(3):315–318, 1980.

[KP89] Andrzej Krasniewski and Slawomir Pilarski. Circular self-test path: A low-cost BIST technique for VLSI circuits. *IEEE Transactions on Computer-Aided Design*, CAD-8(1):46–55, January 1989.

[KTH88] Kwanghyun Kim, Joseph G. Tront, and Dong S. Ha. Automatic insertion of BIST hardware using VHDL. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 9–15, 1988.

[KTH91] Kwanghyun Kim, Joseph G. Tront, and Dong S. Ha. Bides: A BIST design expert system. *Journal of Electronic Testing: Theory and Applications*, 2:165–179, 1991.

[Lal85] Parag K. Lala. *Fault Tolerant and Fault Testable Hardware Design.* Prentice/Hall International, 1985.

[LBDG87] Robert Lisannke, Franc Berglez, Art J. Degeus, and David Gregory. Testability-driven random test-pattern generation. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1082–1083, November 1987.

[Los78] Jacques Losq. Efficiency of random compact testing. *IEEE Transactions on Computers*, C-27(6):516–525, June 1978.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[LWJA92] Tien-Chien Lee, Wayne Wolf, Niraj K. Jha, and John M. Acken. Behavioral synthesis for easy testability in data path allocation. In *Proceedings of IEEE International Conference on Computer Design*, pages 29–32, 1992.

[McL92] Gordon R. McLeod. BIST techniques for ASIC design. In *Proceedings of the IEEE International Test Conference*, pages 496–496, 1992.

[MCN90] MCNC. *OASIS SYSTEM – Open Architecture Silicon Implementation Software*, 1.0 edition, 1990.

[Moh79] Sri Gopal Mohanty. *Lattice Path Counting and Applications.* Academic Press, 1979.

[MPC88] Michael C. MacFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 330–336, 1988.

[MZ91] D. M. Miller and S. Zhang. A study of the fault coverage of LFSR and CA pseudo-random test pattern generators. In *Proceedings 5th Technical Workshop - New Directions in Testing*, 1991.

[Nic92] M. Nicolaidis. Transparent BIST for RAM. In *Proceedings of 6th Workshop on New Directions for Testing*, pages 164–177, May 1992.

[Pap89] Christos A. Papachristou. Rescheduling transformations for high level synthesis. In *International Conference on Circuits and Systems*, pages 766–769, 1989.

[Par76] Kenneth P. Parker. Compact testing: Testing with compressed data. In *International Symposium on Fault-Tolerant Computing*, pages 93–78, 1976.

[Pax90] Vern Paxson. *FLEX - fast lexical analyzer generator.* Free Software Foundation, 2.3 edition, May 1990.

[PBDB87] M. Paraskeva, D. F. Burrows, P. Diamond, and J. Brandford. Accelerated design and automatic test overlay within a stuctured hardware design environment (SHADE). In G. Saucier and E. Read amd J. Trilhe, editors, *Fast-Prototyping of VLSI.* Elsevier Science Publishers B. V. (North-Holland), 1987.

[RS89] Gordon Russell and Ian L. Sayers. *Advanced Simulation and Test Methodologies for VLSI Design.* Van Nostrand Reinhold (International), 1989.

[RT92] Janusz Rajski and Jerzy Tyszer. Accumulator-based compaction of test responses. Technical report, VLSI Design Laboratory, Department of Electrical Engineering, McGill University, 1992.

[Rud90] Martin Rudolph. Feedback-testing by using multiple input signature registers. *Journal of Electronic Testing: Theory and Applications*, 1:214–219, 1990.

[SK90] Chau-chin Su and Charles R. Kime. Computer-aided design of pseudo-exhaustive BIST for semiregular circuits. In *Proceedings of the IEEE International Test Conference*, pages 680–689, 1990.

[SLC75] H. Daniel Schnurmann, Eric Lindbloom, and Robert G. Carpenter. The weighted random test-pattern generator. *IEEE Transactions on Computers*, C-24(7):695–700, 1975.

[SSMM90] Micaela Serra, Terry Slater, Jon C. Muzio, and Michael Miller. The analysis of one-dimensional linear cellular automata and their aliasing properties. *IEEE Transactions on Computer-Aided Design*, CAD-9(7):767–778, July 1990.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1991.

[TS88] K. Totton and S. Shaw. Self-test: the solution to the VLSI test problem? *IEE Proceedings E (Computers and Digital Techniques)*, 135(4):190–195, 1988.

[Tur90] Jon Turino. *Design 'o Test, second edition.* Van Nostrand Reinhold, 1990.

[Tv91] G. Troup and A. J. van de Goor. Logic synthesis of 100-percent testable logic networks. In *Proceedings of IEEE International Conference on Computer Design*, pages 428–431, 1991.

[TW83] Donal T.Tang and Lin S. Woo. Exhasutive test pattern generation with constant weight vectors. *IEEE Transactions on Computers*, C-32(12):1145–1150, December 1983.

[vCD92] Jos van Sas, Francky Catthoor, and Hugo De Man. Optimized BIST strategies for programmable data paths based on cellular automata. In *Proceedings of the IEEE International Test Conference*, pages 110–119, 1992.

[WC91] Wolf and Camposano, editors. *High Level Synthesis Systems.* Kluwer Academic Publishers, 1991.

[WE85] Neil Weste and Kamran Eshraghian. *Principle of CMOS VLSI Design, A Systems Prespective.* Addison Wesley, 1985.

[WEF+85] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault simulation for structured VLSI. *VLSI Systems Design*, 6(12):20–32, December 1985.

[WH92] Hans-Joachim Wunderlich and Sybille Hellebrand. The pseudoexhaustive test of sequential circuits. *IEEE Transactions on Computer-Aided Desig* , CAD-11(1):26–33, January 1992.

[Win89] Th. Winter. High-level test generation for sequential circuits. In *European Test Conference*, pages 314–321, 1989.

[WM86] Laung-Terng Wang and Edward J. McCluskey. Condensed linear feedback shift register (LFSR) testing – a pseudoexhaustive technique. *IEEE Transactions on Computers*, C-35(4):367–370, April 1986.

[Wo'87] Wayne Wolf. Mix-and-match prototyping using objects. In *Fast Prototyping of VLSI*, pages 117–126. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Wol89] Wayne H. Wolf. How to build a hardware description and measurement system on an object-oriented programming language. *IEEE Transactions on Computer-Aided Design*, CAD-8(3):288–301, March 1989.

[Wol91] Wayne Wolf. Object-oriented programming for CAD. *IEEE Computer*, 24(3):35–41, March 1991.

[Wun87] Hans-Joachim Wunderlich. Self test using unequiprobable random patterns. In *International Symposium on Fault-Tolerant Computing*, pages 258–263, 1987.

[Wun88] Hans-Joachim Wunderlich. Multiple distributions for biased random test patterns. In *Proceedings of the IEEE International Test Conference*, pages 236–244, 1988.

[ZB88] Xi-An Zhu and Melvin A. Breuer. A knowldege-based system for selecting test methodologies. *IEEE Design and Test of Computers*, pages 41–59, October 1988.

[ZBM92] S. Zhang, R. Byrne, and D. M. Miller. BIST generators for sequential faults. In *Proceedings of IEEE International Conference on Computer Design*, pages 260–264, 1992.

[ZM90] S. Zhang and D. M. Miller. A comparison of LFSR and cellular automata BIST. In *Canadian Conference on VLSI*, pages 8.4.1–8.4.9, October 1990.

[ZMM91] S. Zhang, D. M. Miller, and J. C. Muzio. Determination of minimal cost one-dimensional linear hybrid cellular automata. *Electronics Letters*, 27:1125–1126, 1991.

Appendix A

Logic III(UVic) Reference Manual

A.1 Introduction

The Logic III(UVic) HDL (Hardware Description Language) is developed to provide a platform for the specification and evaluation of BIST (Build-In Self-Test) embedding. A BIST embedding adds the functionality of self test to a circuit. Logic III(UVic) is based on an extension of the structural description features of Logic III[MCN90]. The test effectiveness of the BIST embeddings are measured with a program that parses a Logic III(UVic) circuit description and performs fault simulation. Fault-free simulations are also provided to enable functional verification.

The goals of the Logic III(UVic) language and its environment are:

1. to easily specify various BIST architectures,
2. to extend the descriptive power of Logic III, and
3. to provide a simulation environment that allows the test quality to be measured.

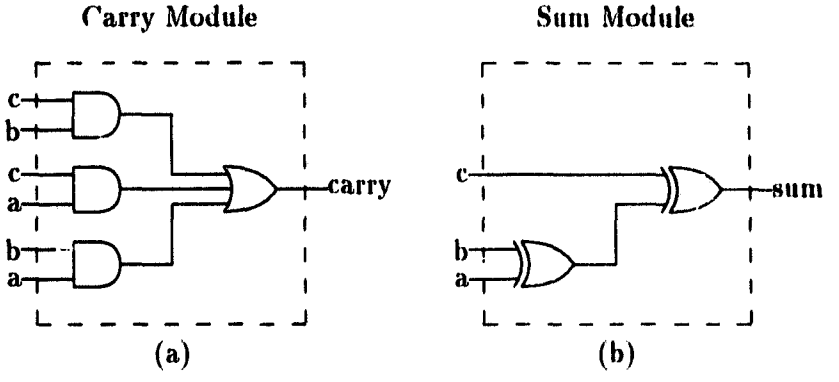
A.2 Basic Concepts

A.2.1 Circuit Structure

The traditional way to describe circuit structure is with schematic diagrams. A carry circuit is shown in Figure A.1a. Figure A.1b shows a sum circuit. Schematic diagrams are composed of symbols, lines, and labels. The symbols represent information transforming

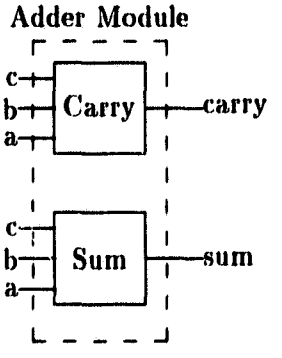
entities, referred to as *modules*. The lines, referred to as *nets*, represent the interconnection of these entities allowing information to be transferred between the entities. Lines with the same labels are connected. For example, in Figure A.1a the lines with the b label are connected.

Figure A.1 Adder and Carry Circuits



Hierarchical circuit description is accomplished by grouping a set of modules and nets together, and then labeling the nets that connect to the outside. Modules contained in a module definition are called *component* modules. The dashed lines in Figure A.1 indicate the grouping. The labeled nets are referred to as *ports*. Using the carry and sum modules, an adder module is created (See Figure A.2). Modules that are not composed of other modules are called *primitive*.

Figure A.2 Hierarchical Modules



The Logic III(UVic) language is designed to represent the structure of a circuit by providing constructs for modules, ports, and nets. Figure A.3 shows the Logic III(UVic)

programs for the carry and sum circuits. The syntax of a module is similar to that of a Pascal procedure. A module's parameter list represents the ports. Variables are used to represent nets. The variable types, **input** and **output**, specify which variables are the module's inputs and outputs. The variable type **node** specifies a net local to the module that must be attached to at least one component's input and at least one component's output.

Figure A.3 Logic III(UVic) code for sum and carry modules

```
net_module xor ( x,y : input; xor_xy : output ); external;
net_module a2 ( x,y : input; and_xy : output ); external;
net_module o3 ( x,y,z : input; or_xyz : output ); external;

net_module carry( a,b,c : input; carry : output );
var
    ab, ac, bc : node;
begin
    a2( a, b, ab);
    a2( a, c, ac);
    a2( b, c, bc);
    o3( ab, ac, bc, carry );
end.

net_module sum( a,b,c : input; sum : output);
var
    ab : node;
begin
    xor( a, b, ab);
    xor( ab, c, sum);
end.
```

The ports and name of the primitive modules **a2**, **o3**, and **xor** are introduced with a module definition where the body is defined as external. The **a2** module is defined in its **net_module** definition, and it is instantiated in the **carry** module. **a2** is a component module of the **carry** module. A module instantiation means that a physical realization of the module is included in the construction of the module in which it is instantiated. Nets are similarly defined in a variable definition and instantiated when they are used in a component module.

Although instantiating a component module uses the same syntax as Pascal procedure calls, the semantics are completely different. The net arguments to a component module and the associated parameters in the component module are variables which represent the same net. Instantiating a module is equivalent to copying the component's body in-place, with the necessary variable name changes to ensure that the different nets remain distinct.

The Logic III(UVic) text for an the adder circuit depicted in Figure A.2 is:

```
net_module adder( x,y,z : input; sum, carry : output );
begin
    sum( x,y,z, sum);
    carry( x,y,z, carry);
end.
```

An equivalent description with the sum and carry modules expanded in-place is:

```
net_module adder( x,y,z : input; sum, carry : output );
var
    s_ab : node;
    c_ab, c_ac, a_bc : node;
begin
    // expanded sum
    xor( x, y, s_ab );
    xor( s_ab, z, sum);
    // expanded carry
    a2( x, y, c_ab);
    a2( x, z, c_ac);
    a2( y, z, c_bc);
    o3( c_ab, c_ac, c_bc, carry )
end.
```

Note that arguments x, y, z and sum to the sum module replace the parameters a, b, c and sum. The local variable ab of sum is renamed s_ab. This avoids a clash with the local variable ab of the carry module. The parameters of carry are similarly replaced. It is the responsibility of a Logic III(UVic) compiler to perform this transformation until only primitive modules remain.

The preceding examples show modules in which the structure is fixed. Logic III(UVic) has the ability to describe modules in which the structure can vary. For example:

```
net_module add_sub( as_flag : integer;
    a,b,c : input; sum, carry : output );
var
    b_inv : node;
begin
    if as_flag = 1 then begin
            adder(a,b,c, sum, carry);
    end
    else begin
            i1(b, b_inv);
            adder(a,b_inv,c, sum, carry)
    end;
end.
```

Depending on the value of the integer parameter **as_flag** this module will produce either a subtractor or an adder module. Customized modules are specified with non-net parameters and/or array parameters. A Logic III(UVic) module definition is therefore a template for the module's structure.

In summary, Logic III(UVic) describes circuits with nets, modules, and ports. Nets are represented by variables of type **node**, **input**, or **output**. Modules are represented by the **net_module** construct and the parameters of a **net_module** represent ports.

A.2.2 Lexical Categories

The text of a Logic III(UVic) design can be grouped into the following lexical categories:

- white spaces,
- comments,
- integer constants,
- string constants,
- operators and delimiters,
- keywords, and
- identifiers.

Any sequence of blanks, tabs, newlines, carriage returns are consider as white space. White space separates other lexical elements, otherwise it has no significance.

Comments are enclosed by (* and *). Comments are ignored by the language. Comments do not nest. A comment including the rest of the line is introduced with //. Since all text between (* and the first occurrence of *) is ignored, // and (* have no effect until *) is encountered. Examples of comments are:

```
(* this a comments *)

// so is this

(* // and this *)
```

An integer constant consists of a sequence of digits. The digits are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Only base 10 integers are supported.

String constants consist of all the characters enclosed by a pair of ". A string constant can not contain a newline character. The following C-language character escapes are recognized:

\r	carriage return
\n	new line
\b	back space
\t	tab
\\	\

String constant examples are:

```
"hello world\n"
"0100 1000 0011"
```

The following sequence of punctuation characters act as delimiters or operators:

```
=   <   >   <=  >=  ..  :=  :-  <-  ,   .
[   ]   (   )   +   -   *   /   '   "
```

The keywords in Logic III(UVic) are:

and	array	begin	circuit	connect	const
display	do	downto	else	end	external
faults_off	for	function	global	if	in
includedef	input	integer	mod	net_module	node
not	observer	of	or	output	reg
string	then	to	use	var	while
write	assert				

The keywords can also appear in upper case.

An identifier is any sequence of at least one letter followed by any number of letters, or digits or '_'s that is not a keyword.

A feature that allows inclusion of files into the source text is provided with:

```
#include "filename"
```

The contents of `filename` appear to Logic III(UVic) as if they are in the test of the design.

