An Emulation-Based Methodology for Integrating Design, Testing and Diagnosis of Application-Specific Integrated Circuits

by

Guru Sivaraman

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

April 22, 1994

Author .... [illegible]

Department of Electrical Engineering and Computer Science

April 22, 1994

Certified by .... [illegible]

Professor Gerald J. Sussman

Department of Electrical Engineering and Computer Science

Thesis Supervisor

Certified by .... [illegible]

Dr. Burnell G. West

Schlumberger Limited

Company Supervisor

Accepted by .... [illegible]

F.R. Morgenthaler

Chairman, Department Committee on Graduate Students

Department of Electrical Engineering and Computer Science

An Emulation-Based Methodology for Integrating Design, Testing and Diagnosis of Application-Specific Integrated Circuits

by

Guru Sivaraman

Submitted to the Department of Electrical Engineering
and Computer Science on April 22, 1994, in partial
fulfillment of the requirements for the degrees of
Bachelor of Science and Master of Science in Computer Science

Abstract

The current Application-Specific Integrated Circuit (ASIC) development cycle lacks completeness and efficiency because it isolates the tasks of design, test and debug, and because of the constraints imposed by the performance bottleneck of software simulation. A new emulation-based methodology is proposed which integrates design and test, making design more efficient and making test generation and fault grading more rigorous. It also introduces design understanding into the production test domain, presenting, for the first time, the opportunity for production line diagnosis. The goals of this thesis were to evaluate the feasibility of the proposed methodology and to identify the issues which need to be resolved before it can fully be implemented.

A sample device was chosen as a test case for the model. Each stage of the proposed process was implemented using this device. Due to the prohibitive cost of using a proper commercial emulation system, the evaluation was performed using a simulation model which we believe captures the nature of emulation sufficiently to provide detailed and valuable analysis. As a result of the evaluation process, performance estimates were made for the proposed technique which suggested a dramatic advantage over current techniques. However, several key concerns were raised about the general applicability of the proposed model to ASIC design tasks. The conclusion of our evaluation was that elements of the proposed methodology offer substantial promise for improving the rigour and efficiency of current design processes. Understanding the limits of their applicability is required to implement them effectively. If the problematic issues we have raised are properly addressed, and this perspective is maintained, then these techniques can be used to substantially enhance the design cycle.

Thesis Supervisor: Professor Gerald J. Sussman
Title: Matsushita Professor of Electrical Engineering

Thesis Supervisor: Dr. Burnell G. West
Title: Technologist, Schlumberger Limited

Acknowledgements

This research project gave me the opportunity to explore an idea which was originally conceived by Burnie West at Schlumberger Limited. His guidance and support have been invaluable to me in the course of my research. I have learned a great deal from Burnie, both as a mentor and as a friend. He has infused me with a sense of the boundless possibilities of research. This project could not have been completed without him.

I have had the privilege and pleasure to work with Gerry Sussman in the Artificial Intelligence Laboratory. Gerry has broadened and refined my perspective on research within the academic environment. He has helped me appreciate the classical ideal of the unselfish, modest scholar who advances knowledge patiently and with honesty. For this I am deeply grateful.

I would like to thank Mark Shirley, at Xerox PARC, for his advice at several stages in my research. I am particularly grateful to two of my colleagues at Schlumberger: to Didier Wimmers, for his tremendous assistance throughout the project; and to Givargis Danialy, for his tireless energy and enthusiasm.

My deepest gratitude is to my family — Amma, Appa, Lalitha and Lars — to whom this work is dedicated. Their boundless, unqualified love has been a constant source of inspiration and sustenance to me; their sacrifice and support has made all of my achievements possible.

I hear my father; I need never fear.

I hear my mother; I shall never be lonely, or want for love.

When I am hungry it is they who provide for me; when I am in dismay, it is they who fill me with comfort.

When I am astonished or bewildered, it is they who make the weak ground firm beneath my soul: it is in them that I put my trust.

When I am sick it is they who send for the doctor; when I am well and happy, it is in their eyes that I know best that I am loved; and it is toward the shining of their smiles that I lift up my heart and in their laughter that I know my best delight.

I hear my father and my mother and they are my giants, my king and my queen, beside whom there are no others so wise or worthy or honourable or brave or beautiful in this world.

I need never fear: nor ever shall I lack for loving-kindness.

James Agee (1956)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Abstract

The current Application-Specific Integrated Circuit (IC) development cycle lacks completeness and efficiency because it isolates the tasks of design, test and debug, and because of the constraints imposed by the performance bottleneck of software simulation. A new emulation-based methodology is proposed which integrates design and test, making design more efficient and making test generation and fault grading more rigorous. It also introduces design understanding into the production test domain, presenting, for the first time, the opportunity for production line diagnosis. The goals of this thesis were to evaluate the feasibility of the proposed methodology and to identify the issues which need to be resolved before it can fully be implemented.

A sample device was chosen as a test case for the model. Each stage of the proposed process was implemented using this device. Due to the prohibitive cost of using a proper commercial emulation system, the evaluation was performed using a simulation model which we believe captures the nature of emulation sufficiently to provide detailed and valuable analysis. As a result of the evaluation process, performance estimates were made for the proposed technique which suggested a dramatic advantage over current techniques. However, several key concerns were raised about the general applicability of the proposed model to IC design tasks. The conclusion of our evaluation was that elements of the proposed methodology offer substantial promise for improving the rigour and efficiency of current design processes. Understanding the limits of their applicability is required to implement them effectively. If the issues we have raised are properly addressed, and this

perspective is maintained, then these techniques can be used to substantially enhance the design cycle.

1.2 Background

ASIC (Application Specific IC) development has undergone tremendous changes in the past 30 years. The process of designing, testing and diagnosing ASICs has benefitted immensely from the emergence of computer-aided design, automatic test generation, and circuit simulation technologies. As the computational power available to developers skyrockets, and as the price of that power plummets, ASIC development is gradually evolving from a manual, time-intensive process to one that is increasingly automated and intelligent.

However, the most modern development cycles are still inefficient and wasteful of resources. There is still too much dependence upon manual design and verification. Devices are still released into the marketplace before they have thoroughly been validated. The development cycle (design, test and debug) is not integrated, with the result that designers do not take advantage of the opportunities that exist for saving time and effort by automating.

What is most strange is that there are clearly more powerful and effective methodologies for developing ASICs than those currently in use. Why is it that chip designers and manufacturers do not take advantage of these alternatives? The reason these approaches are not in use is that they are constrained by the technology bottleneck of software simulation. Although they offer more rigour and efficiency, the computational power required to implement them is not readily available.

1.3 IADE

IADE (Integrated ASIC Development using Emulation) is a proposed methodology for ASIC design, testing and diagnosis, that provides the conceptual completeness, rigour and efficiency that is lacking in current approaches, and promises far better performance than current techniques. It circumvents the computation bottleneck by employing logic emulation, a novel technology that promises both the power and the performance required to implement it.

IADE integrates design, test pattern generation, fault coverage analysis and production-line diagnosis to make the product development cycle both more rigorous and more efficient. Based upon the new technology of programmable hardware emulation, this methodology overcomes the present bottleneck of software simulation in the design and testing cycles. By doing so, it gives designers the flexibility to design for testability. the power to perform thorough and meaningful test generation and fault coverage analysis, and the opportunity to diagnose device failure at the structural testing phase.

1.3.1 IADE vs. the Current Paradigm

IADE combines the process of integrated design validation with production-line test diagnosis to provide a coherent top-down methodology for ASIC development. The structure and rigour it introduces to system design enhance the efficiency, thoroughness and flexibility of the entire design process. It should be noted that the effectiveness of this new methodology is made possible for the most part by the tremendous performance capabilities of hardware emulation.

The way in which IADE changes the process of design. test and debug in ASIC development can be best summarized by following the entire flow of a particular design. Figures 1.1 and 1.2 illustrate the development models for current processes and for the proposed IADE methodology.

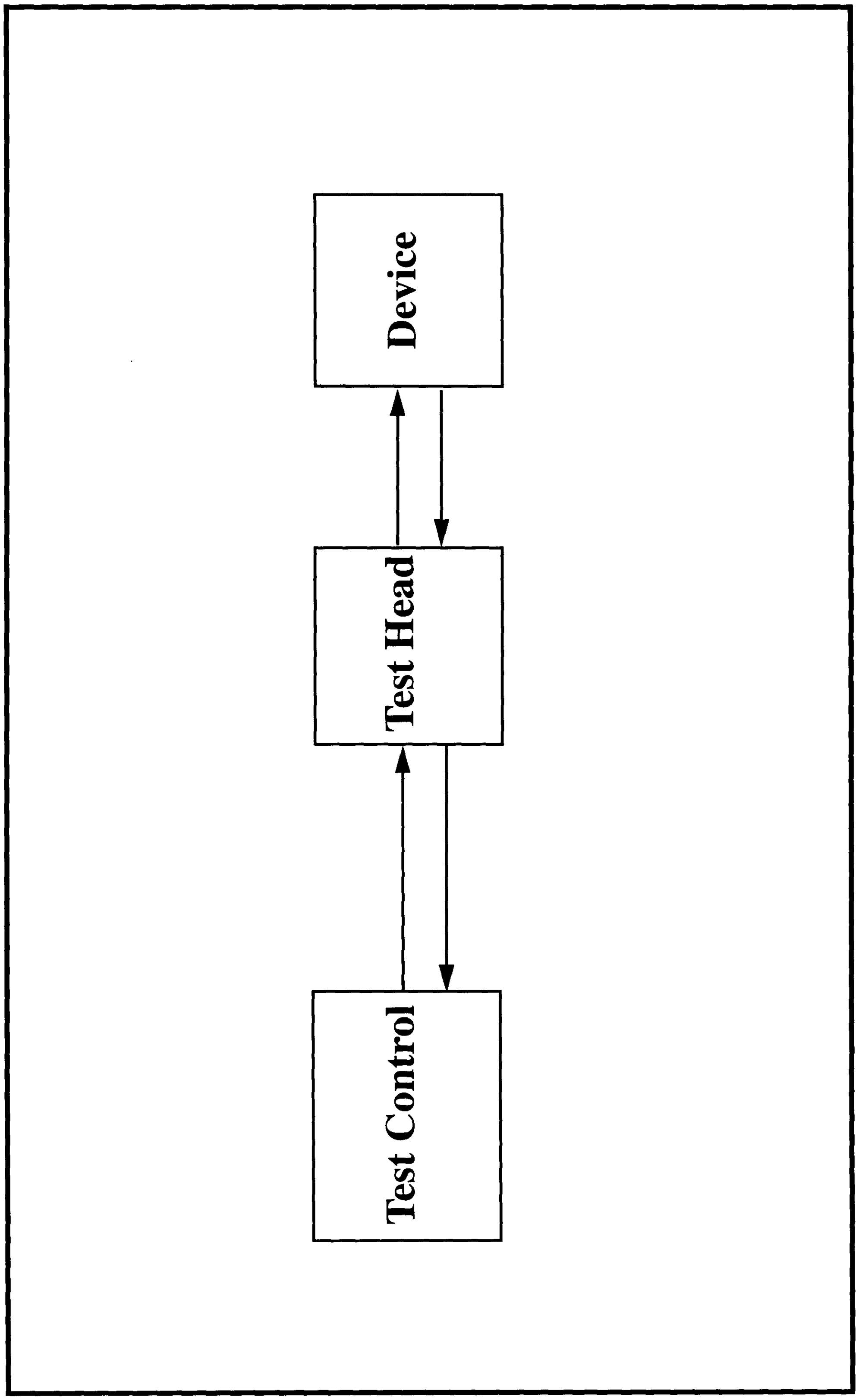

Figure 1.1: Current ASIC Development Model

A typical current process is illustrated in Figure 1.1 After design of the device is achieved and committed to production, test generation produces a test set that will validate the device during manufacturing. This test set is typically generated automatically using structure from design files. Fault grading is used to determine the coverage of the test set. This is currently done using simulation. As devices become increasingly complex, test generation and fault coverage analysis using simulation succumb to time pressure, with the result that designs are often released into production with test sets for which the actual fault coverage has not been determined.

When the design has been committed to production and the test set has been completed, the process of production testing begins. The tester environment has three main conceptual elements: the test pattern store and control section, the test head electronics which controls the actual patterns applied to each pin of the device, and the Device Under Test. The test program which is stored in the control module consists of two parts: the stimulus values to be applied to the DUT, and the output values expected in return.

This program is loaded and configured in the control module, patterns are applied via the test head to the DUT, and the result is that the device either passes or fails. The tester has no inherent knowledge or the DUT's design or intended functionality. The bulk of its complexity goes to managing and storing the test set, and to controlling precise timing of events to the test head.

Figure 1.2 illustrates the proposed IADE mechanism for ASIC design, test and debug. Each phase of development uses a system in which two models of the device are exercised in parallel. In the first phase of development, the design process results in a behaviour model for the device. In the course of designing the device and specifying its interfaces, the engineer develops a rigorous description of the device's intended range of use.

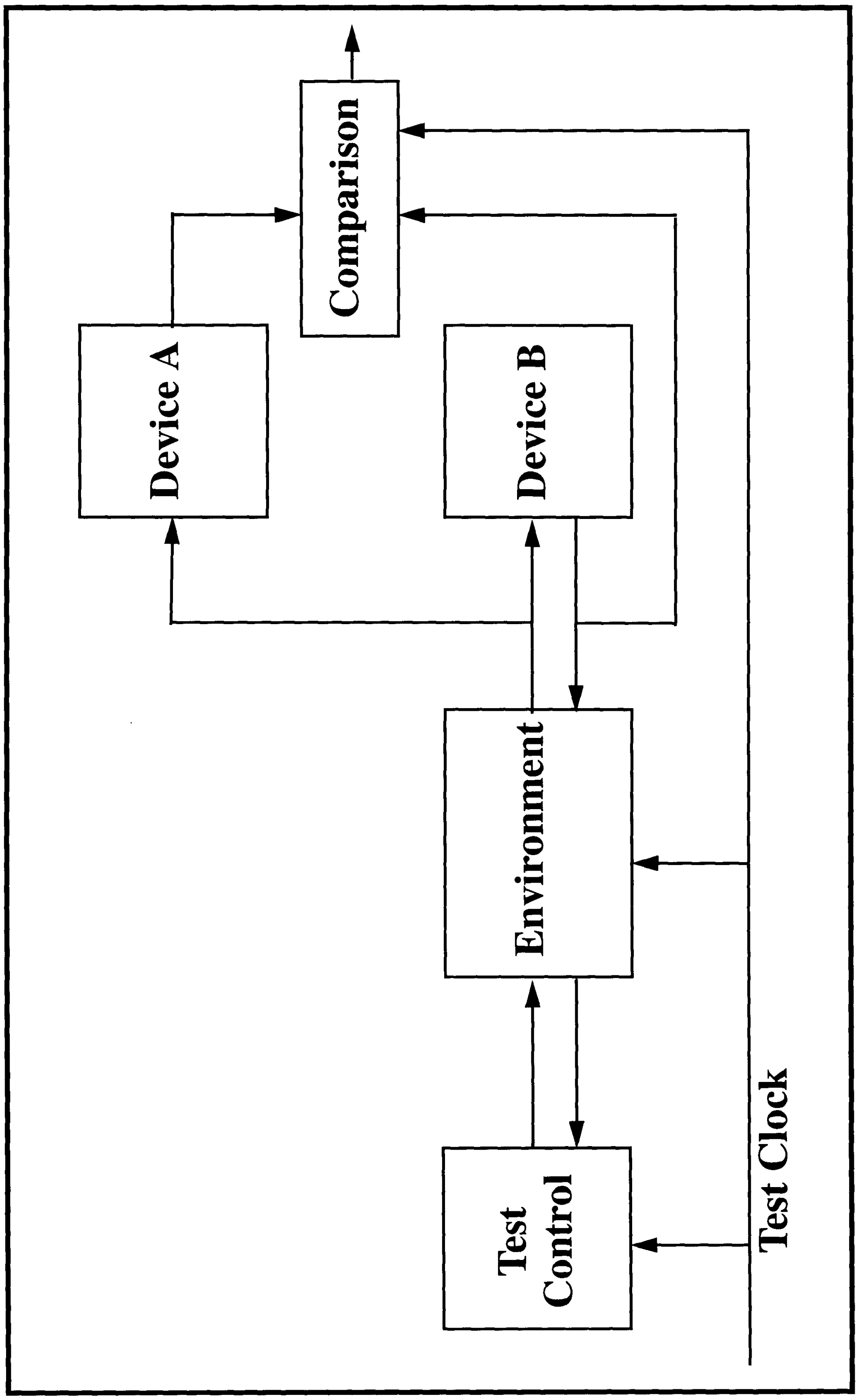

Figure 1.2: IADE Development Model

This can be used to create an environment-level stimulus set that reflects the full range of the device's intended behaviour. The next step is to create the interface between this high-level behaviour description and the device's specific inputs and outputs. This results in the environment model for the device.

The device model is instantiated twice in the design verification system. Device A is a "clean" model, and Device B is a model which can selectively be injected with faults. Initially both models are fault-free. The environment model and the two device models are implemented in gate-array emulation.

Test generation from this point is automatic. There is no need to use software to analyze the design files and produce a test set; instead, by specifying the complete functional behaviour of the device in the stimulus set for the environment, the engineer has completed the task at a higher (and more abstract) level. The environment will take the stimulus description and generate automatically the specific test patterns which that entails. Test generation, therefore, is complete and automatic. Fault grading is also automatic. The stimulus set is applied in parallel to the good and faultable models.

The outputs are compared. When the stimulus set is inadequate to discover injected faults, then the model can be revisited to identify redundancy or poor design. In this way, design is integrated with test generation before committing to silicon. Emulation also allows this process of test generation and fault grading to proceed at hardware speeds, and therefore makes it possible to perform complete validation before committing to production.

When the design has been optimized and finalized, production testing takes place. In this phase, the "good" model for Device A is discarded, and replaced by the production DUT. The DUT (Device A) and the faultable model (Device B) are then exercised in parallel. The patterns produced by the environment model, which is implemented in emulation, are fed directly to the test head electronics and then to the DUT. As a result, the physical tester itself does not require the complex memory management and pattern control subsystem which stores and applies the statically created device-level test set used in current models.

Instead, the equivalent of this test set is generated automatically by the environment model. All that needs to be stored now is the environment stimulus set, which is considerably smaller and more abstract. As a result, the tester environment is smaller and more efficient. It relies upon the emulation subsystem to generate and feed it the patterns it needs to apply to the pins of the DUT. Moreover, the emulated "faultable" model of the device can now be exercised in parallel with the actual DUT to diagnose cluster failures in production. By comparing the outputs from the DUT (Device A) with those from the selectively "faultable" emulation model (Device B), it is possible to diagnose and localize faults which result in device failure. IADE therefore changes the nature of the design and test flow by introducing emulation and environment modeling into the process to provide for dynamic, on-the-fly test generation, and to make unnecessary the sophisticated test control mechanisms which make up the bulk of the complexity of modern testers. It also provides on-the-fly and complete fault coverage analysis, and allows for production line diagnosis of failures.

There are some drawbacks to using IADE. First, emulation models of devices can be run at hardware speeds, but in most cases they cannot yet be run at the speeds of the devices under test. Moreover, unlike simulation, emulation does not provide accurate timing information or information about indeterminate states. Consequently the use of IADE is strictly for functional verification. A timing analyzer would have to be incorporated into the system to complete timing verification. However, it seems that using emulation in combination with simulation in this manner would greatly improve the simplicity and efficiency of design and test. Emulation technology is also rapidly progressing to the point where running speeds of 100 MHz and greater are conceivable in the near future.

1.4 The Thesis Project

This paper describes a thesis research project whose goals were to evaluate the feasibility of the IADE methodology, and to assess the issues which need to be resolved before it can be implemented fully in the design environment. This was achieved by implementing the proposed techniques using a test device being developed at Schlumberger for use in their commercial automatic test equipment (ATE).

The prohibitive cost of commercial emulation systems forced us to conduct our analysis entirely in simulation. This raises the question: how effective could our study be without actually using an emulation system? Consider that there are two aspects to the evaluation: we wanted to estimate quantitative performance characteristics of the proposed model; we also wanted to develop a qualitative understanding of its weaknesses or limitations. Performance comparisons can be made without actually implementing the model in emulation: given cycle length measurements for the simulation model, we can estimate the real time performance of the equivalent emulation model by using a projected emulation clockspeed. For qualitative analysis, we believe that *designing* the system rigorously for emulation would capture emulation issues sufficiently to provide substantial material for analysis, regardless of whether that system was subsequently implemented using simulation or emulation. We are aware that the proposed model needs to be implemented properly in emulation in order to understand all of the issues involved. Nevertheless, we are confident that our initial simulation-based analysis clarifies several key issues, refines our understanding of the concept and identifies directions for future research, at a low cost comparatively. It has, we believe, provided the groundwork for more resource-intensive analysis in future.

1.5 Outline of the Thesis

This paper has three main sections. The first section analyzes the drawbacks of current development cycles. and the means by which IADE attempts to address them. This overview is provided in Chapter 2. The second section describes our test implementation of the methodology, for each phase of the design process. Chapter 3 discusses the specification and design phase. Chapter 4 describes our implementation of fault grading. Chapter 5 describes the process by which we conducted functional verification of physical devices in the production test environment. In Chapter 6, we explore and assess various approaches for diagnosing failing physical parts in production testing. Finally, in Chapter 7, we summarize our results and identify areas for further research.

Chapter 2

The ASIC Development Cycle

2.1 Introduction

The process of developing ASICs consists of five main phases. The first stage is specification and design of the device. This includes behaviour description, gate-level implementation and design verification. The second stage is development of a production-line test set for the device. Once a production set is created, it needs to be fault-graded. This involves establishing requirements for the fault-coverage of the test set, and analyzing the test set to verify that it meets those requirements. The fourth stage of development is production-line testing of manufactured parts using this test set. The final phase is production-line diagnosis of cluster failures, which may occur because of process variance or narrow margins in the design. Diagnosing cluster failures feeds back to design of next-generation devices.

This chapter begins by defining a syntax for describing ASIC development. It then compares the phases of development in a typical current method with those in the proposed method. It describes the drawbacks of currently used methodologies, and the means by which IADE addresses those flaws. It then presents the details of the proposal, and the conceptual basis for its design. Finally, it describes the process by which we will evaluate the proposed methodology.

2.2 A Syntax for Describing ASIC Development

Understanding the goals of the proposed ASIC development methodology first requires developing a syntax with which to describe unambiguously the different steps which take place in the development cycle. Using this syntax we can then examine current techniques

$\mathbf{D_B}$	**- Device - Behaviour Model**
$\mathbf{D_G}$	**- Device - Gate-Level Model**
$\mathbf{D_G^n}$	**- Device Gate-Level Model Injected with Fault n**
$\mathbf{F_D}$	**- Functional Test Set (Device Level Stimulus)**
$\mathbf{S_D}$	**- Structural Test Set (Device Level Stimulus)**
f(x)	**- System f applied with test set x**
$f(\chi) \approx g(\chi) \Leftrightarrow f(\chi) = g(\chi)$	**at each time-step in which f(x) and g(x) are strobed**
P	**- Fabrication Process**
W	**- Wafer**
$\mathbf{P(D_G) = W}$	**- Process P is applied to gate-level model $\mathbf{D_G}$ to produce wafer W**
DUT	**- Device Under Test**
$\mathbf{C_R}$	**- Fault Coverage Required**
$\mathbf{C_M}$	**- Fault Coverage Measured**

Figure 2.1: A Syntax for Describing ASIC Development

and their limitations, and understand how those limitations are addressed by the proposed method. We begin by defining a set of terms which will form the building blocks of our syntax, shown in Figure 2.1.

The first group of terms distinguishes between different device models, each of which is written using a Hardware Definition Language (HDL) such as Verilog: the behavioural HDL model ($\mathbf{D_B}$); the gate-level HDL model ($\mathbf{D_G}$); and the gate-level HDL model into which a particular fault *n* has been injected ($\mathbf{D_G^n}$). The second group of terms distinguishes between different test sets: the test set designed to exercise functionality ($\mathbf{F_D}$); and the test set designed to exercise all paths in the device structure ($\mathbf{F_S}$). When a test set *x* is applied to a device f, its outputs are denoted by $f(x)$. We say that two devices f and g are *equivalent* (denoted by $\approx$) across a test set x if, for all time steps in which they are strobed, $f(x)$ equals $g(x)$. The next set of expressions describes the process by which a production wafer is created from a gate-level device model. Finally, $\mathbf{C_R}$ and $\mathbf{C_M}$ are used to describe the required and measured fault coverage values for a particular test set.

We will use these terms to develop expressions which describe each step in the ASIC development process, both for current methods and for the proposed method. This will provide a clear basis for comparison between the two processes.

2.3 The Current Method

The following sections describe each of the phases of ASIC development within a typical current methodology. What constitutes a typical current methodology? It is important to understand that there is no "set" methodology for developing ASICs. In different design environments, engineers combine available resources with extant process understanding and experience to create design flows which are unique to those environments. Despite this diversity of techniques, we can group design flows into two broad categories: those which are most primitive, relying mainly on manual analysis, and those which make extensive use of fairly sophisticated simulation and modeling tools. We will concentrate our comparison on the latter.

2.3.1 Design and Verification

The first step in designing a device is to develop its behaviour specification. In contemporary environments, this is usually done by developing a functional behaviour model of the device, $\mathbf{D_B}$, using HDL tools. This model defines the input/output behaviour of the device (its functionality) without constraining it to any particular gate-level implementation. Once the behaviour model is specified, design engineers produce a gate-level implementation $\mathbf{D_G}$, either by hand or by using synthesis tools.

The next step in a typical process is to develop a functional test set for the device, $\mathbf{F_D}$. This test set is then used to verify that the gate-level model of the device exhibits its required functionality. $\mathbf{F_D}$ is applied in simulation to both $\mathbf{D_B}$ and $\mathbf{D_G}$, and their outputs are compared. $\mathbf{D_G}$ passes this test if:

$$D_G(F_D) \approx D_B(F_D) \tag{2.1}$$

Once $\mathbf{D_G}$ is verified, the designer's task is considered done. The design is then passed to the test engineering and manufacturing environments.

2.3.2 Test Set Generation

Test engineers begin their task with the assumption that the gate-level design $\mathbf{D_G}$ is correct and does not need modification. Their first task is to develop a production test set for the device. This set is frequently different from the functional test set $\mathbf{F_D}$ used to verify the gate-level model. Usually, it follows the structure of the gate-level model and is designed to exercise all of the paths in $\mathbf{D_G}$. This test set is denoted by $\mathbf{S_D}$. Although automatic test generation software is occasionally used for test development, this task is more commonly done laboriously by hand. The test set that results is expected to exercise the design properly and completely.

2.3.3 Fault Grading

The fault coverage of a test set is measured relative to a particular device and a particular *fault model.* A fault model is simply a set of faults which could exist within the physical device. For example, the *single-stuck-at input fault model* considers all situations in which (exactly) one input to (exactly) one node of the device is stuck to zero or stuck to one. Choosing a fault model produces a set of N possible faults to consider in testing the device. Consider the case of a device with P nodes, each of which has, on average, Q input signals. In this case, a single-stuck-at input fault model produces N possible faults, where

$$N = 2(P \cdot Q) \tag{2.2}$$

For each situation n of the N total possible situations, the test set is applied to the device to determine whether or not it can detect the fault. Fault coverage is given as the proportion of all N possible faults for which the test set $\mathbf{S_D}$ can detect the difference between the device injected with the fault ($\mathbf{D_G}^{\mathbf{n}}$) and the clean device ($\mathbf{D_G}$):

$$C_M = \frac{\langle \sum n | (D_G(S_D) \neq D_G^n(S_D)) \rangle}{\sum n} \tag{2.3}$$

Fault coverage is measured using software simulation tools. This process can often takes upwards of several weeks. Because of the performance limitations of software simulation, the designer is frequently forced to fabricate the circuit before fault-grading is completed. This has two consequences:

First, there is no opportunity for fault-grading results to feed back into design on the same iteration, as the circuit has already been committed to fabrication. The design is revisited only if there is a major functional failure on the wafer produced by the first fabrication:

$$W(S_D) \neq D_B(S_D) \Rightarrow D_G \rightarrow D_G' \tag{2.4}$$

Second, the fault grading results themselves diminish in significance. If the chip is fabricated successfully, fault coverage measurements are sometimes considered useless when they finally arrive, and therefore discarded. Occasionally, if the design is fabricated successfully, but the fault coverage is insufficient, the test set is extended until the desired coverage is achieved:

$$C_M \neq C_R \Rightarrow S_D \rightarrow S_D' \tag{2.5}$$

If the first wafer produces errors, and the design is modified, the initial fault grading model is clearly invalid, and needs to be re-started using the new design.

A much more subtle limitation of this method of test validation is that there is no opportunity for test generation or fault grading to critically question the efficiency or optimality of the gate-level design of the device. The designer is, at once, attempting both to test the gate-level design $\mathbf{D_G}$, *and* to validate the test set $\mathbf{S_D}$. Since the test set is based upon the structure of the gate-level model, the presumption is that *a priori* the design meets its functional requirements optimally and that it remains only to exercise every path in the circuit. In this approach, the automatic pattern generation tools are not concerned with the functionality of the device, but merely with its structure. As a result, the key question, whether the device does what it should, is discarded prematurely when the design phase ends. It is possible only to verify that the circuit matches the structure defined for it.

Because of its circular nature, this process becomes iterative: the initial design is fabricated and exercised by the initial test set; this run may identify errors in the design or demonstrate incompleteness in the test set. If this occurs, the design and test set are corrected and updated, and the design is re-fabricated. The cycle is repeated until no further errors are detected. Ultimately, personal inspection of the circuit and the test set by design engineers is required for final approval of the design.

2.3.4 Production Line Testing and Diagnosis

Once the design has been committed to final production, and the production test set S_D has been finalized, the next step is to convert the test set into a production test program which can run on commercial Automatic Test Equipment (ATE). The test program contains two elements: the actual stimulus applied to the device as input, and expected output data which would be generated from a clean device. Once this set of test vectors is determined, it is programmed on the assembly-line ATE machine by production test engineers.

The production test engineer is unaware of, and indeed, unconcerned with the details of the design and test generation phases. In fact, in many cases, designs are committed to production for which full verification and fault-grading have not been completed, due to the time constraints of product schedules.

The ATE machine tests each DUT on the production line as follows:

$$DUT(S_D) = \{PASS, FAIL\} \tag{2.6}$$

Two main problems can occur in production of a device:

First, devices are sometimes passed by the test program which subsequently fail in the field: these are known as test escapes. The main reason for this problem is that the test set was not properly validated to achieve a level of fault coverage sufficient to handle most field situations. In this case, current methods require the test set to be extended specifically to identify each source of field failure.

Second, the situation frequently occurs in ASIC development and production that a well-designed, well-validated device exhibits an alarming or unacceptable number of clusters of failures in assembly line structural testing. There are a few possible reasons for this phenomenon. One reason may be that in the course of the product's lifetime, process variance becomes significant enough to cause chronic failures. Another common reason is that the mask is defective. Often, devices that have adequate thresholds have their specifications tweaked in such a way as to reduce thresholds and render their designs marginal. For example, a good design at 33 MHz might become shaky and unreliable at 66 MHz. In situations of unacceptable yield, current techniques provide no means by which to diagnose DUT failures on the assembly line.

This proceeds from the basic design of ATE equipment: The first responsibility of the ATE architect is to design equipment which exercises the device with the multi-million test vector set supplied by the customer in as little time as possible, with as much flexibility as possible to examine test result data. The customer (IC designer) configures the equipment with their production test set, and this chip tester is used on the production floor to test each device that is manufactured. There is no design understanding inherent in the current ATE architectures. They are simply large control and memory structures focussed upon timing accuracy and data measurement and analysis capability.

In spite of this, the design of the tester is surprisingly complex, for two reasons. First, since many modern devices are both large and fast, applying test patterns to hundreds of pins at speeds faster than the devices themselves (many devices today run at speeds upward of 100MHz) is a challenging task. Second, a fair portion of the tester's complexity is devoted simply to storing and controlling the flow of millions of test vectors. Consequently, testers are large, expensive, and difficult to maintain.

2.4 The Proposed Method

The method proposed in this research project begins by introducing the following new elements to the syntax described in Figure 2.1. These are illustrated in Figure 2.2. These additions arise from the observation that describing the average device in terms of its input/output behaviour is a very ineffective way to represent its functionality. At the device boundary, behaviour can be described only in terms of transitions in input and output signals. This type of low-level state table does little to convey the general functionality of the device. Consider the example of a 32-bit counter. The state table required to describe the behaviour of this device would be enormous in size. Yet, understanding the functionality of the circuit can be understood much better by abstracting away from its I/O boundary and understanding its role within a larger system.