A.2.3 Program Structure and Scoping

A BNF notation describes the syntax of Logic III(UVic).

```
<text>          → [ <global> | <mod-defn-list> ]+ <circuit>
<global>        → global <directives> <const-defn> <var-defn> end.
<directives>    → <directive> | <directives> <directive>
<directive>     → includedef ( <string-constant> ) ;
<mod-defn-list> → <mod-defn> | <mod-defn-list> <mod-defn>
<mod-defn>      → <fun-defn> | <netmod-defn>
<fun-defn>      → function <identifier> ( <params-defn> ) : <type> ;
                        <mod-body>
<netmod-defn>   → net_module <identifier> ( <params-defn> ); <mod-body>
<mod-body>      → <const-defn> <var-defn> begin <inst-stmt-list> end.
                | external;
<circuit>       → circuit <identifier> ; <const-defn> <var-defn>
                        begin <inst-stmt-list> end.
```

The text of a complete Logic III(UVic) design is:

```
net_module a2( a,b : input; x : output ); external;
net_module o3( a,b,c : input; x : output ); external;

net_module two_of_three( a,b,c : input; b_flag : output );
var
    ab, ac, bc : node;
begin
    a2( a,b, ab);
    a2( a,c, ac);
    a2( b,c, bc);
    o3( ab, ac, bc, b_flag);
end.

circuit main;
var
    r,s,t : input;
    flag : output;
begin
    two_of_three( r,s,t, flag);
end.
```

This example describes a circuit composed of four primitive modules and seven nets. The seven nets are `r`, `s`, `t`, and `flag` from `main`, and `ab`, `ac`, and `bc` from `two_of_three`. The parameters `a`, `b`, `c`, and `b_flag` are replaced by the associated arguments `r`, `s`, `t`, and `flag`.

All constant, variable and module names are visible immediately after they appear in the text. Logic III(UVic) has global and local scope. Variables and constants defined in the <*global*> section are visible inside all modules unless a constant or variable with the

same name is defined in the module's scope. Variables defined in the parameter list and constants and variables defined in the modules <*var-defn*> and <*const-defn*> sections are in the module's scope (i.e., only visible inside the module).

A.3 Types, Variables and Constants

The variable types in Logic III(UVic) can be separated into ones that are part of the structural description and ones that are used to customize the structural description. The string and integer types are used to parameterize the structural description. Variables of these types are directly interpreted by a Logic III(UVic) compiler, by themselves they do not represent any structure. The **node** type represents the interconnection element in the structural description (e.g., the wire). The **input** and **output** types are specialized forms of the **node** type that can occur only in a module's parameter list or in the **var** section of the circuit module. The **input** and **output** types in the **circuit** module specify which variables are the primary inputs and outputs of the circuit. The term *net* will be used when a variable can be either a **node**, **input**, or **output** type. A **reg** type variable represents a 1-bit memory element.

The syntax for variable definitions is:

<*var-defn*> → **var** <*v-defn-list*>
<*v-defn-list*> → <*v-defn*> ; | <*v-defn-list*> <*v-defn*> ;
<*v-defn*> → <*identifier-list*> : <*type*>
<*type*> → <*b-type*> | <*array-type*>
<*b-type*> → **node** | **input** | **output** | **reg** | **integer** | **string**
<*identifier-list*> → <*identifier*> | <*identifier-list*> , <*identifier*>

The parameter variable definition syntax is:

<*params-defn*> → <*v-defn-list*>

The **node** type is an illegal parameter type. The **input** and **output** types can not appear inside a module's <*var-defn*> section. The **input** and **output** types are really relics from Logic III and should be replaced with the **node** type and a type modify which specifies the direction of information flow in to or out of a module.

The two operations defined on net variables are instantiating and aliasing. A net variable is instantiated when it occurs as an argument in a module instantiation. A net variable is aliased to the associated parameter variable of the component module in which it appears as an argument.

In a module's parameter list an **input** type specifies that the signal on the net is generated outside of the module. It is illegal for an **input** variable to become an argument of a

module where the matching parameter's type is **output**. For the **output** variable the signal on the net is either generated by a module instantiated inside the module, or is generated by one of the input nets. Thus it must appear as an argument to a component module in which the matching parameter's type is **output**, or it must be aliased with an input variable.

Variables of type **reg** can be instantiated by placing the variable as an argument to a module and as the target of an update[1] statement. The <*update-stmt*> statement specifies which net drives the input to the memory element. The output of the **reg** variable memory element is accessed by using the variable as an ordinary net variable. In a module instantiation the associated parameter's type must be **input**. Instantiation of a **reg** variable produces the components that implement the memory element.

Integer variable have the usual set of arithmetic and comparison operations defined. The arithmetic operations are: addition, multiplication, subtraction, division, and reminder (modulus). The comparison operations are: equal to, less than, less than or equal to, greater than, and greater than or equal to. The value of an integer variables can be modified with an assignment statement. Built-in integer functions that calculate the minimum, maximum, and logarithm base 2 of their arguments are provided.

Strings can be assigned, compared for equality and sub-strings can be extracted. The syntax for sub-string extraction is identical to that of array slicing. The syntax for array slices is defined in the next section. Other string operations are currently not defined.

The constant section allows the user to give constants more meaningful names and localizes the constant definition to one place, enabling easy modification. A constant definition is visible as soon as it is defined. The syntax is:

<*const-defn*> → **const** <*c-def-list*>
<*c-def-list*> → <*c-def*>; | <*c-def-list*> <*c-def*> ;
<*c-def*> → <*identifier*> = <*i-constant*> | <*identifier*> = <*s-constant*>

A.3.1 Arrays

Array types are also supported in Logic III(UVic). One dimensional arrays of the base types can be created. The syntax to define a variable of an array type is:

<*array-type*> → **array** <*array-size*> **of** <*b-type*>
<*array-size*> → [<*expr1*> .. <*expr2*>] | []

The <*expr1*> expression must evaluate to an integer, and is the lower bound of the array's index range. <*expr2*> is the upper bound of the array's index range. Expressions

[1] The update statement should be removed, since we can figure out from the matching parameter's type whether we have connected to the memory element's input or output

<expr1> and *<expr2>* can depend on the modules parameters, and thus the size of the array is determined when a module is instantiated. The *<expr1>* expression must be less than or equal to the *<expr2>* expression. If the *<array-size>* is **[]**, then the size of the array depends on the associated argument in the module's instantiation statement. Its lower bound is 0 and its upper bound is its size minus one.

For example, given the following array definition and module header:

```
var counter : array [3..10] of node;
net_module bin_counter( cnt : array [] of output );
```

and the instantiation statement:

```
bin_counter( counter );
```

the parameter **cnt** size is (10-3)+1= 8, its first index is 0, and its last index is 7. In Logic III, where the variable size array does not exists, a functionally equivalent module header is:

```
net_module bin_counter(sz : integer;
    cnt : array [0..sz-1] of output );
```

and the instantiation is:

```
bin_counter(8, counter );
```

Although this syntax is accepted by Logic III(UVic), the former is preferable.

The array elements can be referenced as individual elements or as slices. The operation of array slicing allows a subarray, a continuous segment of the array, to be accessed. The array access syntax for a slice and an element are:

<array-slice> → *<identifier>* [*<expr1>* .. *<expr2>*]
<array-elem> → *<identifier>* [*<expr>*]

The *<expr1>* expression must be less than or equal to the *<expr2>* expression. An array slice of size one is still an array. Example array definitions for the variables **arr** and **fred** are:

```
var
    arr : array [1..8] of node;
    fred : array [1..16] of integer;
```

A slice containing the elements 3, 4 and 5 of **arr** is **arr[3..5]**. **fred[5..5]** is a integer array of size 1, while **fred[5]** is an integer. The range of all array access must be in the lower and upper bounds of the array.

The following properties are associated with arrays:

Property	Meaning
first	the array's lower bound
last	the array's upper bound
size	the array's size

The value of an array's property is obtained with the syntax:

<prop-expr> → *<array-expr>* ' *<array-property>*
<array-expr> → *<identifier>* | *<array-slice>*

For the first, last, and size property the property expression evaluates to an integer constant. Using the definition of **fred**, the following expression values are:

```
fred'first = 1
fred'last  = 16
fred'size  = 16
```

In addition to array variables, one dimensional arrays can be created by concatenating scalar variables, array variables, and array slices. The syntax is:

<collection> → [*<primary-list>*]
<primary-list> → *<primary>* | *<primary-list>* , *<primary>*
<primary> → *<identifier>* | *<array-elem>* | *<array-slice>*

For the following variable definitions:

```
var
    a,b,c : node;
    e     : array [5..15] of node;
```

A collection of these variables forming an array can be specified with:

```
[ a,b,c, e[15], e[6..7] ]
```

This collection's type is **array [0..5] of node**. The lower bound of a collection is 0, its upper bound and size is the sum of the sizes of all the items in the collection minus one. An example of using a collection is:

```
var a,b,c,d,e : node;

net_module And( x : array [] of input; y : output); external;

And( [a,b,c,d], e );
```

The collection of scalar elements **a**, **b**, **c**, and **d** form a four element array, that is aliased to the **x** array of **And**. In this case, the **And** module produces a circuit where the **y** output is the Boolean AND of the elements in the **x** array. The collections and variable sized arrays provide a convenient way of describing these kinds of circuits.

A.3.2 Expressions

The syntax for integer, string and net type expression is:

<i-expr> → (*<i-expr>*)
| - *<i-expr>*
| *<i-expr>* *<i-bop>* *<i-expr>*
| *<funct-call>*
| *<primary>*

<i-bop> → + | - | * | / | mod

<funct-call> → min (*<expr-list>*)
| max (*<expr-list>*)
| lg2 (*<expr>*)
| *<identifier>* (*<expr-list>*)

<rel-i-expr> → (*<rel-i-expr>*)
| not *<rel-i-expr>*
| *<rel-i-expr>* and *<rel-i-expr>*
| *<rel-i-expr>* or *<rel-i-expr>*
| *<i-expr>* *<i-rop>* *<i-expr>*

<i-rop> → < | > | <= | >= | =

<s-expr> → *<primary>*

<rel-s-expr> → *<s-expr>* = *<s-expr>*

<n-expr> → *<collection>* | *<primary>*

<expr> → *<i-expr>* | *<s-expr>* | *<n-expr>*

<expr-list> → *<expr>* | *<expr-list>* , *<expr>*

Integer and string expressions are used in the customization of modules. The integer operators **+ - * / mod = <= >=** are identical to Pascal's integer operators. **min** returns the minimum integer of its arguments. **max** returns the maximum integer of its arguments. **log2** returns the ceiling of $\log_2$. The **=** operator for string expression checks for equality.

A.4 Module

A.4.1 Headings

A module realizes the primary unit of design for a circuit. Similar to functions/procedures in Pascal, a module allows the designer to specify components that are used in the construction of a circuit. In a Logic III(UVic), a module is defined once, but can be instantiated zero or more times. Since modules are templates where the structure depends on its input arguments, each module instantiation can yield a different structure. A modules definition can be recursive. A recursive module definition implies that the module structure is customized.

The two types of module definitions are **net_module** and **function** modules. Their syntax is:

<mod-defn> → [*<func-header>* | *<mod-header>*] *<mod-body>*
<func-header> → **function** *<identifier>* (*<var-param-list>*) : *<f-type>* ;
<mod-header> → **net_module** *<identifier>* (*<var-param-list>*) ;
<f-type> → **integer** | **string** | **output** | **array** *<array-size>* **of** *<f-type>*

Functions and net_modules have almost identical semantics, with the exception that functions occur in expression and return a value for use in the expression, while net_modules occur in statements. Functions returning integer or string types do not create structure, but are used in the customizing descriptions. The return type of a function can neither be a **node** nor an **input**. The net parameters of a module define the modules ports. String and integer parameters are used to specify how a module is instantiated. Variable sized arrays also affect how a module is instantiated.

Some example module headers are:

```
net_module adder2( a0,a1,b0,b1 : input; c0,c1,c2 :output);

net_module m_and( x : array [] of input; xs_and : output );

function f_and ( x : array [] of input ) : output;

net_module misr( misr_type : string ;
    obs : array [1..8] of input;
    sig : array [1..8] of output );
```

Since the parameters of **adder2** are only nets, this module is not customizable (i.e., each instantiation produces exactly the same structure). The **m_and** and **f_and** module headers could define an interface to a module that computes the boolean AND of all the elements

in the **x** array. The choice of using a module or a function would depend on how a designer wanted to use the component. These modules are customized by the size of the **x** array. The MISR module instantiation is customized by the **misr_type** parameter.

A.4.2 Module Body

The component modules and their interconnections defining the module's structure are contained in the body. The body is composed of three sections, the first defines the local constants, the second defines the local variables, and the third lists the component modules that are instantiated.

<mod-body> → *<const-defn>* *<var-defn>* **begin** *<inst-stmt-list>* **end.**
<inst-stmt-list>→ *<inst-stmt>* *<inst-stmt-list>* ; *<inst-stmt>*

All local net variables and the customizing string and integer variables are defined in the **var** section. Again variables are visible as soon as they are defined.

The final section in the body consists of a list of instantiation statements to define the component modules. These statements are:

<inst-stmt> → **begin** *<inst-stmt-list>* **end**
| *<connect-stmt>* | *<update-stmt>*
| *<mod-inst>* | *<if-inst-stmt>* | *<if-else-inst-stmt>*
| *<for-inst-stmt>* | *<while-i-stmt>* | *<use-stmt>*

The *<connect-stmt>* statement associates two variables with the same net (i.e., one variable is an alias for another).

<connect-stmt>→ **connect** (*<identifier>* , *<identifier>*)

It is illegal for a *<connect-stmt>* statement to alias two variables where the type is **input**.

The *<update-stmt>* statement is used to instantiate the structure associated with a **reg** variable, and to attach the data input of the instantiated memory element to the given net variable.

<update-stmt> → *<identifier>* **<-** *<variable>*

The *<if-inst-stmt>* and *<if-else-inst-stmt>* statements provide the primary means in Logic III(UVic) of customizing a modules instantiation. For the *<if-inst-stmt>* statement, the statements in the then-clause are instantiated if the condition evaluates as true. For the *<if-else-inst-stmt>* statement, the statements in the else-clause are instantiated if the condition evaluates as false, otherwise the then-clause is instantiated.

<*if-inst-stmt*> → **if** <*rel-expr*> **then** <*inst-stmt*>
<*if-else-inst-stmt*> **if** <*rel-expr*> **then** <*inst-stmt*> **else** <*inst-stmt*>

```
if and_flag = 1 then
    a2(x,y, xy);
else
    o2(x,y, xy);
```

The above code fragment will instantiate the a2 module if **and_flag = 1**, otherwise the **o2** gate is instantiated.

The <*mod-inst*> statement specifies a component module. The module's definition must have already appeared. While net variables are declared in the modules parameter list or **var** section, they are not instantiated until they occur in an argument list of a module. The matching of arguments to parameters in an instantiating statement aliases each argument with its matching parameter.

<*mod-inst*> → <*identifier*> (<*expr-list*>)

If the module name in the <*mod-inst*> statement is the same name as the one in the module header then this modules definition is recursive. The parameters of a recursive module must include non-net parameters and/or variable sized array parameter. For example:

```
net_module And( x : array [] of input; y : output);
var
    mid : integer;
    m1, m2 : input;
begin
    if x'size > 2 then begin
            mid := x'size / 2 ;
            And( x[1..mid], m1 );
            And( x[mid+1..x'size], m2);
            a2(m1, m2, y);
    end
    else if x'size = 2 then
            a2( x[1], x[2], y);
    else
            connect( x[1], y );
end.
```

The above module creates a tree of a2 modules to compute the Boolean AND of its inputs. It is customized by the x array size.

The <*for-inst-stmt*> and <*while-i-stmt*> statements are used to repeatily instantiate the <*inst-stmt*> statements defined in their bodies. The for-statement will instantiated its

statements abs(<*expr2*> - <*expr1*>) + 1 times. The while-statement will instantiate its body as long as its condition is true.