$\mathbf{E_B}$	**- Environment - Behaviour Model**
$\mathbf{E_G}$	**- Environment - Gate-Level Model**
$\mathbf{F_E}$	**- Functional Test Set (Environment Level Stimulus)**

Figure 2.2: New Elements in the Proposed Method

2.4.1 The Environment Abstraction

The proposed method introduces the concept of modeling *both* a device and an abstraction of its environment. The environment model is any model which abstracts from the device boundary to some higher-level description of the device's functionality. The purpose of this abstraction is to clarify the intended functionality of the device and to simplify the task of determining its required functionality. The environment model of the device is itself a device, which can be implemented behaviourally ($\mathbf{E_B}$) or structurally ($\mathbf{E_G}$). The environment abstraction should be specified in such a way that the environment level stimulus $\mathbf{F_E}$ is simple to define and clearly reflects the functional behaviour of the device. The extent to which the environment abstracts from the device is arbitrary. At one extreme, the environment can be an empty box, providing no abstraction. In this case, the environment level stimulus $\mathbf{F_E}$ and device level stimulus $\mathbf{F_D}$ are identical. At the other extreme, the environment models the entire system in which the device operates. In this case, the entire system must be implemented in the environment, and the environment level stimulus $\mathbf{F_E}$ could be as simple as a "Start" button. Typically, the environment abstraction will lie somewhere between these two extremes, the goal being to make the abstraction powerful enough to actually simplify functional specification without making it so powerful that the task of implementing the abstraction becomes equivalent to

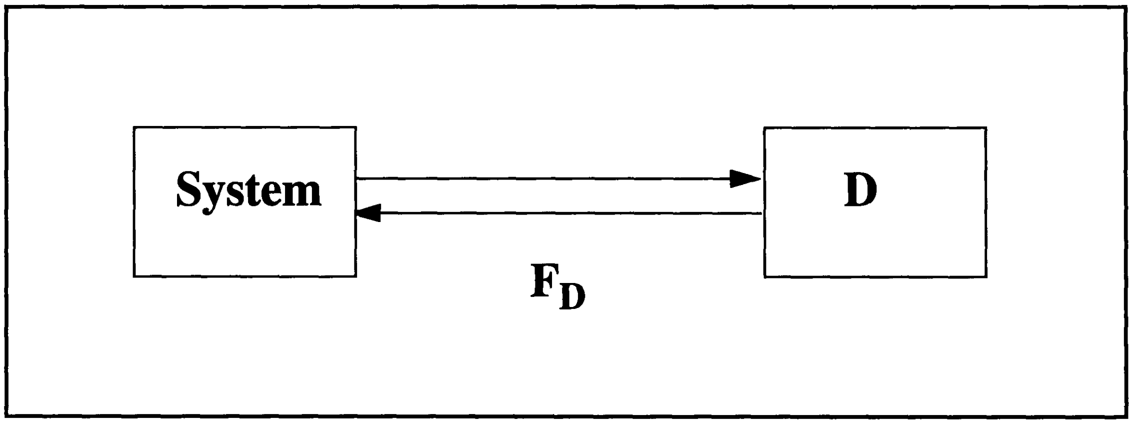

Figure 2.3: Device-System Interaction in Current Methods

developing all of the devices with which the original design interacts. Those details which are unimportant to the functionality of the device should be omitted from the model of its environment.

Once the environment abstraction is determined, it should follow that:

$$E(F_E) = F_D \tag{2.7}$$

Thus, if the environment abstraction is correct, it should translate the high level stimulus set $\mathbf{F_E}$ into the device level stimulus $\mathbf{F_D}$. Furthermore, if expression 2.7 is true, then it also holds true that:

$$D(E(F_E)) = D(F_D) \tag{2.8}$$

Traditional techniques reflect a model in which the device **D** is exercised and analyzed directly across the device input/output interface (Fig. 2.3). The proposed method describes a model in which the device **D** is exercised and analyzed *through* the environment model E (Fig. 2.4). The environment abstraction is a crucial element of the proposed methodology. It provides the means for detaching test validation from the gate-level implementation of the device, thereby allowing fault grading to feed back effectively into design. This process is described in the following sections.

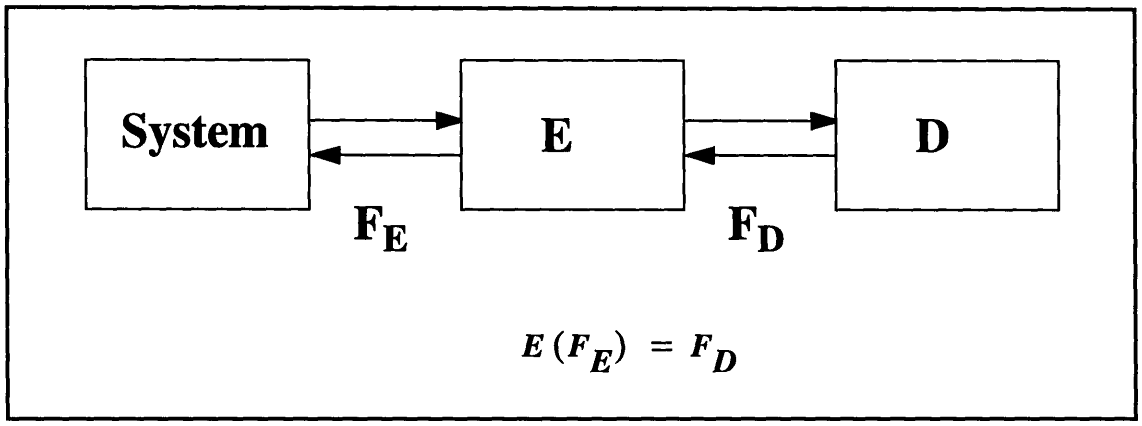

Figure 2.4: Device-System Interaction in IADE

2.4.2 Design and Verification

The IADE model for design and design verification begins with the specification of the device and the environment abstraction. This produces the HDL models $\mathbf{D_B}$ and $\mathbf{E_B}$. At this point, it should be fairly straightforward to determine the basic functional test set for the device, defined at the environment level as $\mathbf{F_E}$.

Once these elements have been defined, the first step is to verify the correctness of the environment model. The environment model makes the correct abstraction from high level stimuli to device level stimuli if:

$$E_B(F_E) = F_D \tag{2.9}$$

where $\mathbf{F_D}$ now represents the device-level functional test set.

The next task is to develop a gate-level model $\mathbf{D_G}$ for the design. The gate-level implementation can be performed either manually or by using synthesis tools. This model can then be verified in simulation by using the model illustrated in Figure 2.5. The test control apparatus applies the test set $\mathbf{F_E}$ to the environment model $\mathbf{E_B}$ which produces the device-level test set $\mathbf{F_D}$. $\mathbf{F_D}$ is applied in parallel to the gate-level and behavioural models of the

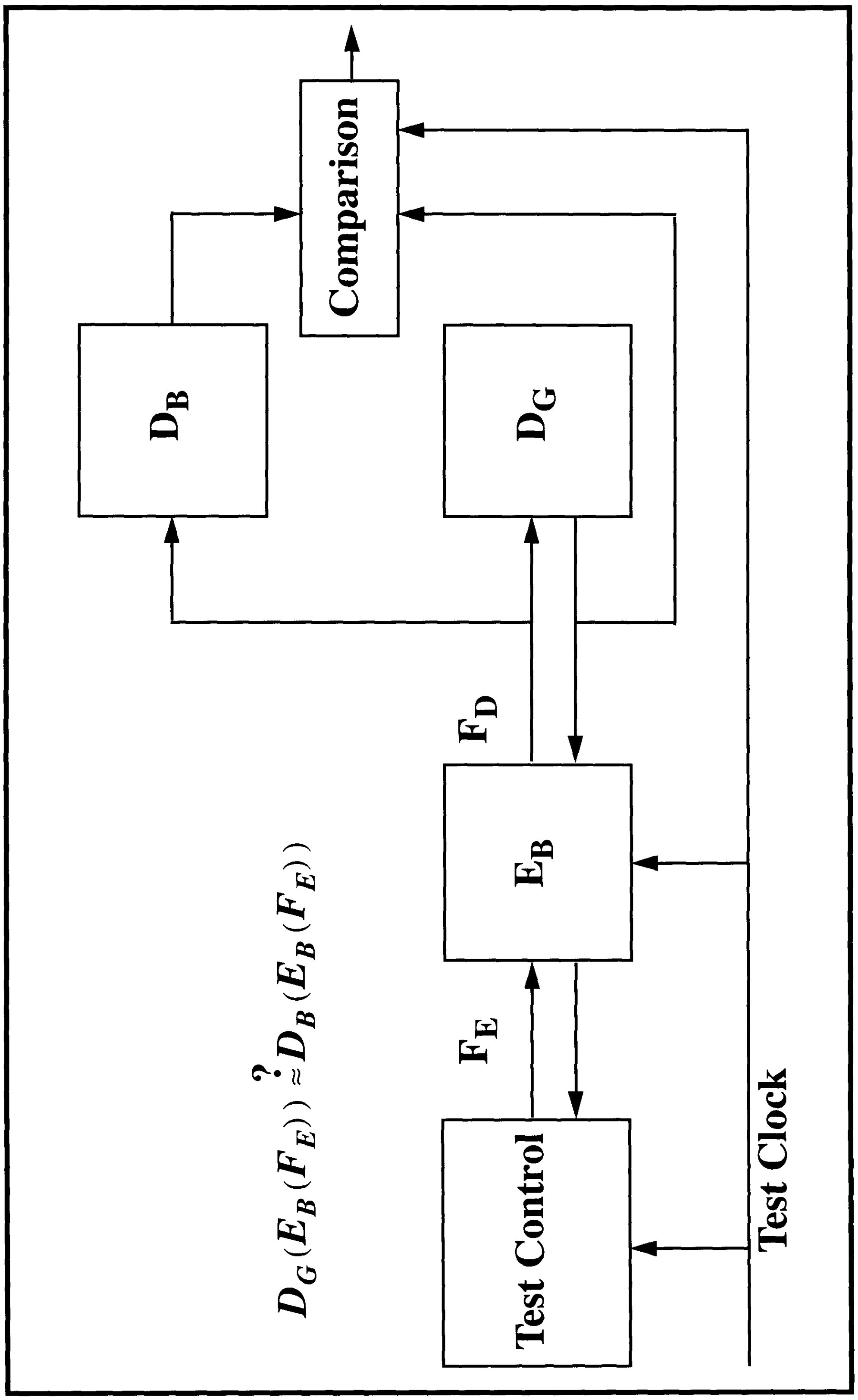

Figure 2.5: IADE Model for Design Verification

device, $\mathbf{D_G}$ and $\mathbf{D_B}$. The outputs of these devices are then compared at intervals which are clocked by the test control module. The gate-level model of the device is correct if:

$$D_G(E_B(F_E)) \approx D_B(E_B(F_E)) \tag{2.10}$$

This process is performed in simulation because it involves behavioural models of the device and the environment. If this has been confirmed, we know that the gate-level implementation we are testing at least satisfies its requirements.

Once the gate-level device model has been developed and verified, the environment model must also be implemented at gate level. When gate-level models are achieved for both the device and its environment, functional verification need no longer be constrained to run in simulation. The proposed fault grading and production test environments use these gate-level models to take advantage of hardware emulation, which requires gate-level design models. It should be noted, however, that implementing a gate-level environment model $\mathbf{E_G}$ is not as complex as implementing the gate-level device model $\mathbf{D_G}$, although initially this appears to be the case. Why is this? The reason is that $\mathbf{D_G}$ is the final product of the entire development process, and consequently, needs to be implemented as cleanly and efficiently as possible. In fact, optimizing $\mathbf{D_G}$ is the goal of the entire design process. Since $\mathbf{E_G}$ is simply a tool to facilitate verification and production testing, it is not necessary to exhaust every effort to make it smaller or faster. Even if it is not optimal, it can still serve its purpose within the system. Indeed, the goal in implementing $\mathbf{E_G}$ should be to produce a functioning gate-level model with the minimum amount of effort. This should not be too difficult to achieve using, for example, basic synthesis tools. $\mathbf{E_G}$ is functionally correct if:

$$E_G(F_E) = E_B(F_E) \tag{2.11}$$

which can be verified using simulation.

2.4.3 Fault Grading

The next phase in the proposed method is fault grading. Unlike in typical current processes, fault grading is performed *before* the gate-level design is committed to fabrication. As a result, the opportunity exists to modify the design as a result of fault grading measurements, an opportunity which is missed by most current development cycles.

The proposed model for fault grading is illustrated in Figure 2.6. In this model, the behaviour model of the environment $\mathbf{E_B}$ is discarded and replaced with the gate-level environment model $\mathbf{E_G}$. Similarly, the behaviour model of the device $\mathbf{D_B}$ is discarded and replaced with a second gate-level model of the device. One of these device models functions as a "good" model, denoted by $\mathbf{D_G}$, which corresponds to the correct gate-level design. The other device model is used as a "faultable" model, denoted by $\mathbf{D_G}^{\mathbf{n}}$. This means it can selectively be injected with faults which correspond to the fault model being used for fault grading, as described in Section 2.3.3. This "faultable" device model can be controlled by means of a fault selector mechanism.

Fault grading of the test set is performed by applying $\mathbf{F_E}$ to the environment model, and applying its output $\mathbf{F_D}$ in parallel to both "good" and "faultable" device models. When the comparator mechanism detects a difference in the outputs of these models, then the test set has successfully detected the fault **n** with which the "faultable" model was injected. The fault selector sequentially injects $\mathbf{D_G}^{\mathbf{n}}$ with each possible fault in the fault model. Each time a different fault is injected, the test set is run again and the outputs of the two models are compared. In this way, we can establish the fault coverage of the test set as:

$$C_M = \frac{\langle \sum n | (D_G(E_G(F_E)) \neq D_G^n(E_G(F_E))) \rangle}{\sum n} \tag{2.12}$$

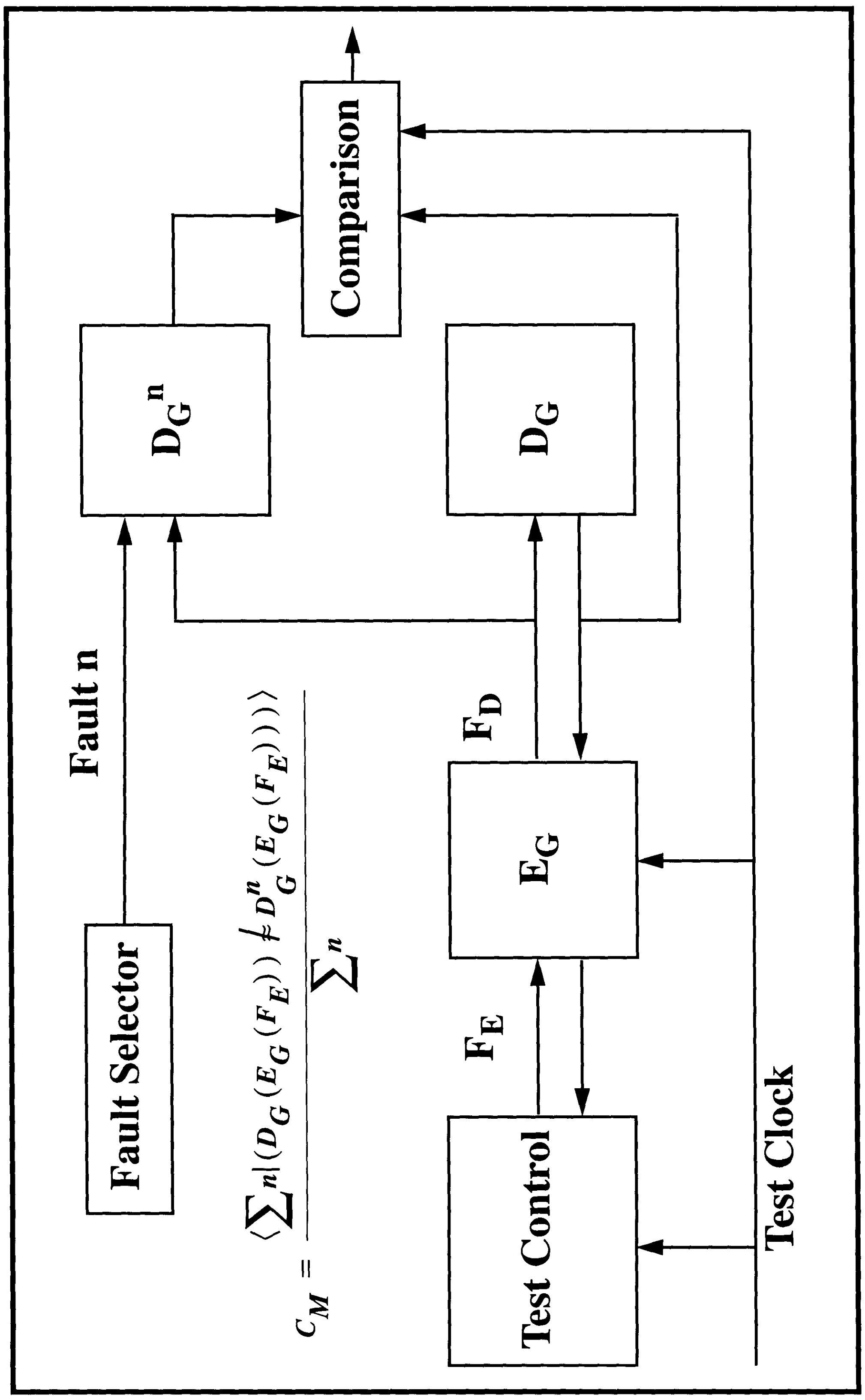
Fault Selector
Fault n
D_G^n
Comparison
D_G
F_D
E_G
F_E
Test Control
Test Clock
$C_M = \frac{\langle \sum n | (D_G(E_G(F_E)) \neq D_G^n(E_G(F_E))) \rangle}{\sum n}$

Figure 2.6: IADE Model for Fault Grading

Because we have replaced all behaviour models in the system with gate-level models, the entire process can now be implemented using hardware logic emulation.

As a result, fault grading can now run at hardware speeds. This is the crucial difference between the proposed process and current techniques: even the most basic current emulation systems run at speeds of 5 to 7 MHz, while current fault grading simulation tools run at speeds of cycles per second. The performance improvement is several orders of magnitude. In the computer industry, it is often said that if a performance improvement in a particular area is substantial enough, it will cause a paradigm shift in that area by enabling new architectures and methodologies to develop, and by shifting the focus of future development to the new bottlenecks which emerge. This is the case with the change from simulation technology to emulation technology.

Fault grading using simulation can take upwards of several weeks to perform, depending upon the size of the device and the breadth of the fault model. The performance gain of 5 to 6 orders of magnitude that is enabled by emulation means this task can now be completed in minutes or hours. As a result, fault grading can be completed, and in fact re-iterated several times, before the production schedule requires the design to be committed to fabrication. How can fault grading feed back effectively into design? To understand the potential of this process, we will consider the following situation in fault grading:

When faults are injected into the "faultable" model which are detected by none of the test patterns in the existing set, the designer is faced with three possibilities: a) the current set of test patterns is inadequate for the full verification of the device; b) the fault lies outside of target fault coverage requirements, and therefore need not be detected to satisfy current coverage requirements; c) the fault is not detected by any pattern in the current test set, but it does *not* affect the functionality of the device within its environment.

The first possibility demonstrates the process by which IADE enables incremental test set generation for complete verification of a device. The second possibility demonstrates the effectiveness of IADE in providing fast, thorough, fault-grading of a set of test patterns. The third possibility demonstrates the power of IADE as a tool for iterative design optimization. In the third case, the emergence of an undetected fault which does not affect the high-level, environment functionality of a device suggests the possible redundancy of that part of the circuit into which the fault was injected. The designer is not bound to the specific implementation of the circuit because test generation is done at a higher, environment level. This abstraction barrier isolates test generation from the specific circuit implementation of the target device. Test generation instead incorporates knowledge of the device's required functionality to constantly revisit every part of the circuit. As a result, fault grading now forces the designer to justify all elements of the design in terms of the device's required functionality. This process of challenging and rationalizing the design forms a feedback path which allows test generation and fault grading to make circuits more efficient and more testable. This is the advantage of integrating design and test in the IADE methodology

2.4.4 Production Line Testing

The production testing model in the IADE methodology differs significantly from the current paradigm. Instead of using test programs which embody stimulus input values and expected output values, the proposed model generates both input values and expected output values on-the-fly, in the same manner as fault grading. This model is illustrated in Figure 2.7.

As in the fault grading model, two devices are exercised in parallel by the environment using the final test set. One of the devices is an emulated gate-level model $\mathbf{D_G}$, and the second device is now the physical Device Under Test (**DUT**), which resides on the

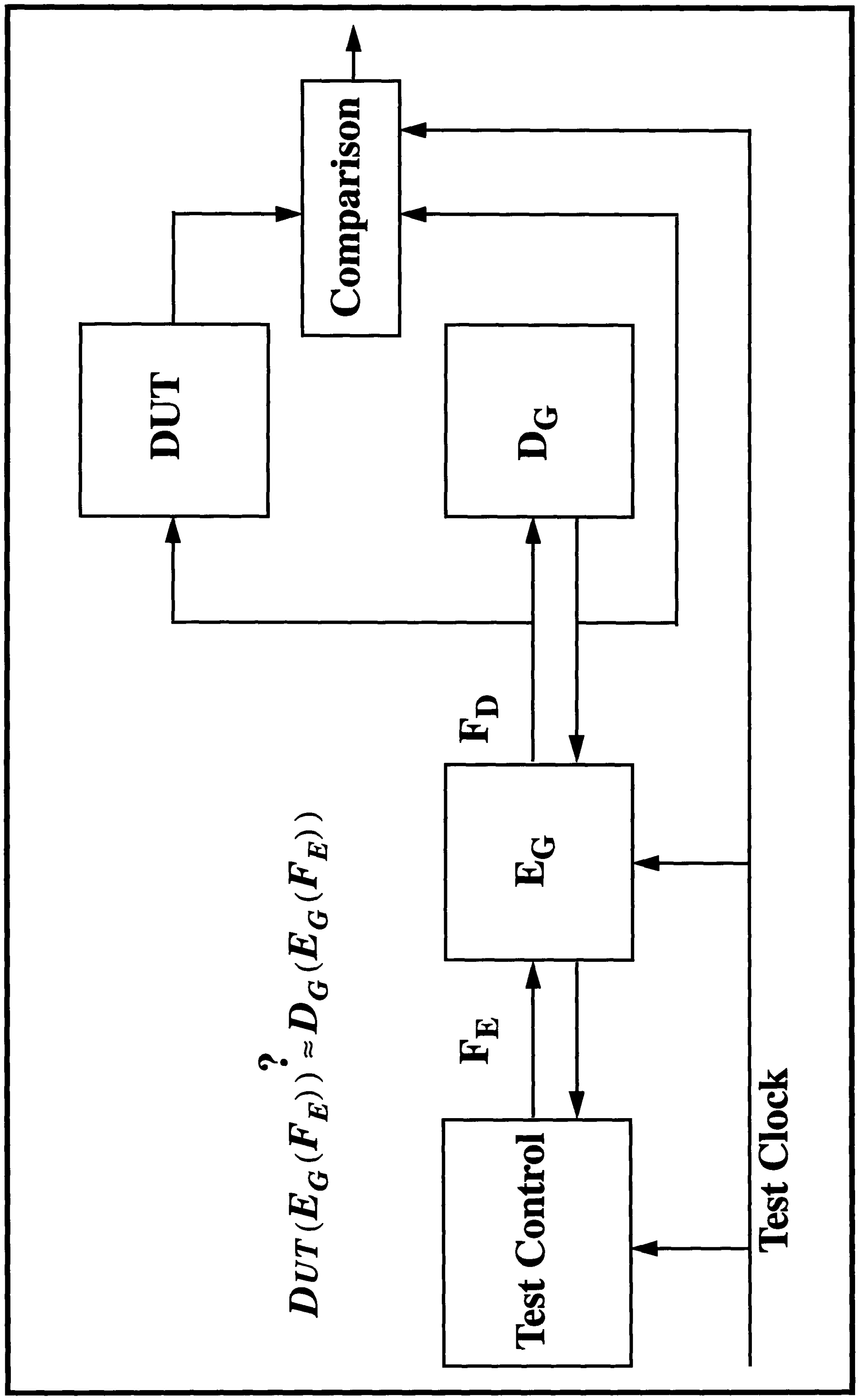

Figure 2.7: IADE Model for Production Line Testing

physical test head and is exercised by the outputs of the environment model. As before, the environment is implemented in emulation.

The component tester is given a much smaller role than it has currently: it now receives the outputs of the environment model as they are generated on-the-fly and applies them to the DUT. Because of the nature of emulation, it makes no difference to the emulated environment model whether it is interacting with an emulated model or an actual device.

The emulated environment applies the test set to the two models in parallel. When the outputs of the DUT and the gate-level emulation match for all of the tests, the DUT has passed:

$$DUT(E_G(F_E)) \approx D_G(E_G(F_E)) \tag{2.13}$$

If they do not match, the DUT fails.

The advantage in this technique is that knowledge of the device's design is introduced into the production testing environment. As a result, the tester need not have any of the bulky memory-management apparatus that is has in current technologies. Now the tester mainly consists of the electronics necessary to apply drive and strobe events as they are produced by the environment. The abstraction introduced by the environment model makes the test set that is actually used much smaller than it is in today's process. Storing this test set is a much smaller and simpler task. Converting this test set into the millions of device test vectors that it represents is the task of the environment.

Although current emulation technology does not always permit the DUT and gate-level emulation model be exercised simultaneously at device speeds, it is advancing fast enough that within a short time it will be possible for the entire system to run at device speeds, in the range of 100 MHz and beyond.

The advantage of IADE is not only in the relative simplicity it brings to the tester domain, through the use of abstraction. It also introduces the potential for intelligent testing, by incorporating design understanding into the tester environment.

2.4.5 Production Line Diagnosis

The proposed model for production line diagnosis is illustrated in Figure 2.8. This model is essentially identical to the model for production testing, with the addition of a diagnostic fault control system which can inject faults selectively into the emulated gate-level device model, now labelled $\mathbf{D_G^n}$.

During normal production testing, $\mathbf{D_G^n}$, which is exercised by the environment is initially free of faults. The DUT and this clean "faultable" model are then exercised in parallel by environmental stimulus. If the comparison signal shows the device outputs of the DUT and the device outputs of the "faultable" model to be identical, then the DUT passes verification. If the comparison signal shows the outputs to be different, then the failing output signature and the test vector at which the outputs differ indicate the area of failure.

At this stage, faults can be sequentially injected into the "faultable" model, and the test set applied to both models for each fault **n**. This process can be used to systematically characterize and localize the fault in the DUT. If the output patterns for the "faultable" model match those for the DUT, that is, when:

$$DUT(E_G(F_E)) \approx D_G^n(E_G(F_E)) \tag{2.14}$$

then the fault which exists in the DUT corresponds to the fault which has been injected in the "faultable" model. The process of injecting faults into the "faultable" model in an intelligent and insightful manner is a crucial area of research in this project. The "intelligence" and usefulness of the test equipment ultimately reflect the effectiveness of its algorithms for identifying routes of inquiry and diagnosis.

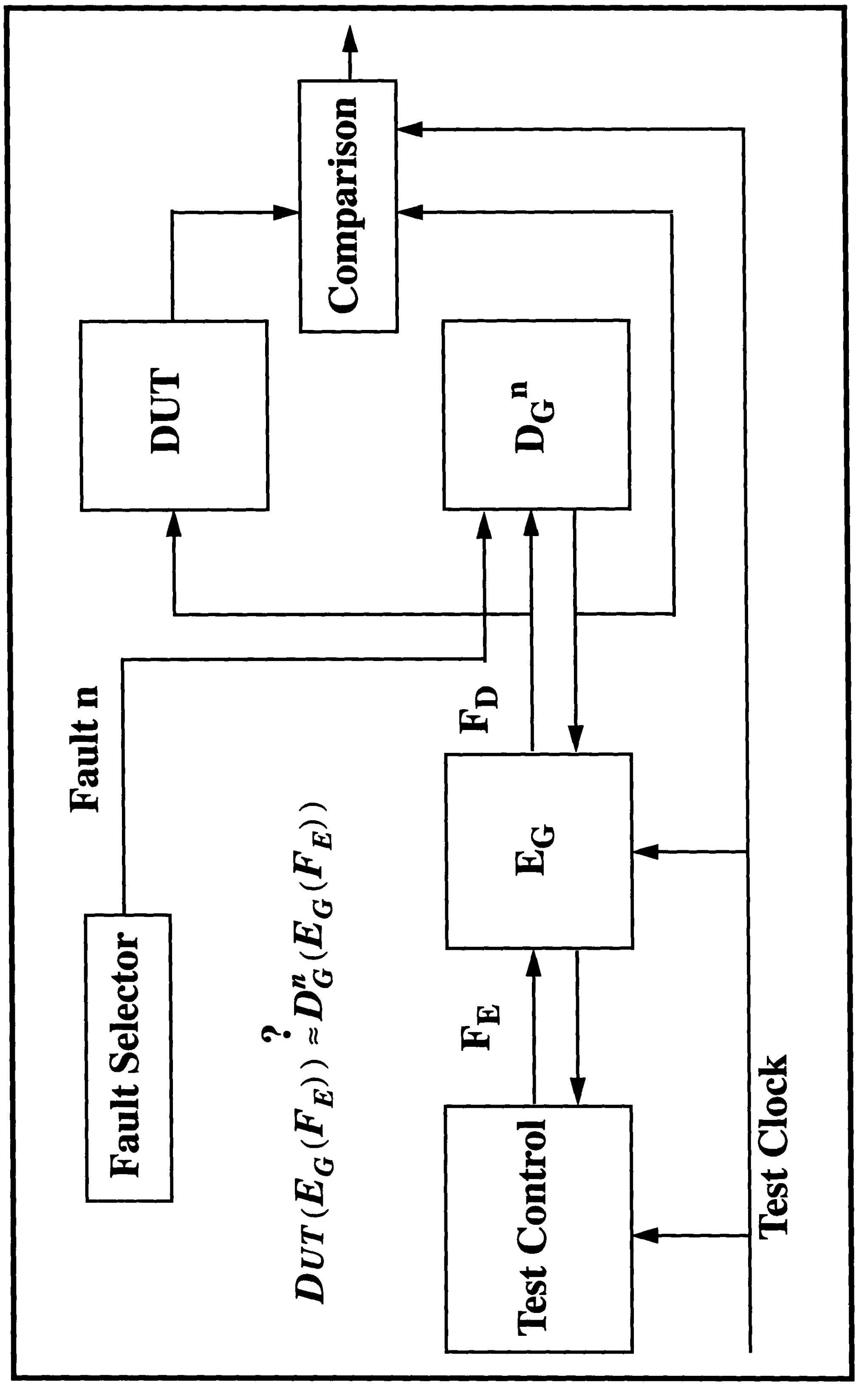

Figure 2.8: IADE Model for Production Line Diagnosis

2.5 Implementing the Proposed Model

Having established the main elements of the proposed design cycle, we faced the challenge of exploring this model effectively given the limited time and resources which constrained our study. We decided that the most straightforward analysis would consist of implementing the model using an existing device as its basis. The device we chose to examine is a custom IC used internally in one of Schlumberger's Automatic Test Equipment (ATE) products. This IC underwent real design, test generation and fault grading, during the period in which we conducted our study. As a result, we were able to follow the development of the actual device in a current design cycle, while simultaneously modeling it in our proposed methodology. We could therefore compare the effectiveness of the two approaches on the same device. As well, we could take advantage of development tasks which had already been completed for the device, such as gate level implementation. Finally, this approach allowed us to test our proposal on actual production parts, and to explore diagnosis on failing parts.

Implementing our proposal required several steps. We were initially given behaviour and gate-level models of the device. We then needed to develop an environment model for the device. Since we could not exercise our system using emulation, it was sufficient to develop a behavioural model for the environment. We then constructed, in simulation, the system illustrated in Figure 2.6. To implement this system, we needed to create a mechanism by which to inject faults into specific signals in the "faultable" device. We also needed to develop the apparatus for comparing the two device models, and a system for logging the results of fault grading. We then would have to develop a functional test suite for the device, and a test control module that could run the entire test set on the system repeatedly. This would allow us to perform fault grading using the proposed system. Once fault grading was performed, we needed to be able to test production parts using our test set. This required interfacing our system to the current ITS9000 Component Test environment in use at Schlumberger, which is the only available system for testing physical parts. Having established this, we would then be able to identify good and bad production parts. FInally, we planned to explore diagnosis algorithms on the S9000 tester, using as examples a production batch of parts given to us by the manufacturer of the actual device.

Each of these tasks is examined in detail in the next few chapters. We describe our implementation, the results we obtained, and any problems or issues which arose in the course of our study.

Chapter 3

Design and Verification

3.1 Introduction

The first step in exploring IADE was to build the device-environment model that would form the emulation module of the system. The behaviour models for the target device, a pair of chips called Drive Control and Response Control, had already been written in Verilog HDL. We needed to establish what we considered to be the appropriate environment abstraction to reflect the functional requirements of this device.

Choosing the appropriate environment model for a device first requires a proper understanding of the device itself: what is its task, and in what range of situations it is expected to perform that task. Clarifying the role of the device in the overall system allows the designer to make this crucial decision: What is the level of abstraction which provides the greatest degree of simplification and relevance while maintaining the flexibility to exercise the device in all circumstances for which it was intended?

3.2 The Sample Circuit

The target device for this environment is the formatter subsystem of the Timing Generator Module on the ITS 9000 Test System. To understand the role of this subsystem it is necessary to understand how the 9000 Test System works. This system is explained in greater detail by West and Napier [13].

The S9000 is based upon a system in which each pin of the device it is testing (DUT) is connected to a drive apparatus and a strobe apparatus. The drive apparatus allows voltages to be applied to the pin, and the strobe apparatus allows measurements of pin voltages to be taken. The process of testing a DUT consists of developing a pattern of drive and strobe events to apply to each pin of the DUT. When timed precisely and applied simultaneously these patterns (a different pattern for each pin) provide a means of observing the functional and timing characteristics of the DUT.

There is a central control system that manages the execution of the overall test program. Each pin of the DUT is controlled independently by its own memory and timing modules, to allow for simultaneous execution of different patterns. This per-pin memory and timing system is housed in the Timing Generator Module.

The Timing Generator Module (TGM) serves the purpose of accessing pattern sequences from memory and applying them to one pin of the DUT on the test head. More specifically, the TGM for each pin, acting in response to stimulus from the central control system, retrieves a pattern from a local cache and converts that into a digital waveform with precise timing to the resolution of 12.5 ps. The waveform that it produces consists of drive and strobe events to apply to the DUT pin it controls. This waveform is sent to a pin electronics module on the test head which converts the events from digital to analog and applies them to the DUT pin.

The TGM is a multi-chip module containing three chips: the Event Logic IC (ELIC), and the two chips which together form our target device, the Response Control IC (RIC) and the Drive Control IC (DIC). RIC and DIC form a subsystem of the TGM. They are controlled by and interface with the ELIC. They also interface with the pin electronics module on the test head.

The operation of the TGM can be defined in two phases. In broad terms, its specifications are as follows: for each event sequence, it receives clocking information, an index into its internal memory, and some other related data. It retrieves the pattern addressed by this index, which consists of drive and strobe events each to occur at a specific time, to the resolution of 12.5 ps. Operating on a master clock of 3.2 ns, it resolves these global times by holding the event to the nearest 3.2 ns step, and then sends the remaining vernier time (within 3.2 ns in 12.5 ps steps), along with the event type information (drives and strobes of different types can be applied) to either DIC or RIC.

DIC handles drive events and outputs to two drivers on the test head. RIC handles strobe events and receives input from two comparators on the test head. Both chips receive the timing vernier and type information, delay the appropriate number of 12.5 ps steps, and apply the drive or strobe event to their outputs (DIC) or inputs (RIC).

RIC and DIC form a compact subsystem which is responsible for executing drive and strobe events within the current 3.2 ns clock cycle to the nearest 12.5 ps step. Together with the ELIC they apply event streams with precise timing to the test head which are the basis of device testing.

The first part of the interface between the ELIC and the formatters is this event stream mechanism, which consists of 4 data busses each for RIC and DIC, on which vernier times and event types are transmitted in pairs of data words.

The second part of the interface between the ELIC and the formatters is the mechanism by which internal registers in the formatters are initialized. These registers serve a number of different functions, from calibrating the analog delay lines inside RIC and DIC, to specifying one of several modes in which the TGM is designed to operate. This interface consists of function bus, data bus and clock lines.

In making the abstraction from a device to its environment, the system designer faces a difficult and seemingly arbitrary choice. In the case of the formatters, the interface between the device and the environment consists simply of those buses and clock lines required for event stream and register setup operations. How far the environment itself should extend, what should be its interface with the system, is a matter of choice. The goal in this case is to make the boundary between the system and the device-environment pair abstract enough that it is not necessary to specify what data is on each event stream bus and register bus on each clock cycle. At the same time, making the interface more and more abstract means incorporating more and more of the system into the environment model. This increases its complexity and unwieldiness. The balance we aimed for was to create an environment that is just powerful enough that its stimulus reflects the correct abstract representation of the behaviour of the formatters, and no more.

In the case of the formatter subsystem, the environment could be specified to perform the role of the ELIC chip for example. This would provide one level of abstraction, making the stimulus to the device-environment model the same as the stimulus to the overall TGM. In fact, the environment for the formatters could be made even more powerful than the TGM, and could contain the TGM within it. In effect, the line between the system and the environment could be placed anywhere from the formatter inputs to the final "start" button that sets the 9000 tester running. It could be correspondingly as simple as a wire connection, or as complex as a full component tester (minus the TGM).

The challenge is to draw that line in such as way that describing the stimulus set that drives the device-environment model is easy and obvious. The conceptual value of the environment model in IADE is that it should perform test generation automatically with the aid of an input stimulus set that is presumed to test the device thoroughly. If the environment and its interface are designed well, creating that complete stimulus set should be very straightforward and it should be easy to verify the completeness of that stimulus set. At the same time, this should be done with the minimum of environment complexity. It is not necessary to build an environment model that is as complex as the tester itself for the sake of making the stimulus set as simple as possible.

3.3 The Environment Model

3.3.1 The Environment-System Interface

For the formatter subsystem of the TGM, we identified what we considered to be a useful interface with the system, one that we felt produced the clearest possible expression of the requirements of the device. In this context, we decided that the environment model need not even be as complicated as the ELIC chip. Instead, it sufficed to produce an environment model with four basic interfaces: register definition, pattern event streams, mux event streams, and dut event streams.

These four interfaces satisfy all of the situations in which the formatters will be used in the S9000 tester: they correspond to register set-up, normal device testing (pattern event streams), device testing under a special mode in which two TGMs are attached to one pin for increased speed (mux event streams), and the returning comparator data from the test head (dut event streams).

The four interfaces are managed in the following way: Each consists of a group of memory files with start and end address markers. There is a control mechanism by which the user can specify which combination of the four interfaces they wish to activate. The user fills the memory files with the events they wish to apply to the test head. Figure 3.1 shows the schematic description of the Formatter Subsystem. Figure 3.2 shows the schematic description of the environment model for the Formatter subsystem. The full system is illustrated in Figure 3.3.

3.3.2 Register Setup

In the case of register setup, there are three memory files. They correspond to the addresses of the registers to be accessed, the function to perform on the registers (read or write), and the data to be placed in the registers on a write.

These three files are accessed in parallel by one start and end address pair. The files themselves may be very large, and may contain many groups of registers for different setup modes. By manipulating the start and end address, the user decides which group to exercise for a given run.

3.3.3 Pattern Event Streams

The interface for pattern event streams consists of two memory files, containing time and type information for all the events in the stream. The time file contains event times referenced to a global starting point, with an lsb resolution of 12.5 ps. The type file contains drive/strobe information. Together these files are accessed with a start and end address pair.

When the model is run, the output at the driver and strober terminals will be the sequence of events specified in that section of the memory files between the start and end address, each event occurring at the time specified in the time file, and of the type specified in the type file.

3.3.4 Mux Event Streams

Mux events are generated when the formatters are used in a particular mode of operation. The TGMs are organized in pairs, so that in one mode, known as pin-mux mode, two TGMs are used to feed one pin on the test head, applying events at double the normal rate. In this mode, the first TGM sends its events not to the test head but to the second TGM, which sends both its own events and the muxed events from its partner to the test head for its pin. The TGM is designed, therefore, to receive both its own pattern event stream and the event stream of its muxed partner.

In order to exercise the formatters properly, the environment model needs to have an interface to this mux event stream. The mux event stream is specified in much the same way as the pattern event stream. The interface consists of a time and type file, with their start and end addresses. When operating in pin-mux mode, the drivers and strobers of the fomatter subsystem should show the combination of pattern events and mux events defined in these two interfaces.

There is one distinction between the pattern events and the mux events. In the real system, mux events that are input to one TGM are the outputs of another TGM, and consequently must have the same timing resolution of 12.5 ps. However, in order to give this device-environment model greater flexibility even than there is on the actual system, we decided to give the mux events a resolution of 1 ps. This relaxation does not break or threaten the validity of the model, because the timing of the mux events are controlled by a different TGM entirely than the one that is being considered in the model. The formatters and environment that receive the mux events have no knowledge of or control over the timing resolution of the mux events. They are simply received and transmitted to the testhead. Therefore, relaxing the timing restrictions in this way is an appropriate modification to make, because it increases the flexibility of the model without violating the specification of the fomatter subsystem itself.

3.3.5 DUT Event Streams

In order for the fomatters (and the TGM) to make use of the drive and strobe events, it is necessary to have events occur on the DUT pin itself. The signals which occur on the DUT pin are passed back to the RIC chip by means of comparator values, achi (above comparator high) and bclo (below comparator low). These values give the test program the feedback it needs to validate the DUT.

In order to properly exercise the formatters, therefore, it is necessary to provide this interface to the environment. The dut interface is similar to those for pattern and mux events, with the exception that instead of event types (such as drive and strobe), it specifies the values of achi and bclo for specific times in the run. Like mux event times, these times have a resolution of 1 ps. The time and value information are specified in two separate files, both of which are accessed by a start and end address pair.

3.3.6 Test Program Flow

Together, the four interfaces, consisting of data files and access addresses, define a user-controlled program of events to apply to the formatter-environment model. The flow of execution can be controlled by means of a test fixture model written in Verilog HDL which determines how these environment inputs change with time. This test fixture is really part of the environment model, since it is required to manage test sequences which are too complex to be described with a single start-end block in memory.

One final layer of abstraction exists for the event stream interfaces. We wrote a conversion program that takes in event streams in a higher-level syntax and converts them into the individual event time and type files required by the environment model. This served only to simplify further the task of listing event sequences. Instead of manually creating binary entries with time and type values for each of the event streams, we could now specify a list of events such as "D1@3.2ns", and have them converted and filed automatically. One could easily envision building more sophisticated, graphical user interfaces to the environment.

Once the environment interface is specified and the environment model itself built, the power of the abstraction becomes apparent. In order to exercise the formatters, it now becomes a matter of writing the files containing register setups and patterns of events to apply to the test head, and simply identifying those block of events in the environment test fixture. In contrast, if the abstraction had not been made, it would have been necessary to specify, cycle by cycle, the values to place on the data busses and wires that form the inputs to the formatters. In our case, this work is done automatically by the environment.

3.4 Results

The importance of our environment abstraction was not immediately apparent to us during our implementation of the proposed model. As we proceeded further in the process, through test generation and fault grading, it became clear to us how substantially choices we had made in developing the environment model affected our entire development process. Several times during our study we were given pause to reflect upon the difficult balance between simplicity of the interface and flexibility of the overall system

We learned several lessons in implementing our environment model. First, we realized that specifying the various interfaces as a series of memory files made the task of test generation more unwieldy than we had anticipated. Exercising a large test set, such as the functional test set used for production, proved to be quite complex. We had envisioned a test program consisting of blocks of events for each of the four interfaces, indexed by start and end addresses. As it turned out, this is not the way in which typical device behaviour takes place. Instead, there are alternating streams of events from one or more of the four interfaces; the four interfaces are never exercised simultaneously. Moreover, there are frequently several sets of events for each interface, which are exercised during different parts of the test program. Fitting behaviour of this nature to our environment interface required a fairly detailed and unique test control fixture for each test set. This made modifying the test set more time-consuming than we expected.

In contrast, our decision to specify events using descriptors such as "D1@3.2ns" proved to be extremely efficient. Making this abstraction saved us an enormous amount of effort, particularly since the functional test set frequently contained repeated cycles of events. Overall, we were astonished by the extent to which the environment model simplified the task of functional test generation. Without this abstraction, we could clearly not have developed a production test set for the device as cleanly or efficiently.

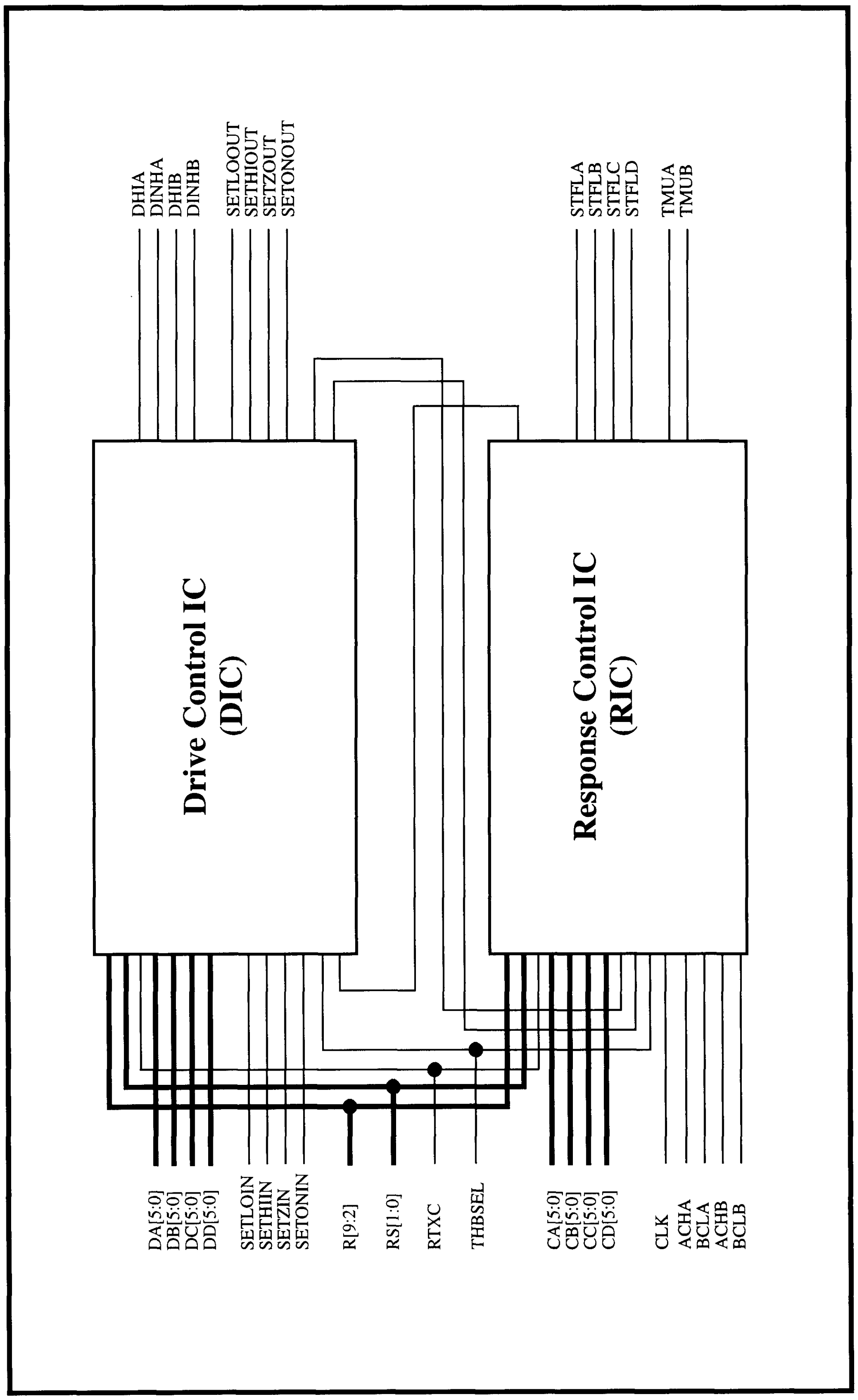

Figure 3.1: Schematic of Formatter System

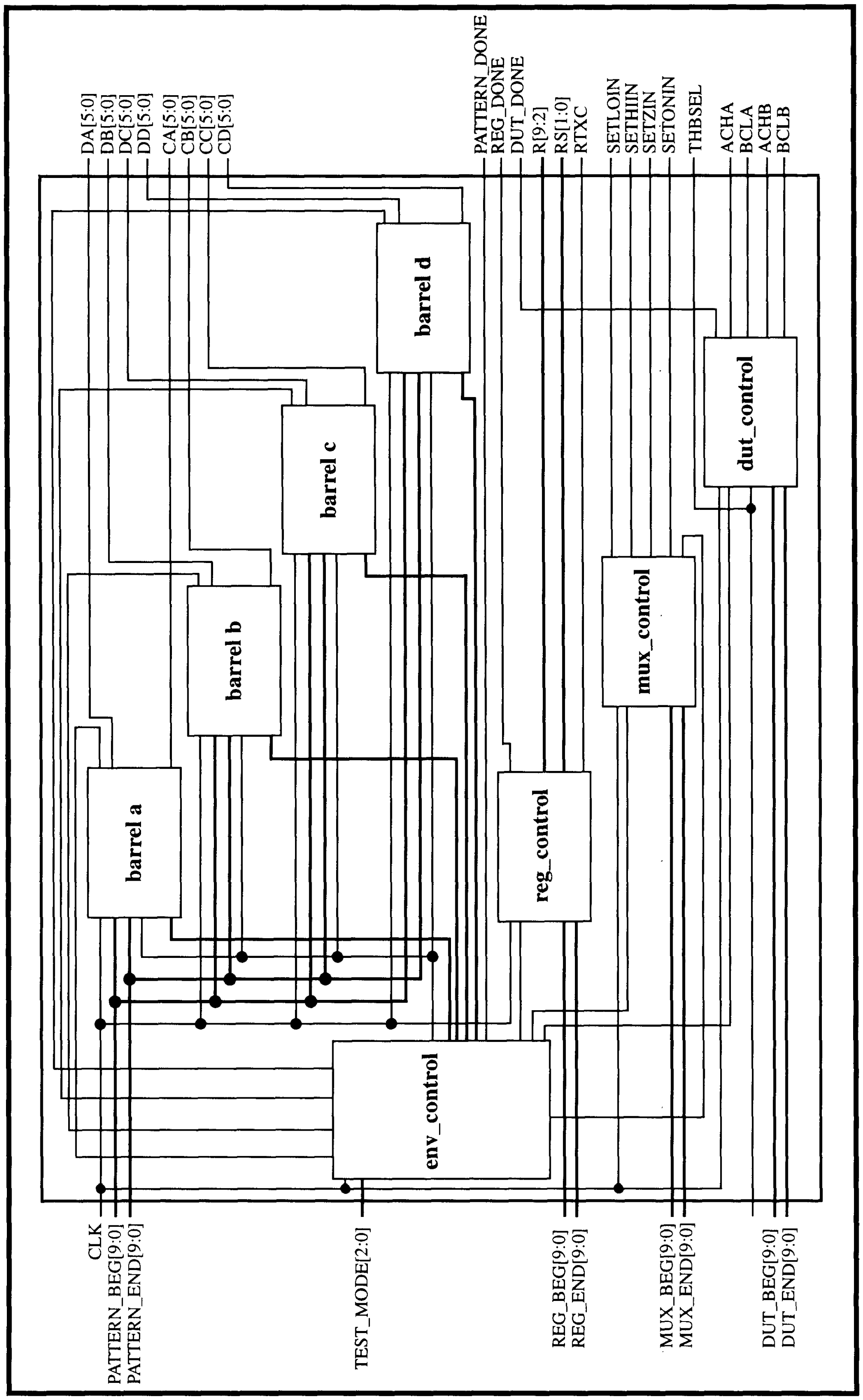

Figure 3.2: Schematic of Environment Model

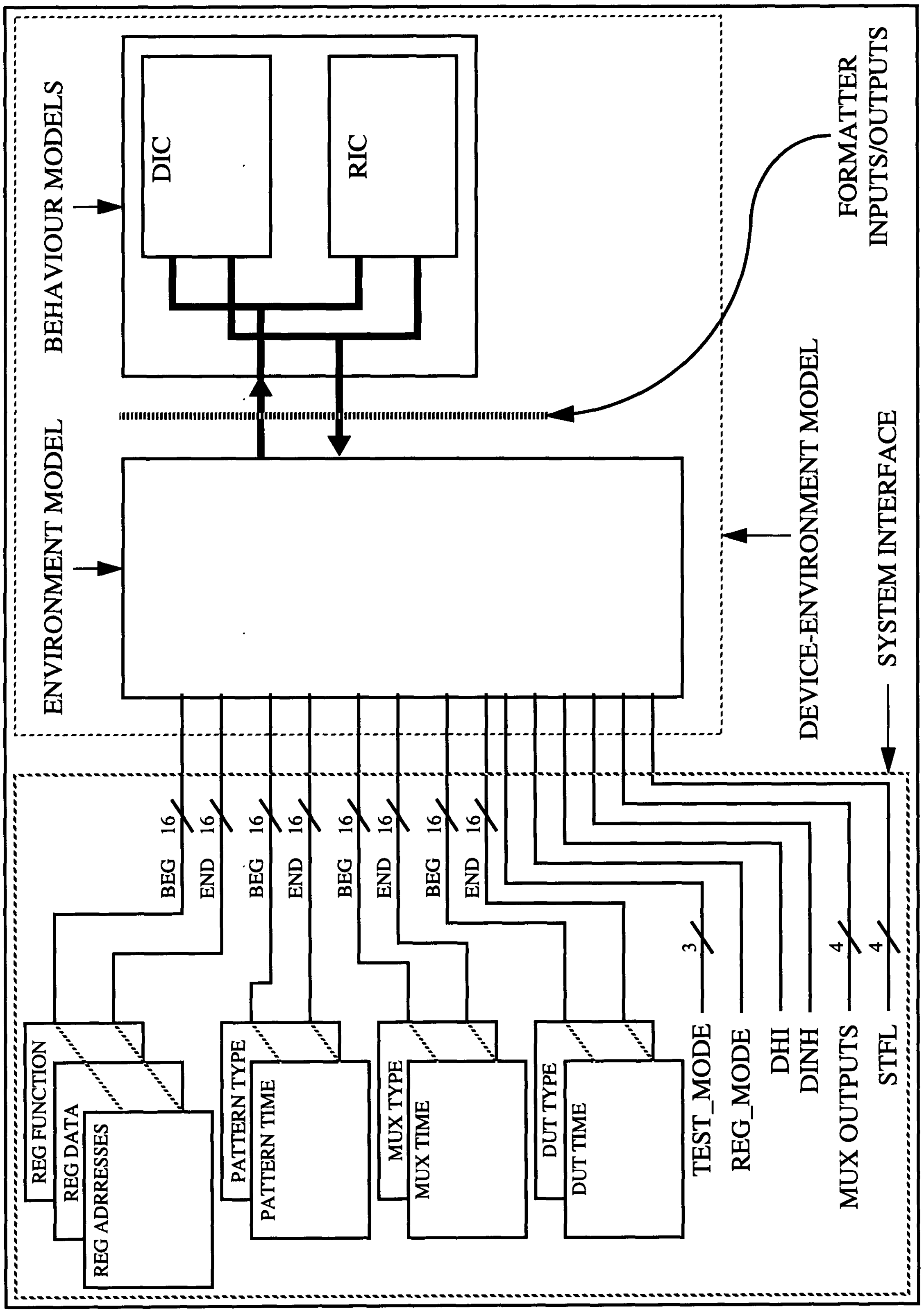

Figure 3.3: The Environment-Formatter System

Chapter 4

Test Generation and Fault Grading

4.1 Introduction

The process of test generation for a device has two goals: first, to write a test set that identifies as many failing devices as possible, as accurately as possible; and second, to measure the quality of that test. Test quality is gauged by a process known as fault grading, which measures the percentage of possible faults detected by the test set, for a given fault model. Fault grading is important because it reflects the balance in testing between effort and its marginal return. It is generally the case in production testing that a small test suite will detect a large percentage of possible faults. Adding the tests required to detect the last few faults can sometimes double or triple the size of this small test suite. As a consequence, developers target a particular percentage of faults they consider acceptable for the production test set to detect. They then build the test set for the device incrementally, adding tests as required it achieves that level of coverage.

The goal in the proposed development process was to approach test generation from a different perspective - one that is based upon functionality rather than structure. Developing an environment model for the device was conceived to facilitate this task by allowing the device to be manipulated with a high-level, functionally defined interface. Fault grading would be performed using the single-stuck-at input fault model. A good analysis of this fault model and of modeling in general is provided by Abramovici [1].

Exploring the proposed model without purchasing a commercial emulation system required compromising some of our objectives. We decided that it would nevertheless be valuable to develop the entire model for fault-grading in simulation, building each component at gate-level so that it could be implemented in emulation without conversion. Running this system in simulation would at least give us some insight into issues raised by the proposed model. Performance could be estimated broadly by comparing the clockspeeds of the two systems.

Implementing the production testing and diagnosis phases without the use of emulation required mapping the proposed model to the current test equipment we had available to us. The system we used was the Schlumberger S9000 FX chip tester. The architecture of this tester is such that programs applied to the device under test contain both applied inputs and expected outputs for each pin. By contrast, the proposed model does not contain expected data for the device under test, and applies test vectors at a higher level. Our solution was as follows:

We executed our device-environment system in simulation, and recorded the resulting device-level input/output behaviour as the applied/expected data in an S9000 test program. We believe this is equivalent to executing the system fully in real time. This method allowed us to use our environment model and environment-level test set to exercise real production parts and perform real diagnosis.

4.2 Test Generation

Implementing the proposed model for the sample circuit (the Formatter subsystem) required developing a functionally-based test set to apply to the system at environment level, both for fault grading and for production testing. Deriving a functional test set for the device requires intimate knowledge of the behaviour specification for the device, as well as an understanding of the system in which that device resides. Despite the benefit of the environment model abstraction, functional test generation is not a simple task.

Fortunately, the design group which developed the sample circuit had already developed a functional test set for the device, albeit designed to operate at the device level itself rather than the environment level. This test set was designed for the regular production testing of the device, which was already entering first fabrication when we began to evaluate the proposed model for fault grading. We decided, in the interest of efficiency, to use this test set as a model for our test set.

Deriving our environment level test set involved analyzing the vectors in this existing test set, choosing those which were functionally based, converting them to environment level stimuli, and discarding any vectors which depended upon structural understanding of the design. This process occasionally identified weaknesses in our choice of environment abstraction - the situation sometimes occurred that we could not manipulate our environment as flexibly as the device itself had been manipulated at the lower level by the production test set. In most cases, however, this was because the test engineer had exercised the device in a non-standard way to take advantage of opportunities for optimization. Although these special cases improved the performance of the test set, they violated the bounds of normal device behaviour in order to do so. We implemented these special vectors within the bounds of normal device operation (which was enforced in some cases, by our environment interface), and consequently achieved the same final results, albeit somewhat less efficiently.

4.2.1 Issues in Test Generation

The test generation process raised several issues which need to be addressed in implementing the proposed model. First, does functional verification suffice for production testing of a device? For the purposes of this evaluation, we restricted our testing of the device to functionality and ignored timing verification. Because emulation systems cannot provide proper modeling of timing constraints in circuits, we envisioned using simulation to address timing issues. Our aim was to off-load as much testing as possible to the emulation environment, retaining simulation only in those situations for which it was indispensable. Ackerman et al. [2] describe a techniques in which emulation and simulation are combined in production testing to maximize efficiency. Although this technique seems feasible, we did not implement a system which combined both technologies to provide complete device verification. Therefore this area needs further exploration before the proposed model can fully be implemented.

A difficulty we encountered in implementing the proposed model was in bundling up the input/output device data as an S9000 test program. We performed the tasks in the following sequence: first, we developed the test set; next, we performed fault-grading in simulation; and finally, we converted the test set to an S9000 program and performed production testing and diagnosis. We later discovered that we should have finished the S9000 conversion before performing fault-grading. The reason is that we were forced to make changes to our production test set in order for it to work properly on the actual tester. These changes reflected two problems: First, we performed tests in simulation which could not be executed in the physical tester, due to its own technological constraints. For example, we were able to apply complex timing sequences to the part which ran fine in simulation, and which theoretically could be applied to the device in normal use, but which could not be exercised by the test equipment. The second problem was that the strobing apparatus we developed for simulation fault-grading consisted of simple comparison of signals at a set strobing period in each clock cycle. In reality, however, the strobing mechanisms required to exercise production parts are far more complex. Specifying all of those mechanisms properly so that the production test worked required changing some vectors. The final production test set, which correctly passed a working device, was different enough from the test set used for fault-grading, that it became necessary to redo fault-grading. It is unclear whether this type of problem will necessitate verifying the production test before doing fault-grading in the proposed model. Because we have not actually implemented the proposed test architecture, this problem remains unaddressed.

4.3 Fault Grading

The task of fault-grading our test set in the proposed system consists of several steps. We began by constructing the system illustrated in Figure2.6, in which the environment model is applied in parallel to two instances of the device model, one of which we designed as "faultable".

Once we had chosen our fault model to be the single-stuck-at input model, we needed a mechanism for injecting these faults into the gate-level model of the device. Our next task was to implement the system in which two device models were executed in parallel. We also needed to develop the strobing apparatus for the system, and the means to summarize fault-grading results. We chose to develop a fault dictionary for future use in diagnosis; we had to develop a mechanism to create this as well. Finally, we faced issues of restarting and initialization in testing, as well as overall performance of our system, which became significant, given our need to run it in simulation.

4.4 The Fault Model

We chose the single-stuck-at fault model primarily for its simplicity and ease of implementation. The principle in this model is that a single signal may be shorted or grounded at any point within a production circuit, causing functional failure. There is a crucial distinction to be made between failure in input signals to gates, and failure in output signals from gates. We initially built a model in which the output of each gate might be stuck at one or stuck at zero. Since our target device contained approximately 500 gates, this produced a fault set of approximately 1000 different potential causes of failure. We began fault grading, and found the results to be very good. Our test set seemed to have no difficulty in identifying gate-output faults. Clearly, however, this fault model is very simplistic. Faulting the output of a single gate may result in input faults to any number of gates, depending upon fan-out. Consequently, gate-output faults are more likely to cause large functionality problems in the circuit, and correspondingly easier to detect in testing. After having run the gate-output fault simulation for some time, we decided that this fault model was too narrow to be of any use in fault diagnosis. We decided instead to change our fault model to consider input signals to gates, rather than output signals from gates. This required considerable changes to our system for injecting faults and re-applying the test set to the device. Nevertheless, it provides a more realistic (although still very narrow) fault model for production testing.

4.5 Modeling a Faulted Signal

The gate-level model we used for our sample device was built and simulated using the Verilog Hardware Definition Language (VHDL), and used as its basis a standard library of gate models written in Verilog. We decided that the cleanest method for injecting a fault into a particular signal in the circuit was to consider the circuit as a network of gates. Any interconnect in the system could then be manipulated as an input line to a particular gate. We modified the model of each gate to permit any one of its inputs to be stuck at one or stuck at zero. Figure 4.1 shows the abstract gate model used in this implementation.

The gate is then controlled as follows: if the enable bit is set to one, the gate behaves as if it were faulted, in the manner prescribed by the fault select code. The number of bits in the fault select code must be sufficient to encode all possible faults for all possible inputs, given the fault model being considered. In our case, the largest number of inputs to any single gate in the device was 12. Correspondingly, our fault select code needed five bits to

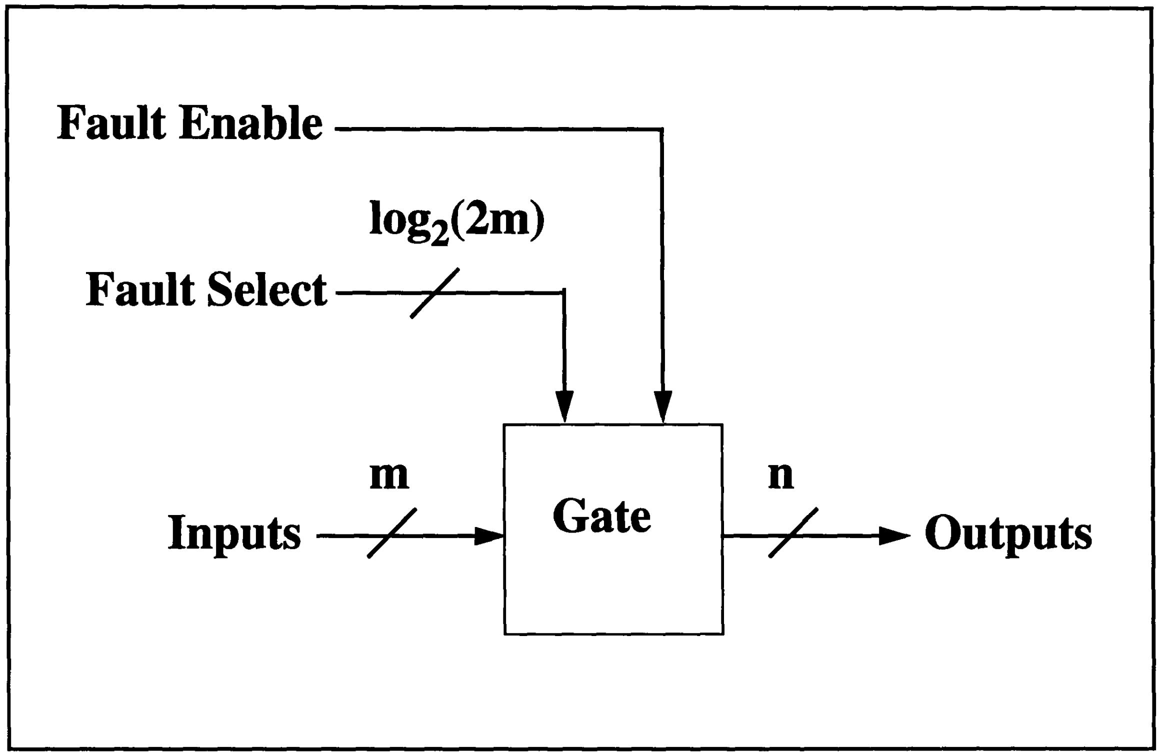

Figure 4.1: Model for a Faultable Gate

encode the 24 possible faults (stuck-at-one and stuck-at-zero for each input line) we might inject into a gate of that size. Table 4.1 shows the fault select coding used in our system.

The fault select bus is fed to the device models, and connected as an input to each gate in the system. Each gate also receives a distinct enable line as an input. The fault control system is then manipulated by choosing the gate to be faulted, and setting that enable line to one. The fault select code then determines how the chosen gate will be faulted.

Choosing to implement the faultable gates in this manner restricted our flexibility to manipulate the system somewhat. For example, our faultable gate model does not permit two signals in a given gate to be faulted simultaneously. As a consequence, we could not modify our fault model for test set validation without redesigning the gate library. For the purposes of this study, we felt that it was sufficient to implement the gate library for one particular fault model. Nevertheless, implementing other, broader fault models using this

Fault Select Code	Type Of Fault
0	Unfaulted
1	Input 1 stuck at zero
2	Input 1 stuck at one
2n-1	Input "n" stuck at zero
2n	Input "n" stuck at one

Table 4.1: Fault Select Coding

structure needs to be explored properly before effective fault grading can be performed in the proposed model. Koeppe [6] and Namitz et al. [8] describe fault insertion mechanisms for broader classes of faults.

4.6 Control Apparatus

The control mechanism reflects a second choice we faced in designing our system. Our target device has roughly 500 gates. We therefore constructed, at the system level, a 10-bit gate select input which contains the index of the gate which is to be faulted. This signal is propagated to the device level, where it is converted to a 1024-bit bus, a distinct line of which is fed to each gate in the device. The bus is controlled so that all of its lines are set to zero, except the line corresponding to the index of the 10-bit gate select input. In this way, the high-level gate select mechanism is used to enable the control line for the chosen gate within the device. This mechanism is clearly not practical for injecting faults into devices with high gate-counts. Moreover, in emulation, a faultable gate will be implemented not by a single gate, but by a group of programmable gates. Controlling the injection of faults in such a system efficiently will be a key issue in implementing the proposed model.