<*for-inst-stmt*> → **for** <*identifier*> := <*expr1*> <*for-dir*> <*expr2*>
do <*inst-stmt*>
<*for-dir*> → **to downto**
<*while-i-stmt*> → **while** <*rel-expr*> **do** <*inst-stmt*>

An example is:

```
for i := 1 to 16 do begin
    a2( x[i], pass, xy[i]);
    exor( xy[i], z[i], out[i] );
end;
```

Sixteen **a2** and **exor** modules will be instantiated by the above **for** statement. An equivalent Logic III(UVic) design fragment consists of sixteen replications of the **a2** and **xor** module statements. Thus the looping construct provides a easy way to specify iterative structures. Inside a for or while statement, the net variables attached to a module's output ports must be an array element, otherwise the scalar net could be attached to more than one output.

The <*use-stmt*> statement is provided by the language to support BIST embedding, and its semantics is discussed in the next section.

A.5 BIST Embedding Support

The language feature that supports BIST embedding is based on the observations that most BIST architectures consist of modification to the memory elements. These modification convert the memory elements into pattern generators and/or response analysis structures. In Logic III(UVic) memory elements are described with **reg** variables. The <*use-stmt*> specifies the structure to be instantiated for the set of reg variables defined in the <*use-stmt*> body.

<*use-stmt*> → **use** <*identifier*> **in** <*inst-stmt*>

The <*identifier*> must be the name of **net_module** with the following signature:

```
net_module embedding(d:array[]of input; q:array[]of output);
```

An embedding module which instantiates each **reg** variable is:

```
net_module register ( d : array [] of input; q : array [] of output );
var
    i : integer;
begin
            for i := 1 to d'size do
                dff(d[i],q[i]);
end.
```

For all **reg** variables the data inputs, the nets defined in the <*update-stmt*>, and the **reg** outputs are collected in **d** and **q** arrays respectively. The embedding module is passed these two arrays when it is instantiated. Since the user defines the embedding module, any structure can be instantiated for the memory elements.

A.6 Language Support of Simulation

In addition to the structural description features of Logic III(UVic), constructs that interface to a simulator are provided. Logic III(UVic) includes features for displaying nets, specifying observation points, and specifying nets where faults are not considered. The simulation interface constructs appear after a module's variable declaration section and before its body. The syntax is:

<*circuit*> → **circuit** <*identifier*> ;
<*const-defn*> <*var-defn*> <*sim-info*> **begin** <*inst-stmt-list*> **end.**
<*mod-body*> → <*const-defn*> <*var-defn*> <*sim-info*> **begin** <*inst-stmt-list*> **end.**
<*sim-info*> → <*display-info*> | <*observer-info*> | <*faults-info*>
<*display-info*> → **display** <*d-item-list*>
<*observer-info*> → **observer** <*primary-list*>
<*faults-info*> → **faults_off** <*primary-list*>
<*primary-list*> → <*primary*> | <*primary-list*> <*primary*>
<*d-item-list*> → <*d-item*> | <*d-item-list*> <*d-item*>
<*d-item*> → <*primary*> **[** : <*format*> **[** : <*fmt-param*>**]** **]**
<*format*> → <*identifier*>
<*fmt-param*> → <*i-constant*>

The <*format*> and <*fmt-param*> values are specified by the simulator.

For example,

```
circuit div7;
const
    X_SIZE = 7; Q_SIZE = 6; R_SIZE = 3; // upper bound
var
    x : array [0..X_SIZE] of input;
    q : array [0..Q_SIZE] of output;
    r : array [0..R_SIZE] of output;
    sig : array [0..9] of node;
display
    x:hex, q:hex, r:hex
observer
    sig ;
faults_off
    r;
begin
    m_div7(x, q, r);
    misr( [q,r], sig);
end.
```

shows the nets **x**, **q**, and **r** in a <*display-info*> section, the net **sig** in a <*observer-info*> section, and the net **r** in a <*faults-info*> section. The simulator is responsible for displaying the values of **x**, **q**, and **r** during fault free simulations. The value of the nets **sig** are used to determine the fault coverage during fault simulation. The fault simulator will not generate any faults associated with the nets **r**.

Appendix B

Lg3 User's Manual

B.1 Introduction

Lg3 is a CAD tool that is designed to aid in the BIST embedding process. This tool parses a Logic III(UVic) circuit description and performs functional and fault simulation. A test script controls the simulations. The test script can be included in the Logic III(UVic) text. The Logic III(UVic) language has features to easily specify BIST embeddings and to interface with a simulator. The tool allows designers to evaluate the test effectiveness of a BIST embedding with fault simulation. The main features of the tool are:

- analysis of the Logic III(UVic) text for syntactic and type errors,
- a functional circuit simulator,
- a fast combinational fault simulator, and a slower sequential fault simulator,
- identification of testing problems at the Logic III(UVic) text level,
- support for common BIST pattern generators,
- simulation control by a test script language,
- compatibility with the OASIS design environment.

The fault simulators use the single stuck-at fault module, with fault collapsing. Faults are reported as *exposed* if during fault simulation the fault effect is observed for at least one response. Any fault which causes the final state to differ from the fault free state is also called a *detected* fault.

B.2 Usage

Lg3 is implemented by a C++ program. It should work in any environment that supports a C++ compiler and several megabytes of main memory. Under UNIX, it is invoked with the command:

```
lg3 [-nf] [-o file] files...
```

where

-n specifies that an RNL netlist file is produced,

-o specifies the name of the RNL netlist file produce, the default name is a.all,

-f specifies that no fan outs are added.

Most simulation results are sent to the standard output.

For example, the command:

```
lg3 sub_modules.lg main.lg
```

invokes the **lg3** tool, that parses the files **sub_modules.lg** and **main.lg**, and performs any simulation specified by a test script contained in main.lg. The command:

```
lg3 -n -o logic/main.all sub_modules.lg main.lg
```

creates the RNL netlist in the file **logic/main.all** from the Logic III(UVic) description. Any simulation script is ignored. After executing the above command, the OASIS system can process the RNL netlist produced.

B.3 Test Script

The test script, which appears after the circuit module, syntax is:

<test-script> → **testing** *<var-defn>* **begin** [*<test-cmd>* ;]+ **end.**
<var-defn> → **var** [*<v-defn>* ;]+
<v-defn> → *<identifier>* : *<type>*
<type> → *<b-type>* | *<array-type>*
<b-type> → **integer** | **string**

```
<array-type>   → array of [ <i-expr> ..  <i-expr> ] of <b-type>
<test-cmd>     → begin [ <test-cmds> ; ]+ end.
               | <sim-cmd> | <gen-assign> | <if-cmd> | <while-cmd>
               | <for-cmd> | <assign-cmd> | <write-cmd>
<sim-cmd>      → simulate ( <i-expr> )
               | fault_simulate ( <i-expr> )
               | par_fault_simulate ( <i-expr> )
               | clear_inputs skip ( <i-expr> )
               | re_init_fault_stats
               | print_non_detected
               | print_non_exposed
<gen-assign>   → <primary> :- <bist-gen>
<bist-gen>     → ca ( <i-expr> [ , <s-constant>] )
               | zca ( <i-expr> [ , <s-constant>] )
               | twodca ( <string> [ , <s-constant>] )
               | lfsr ( <i-expr> [ , <s-constant>] )
               | zlfsr ( <i-expr> [ , <s-constant>] )
               | counter ( <i-expr> [ , <i-expr>] )
               | seq ( <string> )
               | file ( <string> )
<if-cmd>       → if <condition> then <test-cmd>
               | if <condition> then <test-cmd> else <test-cmd>
<while-cmd>    → while <condition> do <test-cmd>
<for-cmd>      → for <identifier> := <i-expr> [downtoto]
                      <i-expr> do <test-cmd>
<write-cmd>    → write( <expr-list> )
```

<*i-expr*> and <*condition*> are integer expressions and boolean expressions as defined in Logic III(UVic). The <*if-cmd*>, <*while-cmd*>, and <*for-cmd*> testing commands are the usually control structures. The <*write-cmd*> is used to print out messages.

The <*gen-assign*> statement is used to bind input sources to a set of nets. The size of the input source is determined by the size of target (i.e., the nets it drives). The first bit of the input source is connected to the first net in the target. The following table describes the input sources.

Generator	Type of Pattern Generator
`ca(` *init_state* `[ , ` *machine*`])`	A minimum cost hybrid cellular automata. The CA's initial state is given by the first parameter. The CA cells that use the 150 rule can be specified by the optional machine parameter.
`zca(` *init_state* `[ , ` *machine*`] )`	A CA with an all zero state.
`twodca(` *init_state, machine, r, c* `)`	A two dimensional CA.
`lfsr (` *init_state* `[ , ` *machine*`] )`	A minimum cost linear feed back shift register. The initial state is given by the first parameter. The machine parameter specifies the tap positions.
`zlfsr (` *init_state* `[ , ` *machine*`] )`	A zero state LFSR.
`counter (` *init_state* `[ , ` *increment* `] )`	A binary counter. The optional parameter, *increment*, specifies the value that is added to the counter.
`seq (` *sequence* `)`	A repeated sequence.
`file (` *filename* `)`	The input patterns are contained in the file with the name, *filename*. A pattern is represent by a line containing ones and zeros.

An example of a Logic III(UVic) design that contains a test script is:

```
net_module a2( x,y : input; z : output); external;
net_module o2( x,y : input; z : output); external;

circuit fred;
var
    a, b, c : input;
    o : output;
    tmp : node;
display
    a, b, c, o;
observer
    o;
begin
    a2( a, b, tmp);
    o2( c, tmp, o);
```

```
end.

testing
begin
    [ a, b, c ] :- counter(0);
    write("simulation for fred\n");
    simulate( 8 );
end.
```

The test script directs **lg3** to simulate the circuit for 8 cycles, with the inputs **a**, **b**, and **c** driven by a binary counter. The net **a** is driven by the least significant bit of the counter. The variables **a**, **b**, **c** and **o** are displayed. All the variables visible to the circuit module are visible to the testing script. The output of **lg3** for the above text is:

```
simulation for fred
a=0 b=0 c=0 o=0
a=1 b=0 c=0 o=0
a=0 b=1 c=0 o=0
a=1 b=1 c=0 o=1
a=0 b=0 c=1 o=1
a=1 b=0 c=1 o=1
a=0 b=1 c=1 o=1
a=1 b=1 c=1 o=1
```

The following table defines the actions caused by the <*sim-cmd*> statements.

Command	Action
simulate (<*i-expr*>)	Simulate for <*i-expr*> cycles, printing out all variables that are specified in display statements. Circuits without memory elements simulate more efficiently.
fault_simulate (<*i-expr*>)	Perform single stuck-at fault for <*i-expr*> cycles. Combinational circuits can use the PPSFP fault simulation algorithm, and thus simulate more quickly then sequential circuits. Sequential circuit fault simulation uses the parallel fault algorithm.

par_fault_simulate (<*i-expr*>)	Perform a fault simulation using the parallel fault algorithm. This command is only useful for getting the detectable faults for a combinational circuit.
re_init_fault_stats	Reset the fault statistics.
print_not_detected	Print out the nets where the fault is not detected.
print_not_exposed	Print out the nets where the fault is not exposed.
clear_inputs	Remove the generator bindings.

An example test script for running the fault simulator and printing out the fault coverage and non-exposed faults for fred is:

```
testing
begin
    [ a, b, c ] :- lfsr( 1 );
    fault_simulate( 2 );
    print_non_exposed;
    write( "faults = ", number_of_faults );
    write( " exposed = ", number_of_exposed, "\n");
end.
```

The above script directs **lg3**, to drive the inputs by a three-bit LFSR, perform the fault simulation for 2 cycles, and then print out the faults that are not exposed and the total number of faults and the total number exposed. The output is:

```
S@0:T.c nfd=0 det@:
S@0:T.o nfd=0 det@:
S@0:T.tmp nfd=0 det@:
faults = 6 exposed = 3
```

Each of the first three lines correspond to one fault. The first line states that the stuck-at-0 fault for the net represented by the variable **c** in the top-level module is not exposed.

The **lg3** program defines the following integer variables:

Variable	Meaning
number_of_gates	The total number of primitive gates.
number_of_signals	The number of unique nets.
number_of_faults	The number of stuck-at faults.
number_of_cycles	The total number of simulation cycles. This variable is reset to 0 after the **re_init_fault_stats** command.
number_of_exposed	The total number of faults that have been exposed since the last **re_init_fault_stats** command.
number_of_detected	The total number of faults that have been detected since the last **re_init_fault_stats** command.

An example of a test script that continues the fault simulation until a specified fault coverage is achieve or the maximum number of patterns has been applied is:

```
testing
var
    ninety_five_fc : integer;
begin
    // code to bind inputs to pattern generators
    ninety_five_fc := number_of_faults -
            ( number_of_faults/20 );
    while (ninety_five_fc > number_of_exposed) and
            (number_of_cycles < 256000) do
    begin
        fault_simulate( 256 );
        write( "fc=", number_of_exposed);
        write( "cyc=", number_of_cycles, "\n");
    end
end.
```

For combinational circuits, more efficient simulations are realized if **fault_simulate**'s argument is a multiple of 256.

An example of a script that tries different pattern generators is:

```
// rest of multiply circuit
circuit mply_top;
var
    x, y : array [1..8] of input;
    z : array [1..16] of output;
display
    x:dec, y:dec, z:dec;
```

```
observer
    z;
begin
    mply(x,y,z);
end.

testing
begin
    [ x, y ] :- zca( 0 );
    fault_simulate( 16384 );
    write( "zca fc = ", number_of_exposed, "\n");
    clear_inputs; re_init_fault_stats;
    [ x, y ] :- lfsr( 1 );
    fault_simulate( 16384 );
    write( "lfsr fc = ", number_of_exposed, "\n");
    clear_inputs; re_init_fault_stats;
    [ x, y ] :- counter( 0 );
    fault_simulate( 16384 );
    write("counter fc = ", number_of_exposed, "\n");
end.
```

This script prints the fault coverage achieved by a zero state CA, a LFSR, and a counter.

Appendix C

BIST Library Listings

Lib/stdinc.lg

```
(*
 * Define the library modules
 *)

global
includedef ("/vlsi/lib/oasis/scmos2.0/standard.def");
var
        Reset, Phi1H, Phi2H, Phi1_test : input;
end.

net_module fan( x : input; z : output ); external;
net_module i1( x : input; z : output ); external;
net_module a2( x, y : input; z : output ); external;
net_module a3( w, x, y : input; z : output ); external;
net_module a4( v, w, x, y : input; z : output ); external;
net_module a5( u, v, w, x, y: input; z : output ); external;
net_module a6( t, u, v, w, x, y: input; z : output ); external;
net_module a7( s, t, u, v, w, x, y: input; z : output ); external;
net_module a8( r, s, t, u, v, w, x, y: input; z : output ); external;
net_module a9( q, r, s, t, u, v, w, x, y: input; z : output ); external;
net_module o2( x, y : input; z : output ); external;
net_module o3( w, x, y : input; z : output ); external;
net_module o4( v, w, x, y : input; z : output ); external;
net_module o5( u, v, w, x, y: input; z : output ); external;
net_module o6( t, u, v, w, x, y: input; z : output ); external;
net_module o7( s, t, u, v, w, x, y: input; z : output ); external;
net_module o8( r, s, t, u, v, w, x, y: input; z : output ); external;
net_module o9( q, r, s, t, u, v, w, x, y: input; z : output ); external;
net_module ai2( x, y : input; z : output ); external;
net_module ai3( w, x, y : input; z : output ); external;
net_module ai4( v, w, x, y : input; z : output ); external;
net_module ai5( u, v, w, x, y: input; z : output ); external;
net_module ai6( t, u, v, w, x, y: input; z : output ); external;
net_module ai7( s, t, u, v, w, x, y: input; z : output ); external;
net_module ai8( r, s, t, u, v, w, x, y: input; z : output ); external;
net_module ai9( q, r, s, t, u, v, w, x, y: input; z : output ); external;
net_module oi2( x, y : input; z : output ); external;
net_module oi3( w, x, y : input; z : output ); external;
net_module oi4( v, w, x, y : input; z : output ); external;
net_module oi5( u, v, w, x, y: input; z : output ); external;
net_module oi6( t, u, v, w, x, y: input; z : output ); external;
net_module oi7( s, t, u, v, w, x, y: input; z : output ); external;
net_module oi8( r, s, t, u, v, w, x, y: input; z : output ); external;
net_module oi9( q, r, s, t, u, v, w, x, y: input; z : output ); external;
net_module aoi22(r,s, t,u : input; z : output ); external;
net_module oai22(r,s, t,u : input; z : output ); external;
net_module aoi21(r,s, t : input; z : output ); external;
net_module oai21(r,s, t : input; z : output ); external;
net_module aoi221(r,s, t,u, v : input; z : output ); external;
net_module aoi222(r,s, t,u, v,w : input; z : output ); external;
net_module oai222(r,s, t,u, v,w : input; z : output ); external;
net_module oai221(r,s, t,u, v: input; z : output ); external;
net_module aoi211(r,s, t, u: input; z : output ); external;
net_module oai211(r,s, t, u: input; z : output ); external;
net_module aoi2221(r,s, t,u, v,w, x : input; z:output ); external;
net_module oai2111(r,s, t, u, v: input; z : output ); external;
net_module aoi2111(r,s, t, u, v: input; z : output ); external;
net_module oai2211(r,s, t, u, v, w: input; z : output ); external;
net_module aoi2211(r,s, t, u, v, w: input; z : output ); external;
net_module oai32(r,s,t, u,v: input; z : output ); external;
net_module oai31(r,s,t, v : input; z : output ); external;
net_module aoi31(r,s,t, u: input; z : output ); external;
net_module oai32(r,s,t, u,v : input; z : output ); external;
net_module aoi32(r,s,t, u,v : input; z : output ); external;
net_module exor( x, y : input; z : output ); external;
net_module exnor( x, y : input; z : output ); external;
net_module buf2 ( i : input; o : output ); external
```

Lib/utility.lg

```
(* ———————— memory stuff ———————— *)

net_module n_register(
        hold : input;
        a : array[] of input;
        q : array[a'first..a'last] of output; );
var
        hold_bar : node;
        dffin, dffin_bar : array[a'first..a'last] of node;
        me               : array[a'first..a'last] of reg;
        i : integer;
begin

    i1 ( hold, hold_bar );
    for i := a'first to a'last do begin
        aoi22( a[i], hold_bar, q[i], hold, dffin_bar[i] );
        i1( dffin_bar[i], dffin[i] );
        me[i] <- dffin[i];
        f=: me[i], q[i]);
    end
end.


net_module n_reg(d : array [] of input;
        q : array [d'first..d'last] of output;);
var
        r : array [ d'first .. d'last ] of reg;
        i : integer;
begin
        for i := d'first to d'last do begin
                r[i] <- d[i];
                connect(r[i], q[i]);
        end;
end.

(* ———————— BIG gates ———————— *)
net_module Or_hidden( flag: integer;
        x : array [0..] of input; y : output);
const
        size = x'size;
        left_size = size/2;
var
        t1, t2: node;
        i : integer;
begin
        if flag = 0 then begin
                if size = 1 then
                        connect( x[0], y)
                else if size = 2 then
                        o2( x[0], x[1], y)
                else if size = 3 then
                        o3( x[0], x[1], x[2], y)
                else if size = 4 then
                        o4( x[0], x[1], x[2], x[3], y)
                else if size > 4 then begin
                        Or_hidden(1, x[0..left_size-1], t1);
                        Or_hidden(1, x[left_size..size-1], t2);

                        a2(t1, t2, y)
                end
        end
        else
                if size = 1 then
                        i1( x[0], y)
                else if size = 2 then
                        oi2( x[0], x[1], y)
                else if size = 3 then
                        oi3( x[0], x[1], x[2], y)
                else if size = 4 then
                        oi4( x[0], x[1], x[2], x[3], y)
                else if size > 4 then begin
                        Or_hidden(0, x[0..left_size-1], t1);
                        Or_hidden(0, x[left_size..size-1], t2);
                        oi2(t1, t2, y)
                end
end.

net_module Or( x : array [] of input; y : output );
begin
        Or_hidden(0, x, y);
end.

net_module Nor_hidden( flag: integer;
        x : array [0..] of input; y : output);
const
        size = x'size;
        left_size = size/2;
var
        t1, t2: node;
        i : integer;
begin
        if flag = 0 then begin
                if size = 1 then
                        i1( x[0], y)
                else if size = 2 then
                        oi2( x[0], x[1], y)
                else if size = 3 then
```

```
                oi3( x[0], x[1], x[2], y)
            else if size = 4 then
                oi4( x[0], x[1], x[2], x[3], y)
            else if size > 4 then begin
                Nor_hidden(1, x[0..left_size-1], t1);
                Nor_hidden(1, x[left_size..size-1], t2);

                oi2(t1, t2, y)
            end
        end
        else
            if size = 1 then
                connect( x[0], y)
            else if size = 2 then
                o2( x[0], x[1], y)
            else if size = 3 then
                o3( x[0], x[1], x[2], y)
            else if size = 4 then
                o4( x[0], x[1], x[2], x[3], y)
            else if   ze > 4 then begin
                Nor_hidden(0, x[0..left_size-1], t1);
                Nor_hidden(0, x[left_size..size-1], t2);
                ai2(t1, t2, y)
            end
end.

net_module Nor( x : array [] of input; y : output );
begin
        Nor_hidden(0, x, y);
end.

net_module And_hidden( flag: integer;
        x : array [0..] of input; y : output);
const
        size = x'size;
        left_size = size/2;
var
        t1, t2: node;
        i : integer;
begin
        if flag = 0 then begin
            if size = 1 then
                connect( x[0], y)
            else if size = 2 then
                a2( x[0], x[1], y)
            else if size = 3 then
                a3( x[0], x[1], x[2], y)
            else if size = 4 then
                a4( x[0], x[1], x[2], x[3], y)
            else if size > 4 then begin
                And_hidden(1, x[0..left_size-1], t1);
                And_hidden(1, x[left_size..size-1], t2);

                oi2(t1, t2, y)
            end
        end
        else
            if size = 1 then
                i1( x[0], y)
            else if size = 2 then
                ai2( x[0], x[1], y)
            else if size = 3 then
                ai3( x[0], x[1], x[2], y)
            else if size = 4 then
                ai4( x[0], x[1], x[2], x[3], y)
            else if size > 4 then begin
                And_hidden(0, x[0..left_size-1], t1);
                And_hidden(0, x[left_size..size-1], t2);
                ai2(t1, t2, y)
            end
end.

net_module And( x : array [] of input; y : output );
begin
        And_hidden(0, x, y);
end.

net_module Xor ( x : array [0..] of input; y : output);
const
        size = x'size;
        left_size := size/2;
var
        t1, t2: node;
        i : integer;
begin
        if size = 1 then
            connect( x[0], y)
        else if size = 2 then
            exor( x[0], x[1], y)
        else if size > 2 then begin
            Xor( x[0..left_size-1], t1);
            Xor( x[left_size..size-1], t2);
            exor(t1, t2, y)
        end
end.

(* ---------- Bitwise Gates ---------- *)
net_module Fan( x : array [] of input;
```

```
        y : array [x'first..x'last] of output );
var
        i : integer;
begin
        for i:= x'first to x'last do
                fan( x[i], y[i] );
end.

function FFan( x : array [] of input )
        :  array [x'first..x'last] of output;
var
        i : integer;
display x;
begin
        for i:= x'first to x'last do
                fan( x[i], FFan[i] );
end.

net_module bitwise_and(
        r:array []; of input;
        s:array [r'first..r'last] of input;
        z : array [r'first..r'last] of output );
var
        i : integer;
begin
        for i:= r'first to r'last do
                a2( r[i], s[i], z[i] );
end.

net_module bitwise_inv(x : array [] of input;
        y : array [x'first..x'last] of output);
var
        i : integer;
begin
        for i := x'first to x'last do
                i1(x[i], y[i]);
end.

net_module bitwise_exor(
        x : array [] of input;
        y : array [x'first..x'last] of input;
        z : array [x'first..x'last] of output;
);
var
        i : integer;
begin
        for i := x'first to x'last do
                exor(x[i], y[i], z[i]);
end.
```

```
net_module mux2( sel, x, y : input; z : output );
var
        sel_b, t : node;
faults_off;
begin
        i1(sel,sel_b);
        aoi22( x, sel_b, y, sel, t);
        i1( t, z);
end.

net_module n_mux2(
        sel : input;
        x : array [] of input;
        y : array [x'first..x'last] of input;
        z : array [x'first..x'last] of output; );
var
        sel_bar : node;
        z_bar : array [x'first..x'last] of node;
        i : integer;
begin
        i1 ( sel, sel_bar );
        for i := x'first to x'last do begin
                aoi22( x[i], sel_bar, y[i], sel, z_bar[i] );
                i1( z_bar[i], z[i])
        end
end.

(* ———————— decoders ———————— *)

net_module min_term_tc( minterm : integer;
        lit : array [] of input;
        lit_bar : array [ lit'first .. lit'last ] of input;
        out : output; );
var
        mask : integer;
        i : integer;
        tmp : array [ lit'first .. lit'last ] of node;
begin
        mask := 1;
        for i := lit'first to lit'last do begin
                if ((minterm / mask) mod 2) = 1 then
                        connect(lit[i], tmp[i])
                else
                        connect(lit_bar[i], tmp[i]);
                mask := mask * 2
        end;
        And(tmp, out);
end.
```

Lib/utility.lg

```
net_module min_term( minterm : integer;
        lit : array [ ] of input; out : output);
var
        mask : integer;
        i : integer;
        tmp : array [ lit'first .. lit'last ] of node;
begin
        mask := 1;
        for i := lit'first to lit'last do begin
                if ((minterm / mask) mod 2) = 1 then
                        connect(lit[i], tmp[i])
                else
                        i1(lit[i], tmp[i]);
                mask := mask * 2
        end;
        And(tmp, out);
end.

net_module sel_mar_term( marterm : integer;
        sel : array [] of input; out : output);
var
        mask : integer;
        i,index : integer;
        tmp : array [ sel'first .. sel'last ] of node;
begin
        if marterm = 0 then
                write("no max terms");
        mask := 1;
        index := sel'first;
        for i := sel'first to sel'last do begin
                if ((mar'erm / mask) mod 2) = 1 then begin
                        connect(sel[i], tmp[index]);
                        index := index + 1;
                end;
                mask := mask * 2
        end;
        Or(tmp[sel'first .index-1], out);
end.

(* select_lines to n_lines decoder *)
net_module decoder( sel : array [] of input, out : array [] of output );
var
        sel_bar : array [sel'first..sel'last] of node;
        index : integer;
begin

        assert( sel'size <= log2(out'size) );
        bitwise_invert(sel, sel_bar);
        (* now generate the AND gates *)
        for index := out'first to out'last do begin
                min_'erm_tc( index-out'firs', sel, sel_bar, out[index] );
        end;
end.

(* ---------- Arithmetic Modules ---------- *)
net_module half_adder2(a, b: input; s, c: output);
begin
        exor(a,b,s);
        a2(a,b,c);
end.

net_module adder2(a, b, cin: input; s, cout: output;);
var
        x1, coutb: node;
begin
        exor(a, b, x1);
        exor(cin,x1, s);
        aoi222(a,b, a,cin, b,cin, coutb);
        i1(coutb, cout);
end.

net_module tc_adder(
        a: array [] of input;
        b: array [a'first..a'last] of input;
        cin: input;
        sum: array [a'first..a'last] of output;
        carry : output ; );
var
        i,size : integer;
        cout : array [a'first..a'last-1] of node;
begin
        size := a'size;
        adder2(a[a'first], b[a'first], cin,
                sum[a'first], cout[a'first]);
        for i := a'first+1 to a'last-1 do
                adder2(a[i], b[i], cout[i-1], sum[i], cout[i]);
        adder2(a[a'last], b[a'last], cout[a'last-1],
                sum[a'last], carry);
end.

net_module n_add_sub(
        add_sub_ctrl : input;
        a,b: array [] of input;
        sum: array [] of output, );
var
        b_ones_comp : array [0..a'size-1] of node;
```

Lib/utility.lg

```
        t1, carry : node;
        tmp1,i : integer;
begin
        for i := 0 to a'size-1 do
                exor(b[i], add_sub_ctrl, b_ones_comp[i] );
        tmp1 := a'size-1;
        tc_adder( a[0..tmp1-1], b_ones_comp[0..tmp1-1],
                add_sub_ctrl, sum[0..tmp1-1], carry );
        exor(a[a'size-1], b_ones_comp[a'size-1], t1);
        exor( t1, carry, sum[a'size-1] );
end.

net_module n_add_sub_c(
        add_sub_ctrl : input;
        a: array [] of input;
        b: array [a'first..a'last] of input;
        sum: array [a'first..a'last] of output;
        carry : output; );
var
        b_ones_comp          : array [ a'first..a'last] of node;
        i                    : integer;
begin
        for i := a'first to b'last do
                exor(b[i], add_sub_ctrl, b_ones_comp[i] );
        tc_adder(a, b_ones_comp, add_sub_ctrl, sum, carry );
end.

net_module accumulator(
        add_sub_mode,hold,load_acc : input;
        load_in     : array [] of input;
        acc_in      : array [load_in'first..load_in'last] of input;
        acc_out     : array [load_in'first..load_in'last] of output;
var
        t_load_in,sum_out: array [load_in'first..load_in'last] of node;
begin
        n_add_sub(add_sub_mode, acc_out, acc_in, sum_out);
        n_mux2(load_acc, sum_out, load_in, t_load_in);
        n_register(hold, t_load_in, acc_out);
end.


net_module n_buf(o : array [] of input;
        g : array [o'first..o'last] of output);
var
        i : integer;
begin
        for i := o'first to o'last do
                buf2(o[i], g[i]);
end.
```

```
(* ---------- A set of utility functions. ---------- *)

function abs( x : integer ) : integer;
begin
        if x < 0 then abs := -x else abs := x
end.

(*
 * computes: x ** y (ala fortran)
 *)
function pow( x, y : integer ) : integer;
var
        i : integer;
        res : integer;
begin
        if y >= 0 then begin
                res := 1;
                for i := 1 to y do
                        res := res * x;
                pow := res;
        end
        else
                pow := 0;
end.

// find first least significant bit
function first_lsb( val : integer ) : integer ;
var
        flag : integer;
begin
        first_lsb := 0;
        flag := 1;
        while first_lsb < 32 and flag = 1 do begin
                if (val mod 2) = 1 then
                        flag := 0
                else begin
                        val := val / 2;
                        first_lsb := first_lsb + 1;
                end
        end
end.

// find first least significant bit
function first_msb( val : integer ) : integer ;
var
        mask, flag : integer;
begin
        first_msb := 31;
```

```
        mask := 1073741824;
        flag := 1;
        while first_msb > 0 and flag = 1 do begin
                if val >= mask then
                        flag := 0
                else begin
                        mask := mask / 2;
                        first_msb := first_msb - 1;
                end
        end
end.

// find the lenght of a string of ones starting at position lsb
function ls_bit_run( x, lsb, msb : integer ) : integer;
var
        len, mask, flag : integer;
begin
        // mask bit for 2^(lsb) note position 1 is bit 0
        mask := pow(2,lsb);
        len := 1;
        flag := 1;
        while len <= ((msb-lsb)+1) and flag = 1 do begin
                if ( (x/mask) mod 2 ) = 0 then
                        flag := 0
                else begin
                        len := len + 1;
                        mask := mask * 2;
                end
        end;
        ls_bit_run := len;
end.


// find the lenght of a string of ones starting at position msb
function ms_bit_run( x, lsb, msb : integer ) : integer;
var
        len, mask, flag : integer;
begin
        // mask bit for 2^(msb-1) note position 1 is bit 0
        mask := pow(2,msb-2);
        len := 1;
        flag := 1;
        while len <= ((msb-lsb)+1) and flag = 1 do begin
                if ( (x/mask) mod 2 ) = 0 then
                        flag := 0
                else begin
                        len := len + 1;
                        mask := mask/2;
                end
        end;
        ms_bit_run := len;
end.
```