Once the apparatus was in place for injecting a single stuck-at fault into a given signal in the system, we needed to build a test control harness which applied the entire production test set to the two device models, strobing and comparing their output signals. We chose to strobe the signals of the models at 95% of each clock cycle. At this point in every clock period, the output lines of the two devices were compared. If the two sets of signals differed in any way, the failure was logged and the test run could be aborted.

The next level of control was the mechanism by which individual faults were injected sequentially into the "faultable" device, the test set applied, and outputs compared for each fault. For each fault, this fault grading harness applied the test set to both devices until a difference in outputs was detected. At that point, the detection was logged, a new fault was injected, and the test set was re-started. If the test set completed without any difference being detected, then this fact was logged, a new fault was injected, and the test set was re-started.

Although the specified clockspeed at which our target device is designed to run is 312.5 MHz, we chose to run our fault grading simulation with a device clockspeed of 100 MHz. The reason for this is that we wanted to be certain that the device exhibits *static behaviour*. This means that, in response to inputs on a given clock cycle, the device will reach steady state before the next clock cycle.

The issue of static behaviour has tremendous relevance when dealing with emulation technology. The issue is this: if emulation cannot be made to run as fast as a device's intended clockspeed, can fault grading be performed at the emulation speed? If a device is specified at 100 MHz, can fault grading be performed using emulation at 5 MHz? This can only be done if there is no difference in the functional signature of the device, at 100 MHz or at 5 MHz. This can only be true if the clock period does not begin to crowd into the propagation delays for the device. If that happens, as is the case with pipelined devices, then the device will not exhibit static behaviour: that is, it will have different signatures at 100 MHz and at 5 MHz. In such a case, fault grading clearly cannot be performed using emulation, since the basis of the process is output strobing at a fixed percentage of the clock period. This issue is a highly complex one, and needs to be analyzed in greater detail if current emulation technology is to be used in the proposed model. It will disappear only when emulation technology has advanced sufficiently to permit devices to be emulated at their specified clockspeeds.

4.7 Initialization

Creating this fault grading control system required understanding the re-start and initialization issues involved with the device. Initially, we found it difficult to implement the system in such a way that each test run is independent from its predecessors and starts from a known state. There are two aspects to this: First, we need to ensure that there exists a way to force the initial internal state of the device to a known configuration. Second, we need to verify that, on the initial test run, and on every test run thereafter, this initial state is achieved before testing begins.

We developed, therefore, an initialization sequence which resets all of the device's state and can be verified at the beginning of each test run, by strobing the internal and external state of the device and datalogging the result. During this initialization period, the output strobing and comparison of the two devices is disabled to prevent premature abortion. When we implemented this system and began the fault grading, we found that our test runs seemed to behave independently of one another.

This result can be misleading, however. We identified a few test runs in which the expected initialization state was *not* achieved when strobing began; this was because the faults injected in the device during those test runs were involved in the initialization sequence itself, and therefore *guaranteed* that the device would not initialize properly.

How can this problem be circumvented? The paradox here is that we would like to consider the initialization sequence outside the domain of the fault injected in the device. However, a fault which affects other normal device behaviour may clearly also hamper the activities which are performed during initialization. A production part with such a fault would initialize improperly just as the model hypothesized in fault grading. By developing and executing a clear initialization sequence, however, we can address the problem of initial state consistently for most situations.

A second, more troubling challenge is the strobing of internal state to verify successful initialization. Strobing the initial state for our device required taking advantage of the accessibility of the internal device design structure through normal behaviour. How simple or feasible this may be for an arbitrary design is unclear. This issue must, however, be addressed in the context of each individual device implemented using the proposed model.

4.8 Reporting

We condensed the reporting for our fault grading phase into two groups: internal and external state for each test run; and fault grading results for each test run. The former consists of state values for the device after each initialization sequence. The latter was formatted as a fault dictionary: there is one entry for each fault injected into the "faultable" device. For each such entry, the dictionary states whether or not the test set identified the fault, and if it did, the details of the detection point - first fail vector and fail signature. Figure 4.2 shows a sample set of entries from the fault dictionary generated for the device. The highlighted columns indicate the fail vector and signature entries which are used later for diagnosis of failing production parts.

Gate	Total Possible Faults	Fault	Failing Vector	Pass/ Fail	Failing Test	Drive	Setout	STFL	TMU	RBus
256	8	1	6098	pass						
256	8	2	536	fail	25	0000	0000	0000	00	000000H0
256	8	3	6098	pass						
256	8	4	536	fail	25	0000	0000	0000	00	000000H0
256	8	5	630	fail	25	1100	LLLL	0000	00	00000000
256	8	6	536	fail	25	0000	0000	0000	00	000000H0
256	8	7	654	fail	25	1100	LLLL	0000	00	00001010
256	8	8	536	fail	25	0000	0000	0000	00	000000H0
257	8	1	2338	fail	26	0010	0X00	0000	11	11111111
257	8	2	157	fail	1	x000	0X00	0000	00	11111111
257	8	3	157	fail	1	x000	0X00	0000	00	11111111
257	8	4	2338	fail	26	1000	0X00	0000	11	11111111
257	8	5	91	fail	1	X000	0000	0000	00	11111111
257	8	6	157	fail	1	x000	0X00	0000	00	11111111
257	8	7	157	fail	1	x000	0X00	0000	00	11111111
257	8	8	91	fail	1	X000	0000	0000	00	11111111

Table 4.2: Sample Entries from Fault Dictionary

The coding for the fail signature is condensed in the following way. Expected values are the values shown by the clean device; measured values are those for the faulted device. '1' indicates that '1' was both expected and measured. The same is true of '0'. 'L' indicates that '1' was expected and, instead, '0' was measured. 'H' indicates that '0' was expected and, instead, '1' was measured. 'x' indicates that '1' was expected, but the measured value was 'undefined'. 'X' indicates that '0' was expected, but the measured value was 'undefined'.

4.9 Modeling Issues

During the fault grading process, we discovered a serious problem in our faultable gate models: some of the faults which should clearly have caused device failure were passing undetected by the test set. We had tested half of the potential faults and our coverage so far was only around 50%. We were very disappointed by this figure, because we expected coverage of at least 75% for our test set.

The only way to understand why the faults were not identified was to trace through the model from the faulted signals to the device outputs during a particular phase of testing. Doing this for a few of our mystery faults identified the source of error: our faultable gate models were inadequate for several of the gates in our library. Specifically, we did not correctly model the latches and flip-flops in the gate library so that undetermined inputs, and transitions involving undetermined states were not propagated properly to their outputs.

This highlights the importance of modeling in the process of design verification and fault grading. When we re-examined the original gate models which had been constructed for the actual design and implementation of our target device, we were surprised by their crudeness. There was no modeling of transitions to or from undetermined states in the models. We analyzed each gate model once more, using transition tables to verify that every possible input situation had been accounted for.

When we re-ran the fault grading simulation using the more comprehensive gate models, we found that a large percentage of faults which previously had been undetected were now propagated to the device outputs and identified by our test set.

An issue which we have not been able to explore is how undetermined states will be addressed in logic emulation. Emulation clearly cannot model undetermined states; the impact this will have on fault grading is unclear. If our experience with this simulation is any indication, the fault grading process in emulation will produce coverage results which lie somewhere between our initial results and those we obtained after modeling undetermined states rigorously. This is because our initial model ignored all transitions involving undetermined states. As a result, a fault which makes a signal undetermined is never recognized in fault grading. Our final model assumed that every signal which becomes undetermined reflects a fault, and detects all such situations. In real hardware, there is no "undetermined" state for a signal. It will either be correct or incorrect. Consequently, it will sometimes propagate out as an error, and at other times propagate out as the correct signal (by chance). The faults which propagate out as errors contribute to fault coverage. Therefore, real emulation coverage measurements will be neither as imprecise as our first model, nor as precise as our final model.

This issue needs to be considered more carefully in assessing the feasibility of emulation as a tool for proper fault grading. Perhaps a combination of simulation (which is already required for timing verification) and emulation will solve the problem.

4.10 Performance

By the time we finished the structures for fault injection, test control, and fault grading, we had built a fairly substantial simulation model in Verilog HDL, containing the environment and device models, and using a test set that was several thousand cycles in length. Fault grading required applying this test set to the system for each possible fault in our fault model. Given the size of our device, there are 3,616 possible single-stuck-at faults for our system.

The first attempt at fault grading resulted in unacceptable system performance. Running on a Sun SparcStation 10 workstation, the model took roughly 24 minutes to complete a single test set run. Running this simulation several thousand times would take

Category	Gross Figure (3616 Faults)	Average Per Fault
Simulation Length (time)	121,795,130,000 ps	33,682,283 ps
Device Clock Period	10 ns	10 ns
Simulation Length (cycles)	12,179,513 cycles	3368 cycles
Real System Time	45:16:12	0:45
Simulation Speed	74.7 Hz	74.7 Hz
Emulation Speed	5 MHz	5 MHz
Projected Real System Time in Emulation	2.436 s	673 μs

Table 4.3: Performance Measurements for Fault Grading Process

roughly 1400 hours of computation in the worst case, and would clearly not be feasible given the time constraints of this study. As well, we anticipated, inevitably, the need to make changes to the model after initial testing.

Considerable effort was spent, therefore, in profiling the Verilog model during execution in order to identify areas for optimization. Through a number of changes, including optimization of the control harness and test set modifications necessitated by production testing requirements, we were ultimately able to reduce the per-test execution time to roughly three minutes, running alone on a Sun SparcStation 10. This brought the task within reasonable bounds. In fact, since the test set was not fully executed in the common case (when failure was detected, and the test aborted), we were able to complete the final fault grading simulation in roughly 46 hours of system time on the Sparc 10. This simulation executed at roughly 80 device cycles per second.

Estimating the emulation performance of this process is straightforward once an emulation clockspeed is selected. Our fault grading system recorded the number of device clock cycles taken by the simulation, which can then be assessed in real terms using any emulation clockspeed we choose to assume. Current commercial emulation systems operate at approximately 5 MHz. Given this conservative estimate, the time required to perform the entire fault grading process for our device would be roughly 2.4 seconds. Table 4.3 shows the performance results achieved in the final simulation, compared with projected performance for emulation, at a speed of 5 MHz.

4.11 Results

Our fault grading results were slightly better than we anticipated. The final fault coverage measured for our production test set using the single-stuck-at fault model was 80.5%. We ran our simulation in sections, because we were able to use two or three machines in parallel overnight. We tried to minimize this partitioning because each Verilog simulation incurs a certain performance overhead. Table 4.4 shows our coverage results for each of the simulations we executed.

From this table, it can be seen that certain gate numbers were not tested. Why is this so? It is because we were able to use the proposed functionally-based fault grading model to identify redundant gates within our sample circuit! In the first few simulations we attempted using the fault grading model, we were puzzled by a group of gates which clearly were not identified by the test set, and which nevertheless did not appear to cause functional failure of the device. Upon closer inspection, it became clear that these gates were completely unrelated to the device's functionality. They were inserted in the HDL gate-level models in order to facilitate the fabrication of the two IC's which form our target device, Response Control (RIC) and Drive Control (DIC). Since RIC and DIC share a similar structure with some variations, their designers used certain "dummy" gates in both models that shared the same number of ports. Internally, these gates simply contained wire

Gates Tested	Total Tests	Passes	Fails	Coverage
6-54	140	32	108	77.1%
72-74	12	0	12	100%
79-119	164	11	153	93.3%
124-126	12	4	8	66.7%
144-350	1440	356	1084	75.3%
351-460	996	176	820	82.3%
461-516	852	125	727	85.3%
TOTALS	3616	704	2912	80.5%

Table 4.4: Fault Grading Results

connections between their input and output ports which were different for RIC and for DIC. Effectively, these gates would disappear in fabrication, and contained no logic at all. They simply allowed the designers to use a common mask for RIC and DIC, while maintaining different wiring for each chip. They are perfect (if unrealistic) examples of "redundant" circuitry in a design; we were surprised and pleased to find these gates in our device. Our fault grading process, based upon a functional test set, attempted to inject faults into these gates. When our test set was unable to identify them, it led us to discover that these gates were unused in the functional behaviour of the device. Therefore, our fault grading succeeded, in a modest way, in identifying redundancy in our design.

As this section describes, however, there are clearly several issues to be addressed in any realistic implementation of the proposed fault grading model using logic emulation. Whether it will prove to be as successful or practical as our model in a commercial emulation-based process still remains to be resolved.

Chapter 5

Production Line Testing

5.1 Introduction

Implementing the proposed model for production testing presented a difficult challenge. The IADE production test environment consists of a software test control module, a logic emulation system and a test head with pin electronics to interface with physical devices. We faced two constraints: we could not afford to incorporate an emulation system into our study; and the only equipment with pin electronics we could use was an existing Schlumberger component tester, the S9000. We needed, therefore, to use the pin electronics of the S9000 to test actual production parts of the sample circuit in the manner proposed in the IADE methodology. To understand the way in which we achieved this task, it is necessary first to understand the current S9000 test environment, and the relationship it bears to the proposed model. This section describes the S9000 production test architecture, the way in which we interfaced our model to it, and the results we obtained for this phase of the proposed process.

5.2 The S9000 Test Program Architecture

The IADE model for production test requires a test control to feed stimulus to the environment model of the device, which in turn translates this stimulus to device-level test vectors, which are applied via pin electronics to the physical Device Under Test (DUT). Outputs of the DUT are compared with the outputs of the emulated "clean" device model, which is exercised in parallel with the DUT. The current S9000 production test architecture is essentially a subsection of this model, excluding only the environment model and second, "clean," device model (and the emulation system in which they are implemented).

Instead, the device is stimulated directly using a test program written using the S9000 tester software. This test set is applied via pin electronics to the physical DUT. The output data which should return from the DUT is not generated by a second device model, but is instead incorporated into the test program as "expect data."

A straightforward way to integrate our simulated device-environment system with the current S9000 architecture would then proceed as follows: We begin by applying our environment stimulus set to the device-environment system in simulation. We then record the device-level input/output behaviour of the resulting simulation. The device input set is in effect equivalent to the device stimulus program which would directly have been applied to the DUT, and the device output set is equivalent to the "expect data" which would have also been incorporated into the test program.

Bundling these vectors together, and formatting them to correspond to a normal S9000 test program, has the effect of capturing the data generated by our IADE implementation, and converting it to a format compatible with the current test environment. As a result, we could convert our environment-level production test set into a device-level production test program that could be applied using current test head electronics to physical parts.

5.3 Issues in Conversion

We wrote a conversion program that filtered data from the simulation model built earlier, and produced a test program that could be executed on the S9000 tester. As described in section 4.2.1, two main issues surfaced in performing this conversion: first, the timing limitations of the S9000 architecture made it impossible to perform certain test activities which has been possible in simulation; and second, the complexity of the S9000 strobing apparatus forced us to re-organize some parts of the test program.

Addressing these two problems, and formatting the test set to run properly as a test program took some effort. This was partly due to the complexity associated with moving from the simplistic simulation environment to the physical world, and the scope and flexibility of the S9000 software, which offers the capability to perform much more sophisticated testing than we required. Despite the simplicity of our device and test, we needed to develop a fairly large test structure to specify all of the possible control parameters in an S9000 program.

Once we had developed familiarity with this environment, it was necessary to make adjustments to the test program until it correctly performed the behaviour we had observed in the simulation environment. When this had been verified, we tested a production part that was known to be good, to confirm that it also passed our test set. We then turned to the set of 17 parts which had been provided to us by the manufacturer of the Formatter ICs.

5.4 Testing the Sample Circuit

In order to understand diagnosis issues within the proposed environment, we chose to apply the various diagnosis methods described to production parts of a sample circuit. The circuit used for this analysis is the TGM Formatter subsystem described in Section 3. The Formatter subsystem, consisting of the Response Control IC (RIC) and the Drive Control IC (DIC), lies on the TGM multi-chip module along with one other device, the Event Logic IC (ELIC), as shown in Figure 6.1.

For the purpose of this study, we chose to consider the RIC and DIC subsystem as our target device, and remove the ELIC from the TGM module, pulling its input/output pins to the edge of the module. This new chip, known as the Formatter Characterization Module, is shown in Figure 6.2. We were provided with an unmarked batch of 17 good and bad

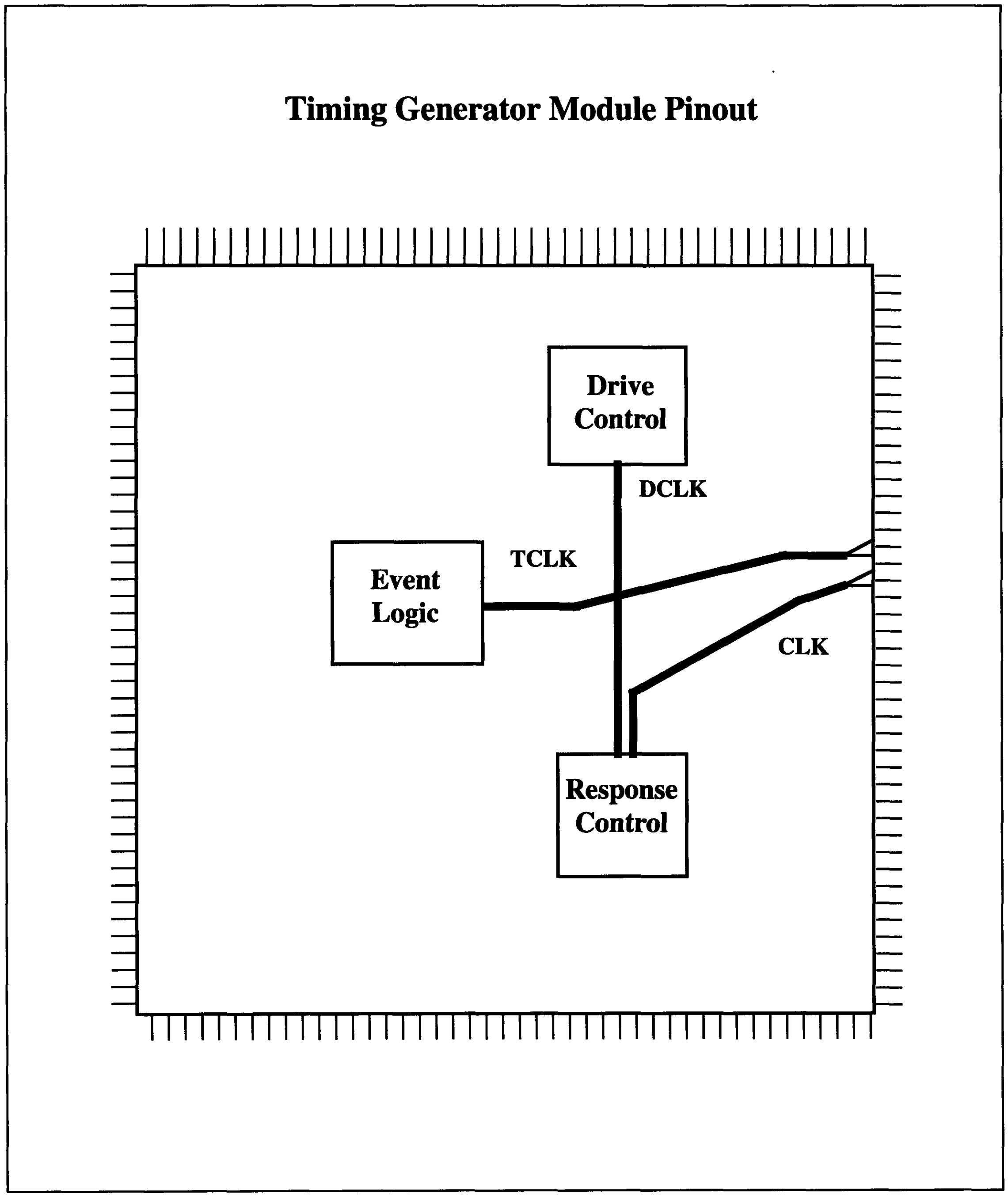

Figure 5.1: Timing Generator Module

production parts. The task was to use the test set developed during the fault grading phase of the study to test these parts and correctly identify the good and bad devices. For those devices which we identified as faulty, the next task was to explore various diagnosis

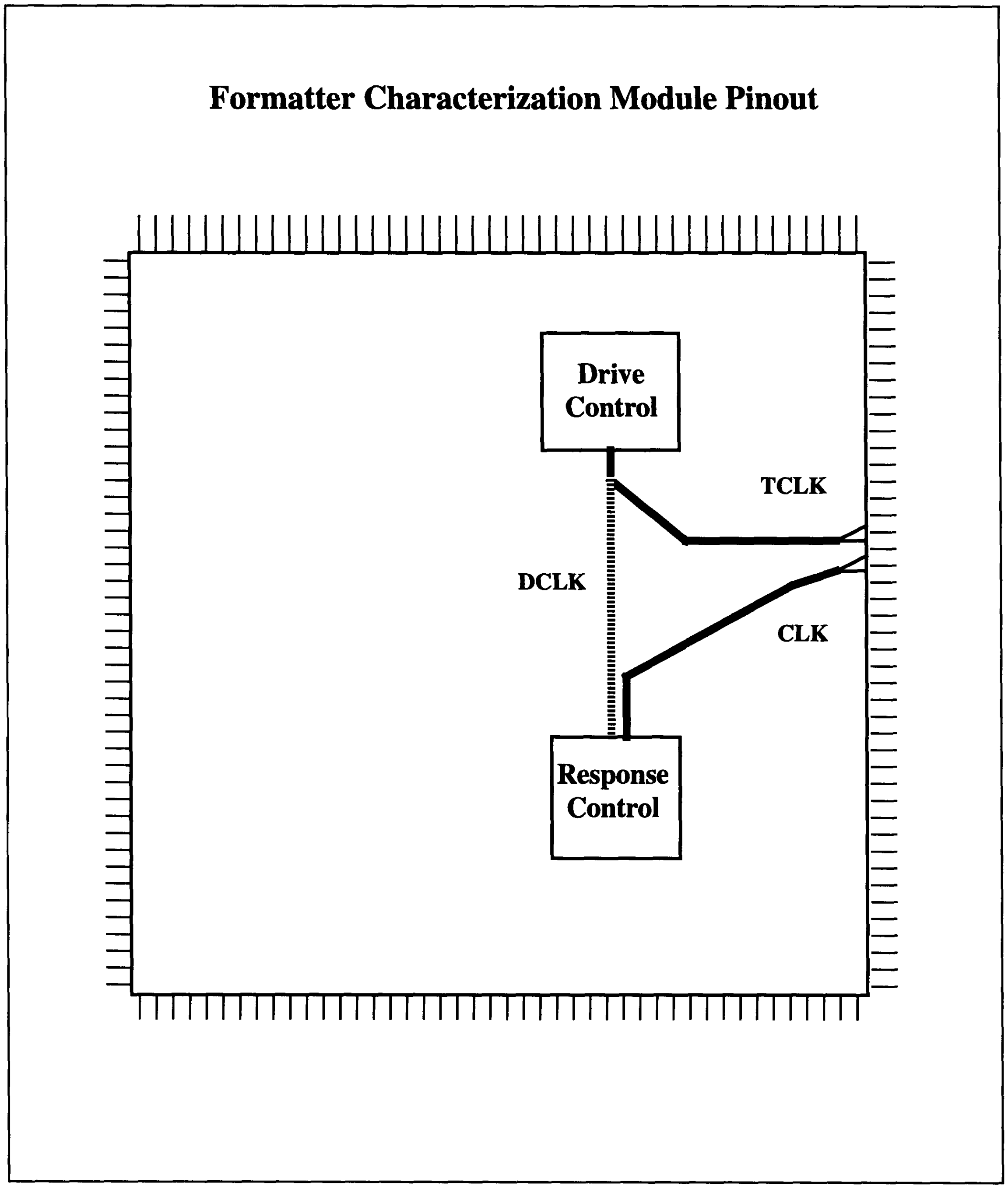

Figure 5.2: Formatter Characterization Module

algorithms to try and determine the cause of failure for each bad part.

The environment in which this process was conducted does *not* match the proposed emulation test environment. Since the resources were not available to implement the proposed environment, we used the current Schlumberger S9000 test environment to conduct the analysis.

5.5 Results

The first step in testing the devices was to exercise them with the production test set we had developed. This test should immediately identify those parts which worked properly. The production test set had two parts. The first test verified the board-level connections of the Formatter Characterization Modules. This is known as a continuity test The second test verified the actual functionality of the RIC and DIC chips on the modules. The purpose of the first test was to filter out those parts which failed because of board-level faults. Since, technically, the device we are concerned with is the formatter subsystem and not the Formatter Characterization Module, we need to be concerned only with faults that occur inside the RIC and DIC chips. Table 6.1 shows the results of this first set of tests.

Of the 17 chips tested, 11 passed both the continuity and functionality tests. These devices need not be considered. Of the remaining 6 parts, two failed both continuity and functionality. As a result, the failures which occurred within these modules might be either at the board level or on the chips themselves. We considered these parts beyond the scope of our study. There remained 4 parts which passed continuity but failed functionality. They are suitable candidates for diagnosis, because they contain errors within the RIC and DIC subsystem, which we considered our sample circuit. These chips are highlighted in Table 6.1, and form the basis for our exploration of diagnosis in the following section.

Chip Number	Continuity	Functionality
2	Pass	Pass
3	Pass	Pass
4	Pass	Pass
5	Pass	Pass
6	Pass	Pass
7	Pass	Pass
8	Pass	Pass
9	Pass	Pass
10	Pass	Pass
11	Pass	Pass
12	Pass	Pass
13	Pass	Fail
15	Pass	Fail
16	Pass	Fail
17	Fail	Fail
18	Fail	Fail
19	Pass	Fail

Table 5.1: Production Test Results for Formatter Characterization Modules

Chapter 6

Production Line Diagnosis

6.1 Introduction

In recent years considerable interest has been generated in the semiconductor industry around the task of increasing production yields and improving fabrication processes. The principal source of feedback to drive this task has been data from production testing. However, the current test environment lacks device design knowledge and therefore does not offer the potential for on-the-fly diagnosis of failing parts. One of the principal goals of the proposed methodology is to introduce device design knowledge to the tester domain and enable production test engineers to diagnose production failures for feedback into design Yamaguchi et al. [15] describe a method of diagnosis which uses electron beam probing and simulation techniques for identifying faulty function blocks. Our method is simpler and more efficient, using instead a faulted device model for comparison and running at emulation speeds.

This section begins by describing the proposed production test and diagnosis environment, and the sample circuit used to evaluate the proposal. Within this environment, diagnosis can be performed according to the traditional model of generating a set of hypotheses, testing out the hypotheses, and discarding failed hypotheses to produce new ones. The model of heuristics by which we carry out this process of generate and test reflects the sophistication of the diagnosis algorithm. The next goal of this section is to describe a variety of current models for diagnosis and examine their applicability to the IC test environment in terms of complexity and effectiveness. Finally, this section describes the process we undertook to practise these techniques upon production parts of the sample circuit and the effectiveness of our diagnosis process.

6.2 Diagnosis Strategies

We have established in sections 2.4.4 and 2.4.5 the IADE model for production test which allows the engineer to dynamically change the structure of the "good" device model $\mathbf{D_G}^{\mathbf{n}}$ used to test the DUT. The emulation system also gives the engineer access to the high-level functional test set $\mathbf{F_E}$ which is applied to the device. In the event that a production DUT fails the standard test set, the engineer now has the ability to perform diagnosis in real time. He knows the fail vector and signature exhibited by the DUT, and he has access to the structure of the device model and the test set itself. Two types of diagnosis can be performed at this point: structural diagnosis and behavioural diagnosis.

Structural diagnosis consists of hypothesizing a variety of structural faults which could account for the failure exhibited by the DUT during the standard production test, implementing each hypothesis using the faultable device model $\mathbf{D_G}^{\mathbf{n}}$, and testing out each hypothesis by exercising it in parallel with the DUT and comparing their outputs.

Behavioural diagnosis can be performed by observing device behaviour in the vicinity of the failure during the standard production test, modifying the test set $\mathbf{F_E}$ and applying it to the system to determine which specific activities of the device result in failure.

Because the proposed test environment provides access to both elements of the system, diagnosis can be performed using one or both of these methods.

6.3 Structural Diagnosis

When a DUT fails the production test, it is because somehow its circuit does not match the circuit of the device as it was designed. The main goal of structural diagnosis is to guess what the actual circuit of the failing DUT might be. If the flawed structure is modeled accurately, then the model's behaviour will match the behaviour of the DUT for all vectors in the test set. The process of hypothesizing the structure of the failing DUT is not restricted to any particular fault model. However, the more sophisticated and accurate the fault model, the more likely it is that a hypothesis will match faults which occur in the real world. For example, a simple model such as the *single-stuck-at fault model* will simplify and shorten the task of building a hypothesis set, but it is very rare for a real DUT to fail in production testing because of a single stuck-at fault.

The stages of structural diagnosis correspond roughly to the traditional AI methods of *Generate and Test*, proposed by Newell [9], and *Generate, Test and Debug*, proposed by Simmons & Davis [13]. The debugger begins by producing an initial hypothesis set, each element of which is a potential structure for the failing DUT. The debugger then implements each hypothesis with the emulated device model using the diagnostic fault selector. Each candidate model is then run in the system in place of the "good" device. If the outputs of the DUT and the hypothesis model $\mathbf{D_G}^\mathbf{n}$ match for all vectors of the test set, then the structure of the DUT has been determined, with respect to the given test set:

$$DUT(E_G(F_E)) \approx D^n_G(E_G(F_E)) \qquad (6.1)$$

The hypothesis set can be developed using a variety of methods, which reflects the basic trade-off between increasing the size of the diagnostic system and increasing its sophistication. The most straightforward approach to generating a hypothesis set is by use of a fault dictionary. A less bulky and more computation-intensive method involves performing backtracing upon the failing outputs of the DUT. The third and most sophisticated method for hypothesis generation is physical localization based upon the structural hierarchy of the device.

6.3.1 Fault Dictionary Look-up

When performing fault-grading for the production test set, the test engineer injects all possible faults for a particular fault model into the device, and records the percentage of those faults which are identified by the test set. At the same time, the engineer can build a fault dictionary for the device. This contains entries, for each fault injected, which record the type of fault injected, whether the device passed or failed when injected with the fault, and, if it failed, the first failing vector and output signature at the fail.

If this dictionary is stored in the system, it can be retrieved during diagnosis and used to produce an initial hypothesis set. Given a failing DUT, the engineer can search the fault dictionary for entries which have the same fail vector and output signature. Each entry which matches these items will suggest one possible fault which could exist in the current failing DUT.

Although fault dictionary look-up can be the quickest and simplest method of candidate generation, it has several drawbacks. First, the dictionary will correspond only to the fault model which was used in grading the test set. Therefore, if the fault model used for grading was relatively narrow, then the hit rate for real failing parts will be correspondingly low. Second, using a fault dictionary requires a large memory overhead for storing the huge number of entries that even simple devices typically require. This overhead may outweigh the computational advantage of a simple table look-up that fault dictionaries provide.

6.3.2 Backtracing

The process of backtracing involves following signals from the failing output pins of the device back into the design model to understand how possible failures at different stages within the device would propagate to its outputs. For each failing pin, the signal it presents is traced back into the device, stage by stage. The presumption is that each submodule which affected this signal could have corrupted it to produce the incorrect output. Candidates are discarded if the circuit is laid out in such a way that faults in these candidates would produce a different set of failing outputs than was observed in the DUT. Chen

& Srihari [3] describe a set of heuristics for ordering the set of candidate submodules in terms of their likelihood to have caused the observed faults.

Although backtracing is more complicated than fault dictionary look-up, developing backtracing algorithms based upon structure files is relatively straightforward. The difficulty with backtracing is that for multiple failing pins, or for multiple-fault models, the computation required to build and order hypothesis sets using backtracing can be substantial.

6.3.3 Physical Localization

Physical localization proceeds from the basic premise that the design of a device is normally partitioned to reflect different elements of its intended functionality. Designers tend to organize the structure of a device by functionality both to provide modularity to their design and to simplify their own understanding of the device.

When a DUT fails in production testing, the failing test vector and output signature can be analyzed to determine what part of the device's functionality was being exercised at the time of failure. Understanding the functionality which failed quickly narrows the region in which to search for candidate failures. This localization can recursively be applied to smaller and smaller parts of the physical circuit, until the resolution is small enough for reasonable fault hypotheses to emerge.

This method overlaps with behavioural diagnosis, as it is closely tied to understanding the device in terms of its required functionality, and as it involves analysis of the production test set. It is also the most abstract and "intelligent" method of diagnosis. Davis & Hamscher [4] provide a comprehensive analysis of this method of Model-Based Troubleshooting. Diagnostic systems, such as Genesereth's *DART* [5], use this type of design knowledge to perform Generate and Test.

6.4 Behavioural Diagnosis

While structural diagnosis suggests candidate structural models of the failing DUT, behavioural diagnosis does not affect the circuit of the faultable device. In fact, it can be particularly effective in situations where parts of the design model are inaccessible, and so structural diagnosis is useless. Behavioural diagnosis requires examining the device from a primarily functional point of view. Based upon knowledge of the functional partitioning of the device and a detailed understanding of the production test set, diagnosis can be performed by manipulating the test set rather than the gate level design.

The test set is typically developed in an organized manner to exercise, in turn, different elements of the device's required behaviour. When a DUT fails in production, the test set can be examined to understand what type of activity caused the DUT to fail. Since the production test set is compressed to optimize the passing of good parts, it will generally be necessary to expand and modify this test set in order to localize precisely the failing activity.