Lib/bist.lg

```
(*
 * BIST generators and response analyzers based
 * on LFSM are found here.
 *
 * Note: the faults have been turned off.
 *)

function lfsr_tap( size, index : integer ) : integer ;
var
        T          : array [3..902] of integer;
begin
        assert(   (size > 0) and (size <= 300) );

        T[   3] :=   -1; T[   4] :=   -1; T[   5] :=   -1; // size=1
        T[   6] :=    1; T[   7] :=   -1; T[   8] :=   -1; // size=2
                .
                .
                .
        T[897] :=   21; T[898] :=    2; T[899] :=    1; // size=299
        T[900] :=    7; T[901] :=   -1; T[902] :=   -1; // size=300

        if (index = 0) then
                lfsr_tap := 1
        else begin
                size := size * 3;
                if (T[size] = index) or (T[size+1] = index)
                     or (T[size+2] = index) then
                        lfsr_tap := 1
                else
                        lfsr_tap := 0
        end
end.

(*
 * function that returns true if the current index for the memory
 * element in the embedding module initial state should be 1.
 *)

function embed_state( p, i : integer) : integer;
begin
        if p = 0 then
                embed_state := 0
        else if p = 1 then begin
                embed_state := 1;
        end
        else if p = 2 then begin
                if i = 1 then //thisthis does not guarantte a 1
                        embed_state := 1
                else
                        embed_state := 0;
        end
        else begin
                if (i mod p) = 0 then
                        embed_state := 1
                else
                        embed_state := 0;
        end;
end.

net_module me( init : integer; d : input; q : output; );
begin
        if init = 1 then cdff(d, q) else dff(d, q);
end.


net_module misr_tap_me( sin, din, tap : input; reg_out : output;);
var
        t1,reg_in : node;
begin
        exor(sin, tap, t1);
        exor(din,t1,reg_in);
        dff(reg_in, reg_out);
end.

net_module misr_me( sin, din : input; reg_out : output; );
var
        reg_in : node;
begin
        exor(sin, din, reg_in);
        dff(reg_in, reg_out);
end.

net_module misr_me_0( sin, din : input; reg_out : output; );
var
        reg_in : node;
begin
        exor(sin, din, reg_in);
        cdff(reg_in, reg_out);
end.

net_module lfsr_tap_me(i : integer; sin, data : input;
        reg_out : output; );
var
        t1 : node;
begin
        exor(sin, data, t1);
        me(i, t1, reg_out);
end.
```

Lib/bist.lg

```
net_module lfsr( pat : array [0..] of output; );
const
        size = sig'size;
var
        i           : integer;
begin
        assert( size > 0);
        me(1, pat[pat'last], pat[0]);
        for i := 1 to pat'last do begin
                if lfsr_tap(size,i) = 0 then begin
                        me(0,pat[i-1], pat[i])
                end
                else begin
                        lfsr_tap_me(0,pat[i-1], pat[pat'last], pat[i]);
                end
        end
end.

net_module init_lfsr( init : integer; pat : array [0..] of output; );
const
        size = pat'size;
var
        i           : integer;
begin
        assert( size > 0);
        me(embed_state(init,1), pat[pat'last], pat[0]);
        for i := 1 to pat'last do begin
                if lfsr_tap(size,i) = 0 then begin
                        me(embed_state(init,i+1),pat[i-1], pat[i])
                end
                else begin
                        lfsr_tap_me(embed_state(init,i+1),pat[i-1],
                                pat[pat'last], pat[i]);
                end
        end
end.

net_module lfsr_z( pat : array [] of output; );
const
        size = pat'size;
var
        i                       : integer;
        z, z_, t_z              : node;
begin
        assert( size > 0 );
        Or(pat[1..pat'last], z);
        zl(z, z_);
        me(1, pat[pat'last], pat[0]);
        if lfsr_tap(size,1) = 0 then
                lfsr_tap_me(0, pat[0], z_, pat[1])
        else begin
                exor( z_, pat[pat'last], t_z);
                lfsr_tap_me(0, pat[0], t_z, pat[1])
        end;

        for i := 2 to pat'last do begin
                if lfsr_tap(size,i) = 0 then
                        me(0, pat[i-1], pat[i])
                else
                        lfsr_tap_me(0, pat[i-1], pat[pat'last], pat[i])
        end
end.

net_module misr( d : array [0..] of input;
        sig : array [d'first..d'last] of output );
const
        size = d'size;
var
        i           : integer;
begin
        assert( size > 0 );
        misr_me(sig[sig'last], d[0], sig[0]);
        for i := 1 to d'last do
                if lfsr_tap(size,i) = 0 then
                        misr_me(sig[i-1], d[i], sig[i])
                else
                        misr_tap_me(sig[i-1],d[i],sig[sig'last],sig[i]);
end.

// big misr with smaller number of inputs,
net_module part_misr( d : array [0..] of input;
        sig : array [0..] of output );
const
        dsize = d'size;
        ssize = sig'size;
var
        i           : integer;
begin
        assert( dsize > 0 );
        assert( dsize <= ssize );
        misr_me(sig[sig'last], d[0], sig[0]);
        for i := 1 to d'last do
                if lfsr_tap(ssize,i) = 0 then
                        misr_me(sig[i-1], d[i], sig[i])
                else
                        misr_tap_me(sig[i-1],d[i],sig[sig'last],sig[i]);
```

Lib/bist.lg

```
        for i := d'last+1 to sig'last do
                if lfsr_tap(ssize,i) = 0 then
                        me(0,sig[i-1], sig[i])
                else
                        lfsr_tap_me(0,sig[i-1], sig[sig'last], sig[i]);
end.

net_module n_sh_register( sh_ctl : input;
        sin : input;
        din : array [] of input;
        sout : output;
        dout : array [din'first..din'last] of output; );
const
        lb = din'first;
        ub = din'last;
var
        m : array [ lb..ub ] of node;
        i : integer;
begin
        mux2( sh_ctl, din[lb], sin, m[lb] );
        dff( m[lb], dout[lb] );
        for i := lb+1 to ub do begin
                mux2( sh_ctl, din[i], dout[i-1], m[i]);
                dff( m[i], dout[i] );
        end;
        connect( dout[ub], sout );
end.

global
var
        act_bisted_nor : input;
        act_bisted_nor_ : input;
end.

net_module Bisted_Nor1( x : array [0..] of input; y : output);
external;

net_module Bisted_Nor( x : array [0..] of input; y : output);
const
        sz = x'size;
        left_sz = sz/2;
var
        i : integer;
        t1, t2: node;
        f_t1, f_t2: node;
        t1_, t2_: node;
begin
        if sz = 1 then
                t1( x[0], y)
        else if sz = 2 then
                o12( x[0], x[1], y)
        else if sz = 3 then
                o13( x[0], x[1], x[2], y)
        else if sz = 4 then
                o14( x[0], x[1], x[2], x[3], y)
        else if sz > 4 then begin
                Bisted_Nor1( x[0..left_sz-1], t1);
                Bisted_Nor1( x[left_sz..sz-1], t2);

                // in normal mode a nor gate
                fan(t1, f_t1);
                ai2(act_bisted_nor, f_t1, t1_);
                fan(t2, f_t2);
                ai2(act_bisted_nor, f_t2, t2_);
                aoi2^0(t1, t2_, t2, t1_, y);
        end;
end.

net_module Bisted_Nor1( x : array [0..] of input; y : output);
const
        sz = x'size; left_sz = sz/2;
var
        i : integer;
        t1, t2: node;
        f_t1, f_t2: node;
        t1_, t2_: node;
begin
        if sz = 1 then
                connect( x[0], y)
        else if sz = 2 then
                o12( x[0], x[1], y)
        else if sz = 3 then
                o13( x[0], x[1], x[2], y)
        else if sz = 4 then
                o14( x[0], x[1], x[2], x[3], y)
        else if sz > 4 then begin
                Bisted_Nor(x[0..left_sz-1], t1);
                Bisted_Nor(x[left_sz..sz-1], t2);
                // in normal mode a nand gate
                fan(t1, f_t1);
                oi12(act_bisted_nor_, f_t1, t1_);
                fan(t2, f_t2);
                oi12(act_bisted_nor_, f_t2, t2_);
                oai22(t1, t2_, t2, t1_, y);
        end;
end.
```

Lib/embed.lg

```
net_module wt_25(
        d : array [1..] of input;
        q : array [d'first..d'last] of output );
var
        i : integer;
begin
        a2( d[d'first], d[d'last], q[d'first]);
        fan( d[d'first+1], q[d'first+1]);
        for i := d'first+2 to d'size do
                a2(d[i-1], d[i], q[i]);
end.

net_module wt_125(
        d : array [1..] of input;
        q : array [d'first..d'last] of output );
var
        i : integer;
begin
        a3( d[d'first], d[d'last-1], d[d'last], q[d'first]);
        a3( d[d'first+1], d[d'last], d[d'first], q[d'first+1]);
        for i := d'first+2 to d'size do
                a3(d[i-2], d[i-1], d[i], q[i]);
end.

net_module wt_25_50(
        d : array [1..] of input;
        q : array [d'first..d'last] of output );
var
        i : integer;
begin
        a2( d[d'first], d[d'last], q[d'first]);
        for i := d'first+1 to d'size do
                if i mod 2 = 0 then
                        a2(d[i-1], d[i], q[i])
                else
                        fan(d[i], q[i]);
end.

(*
 * just embed the normal memory elements
 *)
net_module dff_array (
        d : array [1..] of input;
        q : array [d'first..d'last] of output );
var
        Ld : array [d'first..d'last] of node;
        Lq : array [d'first..d'last] of node;
        i : integer;
faults_off;
begin
        Fan( d, Ld );
        for i := d'first to d'last do
                dff(Ld[i], Lq[i]);
        Fan( Lq, q );
end.

(*
 * combinational approach
 *)
global
var
        comb_generator : array [1..300] of input;
        comb_index     : integer;
end.

function init_comb( x:integer ) : integer;
begin
        comb_index := 1;
        init_comb := 1;
end.

net_module comb_embed(
        obs     : array [] of input;
        gen     : array [obs'first..obs'last] of output );
var
        i : integer;
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
observer
        Lobs;
faults_off
        Lobs;
begin
        Fan(obs, Lobs);
        for i := Lobs'first to Lobs'last do begin
                assert( (comb_index > 0) and
                        (comb_index <= comb_generator'last) );
                connect( comb_generator[comb_index], Lgen[i] );
                comb_index := comb_index + 1;
        end;
        Fan(Lgen, gen);
end.

// conbinational embedding with weights
net_module wt_comb_embed(
        obs     : array [] of input;
        gen     : array [obs'first..obs'last] of output );
var
```

Lib/embed.lg

```
        i : integer;
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
observer
        Lobs;
faults_off
        Lobs;
begin
        Fan(obs, Lobs);
        for i := Lobs'first to Lobs'last do begin
                assert( (comb_index > 0) and
                        (comb_index <= comb_generator'last) );
                connect( comb_generator[comb_index], Lgen[i] );
                comb_index := comb_index + 1;
        end;
        wt_25(Lgen, gen);
end.

net_module finish_comb(dummy:integer);
faults_off
        comb_generator[1..comb_index-1];
begin
        dummy := 1;
end.

(*

(* ------ STUMPS embedding ------- *)

global
var
        stumps_gen : array [1..16] of node;
        stumps_gen_index : integer;
        stumps_obs : array [1..16] of node;
        stumps_obs_index : integer;
        stumps_max_size : integer;
        stumps_init_pattern : integer;

        stumps_shift : node;
end.

function init_stumps( init : integer) : integer ;
begin
        stumps_gen_index := stumps_gen'first;
        stumps_obs_index := stumps_obs'first;
        stumps_init_pattern := init;
        stumps_max_size := 0;
        init_stumps := 1;
end.
```

```
net_module stumps_embed(
        obs     : array [] of input;
        gen     : array [obs'first..obs'last] of output );
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
faults_off;
begin
        assert(stumps_gen_index > 0
                and stumps_gen_index <= stumps_gen'last);
        assert(stumps_obs_index > 0
                and stumps_obs_index <= stumps_obs'last);
        stumps_max_size := max(stumps_max_size, obs'size);

        Fan(obs, Lobs);
        n_sh_register( stumps_shift, stumps_gen[stumps_gen_index],
                Lobs, stumps_obs[stumps_obs_index], Lgen );
        Fan(Lgen, gen);
        stumps_gen_index := stumps_gen_index + 1;
        stumps_obs_index := stumps_obs_index + 1;
end.

net_module stumps_generator_ctrl(gsize : integer );
var
        pat : array[1..gsize] of node;
        i,j : integer;
faults_off;
begin
        assert(  gsize >= stumps_gen_index-1 );
        init_lfsr(stumps_init_pattern,pat);
        j := pat'first;
        for i := stumps_gen'first to stumps_gen_index-1 do begin
                connect( pat[j], stumps_gen[i] );
                j := j + 1;
        end;
end.

net_module stumps_observer( osize : integer );
var
        sig : array[1..osize] of node;
observer
        sig;
faults_off;
begin
        assert( sig'size >= 16 );
        part_misr(stumps_obs[1..stumps_obs_index-1], sig);
end.
```

Lib/embed.lg

```
net_module finish_stumps(dummy:integer);
faults_off
        stumps_obs[1..stumps_obs_index-1],
        stumps_gen[1..stumps_gen_index-1],
        stumps_shift;
begin
        dummy := 1;
end.

(*
(*
 * the classic BILBO
 *)

global
var
        bilbo_sin : array [1..16] of input;
        bilbo_sin_index : integer;
        bilbo_sout: array [1..16] of output;
        bilbo_sout_index : integer;

        bilbo_b1, bilbo_b2 : array[1..16] of input;
        bilbo_index : integer;
end.

function init_bilbo( dummy : integer) : integer ;
begin
        bilbo_index := 1;
        bilbo_sin_index := bilbo_sin'first;
        bilbo_sout_index := bilbo_sout'first;
        init_bilbo := 1;
end.

net_module bilbo_me(d,s : input; q,qb : output);
var
        d_b1, s_b2, d_in : node;
begin
        a2(d, bilbo_b1[bilbo_index], d_b1);
        oi2(s, bilbo_b2[bilbo_index], s_b2);
        exor(d_b1, s_b2, d_in);
        dff(d_in, q);
        i1(q, qb);
end.

net_module bilbo_embed(
        obs     : array [1..] of input;
        gen     : array [obs'first..obs'last] of output );
const
        size = obs'size;
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
        taps    : array [1..size] of node;
        s       : array [0..size] of node;
        feedback: node;
        i, no_taps              : integer;
observer
        Lgen;
faults_off;
begin
        assert(bilbo_index >= bilbo_b1'first
                and bilbo_index <= bilbo_b1'last);
        Fan(obs, Lobs);
        no_taps := 1;
        for i := 1 to Lobs'last do begin
                bilbo_me(Lobs[i], s[i-1], Lgen[i], s[i]);
                if lfsr_tap(size,size-i) = 1 then begin
                        connect(s[i], taps[no_taps] );
                        no_taps := no_taps + 1;
                end;
        end;
        Xor(taps[1..no_taps-1], feedback);
        mux2(bilbo_b1[bilbo_index],
                bilbo_sin[bilbo_sin_index], feedback, s[0]);
        bilbo_sin_index := bilbo_sin_index + 1;
        i1( s[Lobs'last], bilbo_sout[bilbo_sout_index]);
        bilbo_sout_index := bilbo_sout_index + 1;
        bilbo_index := bilbo_index + 1;
        Fan(Lgen, gen);
end.

net_module finish_bilbo(dummy:integer);
faults_off
        bilbo_sin[1..bilbo_sin_index-1],
        bilbo_sout[1..bilbo_sout_index-1],
        bilbo_b1[1..bilbo_index-1],
        bilbo_b2[1..bilbo_index-1];
begin
        dummy := 1;
end.

(*
(*
 * - simple lfsr generator/register, misr/generators
 *)
global
var
        gen_on : array [1..16] of input,
```

Lib/embed.lg

```
        gen_on_index : integer;
        misr_on : array [1..16] of input;
        misr_on_index : integer;
        best_init_pattern : integer;
end.

function init_best( init : integer) : integer ;
begin
        best_init_pattern := init;
        gen_on_index := gen_on'first;
        misr_on_index := misr_on'first;
        init_best := 1;
end.

net_module gen_embed(
        obs         : array [0..] of input;
        gen         : array [obs'first..obs'last] of output );
const
        size = obs'size;
var
        d_s         : array [gen'first..gen'last] of node;
        merge       : array [gen'first..gen'last] of node;
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
        i           : integer;
faults_off;
begin
        Fan(obs, Lobs);
        assert( size > 0);
        mux2(gen_on[gen_on_index],Lobs[gen'first],
                Lgen[gen'last],d_s[gen'first]);
        me(embed_state(best_init_pattern, Lobs'first+1),
                d_s[gen'first], Lgen[gen'first]);
        for i := Lgen'first+1 to Lgen'last do begin
                if lfsr_tap(size,i) = 0 then begin
                        connect(Lgen[i-1], merge[i]);
                end
                else begin
                        exor(Lgen[i-1],Lgen[gen'last],merge[i]);
                        write("tap at ", i, "\n");
                end;
                mux2(gen_on[gen_on_index],Lobs[i], merge[i],d_s[i]);
                me(embed_state(best_init_pattern, i+2),
                        merge[i], Lgen[i]);
        end;
        gen_on_index := gen_on_index + 1;
        Fan(Lgen, gen);
end.
```

```
net_module on_misr_tap_me( ctrl, sin, din, tap : input;
        reg_out : output;);
var
        t1, t2, reg_in : node;
begin
        a2(ctrl, sin, t2);
        exor(t2, tap, t1);
        exor(din,t1,reg_in);
        dff(reg_in, reg_out);
end.

net_module on_misr_me(ctrl, sin, din : input; reg_out : output; );
var
        t1, reg_in : node;
begin
        a2(ctrl, sin, t1);
        exor(t1, din, reg_in);
        dff(reg_in, reg_out);
end.

net_module misr_embed(
        data        : array [0..] of input;
        sig         : array [data'first..data'last] of output );
const
        size = data'size;
var
        Ldata : array [data'first..data'last] of node;
        Lsig : array [sig'first..sig'last] of node;
        i           : integer;
observer
        Lsig;
faults_off;
begin
        write("misr size = ", size, "\n");
        Fan(data, Ldata);
        assert( size > 0 );
        on_misr_me(misr_on[misr_on_index], Lsig[Lsig'last],
                Ldata[0], Lsig[0]);
        for i := 1 to Ldata'last do
                if lfsr_tap(size,i) = 0 then
                        on_misr_me(misr_on[misr_on_index],
                                Lsig[i-1], Ldata[i], Lsig[i])
                else
                        on_misr_tap_me(misr_on[misr_on_index],
                         Lsig[i-1],Ldata[i],Lsig[sig'last],Lsig[i]);
        misr_on_index := misr_on_index + 1;
        Fan(Lsig, sig);
end.
```

Lib/embed.lg

```
net_module gen_misr_embed(
        obs     : array [0..] of input;
        gen     : array [obs'first..obs'last] of output );
var
        Lobs    : array [obs'first..obs'last] of node;
        sig     : array [obs'first..obs'last] of node;
observer
        sig;
faults_off;
begin
        Fan(obs, Lobs);
        gen_embed(obs, gen);
        misr(Lobs, sig);
end.

net_module finish_best(dummy:integer);
faults_off
        gen_on[1..gen_on_index-1], misr_on[1..misr_on_index-1];
begin
        dummy := 1;
end.

(*
 * cstp embedding
 *)
global
var
        cstp_on         : input;
        cstp_chain      : array [1..16] of node;
        cstp_chain_index : integer;
        cstp_init_pattern : integer;
end.

function init_cstp(init_p : integer) : integer;
begin
        cstp_chain_index := 1;
        cstp_init_pattern := init_p;
        init_cstp := 1;
end.

net_module cstp_me(init : integer; ch, d: input; q : output);
var
        d_in, t1 : node;
begin
        a2(cstp_on, ch, t1);
        exor(t1, d, d_in);
        me( init, d_in, q);
end.
```

```
net_module cstp_embed (
        obs     : array [1..] of input;
        gen     : array [obs'first..obs'last] of output );
const
        size = obs'size;
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
        i    : integer;
observer
        Lgen;
faults_off;
begin
        assert( size > 0 );
        assert( cstp_chain_index >= cstp_chain'first and
                cstp_chain_index <= cstp_chain'last);
        Fan(obs, Lobs);
        cstp_me(embed_state(cstp_init_pattern, Lobs'first),
                cstp_chain[cstp_chain_index],
                Lobs[Lobs'first], Lgen[Lobs'first]);
        cstp_chain_index := cstp_chain_index + 1;
        for i := Lobs'first+1 to Lobs'last do
                cstp_me(embed_state(cstp_init_pattern, i),
                        Lgen[i-1], Lobs[i], Lgen[i]);
        connect( Lgen[Lobs'last], cstp_chain[cstp_chain_index] );
        Fan(Lgen, gen);
end.

net_module cstp_scan_embed (
        obs     : array [1..] of input;
        gen     : array [obs'first..obs'last] of output );
const
        size = obs'size;
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
        d_in : array [obs'first..obs'last] of node;
        i    : integer;
faults_off;
begin
        assert( size > 0 );
        assert( cstp_chain_index >= cstp_chain'first and
                cstp_chain_index <= cstp_chain'last);
        Fan(obs, Lobs);
        mux2(cstp_on, Lobs[Lobs'first], cstp_chain[cstp_chain_index],
                d_in[Lobs'first]);
        me( embed_state(cstp_init_pattern, Lobs'first),
                d_in[Lobs'first], Lgen[Lobs'first]);
        cstp_chain_index := cstp_chain_index + 1;
```

```
        for i := Lobs'first+1 to Lobs'last do begin
                mux2(cstp_on, Lobs[i], Lgen[i-1], d_in[i]),
                me(embed_state(cstp_init_pattern,i),d_in[i],Lgen[i]);
        end;
        connect( Lgen[Lobs'last], cstp_chain[cstp_chain_index] );
        Fan(Lgen, gen);
end.

net_module cstp_observe( obs : array [1..] of input; );
const
        size = obs'size;
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [obs'first..obs'last] of node;
        i          : integer;
observer
        Lgen;
faults_off;
begin
        assert( size > 0 );
        assert( cstp_chain_index >= cstp_chain'first and
                cstp_chain_index <= cstp_chain'last);
        Fan(obs, Lobs);
        cstp_me(embed_state(cstp_init_pattern, Lobs'first),
                Lobs[Lobs'first], cstp_chain[cstp_chain_index],
                Lgen[Lobs'first]);
        cstp_chain_index := cstp_chain_index + 1;
        for i := Lobs'first+1 to Lobs'last do
                cstp_me(embed_state(cstp_init_pattern, i),
                        Lgen[i-1], Lobs[i], Lgen[i]);
        connect( Lgen[Lobs'last], cstp_chain[cstp_chain_index] );
end.

net_module finish_cstp( dummy : integer);
faults_off
        cstp_chain[1..cstp_chain_index],cstp_on;
begin
        connect( cstp_chain[cstp_chain_index],
                cstp_chain[cstp_chain'first]);
end.

(*
 * scan embedding
 *)
global
var
        scan_on, scan_in, scan_out      : input;
        scan_chain : array [1..16] of node;
        scan_chain_index : integer;

end.

function init_scan(dummy : integer) : integer;
begin
        scan_chain_index := 1;
        init_scan := 1;
end.

net_module scan_embed (
        obs         : array [1..] of input;
        gen         : array [obs'first..obs'last] of output );
const
        size = obs'size;
var
        Lobs : array [obs'first..obs'last] of node;
        Lgen : array [gen'first..gen'last] of node;
        i          : integer;
        d_in   : array [obs'first..obs'last] of node;
observer
        Lgen;
faults_off;
begin
        Fan(obs, Lobs);
        assert( size > 0 );
        assert( scan_chain_index >= scan_chain'first and
                scan_chain_index <= scan_chain'last);
        mux2(scan_on, Lobs[Lobs'first],
                scan_chain[scan_chain_index],d_in[Lobs'first]);
        cdff(d_in[Lobs'first], Lgen[Lobs'first]);
        scan_chain_index := scan_chain_index + 1;
        for i := Lobs'first+1 to Lobs'last do begin
                mux2(scan_on, Lobs[i], Lgen[i-1], d_in[i]);
                cdff( d_in[i],   Lgen[i]);
        end;
        connect( Lgen[Lobs'last], scan_chain[scan_chain_index] );
        Fan(Lgen, gen);
end.

net_module finish_scan( dummy : integer);
faults_off
        scan_chain[1..scan_chain_index];
begin
        connect( scan_chain[scan_chain_index], scan_out);
        connect( scan_chain[scan_chain'first], scan_in);

end.
```

Appendix D

Case Study Design Listings

Alu/alu.lg

```
(*
 * an n by m unsigned multiplier.
 *)

net_module mpy_by_one ( x : array [1..] of input;
      y : input;
      z : array [1..x'size] of output );
var
      i : integer;
begin
      for i := z'first to z'last do begin
            a2(y, x[i], z[i]);
      end;
end.

net_module mpy_add( x : array [1..] of input;
      y : array [1..x'size] of input;
      s : array [1..x'size+1] of output );
const
      x_sz = x'size;
var
      x1, coutb : node;
      carry : array [1..x'size] of node;
      i : integer;
begin
      half_adder2( x[1], y[1], s[1], carry[i]);
      for i := 2 to x_sz do begin
            adder2( x[i], y[i], carry[i-1], s[i], carry[i]);
      end;
      connect(carry[x_sz], s[x_sz+1]);
end.

net_module multiplier( x,y : array [1..] of input;
      z : array [1..x'size+y'size] of output );
const
      z_sz = z'size;
      y_sz = y'size;
      x_sz = x'size;
      pp_sz = z_sz - 1;
      pp_ind = (pp_sz - x_sz) + 1;
var
      part_prod : array [1..pp_sz] of node;
      part_sum : array [1..x_sz] of node;
      mpy_by_1 : array [1..x_sz] of node;
      mpy_by_1_a : array [1..x_sz] of node;
begin
      if y'size > 2 then begin
            mpy_by_one ( x, y[y_sz], mpy_by_1 );
            multiplier ( x, y[1..y_sz-1], part_prod);
            mpy_add( mpy_by_1, part_prod[pp_ind..pp_sz], z[pp_ind..z_sz]);
            connect( part_prod[1..pp_ind-1], z[1..pp_ind-1] );
      end
      else begin
            mpy_by_one ( x, y[1], mpy_by_1 );
            connect( mpy_by_1[1], z[1] );
            mpy_by_one ( x, y[2], mpy_by_1_a );
            mpy_add( mpy_by_1[2..x_sz], mpy_by_1_a[1..x_sz-1],part_sum);
            connect(part_sum[1..x_sz-1], z[2..x_sz]);
            half_adder2(part_sum[x_sz],mpy_by_1_a[x_sz],z[z_sz-1], z[z_sz]);
      end
end.
```

Alu/alu.lg

```
#include "../../include/stdinc.lg"
#include "../../include/utility.lg"
#include "../../include/mpy.lg"
#include "../../include/bist.lg"
#include "../../include/embed.lg"

net_module sn74181 (
        m : input;
        s : array [1..4] of input;
        a,b : array [1..4] of input;
        cin : input;
        f : array [1..4] of output;
        cout,p,g, eq : output;);
var
        I : array [1..14] of node;
        O : array [72..79] of node;
        L : array [15..71] of node;
begin
        connect(s[1],I[4]);
        connect(s[2],I[3]);
        connect(s[3],I[2]);
        connect(s[4],I[1]);

        i1(b[4], I[5]);
        i1(b[3], I[7]);
        i1(b[2], I[9]);
        i1(b[1], I[11]);

        i1(a[4], I[6]);
        i1(a[3], I[8]);
        i1(a[2], I[10]);
        i1(a[1], I[12]);

        connect(m, I[13]);
        connect(cin, I[14]);
        i1(O[75],f[4]);
        i1(O[74],f[3]);
        i1(O[73],f[2]);
        i1(O[72],f[1]);
        connect(O[79],eq);
        i1(O[78], p);
        i1(O[76], g);
        connect(O[77], cout);
        (*-------------------*)
        i1 (I[5], L[15]);
        a3 (I[5], I[1], I[6], L[16]);
        a3 (I[6], I[2], L[15], L[17]);
        oi2 (L[16], L[17], L[18]);
        a2 (L[15], I[3], L[19]);
        a2 (I[4], I[5], L[20]);
        buf2 (I[6], L[21]);
        oi3 (L[19], L[20], L[21], L[22]);
        i1 (I[7], L[23]);
        a3 (I[7], I[1], I[8], L[24]);
        a3 (I[8], I[2], L[23], L[25]);
        oi2 (L[24], L[25], L[26]);
        a2 (L[23], I[3], L[27]);
        a2 (I[4], I[7], L[28]);
        buf2 (I[8], L[29]);
        oi3 (L[27], L[28], L[29], L[30]);
        i1 (I[9], L[31]);
        a3 (I[9], I[1], I[10], L[32]);
        a3 (I[10], I[2], L[31], L[33]);
        oi2 (L[32], L[33], L[34]);
        a2 (L[31], I[3], L[35]);
        a2 (I[4], I[9], L[36]);
        buf2 (I[10], L[37]);
        oi3 (L[35], L[36], L[37], L[38]);
        i1 (I[11], L[39]);
        a3 (I[11], I[1], I[12], L[40]);
        a3 (I[12], I[2], L[39], L[41]);
        oi2 (L[40], L[41], L[42]);
        a2 (L[39], I[3], L[43]);
        a2 (I[4], I[11], L[44]);
        buf2 (I[12], L[45]);
        oi3 (L[43], L[44], L[45], L[46]);
        i1 (I[13], L[47]);
        ai2 (I[14], L[47], L[48]);
        a3 (I[14], L[42], L[47], L[49]);
        a2 (L[46], L[47], L[50]);
        oi2 (L[49], L[50], L[51]);
        exor (L[42], L[46], L[52]);
        exor (L[52], L[48], O[72]);
        exor (L[34], L[38], L[53]);
        exor (L[53], L[51], O[73]);
        a4 (I[14], L[42], L[34], L[47], L[54]);
        a3 (L[34], L[46], L[47], L[55]);
        a2 (L[38], L[47], L[56]);
        oi3 (L[54], L[55], L[56], L[57]);
        a5 (I[14], L[42], L[34], L[26], L[47], L[58]);
        a4 (L[34], L[26], L[46], L[47], L[59]);
        a3 (L[26], L[38], L[47], L[60]);
        a2 (L[30], L[47], L[61]);
        oi4 (L[58], L[59], L[60], L[61], L[62]);
        exor (L[26], L[30], L[63]);
        exor (L[63], L[57], O[74]);
        exo  (L[18], L[22], L[64]);
        exor (L[64], L[62], O[75]);
```

```
      buf2  (L[22],  L[65]);
      a2  (L[18],  L[30],  L[66]);
      a3  (L[18],  L[26],  L[38],  L[67]),
      a4  (L[18],  L[26],  L[34],  L[46],  L[68]);
      oi4  (L[65],  L[66],  L[67],  L[68],  O[76]);
      ai5  (L[18],  L[26],  L[34],  L[42],  I[14],  L[69]);
      i1  (O[76],  L[70]);
      i1  (L[69],  L[71]);
      o2  (L[70],  L[71],  O[77]);
      ai4  (L[18],  L[26],  L[34],  L[42],  O[78]);
      a4  (O[75],  O[74],  O[73],  O[72],  O[79])
end.

net_module sn74182(
      cin  : input;
      g  : array [0..2] of input;
      p  : array [0..2] of input;
      co   : array [1..3] of output );
var
      co_b: array [1..2] of node;
      t1_b,  t2,  t3  : node;
begin
      oai22(g[0],p[0],  g[0],  cin,  co_b[1]);
      i1(co_b[1],  co[1]);
      oai332(g[1],g[0],cin,  g[1],g[0],p[0],  g[1],p[1],  co_b[2]);
      i1(co_b[2],  co[2]);
      oai32(g[2],g[1],p[1],  g[2],p[2],  t1_b);
      oi4(g[2],g[1],g[0],p[0],  t2);
      oi4(g[2],g[1],g[0],cin,  t3);
      oi3(t1_b  ,t2,t3,  co[3]); // really a3
end.


net_module alu16(
      m  : input;
      s  : array [1..4] of input;
      a,b  : array [1..16] of input;
      cin  : input;
      f  : array [1..16] of output;
      cout,  eq  : output );
var
      e  : array [0..3] of node;
      co  : array [3..3] of node;
      g,  p  : array [0..2] of node;
      c  : array [1..3] of node;
      t_cin  : node;
begin
      sn74182(cin,  g[0..2],  p[0..2],  c);
      sn74181_nc(m,s,a[1..4],b[1..4],cin,f[1..4],p[0],g[0],e[0]);
      sn74181_nc(m,s,a[5..8],b[5..8],c[1],f[5..8],p[1],g[1],e[1]);
      sn74181_nc(m,s,a[9..12],b[9..12],c[2],f[9..12],p[2],g[2],e[2]);
      sn74181_npg(m,s,a[13..16],b[13..16],c[3],f[13..16],co[3],e[3]);
      connect(co[3],  cout);
      a4(e[0],  e[1],  e[2],  e[3],  eq);
end.

circuit alu;
var
      // the register inputs and outputs
      m  : input;  r_m  : node;
      s  : array [1..4] of input;  r_s  : array [1..4] of node;
      x,y  : array [1..8] of input;  r_x,r_y  : array [1..8] of node;
      cin  : input;  r_cin  : node;
      xy,  r_xy  : array [1..16] of node;
      f  : array [1..16] of node;  r_f  : array [1..16] of output;
      cout,eq  : node;  r_cout,r_eq  : output;
display
      r_s,  r_m,  r_cin,  x:dec,  y:dec,  r_xy:dec,r_f:dec,  f:dec,  r_cout,  r_eq;
begin
      use Dff_array in begin
            n_reg(x,r_x);
            n_reg(y,r_y);
            n_reg([m,s,cin],  [r_m,r_s,r_cin]);
      end;
      use stumps_embed in begin
            n_reg(xy,r_xy);
      end;
      use stumps_embed in begin
            n_reg(f,r_f);
            n_reg([cout,eq],  [r_cout,r_eq]);
      end;

      multiplier(r_x,r_y,  xy);
      alu16(r_m,r_s,r_xy,r_f,r_cin,  f,cout,  eq);
end.
```