As a result, the debugger's task is to generate a set of candidate test sequences, each of which localizes the failing behaviour of the device in one way or another. Each of these candidate sequences is put in place of the production test set, and applied to the system. Based upon these results, the debugger should be able to resolve the failing functionality to specific activities. If the device design is partitioned functionally, then this result can be used to determine structural submodules which may have caused the observed failure.

The behavioural approach to diagnosis can very quickly isolate small portions of the circuit, which can then be analyzed using structural techniques. However, this method requires the debugger to have considerable design and functional understanding.

6.5 Analysis of Faulty Parts

Table 5.1 shows the results of testing 17 physical parts in the production test environment. We chose to diagnose those parts which passed continuity testing (which verifies connections on the multi-chip module), but failed functionality. We surmised that these were the parts for which faults existed on the RIC and DIC circuits themselves. Each chip was then analyzed using both structural and behavioural diagnosis methods, to evaluate the complexity and effectiveness of these methods in identifying the faults.

6.5.1 Structural Diagnosis

Because the environment in which we operated did not have emulation capability, the process by which we tested candidate structures was as follows:

1. Generate a hypothesis structure $\mathbf{D_G}^{\mathbf{n}}$.
2. Model this hypothesis and exercise it using the simulated version of the test environment shown in Figure 2.8.
3. Bundle the test set $\mathbf{F_D}$ produced in simulation along with the expected device outputs from the hypothesis model $\mathbf{D_G}^{\mathbf{n}}$ to produce an S9000 test program.
4. Run the test program on the failing DUT within the S9000 test environment.
5. If the DUT passes the test, then its structure matches that of $\mathbf{D_G}^{\mathbf{n}}$

This process is essentially no different from the process proposed for the new tester environment. The only difference is that in the proposed environment, it would be possible to perform these tasks automatically and in real time.

The next task was to generate a hypothesis set for each device. To do this, we applied each of the diagnosis methods described in section 6.2 in increasing order of complexity. Having already developed and stored a single-stuck-at fault dictionary for the test set, the first step was to check the fault dictionary for entries which matched the first fail information for each of the failing devices. If matches were found in the fault dictionary, these faults were considered the first candidates in the hypothesis set. Each of these faults was injected into the design model, and simulated to produce a test program which could be applied to the failing DUT. If the DUT passed the entire test program, then the fault was identified. Two of the failing parts had matches within the fault dictionary. Unfortunately, in each case, the hypothesis structure did not match the structure of the DUT. The DUT would pass the initial vector where it had failed on the "good" test program, but it would then fail later on in the hypothesis program. This showed that in many cases, two different structures will match in behaviour for part of a test set and then diverge when the difference between them is highlighted.

The next step in structural diagnosis was to generate candidates by performing backtracing. Following the stages of the circuit back from the output pins identified a set of candidate submodules for testing. However, examining the fault dictionary entries for each candidate fault showed that none of them failed at the right test vector and with the right signature to correspond to the failing DUTs. In fact, this process proved to be redundant once the fault-dictionary lookup had already been performed. However, when compared with the space overhead imposed by the use of fault dictionaries, this approach proved to be much more efficient in identifying candidates. In real-world situations, performing backtracing is preferable to creating and storing fault-dictionaries during fault grading.

Neither the fault dictionary lookup nor the approach of backtracing diagnosed any of the failing devices correctly. Part of the problem is that both approaches relied upon the *single-stuck-at fault model*, which is too narrow for any useful real-world diagnosis. The final method we tried was a combination of physical localization and behavioural analysis, which proved to be the most effective and intelligent approach to diagnosis.

6.5.2 Physical Localization and Behavioural Diagnosis

When the structural analysis techniques failed, our next strategy was to analyze the fail vectors and signatures of each of the failing DUTs, in order to understand what region of activity caused the device to fail. This would immediately narrow the scope of our search to a particular segment of the design model for the device. It is important to note, however, that identifying the fail cycle for a particular device does not immediately identify the activity which results in failure. The primary reason for this is that production test sets, such as the test set we used to exercise our devices, are highly compressed and optimized in order to maximize throughput on the production line. Consequently, each activity performed in the test suite is not necessarily isolated and checkpointed in such a way that failure in the region of the activity immediately identifies it to be the cause of the problem. More commonly, groups of activities are bundled in such a way that any one of several activities would cause the device to fail within a particular region.

This issue is readily addressed, however, by the debugger. The first goal of behavioural analysis is to identify the group of activities which contains the cause of the failure. This region can then be examined in further detail by expanding each activity into a clearly encapsulated module which is isolated and checkpointed. Each of these modules is then applied as stimulus to the diagnosis system model. Now failing behaviour can be resolved to the granularity required to identify specific stimuli.

By applying this process to the failing devices, we found that all four parts failed the functionality test in the same region: during activities which accessed their internal gain registers (Chips 15 and 19) and delay registers (Chips 13 and 16). These registers are described in Table 3.2. In the case of Chips 13 and 16, it occurred that the test set compressed two activities together, making it unclear which activity caused the failure. As a result, we created a new hypothesis test set for each chip, which would clearly determine the point of failure. In fact, our initial hypothesis as to the cause of failure in both these parts proved incorrect once the new test set was developed and applied.

It emerged finally that all four parts failed during the setting of the internal Linear Delay Line gain and delay registers. Thus behavioural techniques brought us to a very specific and clear structural subsection of the circuit in which to hypothesize reasons for failure.

When we re-examined the circuit diagrams to consider the LDL modules, we found that the Verilog models for these circuits were inaccessible to our fault select mechanism, because they were modeled as mixed analog/digital circuits using behavioural descriptions. As a result, there was no way in which to insert faults into internal gates in these modules. We had reached the limits of our diagnostic localization.

The question arises: does this diagnosis suffice in understanding why our production parts failed? After all, we were not able to identify a single gate-level fault in any of the devices. The answer is that this level of diagnosis is perfectly valid, given the accessibility we had to the models of our devices. In any device design model, there will be some elemental black-box modules which cannot be opened for analysis. Whether this limit is reached at transistor level, or at gate level, or, as in our case, at some higher level, is a somewhat design-specific choice. It will necessarily limit the resolution to which faults can be diagnosed, but it does not invalidate diagnosis or undermine its usefulness.

The following section describes in detail the diagnosis results for each of the four devices we examined in this phase of the research.

Test Vector	Register Memory	Register Bus Fail Signature	Register Being Read
770	90	L111 1111	D820
788	93	L111 11L1	D823
944	119	L111 1111	D820
962	122	L111 11L1	D823
1106	146	L111 1111	D820
1124	149	L111 11L1	D823
1388	193	L111 1111	D820
1406	196	L111 11L1	D823

Table 6.1: Failing Test Vectors for Chip 13

6.6 Diagnosis Results

6.6.1 Chip 13

Chip 13 was the first part we examined which satisfied our criteria for diagnosis: it passed the production test program for continuity and failed for functionality. This meant that the device failure lay somewhere on the RIC and DIC circuits and not on the multi-chip module. Each of the first 8 failures in the test program occurred in the register read/write bus lines. Table 6.2 shows the characteristics of these fail points.

The fault dictionary showed no single stuck-at faults which resulted in failure on the same test vector cycle and with the same signature as we had observed. When the fault dictionary was queried for the fail signature alone, it showed 10 possible faults which caused the same output pins to fail. Inspecting these faults by hand in the circuit model ruled them out as possible causes of failure. The fault dictionary showed no faults which caused the first failure to occur on the same test vector.

Register Memory	Event	Comments
48	Master Reset	Checkpoint
56	Write all 1's - D820	First register write
66	Set Kill Read	Causes all reads to return 0
74	Read back - D820	PASS
82	Clear Kill Read	Disables Set Kill Read
90	Read back - D820	FAIL
93	Read back - D823	FAIL
119	Read back - D820	FAIL
122	Read back - D823	FAIL
146	Read back - D820	FAIL
149	Read back - D823	FAIL
193	Read back - D820	FAIL
196	Read back - D823	FAIL

Table 6.2: Failing Event Region for Chip 13

The behavioural diagnosis proved much more fruitful. We isolated as far as possible the region in which the failures occurred and examined the events which had been applied. Table 6.3 shows the analysis of this region of behaviour.

The ambiguity in this event group is illustrated by the highlighted element in Table 6.3. The problem is that immediately after the registers are set to all 1's, the Set Kill Read register command is executed. This command forces all register reads to return 0 values for the registers. The next vector passes. Then the Set Kill Read option is disabled by Clear Kill Read. The next vector fails.

This could be caused by one of two situations: A) the registers are written correctly, and the Set Kill Read option works fine, but the Clear Kill Read does not properly re-

Register Memory	Event	Comment
48	Master Reset	Checkpoint
56	Write all 1's - D820	First register write
66	Set Kill Read (Disabled)	Causes all reads to return 0
74	Read back - D820	FAIL
77	Read back - D823	FAIL
82	Clear Kill Read (Disabled)	Disables Set Kill Read
90	Read back - D820	FAIL
93	Read back - D823	FAIL
119	Read back - D820	FAIL
122	Read back - D823	FAIL
146	Read back - D820	FAIL
149	Read back - D823	FAIL

Table 6.3: Failing Event Group for Chip 13, New Test Set

enable the register read mechanism, causing the next vector to fail; or B) the resisters are not written correctly, but Set Kill Read is used before they can be tested, and the test fails only after Clear Kill Read enables the registers to be read properly.

In order to test these two hypotheses, we began by simply removing both the Set Kill Read and the Clear Kill Read commands from the test set. Table 6.4 shows the new event group and the results. Now it is clear that the Set Kill Read and Clear Kill Read events did not cause the failure, and that the failure occurred when the registers were originally written. The final diagnosis for Chip 13 was that the delay registers in the LDL submodule of the Drive Control IC were not functioning properly.

Test Vector	Register Memory	Register Bus Fail Signature	Register Being Read
776	91	1111 1L11	D821
950	120	1111 1L11	D821
1112	147	1111 1L11	D821
1394	194	1111 1L11	D821

Table 6.4: Failing Test Vectors for Chip 16

Register Memory	Event	Comments
48	Master Reset	Checkpoint
57	Write all 1's - D821	First register write
66	Set Kill Read	Causes all reads to return 0
75	Read back - D821	PASS
82	Clear Kill Read	Disables Set Kill Read
91	Read back - D821	FAIL
120	Read back - D821	FAIL
147	Read back - D821	FAIL
194	Read back - D821	FAIL

Table 6.5: Failing Event Region for Chip 16

6.6.2 Chip 16

Chip 16 proved to be almost identical to Chip 13 in the diagnosis results we obtained for it. Each of the first 4 failures in the test program occurred in the register read/write bus lines. Table 6.5 shows the characteristics of these fail points.

The fault dictionary showed no single stuck-at faults which resulted in failure on the same test vector cycle and with the same signature as we had observed. When the fault dictionary was queried for the fail signature alone, it showed 4 possible faults which

Register Memory	Event	Comment
48	Master Reset	Checkpoint
57	Write all 1's - D821	First register write
66	Set Kill Read (Disabled)	Causes all reads to return 0
75	Read back - D821	FAIL
82	Clear Kill Read	Disables Set Kill Read
91	Read back - D821	FAIL
120	Read back - D821	FAIL
147	Read back - D821	FAIL

Table 6.6: Failing Event Group for Chip 16, New Test Set

caused the same output pins to fail. Inspecting these faults by hand in the circuit model ruled them out as possible causes of failure. The fault dictionary showed no faults which caused the first failure to occur on the same test vector. Table 6.6 shows the results of isolating and analyzing the region of behaviour in which the failures occurred.

This table illustrates the same ambiguity we found in Chip 13. The failure observed at the device outputs could be due either to the Clear Kill Read event or to the register write event itself. We therefore constructed the same hypothesis in which the Set Kill Read and Clear Kill Read commands were removed from the test set and the system re-run. The results are shown in Table 6.7.

As with Chip 13, the Set Kill Read and Clear Kill Read events did not cause the failure, and that the failure occurred when the registers were originally written. The final diagnosis for Chip 16 was that the delay registers in the LDL submodule of the Drive Control IC were not functioning properly.

Test Vector	Register Memory	Register Bus Fail Signature	Register Being Read
164	24	000L 0001	D860
209	27	1000 100H	D863
329	35	111L 000H	D863
374	38	111L 0000	D862
389	39	000L 0001	D863
419	41	111L 0000	D861
434	42	000L 0001	D862
449	43	0010 001H	D863
464	44	111L 0000	D860
479	45	000L 0001	D861
512	47	111L 1111	D863
770	90	111L 1111	D820
776	91	111L 1111	D821
782	92	111L 1111	D822
788	93	111L 1111	D823
794	94	000L 0001	D860
800	95	001L 0011	D861
806	96	011L 0111	D862
812	97	111L 1111	D863

Table 6.7: Failing Test Vectors for Chip 15

6.6.3 Chip 15

Chips 15 and 19 gave results which were somewhat different from those we found for Chips 13 and 16. For Chip 15, each of the first 19 failures in the test program occurred in the register read/write bus lines. Table 6.8 shows the characteristics of these fail points.

Register Memory	Event	Comment
14	Master Reset	Checkpoint
18-21	Set Gain - D860-D863	First Register Write
24	Apply event and read back delay - D860	FAIL
27	Apply event and read back delay - D863	FAIL
35	Apply event and read back delay - D863	FAIL
38	Apply event and read back delay - D862	FAIL
39	Apply event and read back delay - D863	FAIL
41	Apply event and read back delay - D861	FAIL
42	Apply event and read back delay - D862	FAIL
43	Apply event and read back delay - D863	FAIL
44	Apply event and read back delay - D860	FAIL
45	Apply event and read back delay - D861	FAIL
47	Apply event and read back delay - D863	FAIL
66	Set Kill Read	Causes all reads to return 0
78	Read back - D860	PASS
82	Clear Kill Read	Disables Set Kill Read
90	Read back - D820	FAIL
91	Read back - D821	FAIL
92	Read back - D822	FAIL
93	Read back - D823	FAIL
94	Read back - D860	FAIL
95	Read back - D861	FAIL
96	Read back - D862	FAIL
97	Read back - D863	FAIL

Table 6.8: Failing Event Group for Chip 15

The fault dictionary appeared at first to provide useful information. Querying on both fail vector and signature produced 4 candidates for the hypothesis set. For each of the candidates, the fault was injected into the "faultable" model, and the simulation re-run to produce the device-level test set. This test set was then integrated with the expect data for the faulted model to produce a new test program. Each test program was then used on the 9000 tester to exercise Chip 15. Unfortunately, in each of the four cases, the device still failed, albeit at a later point in the test program. This illustrated the possibility that fault dictionary hits will still diverge from the DUT further on in the test set, despite their similar behaviour at the first fail in the original test program. In this case, the failures occurred when the gain registers in the LDL modules were read. Table 6.9 shows the results of analyzing the event region in which the fails occurred.

It is clear from Table 6.9 that the fail occurred due to the initial register write. There is no ambiguity in the Set Kill Read and Clear Kill Read options as there was in Chips 13 and 16. The final diagnosis for Chip 15 was that the gain registers in the LDL submodule of the Drive Control IC were not functioning properly.

6.6.4 Chip 19

Chip 19, the last of the four chips we tested, turned out to be quite similar to Chip 15 in its test behaviour. Each of the first 25 failures in the test program occurred in the register read/write bus lines. Table 6.10 shows the characteristics of these fail points.

Querying the fault dictionary on both fail vector and signature produced 2 candidates for the hypothesis set. For each of the candidates, the fault was injected into the "faultable" model, and the simulation re-run to see if the failing chip corresponded to the faulted model. Unfortunately, as with Chip 15, neither of these hypotheses matched the behaviour of the failing part through the entire test set.

Test Vector	Register Memory	Register Bus Fail Signature	Register Being Read
164	24	0H01 0001	D860
209	27	1H00 1000	D863
224	28	HH10 0010	D860
239	29	0L00 H100	D861
254	30	1H00 1000	D862
269	31	0H00 1111	D863
284	32	1H00 1H0H	D860
299	33	1H00 1H0H	D861
314	34	0H00 1111	D862
329	35	111L 0H0H	D863
344	36	111L 0H0H	D860
359	37	111L 0H0H	D861
389	39	111L 0H0H	D863
404	40	111L 0000	D860
419	41	111L 0000	D861
434	42	111L 0000	D862
449	43	111L 0000	D863
464	44	111L 0000	D860
479	45	0H0L 0H01	D861
494	46	111L 0000	D862
512	47	000L 0001	D863
788	93	0H10 0H1H	D823
794	94	000H 0000	D860
800	95	111L 1111	D861

Table 6.9: Failing Test Vectors for Chip 19

Register Memory	Event	Comment
14	Master Reset	Checkpoint
18-21	Set Gain - D860-D863	First Register Write
24	Apply event and read back delay - D860	FAIL
27	Apply event and read back delay - D863	FAIL
28	Apply event and read back delay - D860	FAIL
29	Apply event and read back delay - D861	FAIL
30	Apply event and read back delay - D862	FAIL
31	Apply event and read back delay - D863	FAIL
32	Apply event and read back delay - D860	FAIL
33	Apply event and read back delay - D861	FAIL
34	Apply event and read back delay - D862	FAIL
35	Apply event and read back delay - D863	FAIL
36	Apply event and read back delay - D860	FAIL
37	Apply event and read back delay - D861	FAIL
39	Apply event and read back delay - D863	FAIL
40	Apply event and read back delay - D860	FAIL
41	Apply event and read back delay - D861	FAIL
42	Apply event and read back delay - D862	FAIL
43	Apply event and read back delay - D863	FAIL
44	Apply event and read back delay - D860	FAIL
45	Apply event and read back delay - D861	FAIL
46	Apply event and read back delay - D862	FAIL
47	Apply event and read back delay - D863	FAIL
93	Apply event and read back delay - D823	FAIL
94	Apply event and read back delay - D860	FAIL
95	Apply event and read back delay - D861	FAIL

Table 6.10: Failing Event Group for Chip 19

As with Chip 15, the failures occurred when the gain registers in the LDL modules were read. Table 6.11 shows the results of analyzing the event region in which the fails occurred. The fail clearly occurred due to the initial register write. There is no ambiguity in the Set Kill Read and Clear Kill Read options as there was in Chips 13 and 16. The final diagnosis for Chip 19 was that the gain registers in the LDL submodule of the Drive Control IC were not functioning properly.

6.7 Conclusions

We were pleased with the final results of our diagnosis efforts. Given 17 production parts, we were able to use the production test set developed in the IADE model to identify six parts which failed functional verification, four of which we had the capacity to diagnose. We then analyzed these four parts to the limit of our modeling resolution, and were able to diagnose the cause of failure in each part to the elemental level. In following this process, we gained some insight into the relative effectiveness of various diagnosis algorithms within the context of the proposed IADE methodology.

The structural diagnosis techniques of fault dictionary lookup and backtracing perform roughly the same function, we realized. It seems that the trade-off between these two approaches is one of memory versus computation. In most real environments, however, we suspect that the backtracing process, which is fairly straightforward but time-consuming, is more efficient than using a fault dictionary. Even in the sample circuit we considered, the fault model used to develop the fault dictionary was far too simplistic to provide any real diagnosis capability. For more complex circuits, the overhead of generating and storing a large fault dictionary seems inappropriate.

Behavioural diagnosis and structural localization, however, combine to form a fairly effective strategy for diagnosis of a wide range of faults. Behavioural diagnosis worked effectively for us as a high-level filter, to help us focus our analysis very quickly on a small part of the circuit. From this point, structural analysis of the circuit can hopefully be used to resolve the fault to the elemental level. It should be noted, however, that this process was performed manually in our study. Automated a process by which short behavioural hypothesis tests are generated and exercised is a much more involved task. This type of diagnosis is perhaps the most "intelligent"; developing a reliable algorithm for such a process could help greatly toward automating diagnosis in the IADE environment.

Even without an automated debugger, diagnosis can conceivably be performed quite easily in the proposed model, with the refinement of the test control interface. We envision a test environment in which the circuit schematics, which represent the "clean" device model being emulated, are accessible to the test engineer. If there is a simple (graphic?) interface through which the engineer can manually hypothesize faults and insert them easily into the device model, the system can take advantage of the natural intelligence and experience of the test engineer to perform diagnosis. Observing such manual diagnosis can also be used to develop a knowledge base for the system, which can gradually begin to capture the diagnosis expertise of the test engineer, and eventually allow the system to be automated.

Chapter 7

Conclusion

7.1 Project Results

The IADE methodology we have proposed substantially modifies the current ASIC development cycle. It builds upon existing concepts in system design (such as modeling of a device within its environment), test engineering (functionally-based test set generation), and circuit diagnosis. What it brings to IC design is an integrated framework for employing many such techniques in unison, with useful feedback paths between phases of design. This is made possible, for the most part, by the computational impact of using logic emulation in place of traditional simulation techniques. The speed improvement emulation offers is significant enough, in the scale of development schedules, to warrant substantial procedural changes to the current paradigm. We have attempted in this research project to explore the value and applicability of the proposed methodology, by implementing each of its phases for a sample ASIC. Our study leaves us with a moderate optimism for IADE. In the course of our analysis, we have encountered several obstacles to full implementation of the methodology, some of which we have not been able to resolve.

7.2 Methodology Issues

In the device design phase, for example, we are conscious of the fact that our implementation was done in simulation. Until we can explore an entire system fully emulated in conjunction with a software test program interface, it is difficult to predict how complex the control and integration task will be. As well, we did not produce a gate-level design for our environment model. A task that needs to be addressed is implementing the gate-level environment model efficiently and cheaply. Synthesis tools already exist for creating gate-level models from behavioural descriptions; will these become fast enough and robust enough to make developing the gate-level environment model transparent to the designer?

In the phase of test generation and fault grading, we need to explore further the partitioning of the testing task between functional and timing verification before proper commercial device test sets can be developed in IADE. Simulation will surely play a role in this expanded system, for timing analysis cannot be performed without it. This being the case, how difficult will it be to divide the task of testing between the emulation and simulation domains? For fault grading with a single-stuck-at fault model, our faultable gate design proved sufficient in simulation. More complex models will have to be developed to handle different classes of faults.

In production testing, the primary task is the integration of the emulation system with the pin electronics in the physical test head. Another area which requires some effort will be developing a software interface for diagnosis of failing parts. This can either be an automated tool, which uses a combination of behavioural and physical localization techniques to diagnose the design, or an interface to the emulation model that will allow the test engineer to manually diagnose failing parts, and which can perhaps accumulate diagnosis knowledge by observing and learning from their techniques.

7.3 Emulation Issues

What makes emulation so much more efficient than simulation (the use of hardware to instantiate a circuit, rather than software) is also what makes it more difficult to implement in a system such as IADE. In the course of our study, we have come across several issues that need to be explored before emulation can be used in IADE. First, there is the issue of modeling potentially large systems in emulation. Currently, individual commercial emulation modules are not be large enough to model devices with very high gate counts. Modeling faultable gates will further increase the emulation system size required to implement a particular device. There is clearly a cost/benefit compromise involved in using an emulation system of a given size. There are systems available commercially which allow the user to connect multiple emulation modules together for potentially infinite scalability. It remains to be seen how effective the control and partitioning mechanisms will be for these systems. Second, there is the issue of device speed versus emulation speed. We expect emulation technology will advance sufficiently in the near future to permit devices to be run at speed in IADE. Until we reach that point, devices will have to be developed in IADE at emulation speeds. If device behaviour is static at normal speed, then we can expect functional signatures to be identical at normal speed and at emulation speed. If this is not the case, then we need to address the problem of mapping emulation speed signatures (for fault grading and diagnosis) to the corresponding device speed signatures. Finally, there is the problem that emulation cannot model indeterminate ("X") states. It is unclear how much this will affect the task of device verification in production.

7.4 Conclusions

Given the scope and complexity of the proposed IADE methodology, we expected from the outset to encounter many issues which stand in the way of its implementation in any professional IC design environment. Because it offered so much potential to integrate and streamline the design cycle, we felt it was worth a substantial exploration effort. We have not been disappointed in either respect. There are a number of areas which need to be researched further before IADE can be made to work robustly, but it has largely withstood our scrutiny.

We were extremely pleased with the results obtained for test generation and fault grading; IADE indeed proved considerably more compact and efficient than current techniques. Furthermore, implementing an environment model for the device clarified our own understanding of the device as design engineers, an exercise which we feel will prove rewarding in its own right as well as in the IADE fault grading model.

Fault grading in IADE proved very effective as well. We were able not only to quantify the performance comparison between emulation and simulation, but also to see the positive effect of using fault grading to justify a particular gate-level implementation. We feel that this is one of the most significant contributions of IADE to the development cycle.

We were most excited by the possibilities IADE offers for production testing and diagnosis. Given our sample batch of 17 devices, we were able to use various diagnosis techniques to resolve device failure to the elemental level. This gave us an understanding of the relative effectiveness of different diagnosis techniques, and also demonstrated the tremendous value of introducing design knowledge into the production test domain.

If the issues we have raised in this initial feasibility study can be resolved, we feel that IADE can be made viable within some ASIC design environments. We expect that when these issues are further explored, they will simply set constraints on the applicability of the IADE technique to different design problems. It is clear that IADE will not be a general-purpose process for designing all integrated circuits. Nevertheless, if its specific range of usefulness can be identified, it can dramatically improve development cycles for those circuits to which it is best suited.

References

[1] Abramovici, M., Breuer, M. and Friedman, A. *Digital Systems Testing and Testable Design.* W.H. Freeman and Company, New York, N.Y. 1990.

[2] Ackerman, D. et al. *Logic Simulation Using a Hardware Accelerator Together with an Automated Error Event Isolation and Trace Facility.* United States Patent Document, Patent Number 5,146,460. Sept 8, 1992.

[3] Chen, J. and Srihari, S. *A Method for Ordering Candidate Submodules in Fault Diagnosis.* Department of Computer Science, State University of New York, Buffalo, New York. June 29, 1988.

[4] Davis, R. and Hamscher, W.C. *Model-Based Reasoning: Troubleshooting.* Artificial Intelligence Laboratory, Massachusetts Institute of Technology. July 1988.

[5] Genesereth, M. *The Use of Design Descriptions in Automated Diagnosis.* Department of Computer Science, Stanford University. 1984.

[6] Koeppe, S. *Method for Simulating an Open Fault in a Logic Circuit Comprising FETs and Simulation Models for Implementing the Method.* United States Patent Document, Patent Number 4,868,825. Sept 19, 1989.

[7] Markowsky, G. *Diagnosing Single Faults in Fanout-Free Combinational Circuits.* IBM Thomas J. Watson Research Center, Yorktown Heights, New York. February 1978.

[8] Namitz, W. et al. *Programmable Fault Insertion Circuit.* United States Patent Document, Patent Number 5,058,112. Oct 15, 1991.

[9] Newell, A. *Artificial Intelligence and the Concept of Mind.* Computer Models of Language and Thought, eds. Schank and Colby. 1973.

[10] Shirley, M.H. *Generating Distinguishing Tests based on Hierarchical Models and Symptom Information.* Proceedings of the International Conference on Computer Design. 1983.

[11] Shirley, M.H. *Generating Tests by Exploiting Designed Behavior.* Artificial Intelligence Laboratory, Massachusetts Institute of Technology. May 26, 1986.

[12] Shirley, M.H., Wu, P., Davis, R., and Robinson, G. *A Synergistic Combination of Test Generation and Design for Testability.* Proceedings of the IEEE International Test Conference. Sept 1987.

[13] Simmons, R. and Davis, R. *Generate, Test and Debug: Combining Associational Rules and Causal Models.* Artificial Intelligence Laboratory, Massachusetts Institute of Technology. 1987.

[14] West, B.G. and Napier, T. *Sequencer Per Pin Test System Architecture.* Proceedings of the IEEE International Test Conference. Sept 1990.

[15] Yamaguchi, N. et al. *Method of Diagnosing Integrated Logic Circuit.* United States Patent Document, Patent Number 4,996,659. Feb 26, 1991.

Appendix A

Gate Models

A.1 Library of Faultable Gates

```
'timescale 1ns/1ps

module QY1AND2A (Q,QB,X1,X2,EN,FAULT);
 output Q,QB;
 input X1,X2;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 and #(Trise,Tfall) z2 (Q,X1F,X2F);
 not z1(QB,Q);
endmodule

module QY1AND2A_GOOD (Q,QB,X1,X2);
 output Q,QB;
 input X1,X2;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 and #(Trise,Tfall) z2 (Q,X1,X2);
endmodule

module QY1AND3A (Q,QB,X1,X2,X3,EN,FAULT);
 output Q,QB;
 input X1,X2,X3;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F, X3F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 assign X3F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X3;
 and #(Trise,Tfall) z2 (Q,X1F,X2F,X3F);
 not z1(QB,Q);
endmodule

module QY1AND3A_GOOD (Q,QB,X1,X2,X3);
 output Q,QB;
 input X1,X2,X3;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 and #(Trise,Tfall) z2 (Q,X1,X2,X3);
endmodule

module QY1OR2A (Q,QB,X1,X2,EN,FAULT);
 output Q,QB;
 input X1,X2;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 or #(Trise,Tfall) z2 (Q,X1F,X2F);
 not z1(QB,Q);
endmodule

module QY1OR2A_GOOD (Q,QB,X1,X2);
 output Q,QB;
 input X1,X2;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 or #(Trise,Tfall) z2 (Q,X1,X2);
endmodule

module QY1OR3A (Q,QB,X1,X2,X3,EN,FAULT);
 output Q,QB;
 input X1,X2,X3;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F, X3F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 assign X3F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X3;
 or #(Trise,Tfall) z2 (Q,X1F,X2F,X3F);
 not z1(QB,Q);
endmodule

module QY1OR3A_GOOD (Q,QB,X1,X2,X3);
 output Q,QB;
 input X1,X2,X3;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 or #(Trise,Tfall) z2 (Q,X1,X2,X3);
endmodule

module QY1OR4A (Q,QB,X1,X2,X3,X4,EN,FAULT);
 output Q,QB;
 input X1,X2,X3,X4;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F, X3F, X4F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 assign X3F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X3;
 assign X4F = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X4;
 or #(Trise,Tfall) z2 (Q,X1F,X2F,X3F,X4F);
 not z1(QB,Q);
endmodule

module QY1OR4A_GOOD (Q,QB,X1,X2,X3,X4);
 output Q,QB;
 input X1,X2,X3,X4;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 or #(Trise,Tfall) z2 (Q,X1,X2,X3,X4);
endmodule

module QY1OR5A (Q,QB,X1,X2,X3,X4,X5,EN,FAULT);
 output Q,QB;
 input X1,X2,X3,X4,X5;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X2F, X3F, X4F, X5F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X2;
 assign X3F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X3;
 assign X4F = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X4;
 assign X5F = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : X5;
 or #(Trise,Tfall) z2 (Q,X1F,X2F,X3F,X4F,X5F);
 not z1(QB,Q);
endmodule

module QY1OR5A_GOOD (Q,QB,X1,X2,X3,X4,X5);
 output Q,QB;
 input X1,X2,X3,X4,X5;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 not z1(QB,Q);
 or #(Trise,Tfall) z2 (Q,X1,X2,X3,X4,X5);
endmodule

module QY1MX2A (Q,QB,X0,X1,S,EN,FAULT);
 output Q,QB;
 input X0,X1,S;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X0F, X1F, SF;
 assign X0F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X0;
 assign X1F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1;
 assign SF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : S;
 mux2 #(Trise,Tfall) x1 (Q,X0F,X1F,SF);
 not x2 (QB,Q);
endmodule

module QY1MX2A_GOOD (Q,QB,X0,X1,S);
 output Q,QB;
 input X0,X1,S;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 mux2 #(Trise,Tfall) x1 (Q,X0,X1,S);
 not x2 (QB,Q);
endmodule

module QY1MX4A (Q,QB,X0,X1,X2,X3,S1,S0,EN,FAULT);
 output Q,QB;
 input X0,X1,X2,X3,S1,S0;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X0F, X1F, X2F, X3F, S0F, S1F;
 assign X0F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X0;
 assign X1F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1;
 assign X2F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X2;
 assign X3F = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X3;
 assign S1F = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : S1;
 assign S0F = (EN&&FAULT===5'd11) ? 0 : (EN&&FAULT===5'd12) ? 1 : S0;
 mux4 #(Trise,Tfall) x1 (Q,X0F,X1F,X2F,X3F,S1F,S0F);
 not x2 (QB,Q);
endmodule

module QY1MX4A_GOOD (Q,QB,X0,X1,X2,X3,S1,S0);
 output Q,QB;
 input X0,X1,X2,X3,S1,S0;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 mux4 #(Trise,Tfall) x1 (Q,X0,X1,X2,X3,S1,S0);
 not x2 (QB,Q);
endmodule

module QY1DFA (Q,QB,D,CLK,EN,FAULT);
 output Q,QB;
 input D,CLK;
 input EN;
 input [4:0] FAULT;
 reg O, CLKF_OLD;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 wire Q, DF, CLKF;
 assign #(Trise,Tfall) Q=O;
 assign DF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : D;
 assign CLKF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : CLK;
 always @ (posedge CLKF) begin
 if ((CLKF_OLD===1'b0)&&(CLKF===1'b1)) O = DF;
 if ((CLKF===1'bx)&&(DF!==O)) O = 1'bx;
 if ((CLKF_OLD===1'bx)&&(CLKF===1'b1)&&(DF!==O)) O = 1'bx;
 end
 always @(CLKF) #0.001 CLKF_OLD = CLKF;
 assign QB=~Q;
endmodule

module QY1DFA_GOOD (Q,QB,D,CLK);
 output Q,QB;
 input D,CLK;
 reg O;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 assign #(Trise,Tfall) Q=O;
 always @ (posedge CLK) O = D;
endmodule

module QY1DFRA (Q,QB,D,CLK,R,EN,FAULT);
 output Q,QB;
 input D,CLK,R;
 input EN;
 input [4:0] FAULT;
 reg O, RF_OLD, CLKF_OLD;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 wire Q, DF, CLKF, RF;
 assign #(Trise,Tfall) Q=O;
 assign DF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : D;
 assign CLKF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : CLK;
 assign RF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : R;
 always @ (RF) begin
```

```
if (((RF_OLD===1'b0)&&(RF===1'b1))||((RF_OLD===1'b1)&&(RF===1'b0))) O = (RF === 1) ? 1'b0 :
O;
 if ((RF_OLD===1'b0)&&(RF===1'bx)&&(O!==0)) O = 1'bx;
 if ((RF_OLD===1'bx)&&(RF===1'b1)&&(O!==0)) O = 1'b0;
 end
 always @(RF) #0.001 RF_OLD = RF;
 always @ (posedge CLKF) begin
 if ((CLKF_OLD===1'b0)&&(CLKF===1'b1)) O = (RF === 1) ? 1'b0 : (DF===0) ? 1'b0 : (RF===1'bx) ?
1'bx : DF;
 if (CLKF===1'bx) O = ((RF===1)||((O===0)&&(DF===0))) ? 1'b0 : ((RF===1'b0)&&(O===DF)) ? DF :
1'bx;
 if ((CLKF_OLD===1'bx)&&(CLKF===1'b1)) O = ((RF===1)||((O===0)&&(DF===0))) ? 1'b0 :
((RF===1'b0)&&(O===DF)) ? DF : 1'bx;
 end
 always @(CLKF) #0.001 CLKF_OLD = CLKF;
endmodule

module QY1DFRA_GOOD (Q,QB,D,CLK,R);
 output Q,QB;
 input D,CLK,R;
 reg O;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 assign #(Trise,Tfall) Q=O;
 always @ (R) O = (R === 1) ? 1'b0 : O;
 always @ (posedge CLK) O = (R === 1) ? 1'b0 : D;
endmodule

module QY1BUFA (Q,QB,X1,EN,FAULT);
 output Q,QB;
 input X1;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign QB=~Q;
 assign #(Trise,Tfall) Q=X1F;
endmodule

module QY1BUFA_GOOD (Q,QB,X1);
 output Q,QB;
 input X1;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 assign #(Trise,Tfall) Q=X1;
endmodule

module QY1ZEROA (Q,EN,FAULT);
 output Q;
 wire Q, ZERO;
 input EN;
 input [4:0] FAULT;
 assign ZERO=1'b0;
 assign Q = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : ZERO;
endmodule

module QY1ZEROA_GOOD (ZERO);
 output ZERO;
 assign ZERO=1'b0;
endmodule
```

```
module QY1STA (O,I1,I2,EN,FAULT);
 output O;
 input I1,I2;
 input EN;
 input [4:0] FAULT;
 wire O, I1F, I2F;
 assign I1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : I1;
 assign I2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : I2;
 assign O=I2F;
endmodule

module QY1STA_GOOD (O,I1,I2);
 output O;
 input I1,I2;
 assign O=I2;
endmodule

module QY1DIAGA (O,I1,I2,EN,FAULT);
 output O;
 input I1,I2;
 input EN;
 input [4:0] FAULT;
 wire O, I1F, I2F;
 assign I1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : I1;
 assign I2F = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : I2;
 assign O=I1F;
endmodule

module QY1DIAGA_GOOD (O,I1,I2);
 output O;
 input I1,I2;
 assign O=I1;
endmodule

module QY1S2DA (Q,QB,X,EN,FAULT);
 output Q,QB;
 input X;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, XF;
 assign XF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X;
 assign QB=~Q;
 buf #(Trise,Tfall) x1 (Q,XF);
endmodule

module QY1S2DA_GOOD (Q,QB,X);
 output Q,QB;
 input X;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 assign QB=~Q;
 buf #(Trise,Tfall) x1 (Q,X);
endmodule

module QY1D2SDA (Q,X1,X1B,EN,FAULT);
 output Q;
 input X1,X1B;
 input EN;
 input [4:0] FAULT;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 wire Q, X1F, X1BF;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1B;
 d2s #(Trise,Tfall) z1 (Q,X1F,X1BF);
endmodule

module QY1D2SDA_GOOD (Q,X1,X1B);
 output Q;
 input X1,X1B;
 parameter Trise=0.001;
 parameter Tfall=0.001;
 d2s #(Trise,Tfall) z1 (Q,X1,X1B);
endmodule

module QY1OR2DA (Q,QB,X1,X1B,X2,X2B,EN,FAULT);
 output Q,QB;
 input X1,X1B,X2,X2B;
 input EN;
 input [4:0] FAULT;
 wire X1S,X2S;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 wire Q, X1F, X1BF, X2F, X2BF;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1B;
 assign X2F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X2;
 assign X2BF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X2B;
 d2s #(trr_x1,tff_x1) z1 (X1S,X1F,X1BF);
 d2s #(trr_x2,tff_x2) z2 (X2S,X2F,X2BF);
 or z3 (Q,X1S,X2S);
 not z4 (QB,Q);
endmodule

module QY1OR2DA_GOOD (Q,QB,X1,X1B,X2,X2B);
 output Q,QB;
 input X1,X1B,X2,X2B;
 wire X1S,X2S;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 d2s #(trr_x1,tff_x1) z1 (X1S,X1,X1B);
 d2s #(trr_x2,tff_x2) z2 (X2S,X2,X2B);
 or z3 (Q,X1S,X2S);
 not z4 (QB,Q);
endmodule

module QY1OR3DA (Q,QB,X1,X1B,X2,X2B,X3,X3B,EN,FAULT);
 output Q,QB;
 input X1,X1B,X2,X2B,X3,X3B;
 input EN;
 input [4:0] FAULT;
 wire X1S,X2S,X3S;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 parameter trr_x3=0.001;
 parameter tff_x3=0.001;
 wire Q, X1F, X1BF, X2F, X2BF, X3F, X3BF;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1B;
 assign X2F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X2;
 assign X2BF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X2B;
 assign X3F = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : X3;
 assign X3BF = (EN&&FAULT===5'd11) ? 0 : (EN&&FAULT===5'd12) ? 1 : X3B;
 d2s #(trr_x1,tff_x1) z1 (X1S,X1F,X1BF);
 d2s #(trr_x2,tff_x2) z2 (X2S,X2F,X2BF);
 d2s #(trr_x2,tff_x2) z3 (X3S,X3F,X3BF);
 or z4 (Q,X1S,X2S,X3S);
 not z5 (QB,Q);
endmodule

module QY1OR3DA_GOOD (Q,QB,X1,X1B,X2,X2B,X3,X3B);
 output Q,QB;
 input X1,X1B,X2,X2B,X3,X3B;
 wire X1S,X2S,X3S;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 parameter trr_x3=0.001;
 parameter tff_x3=0.001;
 d2s #(trr_x1,tff_x1) z1 (X1S,X1,X1B);
 d2s #(trr_x2,tff_x2) z2 (X2S,X2,X2B);
 d2s #(trr_x2,tff_x2) z3 (X3S,X3,X3B);
 or z4 (Q,X1S,X2S,X3S);
 not z5 (QB,Q);
endmodule

module QY1MX2DA (Q,QB,X0,X0B,X1,X1B,S,SB,EN,FAULT);
 output Q,QB;
 input X0,X0B,X1,X1B,S,SB;
 input EN;
 input [4:0] FAULT;
 wire X0S,X1S,SS;
 parameter trr_x0=0.001;
 parameter tff_x0=0.001;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter delayS=0.001;
 wire Q, X0F, X0BF, X1F, X1BF, SF, SBF;
 assign X0F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X0;
 assign X0BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X0B;
 assign X1F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X1B;
 assign SF = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : S;
 assign SBF = (EN&&FAULT===5'd11) ? 0 : (EN&&FAULT===5'd12) ? 1 : SB;
 d2s #(trr_x0,tff_x0) z1 (X0S,X0F,X0BF);
 d2s #(trr_x1,tff_x1) z2 (X1S,X1F,X1BF);
 d2s #delayS z3 (SS,SF,SBF);
 mux2 z4 (Q,X0S,X1S,SS);
 not z5 (QB,Q);
endmodule

module QY1MX2DA_GOOD (Q,QB,X0,X0B,X1,X1B,S,SB);
 output Q,QB;
 input X0,X0B,X1,X1B,S,SB;
 wire X0S,X1S,SS;
 parameter trr_x0=0.001;
 parameter tff_x0=0.001;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter delayS=0.001;
 d2s #(trr_x0,tff_x0) z1 (X0S,X0,X0B);
 d2s #(trr_x1,tff_x1) z2 (X1S,X1,X1B);
 d2s #delayS z3 (SS,S,SB);
 mux2 z4 (Q,X0S,X1S,SS);
 not z5 (QB,Q);
endmodule

module QY1MX4DA (Q,QB,X0,X0B,X1,X1B,X2,X2B,X3,X3B,S1,S1B,S0,S0B,EN,FAULT);
 output Q,QB;
 input X0,X0B,X1,X1B,X2,X2B,X3,X3B,S1,S1B,S0,S0B;
 input EN;
 input [4:0] FAULT;
 wire X0S,X1S,X2S,X3S,S1S,S0S;
 parameter trr_x0=0.001;
 parameter tff_x0=0.001;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 parameter trr_x3=0.001;
 parameter tff_x3=0.001;
 parameter delayS0=0.001;
 parameter delayS1=0.001;
 wire Q, X0F, X0BF, X1F, X1BF, X2F, X2BF, X3F, X3BF, S1F, S1BF, S0F, S0BF;
 assign X0F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X0;
 assign X0BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X0B;
 assign X1F = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : X1B;
 assign X2F = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : X2;
 assign X2BF = (EN&&FAULT===5'd11) ? 0 : (EN&&FAULT===5'd12) ? 1 : X2B;
 assign X3F = (EN&&FAULT===5'd13) ? 0 : (EN&&FAULT===5'd14) ? 1 : X3;
 assign X3BF = (EN&&FAULT===5'd15) ? 0 : (EN&&FAULT===5'd16) ? 1 : X3B;
 assign S1F = (EN&&FAULT===5'd17) ? 0 : (EN&&FAULT===5'd18) ? 1 : S1;
 assign S1BF = (EN&&FAULT===5'd19) ? 0 : (EN&&FAULT===5'd20) ? 1 : S1B;
 assign S0F = (EN&&FAULT===5'd21) ? 0 : (EN&&FAULT===5'd22) ? 1 : S0;
 assign S0BF = (EN&&FAULT===5'd23) ? 0 : (EN&&FAULT===5'd24) ? 1 : S0B;
 d2s #(trr_x0,tff_x0) z1 (X0S,X0F,X0BF);
 d2s #(trr_x1,tff_x1) z2 (X1S,X1F,X1BF);
 d2s #(trr_x2,tff_x2) z3 (X2S,X2F,X2BF);
 d2s #(trr_x3,tff_x3) z4 (X3S,X3F,X3BF);
 d2s #delayS0 z5 (S0S,S0F,S0BF);
 d2s #delayS1 z6 (S1S,S1F,S1BF);
 mux4 z7 (Q,X0S,X1S,X2S,X3S,S1S,S0S);
 not z8 (QB,Q);
endmodule

module QY1MX4DA_GOOD (Q,QB,X0,X0B,X1,X1B,X2,X2B,X3,X3B,S1,S1B,S0,S0B);
 output Q,QB;
 input X0,X0B,X1,X1B,X2,X2B,X3,X3B,S1,S1B,S0,S0B;
 wire X0S,X1S,X2S,X3S,S1S,S0S;
 parameter trr_x0=0.001;
 parameter tff_x0=0.001;
 parameter trr_x1=0.001;
 parameter tff_x1=0.001;
 parameter trr_x2=0.001;
 parameter tff_x2=0.001;
 parameter trr_x3=0.001;
 parameter tff_x3=0.001;
 parameter delayS0=0.001;
 parameter delayS1=0.001;
 d2s #(trr_x0,tff_x0) z1 (X0S,X0,X0B);
 d2s #(trr_x1,tff_x1) z2 (X1S,X1,X1B);
 d2s #(trr_x2,tff_x2) z3 (X2S,X2,X2B);
 d2s #(trr_x3,tff_x3) z4 (X3S,X3,X3B);
 d2s #delayS0 z5 (S0S,S0,S0B);
 d2s #delayS1 z6 (S1S,S1,S1B);
 mux4 z7 (Q,X0S,X1S,X2S,X3S,S1S,S0S);
 not z8 (QB,Q);
endmodule

module QY1CP1DA (Q,QB,R,RB,CLK,CLKB, EN, FAULT);
 output Q,QB;
 input R,RB,CLK,CLKB;
 input EN;
 input [4:0] FAULT;
 wire RS,CLKS, RF, RBF, CLKF, CLKBF;
 reg Q, RS_OLD, CLKS_OLD;
 parameter delayR=0.001;
 parameter delayCLK=0.001;
 assign RF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : R;
 assign RBF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : RB;
 assign CLKF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : CLK;
 assign CLKBF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : CLKB;
 d2s #delayR z1(RS,RF,RBF);
 d2s z2(CLKS,CLKF,CLKBF);
 assign QB=~Q;
 initial begin
 Q = 1'b0;
 end
 always @(negedge faultdrive.start_run) Q = 1'b0;
 always @ (RS) begin
 if (((RS_OLD===1'b0)&&(RS===1'b1))||((RS_OLD===1'b1)&&(RS===1'b0))) Q = (RS === 1) ? 1'b0 :
Q;
 if ((RS_OLD===1'b0)&&(RS===1'bx)&&(Q!==0)) Q = 1'bx;
 if ((RS_OLD===1'bx)&&(RS===1'b1)) Q = 1'b0;
 end
 always @(RS) #0.001 RS_OLD = RS;
 initial #0.1 forever @ (posedge CLKS) begin
 if ((CLKS_OLD===1'b0)&&(CLKS===1'b1)) #delayCLK Q = (RS === 1'b1) ? 1'b0 : (RS===1'b0) ?
1'b1 : 1'bx;
 if (CLKS===1'bx) #delayCLK Q = (RS===1'b1) ? 1'b0 : ((RS===1'b0)&&(Q===1'b1)) ? 1'b1 : 1'bx;
 if ((CLKS_OLD===1'bx)&&(CLKS===1'b1)) #delayCLK Q = (RS===1'b1) ? 1'b0 :
((RS===1'b0)&&(Q===1'b1)) ? 1'b1 : 1'bx;
 end
 always @(CLKS) #0.001 CLKS_OLD = CLKS;
endmodule

module QY1CP1DA_GOOD (Q,QB,R,RB,CLK,CLKB);
 output Q,QB;
 input R,RB,CLK,CLKB;
 wire RS,CLKS;
 reg Q;
 parameter delayR=0.001;
 parameter delayCLK=0.001;
 d2s #delayR z1(RS,R,RB);
 d2s z2(CLKS,CLK,CLKB);
 assign QB=~Q;
 always @ (RS) Q = (RS === 1) ? 1'b0 : Q;
 always @ (posedge CLKS) #delayCLK Q = (RS === 1) ? 1'b0 : 1'b1;
endmodule

module QY1DFDA (Q,QB,D,DB,CLK,CLKB,EN,FAULT);
 output Q,QB;
 input D,DB,CLK,CLKB;
 input EN;
 input [4:0] FAULT;
 wire DS,CLKS, DF, DBF, CLKF, CLKBF;
 reg Q, CLKS_OLD;initial CLKS_OLD=1'b0;
 parameter delayCLK=0.001;
 assign DF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : D;
 assign DBF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : DB;
 assign CLKF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : CLK;
 assign CLKBF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : CLKB;
 d2s z1(DS,DF,DBF);
 d2s z2(CLKS,CLKF,CLKBF);
 assign QB=~Q;
 always @ (posedge CLKS) begin
 if ((CLKS_OLD===1'b0)&&(CLKS===1'b1)) #delayCLK Q = DS;
 if ((CLKS===1'bx)&&(DS!==Q)) #delayCLK Q = 1'bx;
 if ((CLKS_OLD===1'bx)&&(CLKS===1'b1)&&(DS!==Q)) #delayCLK Q = 1'bx;
 end
 always @(CLKS) #0.001 CLKS_OLD = CLKS;
endmodule

module QY1DFDA_GOOD (Q,QB,D,DB,CLK,CLKB);
 output Q,QB;
 input D,DB,CLK,CLKB;
 wire DS,CLKS;
 reg Q;
 parameter delayCLK=0.001;
 d2s z1(DS,D,DB);
 d2s z2(CLKS,CLK,CLKB);
 assign QB=~Q;
 always @ (posedge CLKS) #delayCLK Q = DS;
endmodule

module QY1DFRDA (Q,QB,D,DB,CLK,CLKB,R,RB,EN,FAULT);
 output Q,QB;
 input D,DB,CLK,CLKB,R,RB;
 input EN;
 input [4:0] FAULT;
 wire DS,CLKS,RS, DF, DBF, CLKF, CLKBF, RF, RBF;
 reg Q, RS_OLD, CLKS_OLD;
 parameter delayCLK=0.001;
 parameter delayR=0.001;
 assign DF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : D;
 assign DBF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : DB;
 assign CLKF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : CLK;
 assign CLKBF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : CLKB;
 assign RF = (EN&&FAULT===5'd9) ? 0 : (EN&&FAULT===5'd10) ? 1 : R;
 assign RBF = (EN&&FAULT===5'd11) ? 0 : (EN&&FAULT===5'd12) ? 1 : RB;
 d2s z1(DS,DF,DBF);
 d2s z2(CLKS,CLKF,CLKBF);
 d2s z3(RS,RF,RBF);
 assign QB=~Q;
 always @ (RS) begin
 if (((RS_OLD===1'b0)&&(RS===1'b1))||((RS_OLD===1'b1)&&(RS===1'b0))) #delayR Q = (RS ===
1) ? 1'b0 : Q;
 if ((RS_OLD===1'b0)&&(RS===1'bx)&&(Q!==0)) Q = 1'bx;
 if ((RS_OLD===1'bx)&&(RS===1'b1)&&(Q!==0)) Q = 1'b0;
 end
 always @(RS) #0.001 RS_OLD = RS;
 always @ (posedge CLKS) begin
 if ((CLKS_OLD===1'b0)&&(CLKS===1'b1)) #delayCLK Q = (RS === 1) ? 1'b0 :(DS===0) ? 1'b0 :
(RS===1'bx) ? 1'bx : DS;
 if (CLKS===1'bx) #delayCLK Q = ((RS===1)||((Q===0)&&(DS===0))) ? 1'b0 :
((RS===1'b0)&&(Q===DS)) ? DS : 1'bx;
 if ((CLKS_OLD===1'bx)&&(CLKS===1'b1)) #delayCLK Q = ((RS===1)||((Q===0)&&(DS===0))) ?
1'b0 : ((RS===1'b0)&&(Q===DS)) ? DS : 1'bx;
 end
 always @(CLKS) #0.001 CLKS_OLD = CLKS;
endmodule

module QY1DFRDA_GOOD (Q,QB,D,DB,CLK,CLKB,R,RB);
```

```
 output Q,QB;
 input D,DB,CLK,CLKB,R,RB;
 wire DS,CLKS,RS;
 reg Q;
 parameter delayCLK=0.001;
 parameter delayR=0.001;
 d2s z1(DS,D,DB);
 d2s z2(CLKS,CLK,CLKB);
 d2s z3(RS,R,RB);
 assign QB=~Q;
 always @ (RS) #delayR Q = (RS === 1) ? 1'b0 : Q;
 always @ (posedge CLKS) #delayCLK Q = (RS === 1) ? 1'b0 : DS;
endmodule

module QY1RSDA (Q,QB,S,SB,R,RB,EN,FAULT);
 output Q,QB;
 input S,SB,R,RB;
 input EN;
 input [4:0] FAULT;
 wire SS,RS, SF, SBF, RF, RBF;
 reg Q, RS_OLD, SS_OLD;initial RS_OLD=1'b0; initial SS_OLD=1'b0;
 parameter delayR=0.001;
 parameter delayS=0.001;
 assign SF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : S;
 assign SBF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : SB;
 assign RF = (EN&&FAULT===5'd5) ? 0 : (EN&&FAULT===5'd6) ? 1 : R;
 assign RBF = (EN&&FAULT===5'd7) ? 0 : (EN&&FAULT===5'd8) ? 1 : RB;
 d2s #delayR z1(RS,RF,RBF);
 d2s #delayS z2(SS,SF,SBF);
 assign QB=~Q;
 initial Q=1'b0;
 always @(negedge faultdrive.start_run) Q = 1'b0;
 always @ (RS) begin
 if (((RS_OLD===1'b0)&&(RS===1'b1))||((RS_OLD===1'b1)&&(RS===1'b0))) Q = (RS===1) ? 1'b0 :
((SS===1) ? 1'b1 : Q);
 if ((RS_OLD===1'b0)&&(RS===1'bx)) Q = ((Q===0)&&(SS===0)) ? 1'b0 : 1'bx;
 if ((RS_OLD===1'bx)&&(RS===1'b1)) Q = 1'b0;
 end
 always @(RS) #0.001 RS_OLD = RS;
 always @ (SS) begin
 if (((SS_OLD===1'b0)&&(SS===1'b1))||((SS_OLD===1'b1)&&(SS===1'b0))) Q = (RS===1) ? 1'b0 :
((SS===1) ? 1'b1 : Q);
 if ((SS_OLD===1'b0)&&(SS===1'bx)) Q = ((Q===1)&&(RS===0)) ? 1'b1 : 1'bx;
 if ((SS_OLD===1'bx)&&(SS===1'b1)) Q = (RS===1) ? 1'b0 : (RS===0) ? 1'b1 : 1'bx;
 end
 always @(SS) #0.001 SS_OLD = SS;
endmodule

module QY1RSDA_GOOD (Q,QB,S,SB,R,RB);
 output Q,QB;
 input S,SB,R,RB;
 wire SS,RS;
 reg Q;
 parameter delayR=0.001;
 parameter delayS=0.001;
 d2s #delayR z1(RS,R,RB);
 d2s #delayS z2(SS,S,SB);
 assign QB=~Q;
 initial Q=1'b0;
 always @ (RS or SS) Q = (RS===1) ? 1'b0 : ((SS===1) ? 1'b1 : Q);
endmodule

module QY1DELDA (Q,QB,X1,X1B,EN,FAULT);
 output Q,QB;
 input X1,X1B;
 input EN;
 input [4:0] FAULT;
 wire Q, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, X1F, X1BF;
 assign X1F = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X1;
 assign X1BF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : X1B;
 parameter delay1 = 0.100;
 parameter delay2 = 0.100;
 parameter delay3 = 0.100;
 parameter delay4 = 0.100;
 parameter delay5 = 0.100;
 parameter delay6 = 0.100;
 parameter delay7 = 0.100;
 parameter delay8 = 0.049;
 assign QB=~Q;
 d2s z1 (Q1,X1F,X1BF);
 assign #delay1 Q2 = Q1;
 assign #delay2 Q3 = Q2;
 assign #delay3 Q4 = Q3;
 assign #delay4 Q5 = Q4;
 assign #delay5 Q6 = Q5;
 assign #delay6 Q7 = Q6;
 assign #delay7 Q8 = Q7;
 assign #delay8 Q = Q8;
endmodule

module QY1DELDA_GOOD (Q,QB,X1,X1B);
 output Q,QB;
 input X1,X1B;
 assign QB=~Q;
 d2s #(1.498,0.001) z1 (Q,X1,X1B);
endmodule

module QY1ONEDA (Q,QB,EN,FAULT);
 output Q,QB;
 input EN;
 input [4:0] FAULT;
 wire Q, Q2;
 assign QB=~Q;
 assign Q = (EN&&FAULT===5'd1) ? 0 : 1;
endmodule

module QY1ONEDA_GOOD (Q,QB);
 output Q,QB;
 assign Q=1'b1;
 assign QB=1'b0;
endmodule

module vbidriv (ECL,Q,X,Y,EN,FAULT);
 output Q;
 inout ECL;
 input X,Y;
 input EN;
 input [4:0] FAULT;
 wire Q, XF, YF;
 assign XF = (EN&&FAULT===5'd1) ? 0 : (EN&&FAULT===5'd2) ? 1 : X;
 assign YF = (EN&&FAULT===5'd3) ? 0 : (EN&&FAULT===5'd4) ? 1 : Y;
 and (strong1,weak0) x1 (ECL,XF,YF);
 buf x2 (Q,ECL);
endmodule

module vbidriv_GOOD (ECL,Q,X,Y);
 output Q;
 inout ECL;
 input X,Y;
 and (strong1,weak0) x1 (ECL,X,Y);
 buf x2 (Q,ECL);
endmodule

module vmono (O,OB,T);
 input T;
 output O,OB;
 reg O;
 wire T,OB;
 parameter delay=2;
 assign OB=~O;
 initial O=1'b0;
 always @ (posedge T)
 if (T === 1'b1) begin
 O=1'b1;
 #delay O=1'b0;
 end
endmodule

module vrs (O,OB,R,S);
 input R,S;
 output O,OB;
 reg O;
 wire R,S,OB;
 parameter delay=0;
 assign OB=~O;
 initial O=1'b0;
 always @ (R or S) #delay O = (R==1) ? 1'b0 : ((S==1) ? 1'b1 : O);
endmodule

module v1 (O);
 output O;
 assign O=1'b1;
endmodule

module v0 (O);
 output O;
 assign O=1'b0;
endmodule

module vnot (O,OB,I);
 parameter delay=0;
 input I;
 output O,OB;
 buf #delay x1 (O,I);
 not x2 (OB,O);
endmodule

module vand2 (O,OB,I1,I2);
 parameter delay=0;
 input I1,I2;
 output O,OB;
 and #delay x1 (O,I1,I2);
 not x2 (OB,O);
endmodule

module vand3 (O,OB,I1,I2,I3);
 parameter delay=0;
 input I1,I2,I3;
 output O,OB;
 and #delay x1 (O,I1,I2,I3);
 not x2 (OB,O);
endmodule
```

```
module vor2 (O,OB,I1,I2);
 parameter delay=0;
 input I1,I2;
 output O,OB;
 or #delay x1 (O,I1,I2);
 not x2 (OB,O);
endmodule

module vor3 (O,OB,I1,I2,I3);
 parameter delay=0;
 input I1,I2,I3;
 output O,OB;
 or #delay x1 (O,I1,I2,I3);
 not x2 (OB,O);
endmodule

module vor4 (O,OB,I1,I2,I3,I4);
 parameter delay=0;
 input I1,I2,I3,I4;
 output O,OB;
 or #delay x1 (O,I1,I2,I3,I4);
 not x2 (OB,O);
endmodule

module vor5 (O,OB,I1,I2,I3,I4,I5);
 parameter delay=0;
 input I1,I2,I3,I4,I5;
 output O,OB;
 or #delay x1 (O,I1,I2,I3,I4,I5);
 not x2 (OB,O);
endmodule

module vmux2 (O,OB,X0,X1,S);
 parameter delay=0;
 input X0,X1,S;
 output O,OB;
 mux2 #delay x1 (O,X0,X1,S);
 not x2 (OB,O);
endmodule

module vmux4 (O,OB,X0,X1,X2,X3,S1,S0);
 parameter delay=0;
 input X0,X1,X2,X3,S1,S0;
 output O,OB;
 mux4 #delay x1 (O,X0,X1,X2,X3,S1,S0);
 not x2 (OB,O);
endmodule

module vdf (O,OB,D,CLK);
 parameter delay=0;
 input D,CLK;
 output O,OB;
 reg O;
 assign OB=~O;
 always @ (posedge CLK) #delay O = D;
endmodule

module vdfr (O,OB,D,CLK,R);
 parameter delay=0;
 input D,CLK,R;
 output O,OB;
 reg O;
 assign OB=~O;
 always @ (R) #delay O = (R === 1) ? 1'b0 : O;
 always @ (posedge CLK) #delay O = (R === 1) ? 1'b0 : D;
endmodule

primitive mux2 (o,x0,x1,s);
 output o;
 input s,x0,x1;
 table
 // x0 x1 s o
 0 ? 0 : 0;
 1 ? 0 : 1;
 ? 0 1 : 0;
 ? 1 1 : 1;
 0 0 x : 0;
 1 1 x : 1;
 endtable
endprimitive

primitive d2s (o,x,xb);
 output o;
 input x,xb;
 table
 // x xb o;
 0 1 : 0;
 1 0 : 1;
 endtable
endprimitive

primitive mux4 (o,x0,x1,x2,x3,s1,s0);
 output o;
 input x0,x1,x2,x3,s1,s0;
 table
 // x0 x1 x2 x3 s1 s0 o
 0 ? ? ? 0 0 : 0;
 1 ? ? ? 0 0 : 1;
 ? 0 ? ? 0 1 : 0;
 ? 1 ? ? 0 1 : 1;
 ? ? 0 ? 1 0 : 0;
 ? ? 1 ? 1 0 : 1;
 ? ? ? 0 1 1 : 0;
 ? ? ? 1 1 1 : 1;
 0 ? 0 ? x 0 : 0;
 1 ? 1 ? x 0 : 1;
 ? 0 ? 0 x 1 : 0;
 ? 1 ? 1 x 1 : 1;
 0 0 ? ? 0 x : 0;
 1 1 ? ? 0 x : 1;
 ? ? 0 0 1 x : 0;
 ? ? 1 1 1 x : 1;
 0 0 0 0 x x : 0;
 1 1 1 1 x x : 1;
 endtable
 endprimitive
```

A.2 Linear Delay Line Models

```
'timescale 1ns / 1ps
/*********************************LDL**********************************/
'define LSB EVENT[5]

module LDL_HP (VOUT,VOUTB,VOTST,VO,VOB,VI,VIB,TYPE_OUT,TYPE_OUTB,bus_out,
               EVENT,bus_in,VITST,
               SEL,EN,MR,F6,TST,TCLK);

 output VOUT,VOUTB,VOTST,VO,VI,VOB,VIB;
 output [1:0] TYPE_OUT,TYPE_OUTB;
 output [9:2] bus_out;
 input [5:0] EVENT;
 input [9:2] bus_in;
 input VITST,SEL,MR;
 input EN;
 input F6,TST,TCLK; // F6 is bit 6 of func (register address)

real            actual_delay;
wire    [9:2]   bus_out;
wire            vin;

reg             VOUT;
reg     [9:2]   GAIN;           // LDL drive gain register
reg     [9:2]   DELAY;          // LDL drive delay register
reg     [4:0]   hold_reg; // register holding MSB of delay
reg     [1:0]   event_reg,type_reg;
reg             next_clock_trigger,trigger;

initial trigger = 0;

assign  bus_out = ((EN==1) ? ((F6 == 0) ? GAIN : DELAY) : 8'b0);

// output event TYPE
assign  TYPE_OUT = type_reg;
assign VOUTB=~VOUT;
assign TYPE_OUTB=~TYPE_OUT;
assign VOB=~VO;
assign VIB=~VI;

always @ (posedge SEL) GAIN = bus_in;

always @ (posedge TCLK)
 begin
 if ('LSB == 1) begin // MSB is in the hold_reg
        DELAY = {hold_reg[3:0],EVENT[3:0]}; // concatenate MSB & LSB of delay
        event_reg = {hold_reg[4],EVENT[4]}; // concatenate event type
 end
 hold_reg = EVENT[4:0]; // DTIME latch into hold_reg every TCLK
 end

always @ (posedge TCLK) begin
 next_clock_trigger = 'LSB;
 @ (negedge TCLK)
 trigger = (next_clock_trigger === 1'bx) ? 1'b0 : next_clock_trigger;
end

always @ (posedge vin) type_reg = event_reg;

/****************************** LDL Core ******************************/
/* the time unit is ns */
/* the values listed below are subject to change. */
```

```
parameter VI_pulse_width = 0.5; // 0.5 ns
parameter VOUT_pulse_width = 1.5; // 1.5 ns
parameter To = 2.5; // zero_delay = 2.5 ns
parameter Cramp = 3; // ramp_capacitor = 3.0 pf
parameter volt_per_LSB = 1.5; // delay_level_delta = 1.5 mV
parameter Imax = 500; // max_gain_current = 500 uA
parameter dI = 1; // ramp_current_delta = 1.0 uA

parameter dV = Cramp * volt_per_LSB;
parameter mdf = 1.0; // 0.7 < Min_Delay_Factor < 1.3

initial    VOUT = 0;

assign    VOTST = (TST === 1) ? VOUT : 1'b0;

assign    VO = VOUT;

assign VI = VITST | (TCLK & trigger);

not #(VI_pulse_width) (VI_,VI);
and (vin,VI,VI_);

always @ (posedge vin) begin
 actual_delay = (GAIN===8'bx) ? (To + mdf*DELAY*dV/(Imax-255*dI))
                              : (To + mdf*DELAY*dV/(Imax-GAIN*dI));
 #(actual_delay) VOUT = 1;
end

always @ (posedge VOUT)
 #(VOUT_pulse_width) VOUT = 0; // reset to 0 after VOUT_pulse_width

endmodule
```