Cordic/cordic.lg

```
#include "../../include/stdinc.lg"
#include "../../include/utility.lg"
#include "../../include/bist.lg"
#include "../../include/embed.lg"

(* ———————————— counter ———————————— *)
net_module counter_cell ( enable, clear_bar : input; data, propagate : output);
var
    data_in, nxt : node;
    cnt_me       : reg;
faults_off
    data_in, propagate, data;
begin
    a2(data, enable, propagate);
    exor(data, enable, nxt);
    a2( nxt, clear_bar, data_in);
    cnt_me <- data_in;
    connect(cnt_me, data);
end.

net_module counter( enable, clear : input; count : array [] of output );
var
    ena          : array [count'first .. count'last-1  of node;
    nxt          : node;
    cnt_me       : reg;
    data_in      : node;
    clear_bar    : node;
    i            : integer;
begin
    i1(clear, clear_bar);
    counter_cell( enable, clear_bar, count[count'first], ena[count'first]);
    for i := count'first+1 to count'last-1 do begin
        counter_cell(ena[i-1], clear_bar, count[i], ena[i]);
    end;
    exor(ena[count'last-1],count[count'last], nxt);
    a2(nxt, clear_bar, data_in);
    cnt_me <- data_in;
    connect(cnt_me, count[count'last] );
end.

(* ———————————— shifter ———————————— *)
net_module sign_extend(
    shft    : array [] of input;
    sign    : input;
    data    : array [] of input;
    extend  : array [data'first..data'last] of output );
var
    propagate: array [shft'first..shft'last] of node;
    sign_ext : array [shft'first..shft'last] of node;
    i,j      : integer;
begin
    for i := shft'first to shft'last do
        Or( shft[i..shft'last], propagate[i] );
    i := data'last;
    j := propagate'first;
    while j <= propagate'last do begin
        a2( sign, propagate[j], sign_ext[j]);
        o2( data[i], sign_ext[j], extend[i] );
        i := i - 1;
        j := j + 1;
    end;
    if i >= data'first then
        connect( data[data'first..i], extend[data'first..i] );
end.

net_module sign_extention( left_sh, right_sh : integer;
    sh_ctl : array [(0-left_sh)..] of input;
    sign   : input;
    data   : array [] of input;
    sign_ext: array [data'first..data'last+1] of output );
var
    sign_ena        : node;
begin
    connect( sign, sign_ext[sign_ext'last] );
    if right_sh > 0 then
        sign_extend( sh_ctl[1..right_sh], sign, data,
            sign_ext[data'first..data'last] )
    else
        connect( data, sign_ext[data'first..data'last] );
end.

net_module shifter_bit_position(pos, left_sh, right_sh : integer;
    data    : array [0..] of input;
    sh_ctl  : array [(0-left_sh)..] of input;
    sh_data: output );
var
    and_data : array [data'first..data'last] of node;
    data_lb  : integer;
    data_ub  : integer;
    shift_lb : integer;
    shift_ub : integer;
begin
    if (pos-left_sh) > data'first then begin
        data_lb := pos-left_sh;
        shift_lb := sh_ctl'first;
    end
    else begin
        data_lb := data'first;
```

```
        shift_lb := sh_ctl'first - (data'first+(pos-left_sh));
    end;

    if (pos+right_sh) < data'last then begin
        data_ub := pos+right_sh;
        shift_ub := sh_ctl'last;
    end
    else begin
        data_ub := data'last;
        shift_ub := sh_ctl'last - ((right_sh+pos)-data'last);
    end;

    bitwise_and(data[data_lb..data_ub],sh_ctl[shift_lb..shift_ub],
          and_data[data_lb..data_ub]);
    Or( and_data[data_lb..data_ub], sh_data );
end.

(*
 * arithmetic shifter - performs left and right shifts
 * left_sh - maximum left shift
 * right_sh - maximum right shift
 * d - twos complement d'size-bit number, sign bit is d[d'last]
 * sh_ctl - shift control
 * sh_d - shifted result
 *)
net_module arithmetic_shifter( left_sh, right_sh : integer ;
    d     : array [0..] of input;
    sh_ctl     : array [(0-left_sh)..right_sh] of input;
    sh_d     : array [0..] of output   );
var
    i     : integer;
    sign_bit: node;
    shift     : array [sh_d'first..sh_d'last-1] of node ;
begin
    connect( d[d'last], sign_bit );
    (*
     * shift the data bits, the sign bit is not shifted.
     *)
    for i := 0 to sh_d'last-1 do begin
        shifter_bit_position(i, left_sh, right_sh,
            d[0..d'last-1], sh_ctl, shift[i]);
    end;
    sign_extention( left_sh, right_sh, sh_ctl, sign_bit, shift, sh_d);
end.

(* ---------- Cordic Data Path ---------- *)
net_module cor_datapath_controller(
    rv_mode : input;
    sign_y : input;
    sign_z : input;
    x_mode : output;
    y_mode : output;
    z_mode : output );
var
    rv_bar          : node;
    sign_z_bar      : node;
    xz_mode         : node;
begin
    i1( rv_mode, rv_bar );
    i1( sign_z, sign_z_bar);
    aoi22(sign_z_bar, rv_bar, sign_y, rv_mode, y_mode);
    i1( y_mode, xz_mode );
    connect( xz_mode, x_mode );
    connect( xz_mode, z_mode );
end.

(*
 * state s0: x_hd = y_hd = z_hd = 1; if start then state = s1;
 * state s1: x_ld = y_ld = z_ld = 1; state = s2;
 * state s2: enable = 1; if finished then state = s0;
 *
 * s0 = 00, s1 = 01, s2 = 11 : encodings
 *
 * next state equations:
 * S0+ = S1*finished'
 * S1+ = S0'*start + S1*finished'
 *)

function f_and2(x,y:input) : output;
begin
    a2(x,y, f_and2);
end.

net_module cor_seq_controller(
    start, finished          : input;
    load, hold, enable       : output );
var
    state              : array [0..1] of reg;
    nxt_state          : array [0..1] of node;
    S0_, finished_     : node;
display
    state, load, hold, enable;
faults_off
    state[1];
begin
    i1( state[0], S0_);
    i1( finished, finished_);
    a2( state[1], finished_, nxt_state[0] );
```

```
    o2(f_and2(S0_,start), nxt_state[0], nxt_state[1]);
    state[0] <- nxt_state[0];
    state[1] <- nxt_state[1];
    oi2( state[0], state[1], hold );
    a2( state[0], state[1], enable );
    a2( S0_, state[1], load);
end.

net_module cordic_controller (
    start, rv_mode, lc_mode     : input;
    y_sign, z_sign              : input;
    shift_sel                   : array [] of output;
    x_md, x_hd, x_ld            : output;
    y_md, y_hd, y_ld            : output;
    z_md, z_hd, z_ld            : output );
const
    counter_size = log2(shift_sel'size);
var
    count                       : array [1..counter_size] of node;
    enable_finished             : node;
    hold                        : node;
    load, load_                 : node;
begin
    cor_seq_controller( start, finished, load, hold, enable );
    i1( load, load_);
    o2( hold, f_and2(load_lc_mode), x_hd );
    connect( hold, y_hd );
    connect( hold, z_hd );
    connect( load, x_ld );
    connect( load, y_ld );
    connect( load, z_ld );

    // use the load line to also clear the counter
    counter( enable, hold, count );
    decoder( count, shift_sel);
    connect( shift_sel[shift_sel'last], finished );

    cor_datapath_controller( rv_mode, y_sign, z_sign, x_md, y_md, z_md );
end.

net_module cordic_circular_table_bit_position (
    cor_int_size    : integer;
    cor_size        : integer;
    index           : integer;
    shift_ctl       : array [] of input;
    cordic_bit      : output );
const
    atan_first = -2;    // entry contains atan(2^(-2)
    atan_last = 29;
    table_int_size = 1;
    table_frac_size = 29;
var
    (*
     * the atan table is stored in an integer array
     * in a fixed point format.
     *)
    cor_table       : array [atan_first..atan_last] of integer;
    sel_bit         : integer;
    sel_mask        : integer;
    or_sel_bits     : integer;
    tab_index, i    : integer;
begin
    (* start of table definition *)
    cor_table[-2] := 1423585876; cor_table[-1] := 1188791883;
    cor_table[0] := 843314856; cor_table[1] := 497837829;
    cor_table[2] := 263043836; cor_table[3] := 133525158;
    cor_table[4] := 67021686; cor_table[5] := 33543515;
    cor_table[6] := 16775850; cor_table[7] := 8388437;
    cor_table[8] := 4194282; cor_table[9] := 2097149;
    cor_table[10] := 1048575; cor_table[11] := 524287;
    cor_table[12] := 262143; cor_table[13] := 131071;
    cor_table[14] := 65535; cor_table[15] := 32767;
    cor_table[16] := 16383; cor_table[17] := 8191;
    cor_table[18] := 4095; cor_table[19] := 2047;
    cor_table[20] := 1023; cor_table[21] := 511;
    cor_table[22] := 255; cor_table[23] := 127;
    cor_table[24] := 63; cor_table[25] := 31;
    cor_table[26] := 15; cor_table[27] := 7;
    cor_table[28] := 3; cor_table[29] := 1;
    (* end of table definition *)

    assert( table_int_size < cor_int_size );
    // assumes index is 0 based, and ints are 32 bits
    sel_bit := 31-( (table_int_size-cor_int_size) + (cor_size-index));
    sel_mask := pow(2,sel_bit);

    or_sel_bits := 0;
    for i := shift_ctl'first to shift_ctl'last do begin
        tab_index := (i-shift_ctl'first) - table_int_size;
        if ((cor_table[tab_index] / sel_mask) mod 2) = 1 then begin
            or_sel_bits := or_sel_bits + pow(2,(i-shift_ctl'first));
        end;
    end;
    sel_max_term(or_sel_bits, shift_ctl, cordic_bit)
end.

net_module cordic_linear_table_bit_position (
    cor_size        : integer;
```

```
    index          : integer;
    shift_ctl      : array [] of input;
    cordic_bit     : outpnt );
var
    ctl_index      : integer;
begin
    // again zero based index assumption
    ctl_index := shift_ctl'first + ((cor_size-1)-index);
    connect( shift_ctl[ctl_index], cordic_bit );
end.

net_module cordic_table (
    bits_before_radix : integer;
    lc_md             : input;
    shift_ctl         : array [] of input;
    z_table           : array [] of output );
var
    lin     :    array [z_table'first..z_table'last-1] of node;
    cir     :    array [z_table'first..z_table'last-1] of node;
    z_table_bar:    array [z_table'first..z_table'last-1] of node;
    zero    :    node;
    a, a_bar:    node;
    i       :    integer;
begin
    connect( shift_ctl[shift_ctl'first], a);
    i1(a, a_bar);
    a2(a, a_bar, zero);
    connect(zero, z_table[z_table'last] );

    for i := z_table'first to z_table'last-1 do
        cordic_circular_table_bit_position(bits_before_radix,
            z_table'size-1, i, shift_ctl, cir[i] );

    for i := z_table'first to z_table'last-1 do
        cordic_linear_table_bit_position( z_table'size-1,
            i, shift_ctl, lin[i] );

    n_mux2( lc_md, cir, lin, z_table[z_table'first..z_table'last-1] );
end.


net_module cordic(
    start, rv_mode, lc_mode    : input;
    x_load                     : array [] of input;
    y_load, z_load             : array [x_load'first..x_load'last] of input;
    x_acc, y_acc, z_acc        : array [x_load'first..x_load'last] of output);
const
    left_sh  = 1;
    right_sh = x_load'size-1;
var
    shift_sel          : array [ -left_sh..right_sh ] of node;
    x_sh, y_sh, z_table    : array [x_load'first..x_load'last] of node;
    x_md, x_hd, x_ld       : node ;
    y_md, y_hd, y_ld       : node ;
    z_md, z_hd, z_ld       : node ;
    x_r, y_r, z_r          : array [x_load'first..x_load'last] of reg;
    st_r, rv_r, lc_r       : reg;
begin
    use Dff_array in begin
        cordic_controller( st_r, rv_r, lc_r,
            y_acc[y_acc'last], z_acc[z_acc'last], shift_sel,
            x_md, x_hd, x_ld, y_md, y_hd, y_ld, z_md, z_hd, z_ld );

        accumulator( x_md, x_hd, x_ld, x_r, x_sh, x_acc);
        arithmetic_shifter( left_sh, right_sh, y_acc, shift_sel, x_sh);

        accumulator( y_md, y_hd, y_ld, y_r, y_sh, y_acc);
        arithmetic_shifter( left_sh, right_sh, x_acc, shift_sel, y_sh);

        accumulator( z_md, z_hd, z_ld, z_r, z_table, z_acc);
        // the table produces a sign bit which is always zero.
        cordic_table (left_sh, lc_r, shift_sel, z_table );
    end;
    use Dff_array in begin
        st_r <- start; rv_r <- rv_mode; lc_r <- lc_mode;
        x_r <- x_load; y_r <- y_load; z_r <- z_load;
    end;
end.

circuit cordic_main;
const
    word_size = 16;
var
    st, rv, lc : input;
    x, y, z                : array [1..word_size] of input;
    x_acc, y_acc, z_acc    : array [1..word_size] of output;
begin
    cordic( st, rv, lc, x, y, z, x_acc, y_acc, z_acc);
end.
```