Appendix B

Device Models

B.1 Drive Control IC

```
'timescale 1ps/1ps
/****************************** Drive Control ******************************/
module drive_control_gate (DHIA,DINHA,DHIB,DINHB,
                           SETLOOUT,SETHIOUT,SETZOUT,SETONOUT,
                           TMUPA,TMUPB,
                           R,RS,RTXC,
                           SETLOIN,SETHIIN,SETZIN,SETONIN,
                           DA,DB,DC,DD,
                           THBSEL,DCLK, FAULT_SEL, GATE_SEL);
 output DHIA,DINHA,DHIB,DINHB; // to PE
 output SETLOOUT,SETHIOUT,SETZOUT,SETONOUT; // to even pin
 output TMUPA,TMUPB; // to Response Control's TMU Mux
 inout [9:2] R; // BDE from Event Logic Chip
 input [1:0] RS; // BSE from Event Logic Chip
 input RTXC; // BTXCE from Event Logic Chip
        /* RTXC is pulsed only when RS is 3 (function cycle)
        or the register in Drive/Response Control is addressed
        if RS is 0 or 1 (read/write cycle) */
 input SETLOIN,SETHIIN,SETZIN,SETONIN; // from odd pin
 input [5:0] DA,DB,DC,DD; // from ESS
 input THBSEL,DCLK;
 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

wor     [9:2]    R;
wire    [9:2]    data;
wire    [7:0]    func;
wire    [5:0]    funcb;
wire [1023:0] GATE_NUM;
assign GATE_NUM = 1 << GATE_SEL;

wire TMUPAB,TMUPBB; // to Response Control's TMU Mux
wire DHIAB,DINHAB,DHIBB,DINHBB,NDHI,NDINH;
wire SETLOOUTB,SETHIOUTB,SETZOUTB,SETONOUTB;
wire SETLOINB,SETHIINB,SETZINB,SETONINB;
wire    DVOUTA,DVOUTB,DVOUTC,DVOUTD;
wire    DVOUTAB,DVOUTBB,DVOUTCB,DVOUTDB;
wire    DVOTSTA,DVOTSTB,DVOTSTC,DVOTSTD;
wire    DVOA,DVOB,DVOC,DVOD;
wire    DVIA,DVIB,DVIC,DVID;
wire    DVOAB,DVOBB,DVOCB,DVODB;
wire    DVIAB,DVIBB,DVICB,DVIDB;
wire    [3:0]    DVO = {DVOD,DVOC,DVOB,DVOA}, // to TMU
                 DVI = {DVID,DVIC,DVIB,DVIA}; // to TMU
wire    [3:0]    NDVO = {DVODB,DVOCB,DVOBB,DVOAB}, // to TMU
                 NDVI = {DVIDB,DVICB,DVIBB,DVIAB}; // to TMU
wire    PMXDHI,PMXDINH; // to TMU for device test purpose
wire    PMXDHIB,PMXDINHB; // to TMU for device test purpose
wire    [1:0]    TYPEA,TYPEB,TYPEC,TYPED;
wire    [1:0]    TYPEAB,TYPEBB,TYPECB,TYPEDB;
wor     [9:2]    bus_out; // bus from each LDL
wire    [9:2]    bus_in = R; // bus from Registers to LDLs
wire [9:2] RI;
wire      SELA,SELB,SELC,SELD; // indicate which LDL is selected
wire      ENA,ENB,ENC,END; // indicate which LDL is selected

wire      TGICRESET; // MR (master reset)
wire [4:0] HSPATHBSEL;
wire [2:1] HSPATHASEL;
wire [2:0] PECONTROL;
wire [1:0] PINSTATUS;

 QY1ZEROA X096 (ZERO, GATE_NUM[1], FAULT_SEL); // 0 input

 QY1BUFA X015 (N062,N004,RS[1], GATE_NUM[6], FAULT_SEL);// 1 input
 QY1BUFA X001 (N066,N001,RI[5], GATE_NUM[7], FAULT_SEL);

 QY1AND2A X035 (SELCLK,N055,N005,WCLK, GATE_NUM[34], FAULT_SEL);// 2 input
 QY1AND2A X036 (SELA,N051,ENA,SELCLK, GATE_NUM[35], FAULT_SEL);
 QY1AND2A X037 (SELB,N050,ENB,SELCLK, GATE_NUM[36], FAULT_SEL);
 QY1AND2A X038 (SELC,N049,ENC,SELCLK, GATE_NUM[37], FAULT_SEL);
 QY1AND2A X039 (SELD,N048,END,SELCLK, GATE_NUM[38], FAULT_SEL);
 QY1AND2A X004 (TGICRESET,N065,N003,FNCCLK, GATE_NUM[39], FAULT_SEL);
 QY1DFA X005 (func[7],N061,RI[9],FNCCLK, GATE_NUM[40], FAULT_SEL);
 QY1DFA X006 (func[6],N060,RI[8],FNCCLK, GATE_NUM[41], FAULT_SEL);
 QY1DFA X007 (func[5],funcb[5],RI[7],FNCCLK, GATE_NUM[42], FAULT_SEL);
 QY1DFA X008 (func[4],funcb[4],RI[6],FNCCLK, GATE_NUM[43], FAULT_SEL);
 QY1DFA X009 (func[3],funcb[3],RI[5],FNCCLK, GATE_NUM[44], FAULT_SEL);
 QY1DFA X010 (func[2],funcb[2],RI[4],FNCCLK, GATE_NUM[45], FAULT_SEL);
 QY1DFA X011 (func[1],funcb[1],RI[3],FNCCLK, GATE_NUM[46], FAULT_SEL);
 QY1DFA X012 (func[0],funcb[0],RI[2],FNCCLK, GATE_NUM[47], FAULT_SEL);
 QY1DFA X063 (HSPATHBSEL[1],N096,RI[9],HSBLDCLK, GATE_NUM[48], FAULT_SEL);
 QY1DFA X064 (HSPATHBSEL[0],N095,RI[8],HSBLDCLK, GATE_NUM[49], FAULT_SEL);
 QY1DFA X060 (HSPATHBSEL[4],N094,RI[4],HSBLDCLK, GATE_NUM[50], FAULT_SEL);
 QY1DFA X061 (HSPATHBSEL[3],N093,RI[3],HSBLDCLK, GATE_NUM[51], FAULT_SEL);
 QY1DFA X062 (HSPATHBSEL[2],N092,RI[2],HSBLDCLK, GATE_NUM[52], FAULT_SEL);
 QY1DFA X097 (HSPATHASEL[1],N091,RI[9],HSALDCLK, GATE_NUM[53], FAULT_SEL);
 QY1DFA X065 (HSPATHASEL[2],N090,RI[2],HSALDCLK, GATE_NUM[54], FAULT_SEL);
 QY1DIAGA S01 (N137,func[2],funcb[2], GATE_NUM[55], FAULT_SEL);
 QY1DIAGA S02 (N136,func[1],funcb[1], GATE_NUM[56], FAULT_SEL);
 QY1DIAGA S03 (N135,HSLDEN,Z, GATE_NUM[57], FAULT_SEL);
 QY1DIAGA S04 (N134,func[4],funcb[4], GATE_NUM[58], FAULT_SEL);
 QY1DIAGA S05 (N133,TGICDIAGM_bit5,TGICDIAGM_bit6, GATE_NUM[59], FAULT_SEL);
 QY1DIAGA S10 (N142,PECONTROL[0],ZERO, GATE_NUM[60], FAULT_SEL);
 QY1DIAGA S07 (N132,PECONTROL[1],HSPATHASEL[1], GATE_NUM[61], FAULT_SEL);
 QY1DIAGA S08 (N131,TGICDIAGM_bit1,ZERO, GATE_NUM[62], FAULT_SEL);
 QY1DIAGA S09 (N130,ZERO,PINSTATUS[1], GATE_NUM[63], FAULT_SEL);
 QY1DIAGA S06 (N129,ZERO,func[0], GATE_NUM[64], FAULT_SEL);
 QY1DIAGA S11 (N138,TGICDIAGM_bit0,ZERO, GATE_NUM[65], FAULT_SEL);
 QY1DIAGA S12 (N139,ZERO,PINSTATUS[0], GATE_NUM[66], FAULT_SEL);
 QY1DIAGA S13 (N128,ZERO,EVENTMODE_bit7, GATE_NUM[67], FAULT_SEL);
 QY1DIAGA S14 (N127,ZERO,TGICDIAGM_bit6, GATE_NUM[68], FAULT_SEL);
 QY1DIAGA S16 (N140,EVENTMODE_bit3,ZERO, GATE_NUM[69], FAULT_SEL);
 QY1DIAGA S18 (N143,PECONTROL[2],HSPATHASEL[2], GATE_NUM[70], FAULT_SEL);
 QY1DIAGA S21 (N073,NOT06,ZERO, GATE_NUM[71], FAULT_SEL);
 QY1OR2A X030 (N045,S1,PECLRSET,EVCLRSET, GATE_NUM[72], FAULT_SEL);
 QY1OR2A X031 (N044,S0,PECLRSET,TGCLRSET, GATE_NUM[73], FAULT_SEL);
 QY1OR2A X098 (N057,NOT06,HSLDENB,func[0], GATE_NUM[74], FAULT_SEL);
 QY1STA S15 (N126,ZERO,TGICDIAGM_bit5, GATE_NUM[75], FAULT_SEL);
 QY1STA S20 (N141,ZERO,HSPATHBSEL[2], GATE_NUM[76], FAULT_SEL);
 QY1STA S19 (N125,ZERO,TGICDIAGM_bit2, GATE_NUM[77], FAULT_SEL);
 QY1STA S17 (N124,ZERO,HSPATHBSEL[3], GATE_NUM[78], FAULT_SEL);
 vbidriv P66 (R[2],RI[2],data[2],OUTEN, GATE_NUM[79], FAULT_SEL);
 vbidriv P67 (R[3],RI[3],data[3],OUTEN, GATE_NUM[80], FAULT_SEL);
 vbidriv P68 (R[4],RI[4],data[4],OUTEN, GATE_NUM[81], FAULT_SEL);
 vbidriv P69 (R[5],RI[5],data[5],OUTEN, GATE_NUM[82], FAULT_SEL);
 vbidriv P70 (R[6],RI[6],data[6],OUTEN, GATE_NUM[83], FAULT_SEL);
vbidriv P71 (R[7],RI[7],data[7],OUTEN, GATE_NUM[84], FAULT_SEL);
vbidriv P72 (R[8],RI[8],data[8],OUTEN, GATE_NUM[85], FAULT_SEL);
vbidriv P73 (R[9],RI[9],data[9],OUTEN, GATE_NUM[86], FAULT_SEL);

QY1AND3A X032 (HSBLDCLK,N047,HSLDEN,func[0],WCLK, GATE_NUM[146],
                    FAULT_SEL);// 3input
QY1AND3A X033 (HSALDCLK,N046,HSLDEN,funcb[0],WCLK, GATE_NUM[147], FAULT_SEL);
QY1AND3A X013 (FNCCLK,N064,RTXC,RS[0],RS[1], GATE_NUM[148], FAULT_SEL);
QY1AND3A X014 (WCLK,N063,RTXC,RS[0],N004, GATE_NUM[149], FAULT_SEL);
QY1AND3A X040 (N006,N086,RI[7],TGCLRSET,WCLK, GATE_NUM[150], FAULT_SEL);
QY1AND3A X041 (N007,N085,RI[6],TGCLRSET,WCLK, GATE_NUM[151], FAULT_SEL);
QY1AND3A X042 (N008,N084,RI[5],TGCLRSET,WCLK, GATE_NUM[152], FAULT_SEL);
QY1AND3A X045 (N011,N083,RI[2],TGCLRSET,WCLK, GATE_NUM[153], FAULT_SEL);
QY1AND3A X044 (N010,N082,RI[3],TGCLRSET,WCLK, GATE_NUM[154], FAULT_SEL);
QY1AND3A X043 (N009,N081,RI[4],TGCLRSET,WCLK, GATE_NUM[155], FAULT_SEL);
QY1AND3A X047 (N013,N080,RI[8],TGCLRSET,WCLK, GATE_NUM[156], FAULT_SEL);
QY1AND3A X046 (N012,N079,RI[9],TGCLRSET,WCLK, GATE_NUM[157], FAULT_SEL);
QY1AND3A X056 (N014,N078,RI[7],EVCLRSET,WCLK, GATE_NUM[158], FAULT_SEL);
QY1AND3A X057 (N015,N077,RI[3],EVCLRSET,WCLK, GATE_NUM[159], FAULT_SEL);
QY1AND3A X069 (N016,N099,RI[2],PECLRSET,WCLK, GATE_NUM[160], FAULT_SEL);
QY1AND3A X070 (N017,N098,RI[9],PECLRSET,WCLK, GATE_NUM[161], FAULT_SEL);
QY1AND3A X071 (N018,N097,RI[8],PECLRSET,WCLK, GATE_NUM[162], FAULT_SEL);
QY1DFRA X048 (TGICDIAGM_bitF,N076,func[0],N006,TGICRESET, GATE_NUM[163],
                    FAULT_SEL);
QY1DFRA X049 (TGICDIAGM_bit6,N075,func[0],N007,TGICRESET, GATE_NUM[164],
                    FAULT_SEL);
QY1DFRA X050 (TGICDIAGM_bit5,N074,func[0],N008,TGICRESET, GATE_NUM[165],
                    FAULT_SEL);
QY1DFRA X051 (TGICDIAGM_bit4,TGICDIAGM_bit4B,func[0],N009,TGICRESET,
                    GATE_NUM[166], FAULT_SEL);
QY1DFRA X052 (TGICDIAGM_bit3,N072,func[0],N010,TGICRESET, GATE_NUM[167],
                    FAULT_SEL);
QY1DFRA X053 (TGICDIAGM_bit2,N071,func[0],N011,TGICRESET, GATE_NUM[168],
                    FAULT_SEL);
QY1DFRA X054 (TGICDIAGM_bit1,N070,func[0],N012,TGICRESET, GATE_NUM[169],
                    FAULT_SEL);
QY1DFRA X055 (TGICDIAGM_bit0,N069,func[0],N013,TGICRESET, GATE_NUM[170],
                    FAULT_SEL);
QY1DFRA X058 (EVENTMODE_bit7,N068,func[0],N014,TGICRESET, GATE_NUM[171],
                    FAULT_SEL);
QY1DFRA X059 (EVENTMODE_bit3,N067,func[0],N015,TGICRESET, GATE_NUM[172],
                    FAULT_SEL);
QY1DFRA X066 (PECONTROL[2],N089,func[0],N016,TGICRESET, GATE_NUM[173], FAULT_SEL);
QY1DFRA X067 (PECONTROL[1],N088,func[0],N017,TGICRESET, GATE_NUM[174], FAULT_SEL);
QY1DFRA X068 (PECONTROL[0],N087,func[0],N018,TGICRESET, GATE_NUM[175], FAULT_SEL);
QY1MX2A X079 (N026,N122,N143,HSPATHBSEL[2],N129, GATE_NUM[176], FAULT_SEL);
QY1MX2A X095 (data[2],N109,N034,bus_out[2],LDLRD, GATE_NUM[177], FAULT_SEL);
QY1MX2A X078 (N025,N108,ZERO,HSPATHBSEL[3],N129, GATE_NUM[178], FAULT_SEL);
QY1MX2A X094 (data[3],N107,N033,bus_out[3],LDLRD, GATE_NUM[179], FAULT_SEL);
QY1MX2A X077 (N024,N120,ZERO,HSPATHBSEL[4],N129, GATE_NUM[180], FAULT_SEL);
QY1MX2A X093 (data[4],N106,N032,bus_out[4],LDLRD, GATE_NUM[181], FAULT_SEL);
QY1MX2A X076 (N029,N118,ZERO,ZERO,N129, GATE_NUM[182], FAULT_SEL);
QY1MX2A X092 (data[5],N105,N031,bus_out[5],LDLRD, GATE_NUM[183], FAULT_SEL);
QY1MX2A X075 (N022,N116,ZERO,ZERO,N129, GATE_NUM[184], FAULT_SEL);
QY1MX2A X091 (data[6],N104,N030,bus_out[6],LDLRD, GATE_NUM[185], FAULT_SEL);
QY1MX2A X074 (N021,N114,ZERO,ZERO,N129, GATE_NUM[186], FAULT_SEL);
QY1MX2A X090 (data[7],N103,N023,bus_out[7],LDLRD, GATE_NUM[187], FAULT_SEL);
QY1MX2A X073 (N020,N112,N142,HSPATHBSEL[0],N129, GATE_NUM[188], FAULT_SEL);
QY1MX2A X089 (data[8],N102,N028,bus_out[8],LDLRD, GATE_NUM[189], FAULT_SEL);
QY1MX2A X072 (N019,N110,N132,HSPATHBSEL[1],N129, GATE_NUM[190], FAULT_SEL);
QY1MX2A X088 (data[9],N101,N027,bus_out[9],LDLRD, GATE_NUM[191], FAULT_SEL);
QY1OR3A X024 (LDLRDB,LDLRD,func[7],funcb[5],N134, GATE_NUM[192], FAULT_SEL);
QY1OR3A X023 (N043,X,func[3],func[2],LDLRDB, GATE_NUM[193], FAULT_SEL);
QY1OR3A X026 (N056,ENA,func[1],LDLRDB,func[0], GATE_NUM[194], FAULT_SEL);
```

```
QY1OR3A X027 (N052,ENB,func[1],LDLRDB,funcb[0], GATE_NUM[195], FAULT_SEL);
QY1OR3A X028 (N053,ENC,funcb[1],LDLRDB,func[0], GATE_NUM[196], FAULT_SEL);
QY1OR3A X029 (N054,END,funcb[1],LDLRDB,funcb[0], GATE_NUM[197], FAULT_SEL);

QY1OR4A X020 (MSB0,N042,func[7],func[6],func[5],func[4], GATE_NUM[250],
                    FAULT_SEL);// 4 input
QY1OR4A X016 (N041,PECLRSET,N137,N136,func[3],MSB0, GATE_NUM[251], FAULT_SEL);
QY1OR4A X017 (N040,EVCLRSET,func[3],funcb[2],func[1],MSB0, GATE_NUM[252], FAULT_SEL);
QY1OR4A X018 (N039,TGCLRSET,funcb[3],func[2],funcb[1],MSB0, GATE_NUM[253], FAULT_SEL);
QY1OR4A X019 (HSLDENB,HSLDEN,func[3],funcb[2],funcb[1],MSB0, GATE_NUM[254],
                    FAULT_SEL);
QY1OR4A X025 (N038,N005,LDLRDB,func[6],func[3],func[2], GATE_NUM[255], FAULT_SEL);
QY1OR4A X003 (N059,N003,RI[9],RI[8],N001,N002, GATE_NUM[256], FAULT_SEL);

QY1OR5A X002 (N002,N058,RI[7],RI[6],RI[4],RI[3],RI[2], GATE_NUM[428], FAULT_SEL);// 5 input
QY1OR5A X021 (N036,Z,MSB0,func[3],func[2],funcb[1],func[0], GATE_NUM[429], FAULT_SEL);
QY1OR5A X022 (N035,Y,N135,TGCLRSET,EVCLRSET,PECLRSET,X, GATE_NUM[430],
                    FAULT_SEL);
QY1OR5A X034 (N037,OUTEN,N073,N133,RS[0],RS[1],Y, GATE_NUM[431], FAULT_SEL);

QY1MX4A X087 (N034,N123,N026,ZERO,N125,N141,S1,S0, GATE_NUM[436],
                    FAULT_SEL);   // 6 input
QY1MX4A X086 (N033,N100,N025,N140,TGICDIAGM_bit3,N124,S1,S0, GATE_NUM[437],
                    FAULT_SEL);
QY1MX4A X085 (N032,N121,N024,ZERO,TGICDIAGM_bit4,ZERO,S1,S0,
                    GATE_NUM[438], FAULT_SEL);
QY1MX4A X084 (N031,N119,N029,ZERO,N126,ZERO,S1,S0, GATE_NUM[439], FAULT_SEL);
QY1MX4A X083 (N030,N117,N022,ZERO,N127,ZERO,S1,S0, GATE_NUM[440], FAULT_SEL);
QY1MX4A X082 (N023,N115,N021,N128,TGICDIAGM_bitF,ZERO,S1,S0,
                    GATE_NUM[441], FAULT_SEL);
QY1MX4A X081 (N028,N113,N020,ZERO,N138,N139,S1,S0, GATE_NUM[442], FAULT_SEL);
QY1MX4A X080 (N027,N111,N019,ZERO,N131,N130,S1,S0, GATE_NUM[443], FAULT_SEL);

/****************************** sub-module *******************************/
/* The LDL includes Event Logic Interface */

LDL_HPLDL_A(DVOUTA,DVOUTAB,DVOTSTA,DVOA,DVOAB,DVIA,DVIAB,TYPEA,TYPEAB,
            bus_out,DA,bus_in,DVOTSTD,SELA,ENA,
            TGICRESET,func[6],TGICDIAGM_bit4,DCLK);

LDL_HPLDL_B(DVOUTB,DVOUTBB,DVOTSTB,DVOB,DVOBB,DVIB,DVIBB,TYPEB,TYPEBB,
            bus_out,DB,bus_in,DVOTSTA,SELB,ENB,
            TGICRESET,func[6],TGICDIAGM_bit4,DCLK);

LDL_HPLDL_C(DVOUTC,DVOUTCB,DVOTSTC,DVOC,DVOCB,DVIC,DVICB,TYPEC,TYPECB,
            bus_out,DC,bus_in,DVOTSTB,SELC,ENC,
            TGICRESET,func[6],TGICDIAGM_bit4,DCLK);

LDL_HPLDL_D(DVOUTD,DVOUTDB,DVOTSTD,DVOD,DVODB,DVID,DVIDB,TYPED,TYPEDB,
            bus_out,DD,bus_in,DVOTSTC,SELD,END,
            TGICRESET,func[6],TGICDIAGM_bit4,DCLK);

assign SETLOINB=~SETLOIN;
assign SETHIINB=~SETHIIN;
assign SETZINB=~SETZIN;
assign SETONINB=~SETONIN;
drive_logic_fault drive_logic_fault(DHIA,DINHA,DHIB,DINHB,DHI,DINH,
                                    SETLOOUT,SETHIOUT,SETZOUT,SETONOUT,
                                    PMXDHI,PMXDINH,
                                    SETLOIN,SETHIIN,SETZIN,SETONIN,
                                    DVOUTA,DVOUTB,DVOUTC,DVOUTD,
                                    TYPEA,TYPEB,TYPEC,TYPED,
                                     DHIAB,DINHAB,DHIBB,DINHBB,NDHI,NDINH,
                                    SETLOOUTB,SETHIOUTB,SETZOUTB,SETONOUTB,
                                   PMXDHIB,PMXDINHB,
                                   SETLOINB,SETHIINB,SETZINB,SETONINB,
                                   DVOUTAB,DVOUTBB,DVOUTCB,DVOUTDB,
                                   TYPEAB,TYPEBB,TYPECB,TYPEDB,
                                   THBSEL,PECONTROL[2],PECONTROL[1],PECONTROL[0],
                                   EVENTMODE_bit3,TGICDIAGM_bit3,TGICDIAGM_bit2,
                                   FAULT_SEL, GATE_SEL);

drive_TMU_fault drive_TMU_fault(TMUPA,TMUPB,DVO,DVI,DHI,DINH,PMXDHI,PMXDINH,
                                   TMUPAB,TMUPBB,NDVO,NDVI,NDHI,NDINH,PMXDHIB,
                                   PMXDINHB,HSPATHBSEL[3:2],TGICDIAGM_bitF,
                                   TGICDIAGM_bit3,TGICDIAGM_bit1,TGICDIAGM_bit0,
                                   FAULT_SEL, GATE_SEL);

endmodule

/******************************* Drive Logic *******************************/
/* Drive Logic's main function is to set/reset DHI and DINH, and create
 SETLOOUT,SETHIOUT,SETZOUT and,SETONOUT pulses.(SET_OUT pulses)
 Whenver there is a DVOUT* coming in, Drive Logic will decode its type
 to determine its action.
 Whenever there is a SET_IN pulse coming, Drive Logic may set/reset
 DHI/DINH.
 If RDCF is set, then RDIN and RDHI is used to determine the status
 of DHI/DINH and disregard of any DVOUT* and SET_IN pulses.
 PMXDHI and PMXDINH are used for die testing purpose. When TSTPULSE is
 set, SET_OUT pulses will set/reset them to indicate the presence of the
 pulses.
 When TSTVOH is set, the outputs of SET_OUT remains high and this is
 used to check the VOH of these outputs
*/

module drive_logic_fault (DHIA,DINHA,DHIB,DINHB,DHI,DINH,
                          SETLOOUT,SETHIOUT,SETZOUT,SETONOUT,
                          PMXDHI,PMXDINH,
                          SETLOIN,SETHIIN,SETZIN,SETONIN,
                          DVOUTA,DVOUTB,DVOUTC,DVOUTD,
                          TYPEA,TYPEB,TYPEC,TYPED,
                          DHIAB,DINHAB,DHIBB,DINHBB,NDHI,NDINH,
                          SETLOOUTB,SETHIOUTB,SETZOUTB,SETONOUTB,
                          PMXDHIB,PMXDINHB,
                          SETLOINB,SETHIINB,SETZINB,SETONINB,
                          DVOUTAB,DVOUTBB,DVOUTCB,DVOUTDB,
                          TYPEAB,TYPEBB,TYPECB,TYPEDB,
                          THBSEL,RDCF,RDIN,RDHI,PMM,TSTPULSE,TSTVOH,
                          FAULT_SEL, GATE_SEL);

 output DHIA,DINHA,DHIB,DINHB;
 output DHIAB,DINHAB,DHIBB,DINHBB;
 output DHI,DINH; // to TMU Mux
 output NDHI,NDINH; // to TMU Mux (comp)
 output SETLOOUT,SETHIOUT,SETZOUT,SETONOUT; // Pin Mux Mode output
 output SETLOOUTB,SETHIOUTB,SETZOUTB,SETONOUTB; // Pin Mux Mode output (comp)
 output PMXDHI,PMXDINH; // device test purpose
 output PMXDHIB,PMXDINHB; // device test purpose (comp)

 input SETLOIN,SETHIIN,SETZIN,SETONIN; // Pin Mux Mode input
 input SETLOINB,SETHIINB,SETZINB,SETONINB; // Pin Mux Mode input (comp)
 input DVOUTA,DVOUTB,DVOUTC,DVOUTD; // LDL output pulse
 input DVOUTAB,DVOUTBB,DVOUTCB,DVOUTDB; // LDL output pulse (comp)
 input [1:0] TYPEA,TYPEB,TYPEC,TYPED; // Event type
 input [1:0] TYPEAB,TYPEBB,TYPECB,TYPEDB; // Event type (comp)
 input THBSEL;
 input RDCF,RDIN,RDHI; // PECONTROL[2:0]
 input PMM; // bit 3 of EVENTMODE
 input TSTPULSE; // bit 3 of TGICDIAGM
 input TSTVOH; // bit 2 of TGICDIAGM
 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

wire [1023:0] GATE_NUM;
assign GATE_NUM = 1 << GATE_SEL;

 QY1ONEDA X84 (ONE,ONEB, GATE_NUM[2], FAULT_SEL);// 0 input

 QY1S2DA X53 (N020,N101,TSTVOH, GATE_NUM[8], FAULT_SEL);// 1 input
 QY1S2DA X54 (N092,N104,RDCF, GATE_NUM[9], FAULT_SEL);
 QY1S2DA X59 (N157,N158,RDIN, GATE_NUM[10], FAULT_SEL);
 QY1S2DA X60 (N079,N078,PMM, GATE_NUM[11], FAULT_SEL);
 QY1S2DA X65 (N019,N018,TSTPULSE, GATE_NUM[12], FAULT_SEL);
 QY1S2DA X71 (N022,N021,THBSEL, GATE_NUM[13], FAULT_SEL);
 QY1S2DA X83 (N106,N107,RDHI, GATE_NUM[14], FAULT_SEL);

 QY1DELDA X88 (N133,N134,N115,N085, GATE_NUM[87], FAULT_SEL);// 2 input
 QY1DELDA X92 (N135,N136,N116,N089, GATE_NUM[88], FAULT_SEL);
 QY1DELDA X97 (N137,N138,N010,N011, GATE_NUM[89], FAULT_SEL);
 QY1DELDA X101 (N139,N140,N012,N013, GATE_NUM[90], FAULT_SEL);

 QY1OR2DA X01 (N090,N091,TYPEAB[1],TYPEA[1],TYPEAB[0],TYPEA[0], GATE_NUM[257],
                  FAULT_SEL);// 4 input
 QY1OR2DA X02 (N094,N095,TYPEBB[1],TYPEB[1],TYPEBB[0],TYPEB[0], GATE_NUM[258],
                  FAULT_SEL);
 QY1OR2DA X03 (N098,N099,TYPECB[1],TYPEC[1],TYPECB[0],TYPEC[0], GATE_NUM[259],
                  FAULT_SEL);
 QY1OR2DA X04 (N096,N097,TYPEDB[1],TYPED[1],TYPEDB[0],TYPED[0], GATE_NUM[260],
                  FAULT_SEL);
 QY1OR2DA X05 (N059,N060,TYPEAB[1],TYPEA[1],TYPEA[0],TYPEAB[0], GATE_NUM[261],
                  FAULT_SEL);
 QY1OR2DA X06 (N057,N058,TYPEBB[1],TYPEB[1],TYPEB[0],TYPEBB[0], GATE_NUM[262],
                  FAULT_SEL);
 QY1OR2DA X07 (N001,N054,TYPECB[1],TYPEC[1],TYPEC[0],TYPECB[0], GATE_NUM[263],
                  FAULT_SEL);
 QY1OR2DA X08 (N055,N056,TYPEDB[1],TYPED[1],TYPED[0],TYPEDB[0], GATE_NUM[264],
                  FAULT_SEL);
 QY1OR2DA X09 (N028,N036,TYPEA[1],TYPEAB[1],TYPEA[0],TYPEAB[0], GATE_NUM[265],
                  FAULT_SEL);
 QY1OR2DA X10 (N029,N037,TYPEB[1],TYPEBB[1],TYPEB[0],TYPEBB[0], GATE_NUM[266],
                  FAULT_SEL);
 QY1OR2DA X11 (N032,N039,TYPEC[1],TYPECB[1],TYPEC[0],TYPECB[0], GATE_NUM[267],
                  FAULT_SEL);
 QY1OR2DA X12 (N035,N038,TYPED[1],TYPEDB[1],TYPED[0],TYPEDB[0], GATE_NUM[268],
                  FAULT_SEL);
 QY1OR2DA X13 (N088,N118,DVOUTAB,DVOUTA,N090,N091, GATE_NUM[269], FAULT_SEL);
 QY1OR2DA X14 (N119,N093,DVOUTBB,DVOUTB,N094,N095, GATE_NUM[270], FAULT_SEL);
 QY1OR2DA X15 (N083,N120,DVOUTCB,DVOUTC,N098,N099, GATE_NUM[271], FAULT_SEL);
 QY1OR2DA X16 (N121,N084,DVOUTDB,DVOUTD,N096,N097, GATE_NUM[272], FAULT_SEL);
 QY1OR2DA X17 (N114,N122,DVOUTAB,DVOUTA,N059,N060, GATE_NUM[273], FAULT_SEL);
 QY1OR2DA X18 (N123,N063,DVOUTBB,DVOUTB,N057,N058, GATE_NUM[274], FAULT_SEL);
 QY1OR2DA X19 (N066,N124,DVOUTCB,DVOUTC,N001,N054, GATE_NUM[275], FAULT_SEL);
 QY1OR2DA X20 (N125,N067,DVOUTDB,DVOUTD,N055,N056, GATE_NUM[276], FAULT_SEL);
 QY1OR2DA X21 (N112,N126,DVOUTAB,DVOUTA,N028,N036, GATE_NUM[277], FAULT_SEL);
 QY1OR2DA X22 (N127,N113,DVOUTBB,DVOUTB,N029,N037, GATE_NUM[278], FAULT_SEL);
 QY1OR2DA X23 (N045,N128,DVOUTCB,DVOUTC,N032,N039, GATE_NUM[279], FAULT_SEL);
 QY1OR2DA X24 (N129,N046,DVOUTDB,DVOUTD,N035,N038, GATE_NUM[280], FAULT_SEL);
 QY1OR2DA X25 (N033,N130,DVOUTAB,DVOUTA,N036,N028, GATE_NUM[281], FAULT_SEL);
 QY1OR2DA X26 (N131,N034,DVOUTBB,DVOUTB,N037,N029, GATE_NUM[282], FAULT_SEL);
 QY1OR2DA X27 (N027,N026,DVOUTCB,DVOUTC,N039,N032, GATE_NUM[283], FAULT_SEL);
 QY1OR2DA X28 (N132,N117,DVOUTDB,DVOUTD,N038,N035, GATE_NUM[284], FAULT_SEL);
```

```
QY1OR2DA X29 (N086,N087,N118,N088,N093,N119, GATE_NUM[285], FAULT_SEL);
QY1OR2DA X30 (N080,N081,N120,N083,N084,N121, GATE_NUM[286], FAULT_SEL);
QY1OR2DA X31 (N070,N071,N122,N114,N063,N123, GATE_NUM[287], FAULT_SEL);
QY1OR2DA X32 (N072,N073,N124,N066,N067,N125, GATE_NUM[288], FAULT_SEL);
QY1OR2DA X33 (N048,N049,N126,N112,N113,N127, GATE_NUM[289], FAULT_SEL);
QY1OR2DA X34 (N050,N051,N128,N045,N046,N129, GATE_NUM[290], FAULT_SEL);
QY1OR2DA X35 (N030,N031,N130,N033,N034,N131, GATE_NUM[291], FAULT_SEL);
QY1OR2DA X36 (N024,N025,N026,N027,N117,N132, GATE_NUM[292], FAULT_SEL);
QY1OR2DA X37 (N061,N047,N086,N087,N080,N081, GATE_NUM[293], FAULT_SEL);
QY1OR2DA X38 (N069,N074,N070,N071,N072,N073, GATE_NUM[294], FAULT_SEL);
QY1OR2DA X39 (N052,N053,N048,N049,N050,N051, GATE_NUM[295], FAULT_SEL);
QY1OR2DA X40 (N023,N044,N030,N031,N024,N025, GATE_NUM[296], FAULT_SEL);
QY1OR2DA X41 (SETDHI,SETDHIB,N061,N047,SETHIIN,SETHIINB, GATE_NUM[297],
                    FAULT_SEL);
QY1OR2DA X42 (N075,N076,N047,N061,N078,N079, GATE_NUM[298], FAULT_SEL);
QY1OR2DA X43 (N082,N077,N074,N069,N078,N079, GATE_NUM[299], FAULT_SEL);
QY1OR2DA X44 (RSETDHI,RSETDHIB,N069,N074,SETLOIN,SETLOINB, GATE_NUM[300],
                    FAULT_SEL);
QY1OR2DA X45 (SETOFF,SETOFFB,N052,N053,SETZIN,SETZINB, GATE_NUM[301],
                    FAULT_SEL);
QY1OR2DA X46 (N040,N041,N053,N052,N078,N079, GATE_NUM[302], FAULT_SEL);
QY1OR2DA X47 (N042,N043,N044,N023,N078,N079, GATE_NUM[303], FAULT_SEL);
QY1OR2DA X48 (RSETOFF,RSETOFFB,N023,N044,SETONIN,SETONINB, GATE_NUM[304],
                    FAULT_SEL);
QY1CP1DA X49 (N062,N064,N159,N160,SETDHI,SETDHIB, GATE_NUM[305], FAULT_SEL);
QY1CP1DA X50 (N108,N109,N161,N162,RSETDHI,RSETDHIB, GATE_NUM[306], FAULT_SEL);
QY1CP1DA X55 (N100,N105,N163,N164,SETOFF,SETOFFB, GATE_NUM[307], FAULT_SEL);
QY1CP1DA X56 (N110,N111,N165,N166,RSETOFF,RSETOFFB, GATE_NUM[308], FAULT_SEL);
QY1OR2DA X63 (SETHIOUT,SETHIOUTB,N115,N085,N020,N101, GATE_NUM[309], FAULT_SEL);
QY1OR2DA X64 (SETLOOUT,SETLOOUTB,N116,N089,N020,N101, GATE_NUM[310],
                    FAULT_SEL);
QY1OR2DA X68 (SETZOUT,SETZOUTB,N010,N011,N020,N101, GATE_NUM[311], FAULT_SEL);
QY1OR2DA X69 (SETONOUT,SETONOUTB,N012,N013,N020,N101, GATE_NUM[312],
                    FAULT_SEL);
QY1RSDA X70 (DHI,NDHI,N065,N068,N009,N103, GATE_NUM[313], FAULT_SEL);
QY1RSDA X72 (DINH,NDINH,N007,N008,N102,N006, GATE_NUM[314], FAULT_SEL);
QY1OR2DA X73 (DHIBB,DHIB,NDHI,DHI,N021,N022, GATE_NUM[315], FAULT_SEL);
QY1OR2DA X74 (DHIAB,DHIA,NDHI,DHI,N022,N021, GATE_NUM[316], FAULT_SEL);
QY1OR2DA X75 (N016,N017,SETHIOUTB,SETHIOUT,N018,N019, GATE_NUM[317], FAULT_SEL);
QY1OR2DA X76 (N014,N015,SETLOOUTB,SETLOOUT,N018,N019, GATE_NUM[318],
                    FAULT_SEL);
QY1OR2DA X77 (DINHBB,DINHB,NDINH,DINH,N021,N022, GATE_NUM[319], FAULT_SEL);
QY1OR2DA X78 (DINHAB,DINHA,NDINH,DINH,N022,N021, GATE_NUM[320], FAULT_SEL);
QY1OR2DA X79 (N005,N004,SETZOUTB,SETZOUT,N018,N019, GATE_NUM[321], FAULT_SEL);
QY1OR2DA X80 (N003,N002,SETONOUTB,SETONOUT,N018,N019, GATE_NUM[322],
                    FAULT_SEL);
QY1RSDA X81 (PMXDHI,PMXDHIB,N017,N016,N015,N014, GATE_NUM[323], FAULT_SEL);
QY1RSDA X82 (PMXDINH,PMXDINHB,N004,N005,N002,N003, GATE_NUM[324], FAULT_SEL);
QY1CP1DA X85 (N149,N150,N115,N085,N076,N075, GATE_NUM[325], FAULT_SEL);
QY1DELDA X86 (N141,N142,N133,N134, GATE_NUM[326], FAULT_SEL);
QY1RSDA X87 (N115,N085,N149,N150,N141,N142, GATE_NUM[327], FAULT_SEL);
QY1CP1DA X89 (N151,N152,N116,N089,N077,N082, GATE_NUM[328], FAULT_SEL);
QY1DELDA X90 (N143,N144,N135,N136, GATE_NUM[329], FAULT_SEL);
QY1RSDA X91 (N116,N089,N151,N152,N143,N144, GATE_NUM[330], FAULT_SEL);
QY1CP1DA X94 (N153,N154,N010,N011,N041,N040, GATE_NUM[331], FAULT_SEL);
QY1DELDA X95 (N145,N146,N137,N138, GATE_NUM[332], FAULT_SEL);
QY1RSDA X96 (N010,N011,N153,N154,N145,N146, GATE_NUM[333], FAULT_SEL);
QY1CP1DA X98 (N155,N156,N012,N013,N043,N042, GATE_NUM[334], FAULT_SEL);
QY1DELDA X99 (N147,N148,N139,N140, GATE_NUM[335], FAULT_SEL);
QY1RSDA X100 (N012,N013,N155,N156,N147,N148, GATE_NUM[336], FAULT_SEL);
QY1OR2DA X102 (N159,N160,DHI,NDHI,N092,N104, GATE_NUM[337], FAULT_SEL);
QY1OR2DA X103 (N161,N162,NDHI,DHI,N092,N104, GATE_NUM[338], FAULT_SEL);
QY1OR2DA X104 (N163,N164,DINH,NDINH,N092,N104, GATE_NUM[339], FAULT_SEL);
QY1OR2DA X105 (N165,N166,NDINH,DINH,N092,N104, GATE_NUM[340], FAULT_SEL);
```

```
QY1MX2DA X61 (N065,N068,N062,N064,N106,N107,N092,N104, GATE_NUM[444],
                        FAULT_SEL);// 6 input
QY1MX2DA X62 (N009,N103,N108,N109,N107,N106,N092,N104, GATE_NUM[445], FAULT_SEL);
QY1MX2DA X66 (N007,N008,N100,N105,N157,N158,N092,N104, GATE_NUM[446], FAULT_SEL);
QY1MX2DA X67 (N102,N006,N110,N111,N158,N157,N092,N104, GATE_NUM[447], FAULT_SEL);

endmodule

/****************************** Drive TMU Mux ******************************/
/* The main fucnction of Drive TMU Mux is to output several signals through
 TMUPA,TMUPB to RESPONSE CONTROL. HSPATHBSEL(bit 3 and 2) is used to
 indicate which LDL and TGICDIAGM(bit 15,3,1 and 0) is used to select
 which signal to outputs (TMUPA/TMUPB). The following table relates the
 output signals to the codes of HSPATHBSEL and TGICDIAGM(LPBK,DRSTAT and
 STRBZ):

 HSPATHBSEL[3:2] LPBK,DRSTAT,STRBZ S9100 Signal(TMUPA/TMUPB)
 --------------- ----------------- -------------------------
        00       0XX               DVIA/DVOA
        01       0XX               DVIB/DVOB
        10       0XX               DVIC/DVOC
        11       0XX               DVID/DVOD
        XX       100               DHI/DINH
        XX       101               PMXDHI/PMXDINH <-- NEW
        XX       110               DHI/DHI_ <-- NEW
        XX       111               DINH_/DINH_ <-- NEW

 Bit 3 of TGICDIAGM (TSTPULSE) is used for die testing. When it is
 set, and the selected signal is DVI* or DVO*, the output will toggle
 instead of output the pulses whenever the selected LDL creates a DVI
 pulse or DVO pulse respectively.
*/

module drive_TMU_fault(TMUPA,TMUPB,DVO,DVI,DHI,DINH,PMXDHI,PMXDINH,
 TMUPAB,TMUPBB,DVOB,DVIB,DHIB,DINHB,PMXDHIB,PMXDINHB,
 HSPATHBSEL,LPBK,TSTPULSE,DRSTAT,STRBZ, FAULT_SEL, GATE_SEL);
 output TMUPA,TMUPB; // to RESPONSE CONTROL's TMU Mux
 output TMUPAB,TMUPBB; // to RESPONSE CONTROL's TMU Mux
 input [3:0] DVO,DVI; // DVI[0]/DVO[0] = DVIA/DVOA
                      // DVI[1]/DVO[1] = DVIB/DVOB
                      // DVI[2]/DVO[2] = DVIC/DVOC
                      // DVI[3]/DVO[3] = DVID/DVOD
 input [3:0] DVOB,DVIB;
 input DHI,DINH; // from Drive logic
 input DHIB,DINHB; // from Drive logic (comp)
 input PMXDHI,PMXDINH; // from Drive Logic
 input PMXDHIB,PMXDINHB; // from Drive Logic
 input [3:2] HSPATHBSEL;
 input LPBK, // bit 15 of TGICDIAGM
 TSTPULSE, // bit 3 of TGICDIAGM
 DRSTAT, // bit 1 of TGICDIAGM
 STRBZ; // bit 0 of TGICDIAGM

 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

 wire [1023:0] GATE_NUM;
 assign GATE_NUM = 1 << GATE_SEL;

 wire      DVIN,DVOUT;

 QY1S2DA X01 (N11,N12,HSPATHBSEL[3], GATE_NUM[15], FAULT_SEL);// 1 input
 QY1S2DA X02 (N09,N10,HSPATHBSEL[2], GATE_NUM[16], FAULT_SEL);
```

```
QY1S2DA X03 (N07,N08,TSTPULSE, GATE_NUM[17], FAULT_SEL);
QY1S2DA X04 (N01,N02,DRSTAT, GATE_NUM[18], FAULT_SEL);
QY1S2DA X05 (N03,N04,STRBZ, GATE_NUM[19], FAULT_SEL);
QY1S2DA X06 (N05,N06,LPBK, GATE_NUM[20], FAULT_SEL);

QY1MX2DA X11 (TMUPA,TMUPAB,DVIN,DVINB,N13,N14,N05,N06, GATE_NUM[448],
                    FAULT_SEL);   // 6 input
QY1MX2DA X12 (DVIN,DVINB,N15,N16,DVIPULSE,DVIPULSEB,N07,N08, GATE_NUM[449],
                    FAULT_SEL);
QY1DFRDA X13 (DVIPULSE,DVIPULSEB,DVIPULSEB,DVIPULSE,N15,N16,N08,N07,
                    GATE_NUM[450], FAULT_SEL);
QY1MX2DA X14 (DVOUT,DVOUTB,N17,N18,DVOPULSE,DVOPULSEB,N07,N08,
                    GATE_NUM[451], FAULT_SEL);
QY1DFRDA X15 (DVOPULSE,DVOPULSEB,DVOPULSEB,DVOPULSE,N17,N18,N08,N07,
                    GATE_NUM[452], FAULT_SEL);
QY1MX2DA X16 (TMUPB,TMUPBB,DVOUT,DVOUTB,N19,N20,N05,N06, GATE_NUM[453],
                    FAULT_SEL);

QY1MX4DA X07 (N13,N14,DHI,DHIB,PMXDHI,PMXDHIB,DHI,DHIB,
                    DINHB,DINH,N01,N02,N03,N04, GATE_NUM[502], FAULT_SEL); // 12 input
QY1MX4DA X08 (N15,N16,DVI[0],DVIB[0],DVI[1],DVIB[1],DVI[2],DVIB[2],DVI[3],
                    DVIB[3],N11,N12,N09,N10, GATE_NUM[503], FAULT_SEL);
QY1MX4DA X09 (N17,N18,DVO[0],DVOB[0],DVO[1],DVOB[1],DVO[2],DVOB[2],DVO[3],
                    DVOB[3],N11,N12,N09,N10, GATE_NUM[504], FAULT_SEL);
QY1MX4DA X10 (N19,N20,DINH,DINHB,PMXDINH,PMXDINHB,DHIB,DHI,DINHB,
                    DINH,N01,N02,N03,N04, GATE_NUM[505], FAULT_SEL);

endmodule
```