Gcd/gcd.lg

```
#include "/home/rbyrne/Bist/include/stdinc.lg"
#include "/home/rbyrne/Bist/include/utility.lg"
#include "/home/rbyrne/Bist/include/dff.lg"
#include "/home/rbyrne/Bist/include/bist.lg"
#include "/home/rbyrne/Bist/include/embed.lg"

global
const
        bisted_nors = 1;
end.

net_module half_sub( x,y : input; d,b : output);
var
        x_ : node;
begin
        exor(x,y, d);
        i1(x, x_);
        a2(x_, y, b);
end.

net_module full_sub( x,y, b : input, d, bout : output );
var
        e1, x_, bout_ : node;
begin
        exor(x, y, e1);
        exor(e1, b, d);
        i1(x, x_);
        aoi222(x_y, x_b, y,b, bout_);
        i1(bout_, bout);
end.

net_module subtractor (
        x : array [] of input;
        y : array [x'first..x'last] of input;
        d : array [x'first..x'last] of output;
        b : output );
var
        i : integer;
        bb : array [x'first..x'last] of node.
begin
        half_sub( x[x'first], y[x'first], d[x'first], bb[x'first]);
        for i := x'first+1 to x'last do
                full_sub(x[i], y[i], bb[i-1], d[i], bb[i]).
        connect(bb[x'last], b);
end.

net_module mux3 ( y, z, s0, s1, s2 : input; o : output);
var
        o_ : node;
begin
        aoi222(x,s0, y,s1, z,s2, o_);
        i1(o_, o);
end.

net_module mux3_array( x : array [] of input;
        y, z : array [x'first..x'last] of input;
        s0, s1, s2 : input;
        o : array [x'first..x'last] of output);
var
        i : integer;
begin
        for i := x'first to x'last do
                mux3( x[i], y[i], z[i], s0, s1, s2, o[i] );
end.

(*
 * st+ = z'*st + s*st'
 * notbusy = st'
 * xs = lt' * st
 * xl = yl = s * st'
 * xw = lt * st
 * yw = (lt + y_eq_0) * st
 * yh = s'*st' + lt *st*y_eq_0'
 *)
net_module gcd_controller( s, z, y_eq_0, lt : input;
        xs, xl, xw, yh, yl, yw, notbusy : output );
var
        st_nxt : node;
        st     : reg;
        s_, z_, st_, lt_: node;
        t1,t2,t3       : node;
        yw_, y_eq_0_ : node;
faults_off
        y_eq_0_;
begin
        i1(s, s_);
        i1(z, z_);
        i1( y_eq_0, y_eq_0_);
        i1(st, st_); connect(st_ notbusy);
        i1(lt, lt_);
        a2( s, st_, xl); connect( xl, yl);
        a2( lt_, st, xs);
        a2( lt, st, xw);
        oai21(lt, y_eq_0, st, yw_); i1(yw_ yw);
        a2( z_, st, t1);
        o2( t1, xl, st_nxt);
        a2( s_, st_ t2);
        a2( xs, y_eq_0_ t3);
```

Bill/bill.lg

```
        o2( t2,  t3, yh);
        st <- st_nxt;
end.

net_module gcd(
        start : input;
        x : array [] of input;
        y : array [x'first..x'last] of input;
        xy_gcd : array [x'first..x'last] of output;
        notbusy : output; );
var
        x_r : array [x'first..x'last] of reg;
        y_r : array [x'first..x'last] of reg;
        x_r_in : array [x'first..x'last] of node;
        y_r_in : array [x'first..x'last] of node;
        xy_sub : array [x'first..x'last] of node;
        xs, xl, xw : node; // s-Subtractor, l-Load, w-sWap
        yh, yl, yw : node; // h-Hold, l-Load, w-sWap
        zero0, zero1, zero, x_lt_y : node;
        i : integer;
display
        zero0, zero1, x_r:dec, y_r:dec;
begin
        mux3_array(xy_sub, x, y_r, xs, xl, xw,  x_r_in);
        mux3_array(y_r, y, x_r,     yh, yl, yw,  y_r_in);
        use dff_array in
                for i := x'first to x'las  do x_r[i] <- x_r_in[i];
        use dff_array in
                for i := x'first to x'last do y_r[i] <- y_r_in[i];
        subtractor( x_r, y_r, xy_sub, x_lt_y);
        if bisted_nors = 1 then begin
                Bisted_Nor( xy_sub, zero0);
                Bisted_Nor( y_r, zero1);
        end
        else begin
                Nor( xy_sub, zero0);
                Nor( y_r, zero1);
        end;
        o2(zero0, zero1, zero);
        use Dff_array in
                gcd_controller(start, zero, zero1, x_lt_y,
                        xs, xl, xw, yh, yl, yw, notbusy);
        connect( y_r, xy_gcd);
end.
```

```
(*****************************************************
   Developed by William Gardner
   String Searching Circuit - MMT (n-by-m mesh tree) algorithm
 *****************************************************
CODING CONVENTIONS:                          SAMPLE
- module/circuit I/O variable                Width Start with cap*
- local variables                            store All small letters*
* for all variables, suffix "_" indicates a real net or register;
   without suffix indicates a compile-time parameter;
- constants and module names                 SIZE All caps
 ***************************************************)

#include "../include/stdinc.lg"
#include "../include/dff.lg"

global
const
        ROWS=8; // max pattern bit length ("m")
        COLS=16;// max text bit length ("n"); m<=n

        VROWS=8;// ROWS rounded up to power of 2
        VCOLS=16;// ditto for COLS

        TRUE=1;
        FALSE=0;
var
        // internode connections for leaf nodes & trees
        leaf_row_init_, leaf_row_data_,
        leaf_col_init_, leaf_col_data_,
        leaf_result_ : array[1..VROWS*VCOLS] of node;
end.

(*********************
 leaf_index converts a pair of row/column numbers to an index
 into the various leaf_... arrays
*********************)
function leaf_index( Row_no, Col_no : integer ) : integer;
begin
        leaf_index := (Row_no-1)*VCOLS + Col_no;
end.

(*********************
 overlap returns TRUE if the ranges L1..R1 (left/right)
 and L2..R2 have any numbers in common
*********************)
function overlap( L1, R1, L2, R2 : integer ) : integer;
begin
        if R1<L2 or L1>R2 then overlap := FALSE
        else overlap := TRUE;
```

```
end.


(***************** ***
  Leaf comparator node
**********************
INPUTS    Row_init_ 0=comparison->FF; 1=init FF to 1
          Col_init_ same
          Row_data_ row (pattern) bit to be compared
          Col_data_ column (text) bit to be compared

OUTPUT    Result_      0=no match; 1=every bit matched

CONNECTIONS
     (1) Row_<init, data> come from row parents,
     Col_<init, data> from column parents.
     (2)  Result goes to diagonal parent.
**********************)
net_module  LEAF(Row_init_,Col_init_,Row_data_,Col_data_:input;
                 Result_ : output );
var
          equiv_, match_, store_it_ : node;
          store_ : reg;
begin
          // compare row/col data
          exnor( Row_data_, Col_data_, equiv_ );
          // AND with result so
          // far to give statusof match
          a2( equiv_, store_, match_ );
          o3( match_, Row_init_, Col_init_, store_it_ );
          // if either init line set
          // => store 1output register
          store_ <- store_it_;
          // node output to diagonal parent
          connect( store_, Result_ );
end.


(**********************
  Recursive tree builders - common inputs
**********************
INPUTS    L_virt      bottom level virtual node numbers: left
          R_virt       right (left<right)
          L_real      actual node numbers to create: left
          R_real       right (left<right)

NOTES
     (1) L_virt and R_virt determine the depth of the tree to be
     built. For example, L=1 and R=8 would create a 4-level
     binary tree, with leaves numbered from 1 on the left to 8 on
     the right. Normally, L_real and R_real will be the same as
     L_virt and R_virt respectively.
     (2) In the case of a tree that is not fully populated, L_real
     and R_real may specify a smaller range, e.g., 6 and 8 for the
     example above. In this case, all the levels are generated to
     satisfy the structure of virtual tree, but actual hardware
     connections are only made to the real nodes, with single
     branches (vs. binary branches) stretching down to them.
     (3) As recursive calls are generated, L_virt and R_virt
     shrink with each level down to the same number, while L_real
     and R_real are passed intact as the overriding parameters.
**********************)

(**********************
  Row tree (recursive)
**********************
INPUTS    Init_       init signal to be passed down tree
          Data_       pattern bit to be passed down tree
          Row_no      no. of row for this tree [1..ROWS]
          L_virt      column nos. (see common inputs)
          R_virt
          L_real
          R_real

GLOBALS   leaf_row_init_      array of leaf inputs to connect to
          leaf_row_data_      ditto

NOTES
     (1) This module generates a binary tree to connect to the
     designated leaf comparator nodes. (2) If L_virt = R_virt,
     the connection takes place. Otherwise the tree splits and
the module is called recursively to generate the left half and
right half of the connections respectively. Flip-flops are
created at this level to buffer the init/data signals
     (3) The leaves will be numbered by columns from L_real to
     R_real. This column no. and Row_no for the tree are used to
index the correct leaf node.
**********************)
net_module  ROW_TREE( Init_ Data_ : input;
          Row_no, L_virt, R_virt, L_real, R_real : integer );
var
          store_init_ store_data_ : reg;
          new_R : integer;
begin
          // finally down at leaf level
          if L_virt = R_virt then begin
               connect( Init_
                 leaf_row_init_[leaf_index(Row_no,L_virt)]);
```

```
            connect( Data_,
              leaf_row_data_[leaf_index( Row_no,L_virt)]);
        end
        // generate subtree: divide leaf numbers in half
        else begin
            new_R := L_virt+(R_virt-L_virt)/2;

            if overlap(L_virt,new_R,L_real,R_real) = TRUE then
                ROW_TREE( store_init_, store_data_,
                  Row_no, L_virt, new_R, L_real, R_real );

            if overlap(new_R+1,R_virt,L_real,R_real ) = TRUE then
                ROW_TREE( store_init_, store_data_,
                  Row_no, new_R+1, R_virt, L_real, R_real );

            // create output registers at this level
            store_init_ <- Init_;
            store_data_ <- Data_;
        end
end.


(**********************
  Column tree (recursive)
**********************
INPUTS    Init_       init signal to be passed down tree
          Data_       text bit to be passed down tree
          Col_no      no. of column for this tree [1..COLS]
          L_virt      row nos. (see common inputs)
          R_virt
          L_real
          R_real

GLOBALS   leaf_col_init_      array of leaf inputs to connect to
          leaf_col_data_      ditto

NOTES
      (1) This module generates a binary tree to connect to the
      designated leaf comparator nodes. (2) If L_virt = R_virt,
      the connection takes place. Otherwise the tree splits and
the module is called recursively to generate the left half and
right half of the connections respectively. Flip-flops are
created at this level to buffer the init/data signals
      (3) The leaves will be numbered by rows from L_real to
      R_real. This row no. and Col_no for the tree are used to
      index the correct leaf node.
**********************)
net_module COL_TREE( Init_, Data_ : input;
          Col_no, L_virt, R_virt, L_real, R_real : integer );
var
        store_init_, store_data_ : reg;
        new_R : integer;
begin
        // finally down at leaf level
        if L_virt = R_virt then begin
            connect( Init_,
                leaf_col_init_[ leaf_index( L_virt, Col_no ) ] );
            connect( Data_,
                leaf_col_data_[ leaf_index( R_virt, Col_no ) ] );
        end
        // generate subtree: divide leaf numbers in half
        else begin
            new_R := L_virt+(R_virt-L_virt)/2;

            if overlap(L_virt,new_R,L_real,R_real)=TRUE then
                COL_TREE( store_init_, store_data_,
                          Col_no, L_virt, new_R, L_real, R_real );

            if overlap(new_R+1,R_virt,L_real,R_real)=TRUE then
                COL_TREE( store_init_, store_data_,
                          Col_no, new_R+1, R_virt, L_real, R_real );
        // create output registers at this level
            store_init_ <- Init_;
            store_data_ <- Data_;
        end
end.

(**********************
  Diagonal tree (recursive)
**********************
INPUTS    Col_no      # of column where row-1 starts [1..DIAGS]
          L_virt      column nos. (see common inputs)
          R_virt
          L_real
          R_real

OUTPUT    Result_     result bit to be passed up tree

GLOBAL    leaf_result_          array of leaf outputs to connect to

NOTES
      (1) This module generates a binary tree to connect to the
      designated leaf comparator nodes.
      (2) If L_virt = R_virt,
      the connection takes place. Otherwise the tree splits and
the module is called recursively to generate the left half and
right half of the connections respectively. Flip-flops are
created at upper levels to buffer the AND'ed results from the
```

```
subtree.
     (3) The leaves will be numbered by column nos from L_real to
     R_real. This column no. and Row_no for the tree are used to
index the correct leaf node.
**********)
net_module DIAG_TREE( Col_no, L_virt,
          R_virt, L_real, R_real : integer;
          Result_ : output );
var
          result_L_, result_R_ : node; // results from subtree
          store_it_ : node;
          store_ : reg;
          new_R : integer;
begin
          if L_virt = R_virt then          // finally down at leaf level
               connect( Result_,
                    leaf_result_[ leaf_index(L_virt, Col_no+L_virt-1) ] )
          // generate subtree: divide leaf numbers in half
          else begin
               new_R := L_virt+(R_virt-L_virt)/2;

               DIAG_TREE(Col_no,L_virt,new_R,L_real,R_real,result_L_);

               if overlap(new_R+1,R_virt,L_real,R_real)=TRUE then begin
                    DIAG_TREE(Col_no,new_R+1,R_virt,L_real,R_real,result_R_);

                    // AND child results together
                    a2( result_L_, result_R_, store_it_ );
                    store_ <- store_it_              // output register
                    end

               else store_ <- result_L_;

          connect( Result_, store_ );// node output to parent
          end;

end.


(**********************************************************************
          String Searching Circuit - top level description
**********************************************************************)
circuit string_mesh;

var       // primary I/O
          P_data_, P_init_ : array[1..ROWS] of input; // pattern
          T_data_, T_init_ : array[1..COLS] of input; // text

          // match output for diag tree
          Match_ : array[1..COLS] of output;

          // compile-time variables
          i, r, c : integer;

display
          P_init_, P_data_, T_init_, T_data_, Match_;
observer
          Match_;

begin
     use Dff_array in begin

          // 1st create the m-by-n array of leaf comparator nodes
          // but suppress the unused lower-left triangle
          for r := 1 to ROWS do
               for c := r to COLS do begin

                    i := leaf_index( r, c );
                    LEAF( leaf_row_init_[i], leaf_col_init_[i],
                      leaf_row_data_[i], leaf_col_data_[i],
                      leaf_result_[i] );
               end;

          // next create the 3 sets of binary trees: row trees first
          for i := 1 to ROWS do
                    ROW_TREE(P_init_[i],P_data_[i],i,1,VCOLS,i,COLS);

          // now column trees
          for i := 1 to COLS do
                    COL_TREE( T_init_[i], T_data_[i],
                      i, 1, VROWS, 1, min(i, ROWS) );

          // finally diagonal trees
          for i := 1 to COLS do
                    DIAG_TREE( i, 1, VROWS, 1,
                      COLS-max(i, COLS-ROWS+1)+1, Match_[i] );

     end;

end.
```

Blow-up version of Figure 5.2
page 121 of A High-level Language
and CAD environment for BIST Embedding
by Rodrigue Byrne