B.2 Response Control IC

```
'timescale 1ps/1ps
/*************************** Response Control ***************************/
module response_control_gate (STFLA,STFLB,STFLC,STFLD,
                                        TMUA,TMUB,DCLK,
                                        R,RS,RTXC,
                                        ACHA,BCLA,ACHB,BCLB,
                                        TMUPA,TMUPB,
                                        CA,CB,CC,CD,
                                        THBSEL,CLK, FAULT_SEL, GATE_SEL);
 output STFLA,STFLB,STFLC,STFLD; // to Event Logic Chip
 output TMUA,TMUB; // to TMU
 output DCLK; // to DRIVE CONTROL
 inout [9:2] R; // BDE from Event Logic Chip
 input [1:0] RS; // BSE from Event Logic Chip
 input RTXC; // BTXCE from Event Logic Chip
 input ACHA,BCLA,ACHB,BCLB; // from Test Head
 input TMUPA,TMUPB; // from Drive Control
 input [5:0] CA,CB,CC,CD; // from ESS
 input THBSEL,CLK;
 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

 wire [1023:0] GATE_NUM;
 assign GATE_NUM = 1 << GATE_SEL;
 wire ACHAB,BCLAB,ACHBB,BCLBB; // from Test Head
 wire TMUPAB,TMUPBB; // from Drive Control

wor     [9:2]    R;
wire    [9:2]    RI;
wire    [9:2]    data;
wire    [7:0]    func;
wire    [5:0]    funcb;
wire [2:1] HSPATHASEL;
wire [4:0] HSPATHBSEL;
wire [2:0] PECONTROL;
wire     CVOUTA,CVOUTB,CVOUTC,CVOUTD;
wire     CVOTSTA,CVOTSTB,CVOTSTC,CVOTSTD;
wire     CVOA,CVOB,CVOC,CVOD;
wire     CVIA,CVIB,CVIC,CVID;
wire     CVOAB,CVOBB,CVOCB,CVODB;
wire     CVIAB,CVIBB,CVICB,CVIDB;
wire     [3:0]    CVO = {CVOD,CVOC,CVOB,CVOA}, // to TMU
                  CVI = {CVID,CVIC,CVIB,CVIA}; // to TMU
wire     [3:0]    NCVO = {CVODB,CVOCB,CVOBB,CVOAB}, // to TMU
                  NCVI = {CVIDB,CVICB,CVIBB,CVIAB}; // to TMU
wire     [1:0]    TYPEA,TYPEB,TYPEC,TYPED;
wire     [1:0]    TYPEAB,TYPEBB,TYPECB,TYPEDB;
wor      [9:2]    bus_out;
wire     [9:2]    bus_in = R; // bus to LDLs
wire CLKB;
wire ACH,ACHB,BCL,BCLB;

wire     SELA,SELB,SELC,SELD; // indicate which LDL is selected

parameter clock_buffer_delay=0.8;
wire     #clock_buffer_delay DCLK = CLK;

wire     [1:0]    PINSTATUS; // PINSTATUS[1:0] = {ACH,BCL}
wire     TGICRESET;
wire ENA,ENB,ENC,END;
wire NACH,NBCL;
```

```
assign CLKB=~CLK;

LDL_HPLDL_A(CVOUTA,CVOUTAB,CVOTSTA,CVOA,CVOAB,CVIA,CVIAB,TYPEA,
            TYPEAB,bus_out,CA,bus_in,CVOTSTD,
            SELA,ENA,TGICRESET,func[6],TGICDIAGM_bit4,CLK);

LDL_HPLDL_B(CVOUTB,CVOUTBB,CVOTSTB,CVOB,CVOBB,CVIB,CVIBB,TYPEB,
            TYPEBB,bus_out,CB,bus_in,CVOTSTA,
            SELB,ENB,TGICRESET,func[6],TGICDIAGM_bit4,CLK);

LDL_HPLDL_C(CVOUTC,CVOUTCB,CVOTSTC,CVOC,CVOCB,CVIC,CVICB,TYPEC,
            TYPECB,bus_out,CC,bus_in,CVOTSTB,
            SELC,ENC,TGICRESET,func[6],TGICDIAGM_bit4,CLK);

LDL_HPLDL_D(CVOUTD,CVOUTDB,CVOTSTD,CVOD,CVODB,CVID,CVIDB,TYPED,
            TYPEDB,bus_out,CD,bus_in,CVOTSTC,
            SELD,END,TGICRESET,func[6],TGICDIAGM_bit4,CLK);

strobe_logic_fault strobe_logic_fault (STFLA,STFLB,STFLC,STFLD,
                        CVOUTA,CVOUTB,CVOUTC,CVOUTD,
                        CVIA,CVIB,CVIC,CVID,
                        TYPEA,TYPEB,TYPEC,TYPED,
                        ACH,BCL,
                        CLK,
                        STFLAB,STFLBB,STFLCB,STFLDB,
                        CVOUTAB,CVOUTBB,CVOUTCB,CVOUTDB,
                        CVIAB,CVIBB,CVICB,CVIDB,
                        TYPEAB,TYPEBB,TYPECB,TYPEDB,
                        NACH,NBCL,
                        CLKB,EVENTMODE_bit7, FAULT_SEL, GATE_SEL);

assign TMUPAB=~TMUPA;
assign TMUPBB=~TMUPB;
assign ACHAB=~ACHA;
assign ACHBB=~ACHB;
assign BCLAB=~BCLA;
assign BCLBB=~BCLB;

response_TMU_fault response_TMU_fault(TMUA,TMUB,
                             ACH,BCL,
                             TMUPA,TMUPB,CVO,CVI,
                             ACHA,BCLA,ACHB,BCLB,
                             TMUAB,TMUBB,
                             NACH,NBCL,
                             TMUPAB,TMUPBB,NCVO,NCVI,
                             ACHAB,BCLAB,ACHBB,BCLBB,
                             PINSTATUS[1],PINSTATUS[0],
                             THBSEL,HSPATHASEL,HSPATHBSEL,
                             TGICDIAGM_bitF,TGICDIAGM_bit3,
                             FAULT_SEL, GATE_SEL);

 QY1ZEROA X096 (ZERO, GATE_NUM[3], FAULT_SEL);// 0 input

 QY1BUFA X001 (N066,N001,RI[5], GATE_NUM[21], FAULT_SEL); // 1 input
 QY1BUFA X015 (N062,N004,RS[1], GATE_NUM[22], FAULT_SEL);

 vbidriv P66 (R[2],RI[2],data[2],OUTEN, GATE_NUM[91], FAULT_SEL);// 2 input
 vbidriv P67 (R[3],RI[3],data[3],OUTEN, GATE_NUM[92], FAULT_SEL);
 vbidriv P68 (R[4],RI[4],data[4],OUTEN, GATE_NUM[93], FAULT_SEL);
 vbidriv P69 (R[5],RI[5],data[5],OUTEN, GATE_NUM[94], FAULT_SEL);
 vbidriv P70 (R[6],RI[6],data[6],OUTEN, GATE_NUM[95], FAULT_SEL);
 vbidriv P71 (R[7],RI[7],data[7],OUTEN, GATE_NUM[96], FAULT_SEL);
 vbidriv P72 (R[8],RI[8],data[8],OUTEN, GATE_NUM[97], FAULT_SEL);
 vbidriv P73 (R[9],RI[9],data[9],OUTEN, GATE_NUM[98], FAULT_SEL);
```

```
QY1AND2A X004 (TGICRESET,N065,N003,FNCCLK, GATE_NUM[99], FAULT_SEL);
QY1AND2A X039 (SELD,N048,END,SELCLK, GATE_NUM[100], FAULT_SEL);
QY1AND2A X038 (SELC,N049,ENC,SELCLK, GATE_NUM[101], FAULT_SEL);
QY1AND2A X037 (SELB,N050,ENB,SELCLK, GATE_NUM[102], FAULT_SEL);
QY1AND2A X036 (SELA,N051,ENA,SELCLK, GATE_NUM[103], FAULT_SEL);
QY1AND2A X035 (SELCLK,N055,N005,WCLK, GATE_NUM[104], FAULT_SEL);
QY1DFA X065 (HSPATHASEL[2],N090,RI[2],HSALDCLK, GATE_NUM[105], FAULT_SEL);
QY1DFA X097 (HSPATHASEL[1],N091,RI[9],HSALDCLK, GATE_NUM[106], FAULT_SEL);
QY1DFA X062 (HSPATHBSEL[2],N092,RI[2],HSBLDCLK, GATE_NUM[107], FAULT_SEL);
QY1DFA X061 (HSPATHBSEL[3],N093,RI[3],HSBLDCLK, GATE_NUM[108], FAULT_SEL);
QY1DFA X060 (HSPATHBSEL[4],N094,RI[4],HSBLDCLK, GATE_NUM[109], FAULT_SEL);
QY1DFA X064 (HSPATHBSEL[0],N095,RI[8],HSBLDCLK, GATE_NUM[110], FAULT_SEL);
QY1DFA X063 (HSPATHBSEL[1],N096,RI[9],HSBLDCLK, GATE_NUM[111], FAULT_SEL);
QY1DFA X012 (func[0],funcb[0],RI[2],FNCCLK, GATE_NUM[112], FAULT_SEL);
QY1DFA X011 (func[1],funcb[1],RI[3],FNCCLK, GATE_NUM[113], FAULT_SEL);
QY1DFA X010 (func[2],funcb[2],RI[4],FNCCLK, GATE_NUM[114], FAULT_SEL);
QY1DFA X009 (func[3],funcb[3],RI[5],FNCCLK, GATE_NUM[115], FAULT_SEL);
QY1DFA X008 (func[4],funcb[4],RI[6],FNCCLK, GATE_NUM[116], FAULT_SEL);
QY1DFA X007 (func[5],funcb[5],RI[7],FNCCLK, GATE_NUM[117], FAULT_SEL);
QY1DFA X006 (func[6],N060,RI[8],FNCCLK, GATE_NUM[118], FAULT_SEL);
QY1DFA X005 (func[7],N061,RI[9],FNCCLK, GATE_NUM[119], FAULT_SEL);
QY1DIAGA S15 (N126,ZERO,TGICDIAGM_bit5, GATE_NUM[120], FAULT_SEL);
QY1DIAGA S20 (N141,ZERO,HSPATHBSEL[2], GATE_NUM[121], FAULT_SEL);
QY1DIAGA S19 (N125,ZERO,TGICDIAGM_bit2, GATE_NUM[122], FAULT_SEL);
QY1DIAGA S17 (N124,ZERO,HSPATHBSEL[3], GATE_NUM[123], FAULT_SEL);
QY1OR2A X031 (N044,S0,HSPABSET,TGCLRSET, GATE_NUM[124], FAULT_SEL);
QY1OR2A X030 (N045,S1,HSPABSET,EVCLRSET, GATE_NUM[125], FAULT_SEL);
QY1OR2A X098 (N057,NOT06,HSLDENB,func[0], GATE_NUM[126], FAULT_SEL);
QY1STA S18 (N143,PECONTROL[2],HSPATHASEL[2], GATE_NUM[127], FAULT_SEL);
QY1STA S16 (N140,EVENTMODE_bit3,ZERO, GATE_NUM[128], FAULT_SEL);
QY1STA S14 (N127,ZERO,TGICDIAGM_bit6, GATE_NUM[129], FAULT_SEL);
QY1STA S13 (N128,ZERO,EVENTMODE_bit7, GATE_NUM[130], FAULT_SEL);
QY1STA S12 (N139,ZERO,PINSTATUS[0], GATE_NUM[131], FAULT_SEL);
QY1STA S11 (N138,TGICDIAGM_bit0,ZERO, GATE_NUM[132], FAULT_SEL);
QY1STA S06 (N129,ZERO,func[0], GATE_NUM[133], FAULT_SEL);
QY1STA S09 (N130,ZERO,PINSTATUS[1], GATE_NUM[134], FAULT_SEL);
QY1STA S08 (N131,TGICDIAGM_bit1,ZERO, GATE_NUM[135], FAULT_SEL);
QY1STA S07 (N132,PECONTROL[1],HSPATHASEL[1], GATE_NUM[136], FAULT_SEL);
QY1STA S10 (N142,PECONTROL[0],ZERO, GATE_NUM[137], FAULT_SEL);
QY1STA S04 (N134,func[4],funcb[4], GATE_NUM[138], FAULT_SEL);
QY1STA S03 (N135,HSLDEN,Z, GATE_NUM[139], FAULT_SEL);
QY1STA S02 (N136,func[1],funcb[1], GATE_NUM[140], FAULT_SEL);
QY1STA S01 (N137,func[2],funcb[2], GATE_NUM[141], FAULT_SEL);
QY1STA S05 (N133,TGICDIAGM_bit5,TGICDIAGM_bit6, GATE_NUM[142], FAULT_SEL);
QY1STA S21 (N073,NOT06,ZERO, GATE_NUM[143], FAULT_SEL);

QY1AND3A X071 (N018,N097,RI[8],HSPABSET,WCLK, GATE_NUM[198], FAULT_SEL);// 3 input
QY1AND3A X070 (N017,N098,RI[9],HSPABSET,WCLK, GATE_NUM[199], FAULT_SEL);
QY1AND3A X069 (N016,N099,RI[2],HSPABSET,WCLK, GATE_NUM[200], FAULT_SEL);
QY1AND3A X057 (N015,N077,RI[3],EVCLRSET,WCLK, GATE_NUM[201], FAULT_SEL);
QY1AND3A X056 (N014,N078,RI[7],EVCLRSET,WCLK, GATE_NUM[202], FAULT_SEL);
QY1AND3A X046 (N012,N079,RI[9],TGCLRSET,WCLK, GATE_NUM[203], FAULT_SEL);
QY1AND3A X047 (N013,N080,RI[8],TGCLRSET,WCLK, GATE_NUM[204], FAULT_SEL);
QY1AND3A X043 (N009,N081,RI[4],TGCLRSET,WCLK, GATE_NUM[205], FAULT_SEL);
QY1AND3A X044 (N010,N082,RI[3],TGCLRSET,WCLK, GATE_NUM[206], FAULT_SEL);
QY1AND3A X045 (N011,N083,RI[2],TGCLRSET,WCLK, GATE_NUM[207], FAULT_SEL);
QY1AND3A X042 (N008,N084,RI[5],TGCLRSET,WCLK, GATE_NUM[208], FAULT_SEL);
QY1AND3A X041 (N007,N085,RI[6],TGCLRSET,WCLK, GATE_NUM[209], FAULT_SEL);
QY1AND3A X040 (N006,N086,RI[7],TGCLRSET,WCLK, GATE_NUM[210], FAULT_SEL);
QY1AND3A X014 (WCLK,N063,RTXC,RS[0],N004, GATE_NUM[211], FAULT_SEL);
QY1AND3A X013 (FNCCLK,N064,RTXC,RS[0],RS[1], GATE_NUM[212], FAULT_SEL);
QY1AND3A X033 (HSALDCLK,N046,HSLDEN,funcb[0],WCLK, GATE_NUM[213], FAULT_SEL);
QY1AND3A X032 (HSBLDCLK,N047,HSLDEN,func[0],WCLK, GATE_NUM[214], FAULT_SEL);
QY1DFRA X068 (PECONTROL[0],N087,func[0],N018,TGICRESET, GATE_NUM[215], FAULT_SEL);
```

```
QY1DFRA X067 (PECONTROL[1],N088,func[0],N017,TGICRESET, GATE_NUM[216], FAULT_SEL);
QY1DFRA X066 (PECONTROL[2],N089,func[0],N016,TGICRESET, GATE_NUM[217], FAULT_SEL);
QY1DFRA X059 (EVENTMODE_bit3,N067,func[0],N015,TGICRESET, GATE_NUM[218],
                        FAULT_SEL);
QY1DFRA X058 (EVENTMODE_bit7,N068,func[0],N014,TGICRESET, GATE_NUM[219],
                        FAULT_SEL);
QY1DFRA X055 (TGICDIAGM_bit0,N069,func[0],N013,TGICRESET, GATE_NUM[220],
                        FAULT_SEL);
QY1DFRA X054 (TGICDIAGM_bit1,N070,func[0],N012,TGICRESET, GATE_NUM[221],
                        FAULT_SEL);
QY1DFRA X053 (TGICDIAGM_bit2,N071,func[0],N011,TGICRESET, GATE_NUM[222],
                        FAULT_SEL);
QY1DFRA X052 (TGICDIAGM_bit3,N072,func[0],N010,TGICRESET, GATE_NUM[223],
                        FAULT_SEL);
QY1DFRA X051 (TGICDIAGM_bit4,TGICDIAGM_bit4B,func[0],N009,TGICRESET,
                        GATE_NUM[224], FAULT_SEL);
QY1DFRA X050 (TGICDIAGM_bit5,N074,func[0],N008,TGICRESET, GATE_NUM[225],
                        FAULT_SEL);
QY1DFRA X049 (TGICDIAGM_bit6,N075,func[0],N007,TGICRESET, GATE_NUM[226],
                        FAULT_SEL);
QY1DFRA X048 (TGICDIAGM_bitF,N076,func[0],N006,TGICRESET, GATE_NUM[227],
                        FAULT_SEL);
QY1MX2A X088 (data[9],N101,N027,bus_out[9],LDLRD, GATE_NUM[228], FAULT_SEL);
QY1MX2A X072 (N019,N110,N132,HSPATHBSEL[1],N129, GATE_NUM[229], FAULT_SEL);
QY1MX2A X089 (data[8],N102,N028,bus_out[8],LDLRD, GATE_NUM[230], FAULT_SEL);
QY1MX2A X073 (N020,N112,N142,HSPATHBSEL[0],N129, GATE_NUM[231], FAULT_SEL);
QY1MX2A X090 (data[7],N103,N023,bus_out[7],LDLRD, GATE_NUM[232], FAULT_SEL);
QY1MX2A X074 (N021,N114,ZERO,ZERO,N129, GATE_NUM[233], FAULT_SEL);
QY1MX2A X091 (data[6],N104,N030,bus_out[6],LDLRD, GATE_NUM[234], FAULT_SEL);
QY1MX2A X075 (N022,N116,ZERO,ZERO,N129, GATE_NUM[235], FAULT_SEL);
QY1MX2A X092 (data[5],N105,N031,bus_out[5],LDLRD, GATE_NUM[236], FAULT_SEL);
QY1MX2A X076 (N029,N118,ZERO,ZERO,N129, GATE_NUM[237], FAULT_SEL);
QY1MX2A X093 (data[4],N106,N032,bus_out[4],LDLRD, GATE_NUM[238], FAULT_SEL);
QY1MX2A X077 (N024,N120,ZERO,HSPATHBSEL[4],N129, GATE_NUM[239], FAULT_SEL);
QY1MX2A X094 (data[3],N107,N033,bus_out[3],LDLRD, GATE_NUM[240], FAULT_SEL);
QY1MX2A X078 (N025,N108,ZERO,HSPATHBSEL[3],N129, GATE_NUM[241], FAULT_SEL);
QY1MX2A X095 (data[2],N109,N034,bus_out[2],LDLRD, GATE_NUM[242], FAULT_SEL);
QY1MX2A X079 (N026,N122,N143,HSPATHBSEL[2],N129, GATE_NUM[243], FAULT_SEL);
QY1OR3A X023 (N043,X,func[3],func[2],LDLRDB, GATE_NUM[244], FAULT_SEL);
QY1OR3A X024 (LDLRDB,LDLRD,func[7],funcb[5],N134, GATE_NUM[245], FAULT_SEL);
QY1OR3A X026 (N056,ENA,func[1],LDLRDB,func[0], GATE_NUM[246], FAULT_SEL);
QY1OR3A X027 (N052,ENB,func[1],LDLRDB,funcb[0], GATE_NUM[247], FAULT_SEL);
QY1OR3A X028 (N053,ENC,funcb[1],LDLRDB,func[0], GATE_NUM[248], FAULT_SEL);
QY1OR3A X029 (N054,END,funcb[1],LDLRDB,funcb[0], GATE_NUM[249], FAULT_SEL);

QY1OR4A X003 (N059,N003,RI[9],RI[8],N001,N002, GATE_NUM[341], FAULT_SEL);// 4 input
QY1OR4A X025 (N038,N005,LDLRDB,func[6],func[3],func[2], GATE_NUM[342], FAULT_SEL);
QY1OR4A X019 (HSLDENB,HSLDEN,func[3],funcb[2],funcb[1],MSB0, GATE_NUM[343],
                        FAULT_SEL);
QY1OR4A X018 (N039,TGCLRSET,funcb[3],func[2],funcb[1],MSB0, GATE_NUM[344], FAULT_SEL);
QY1OR4A X017 (N040,EVCLRSET,func[3],funcb[2],func[1],MSB0, GATE_NUM[345], FAULT_SEL);
QY1OR4A X016 (N041,HSPABSET,N137,N136,func[3],MSB0, GATE_NUM[346], FAULT_SEL);
QY1OR4A X020 (MSB0,N042,func[7],func[6],func[5],func[4], GATE_NUM[347], FAULT_SEL);

QY1OR5A X022 (N035,Y,N135,TGCLRSET,EVCLRSET,HSPABSET,X, GATE_NUM[432],
                        FAULT_SEL);// 5 input
QY1OR5A X021 (N036,Z,MSB0,func[3],func[2],funcb[1],func[0], GATE_NUM[433], FAULT_SEL);
QY1OR5A X002 (N002,N058,RI[7],RI[6],RI[4],RI[3],RI[2], GATE_NUM[434], FAULT_SEL);
QY1OR5A X034 (N037,OUTEN,N073,N133,RS[0],RS[1],Y, GATE_NUM[435], FAULT_SEL);

QY1MX4A X080 (N027,N111,N019,ZERO,N131,N130,S1,S0, GATE_NUM[454],
                        FAULT_SEL);    // 6 input
QY1MX4A X081 (N028,N113,N020,ZERO,N138,N139,S1,S0, GATE_NUM[455], FAULT_SEL);
QY1MX4A X082 (N023,N115,N021,N128,TGICDIAGM_bitF,ZERO,S1,S0,
                              GATE_NUM[456], FAULT_SEL);
 QY1MX4A X083 (N030,N117,N022,ZERO,N127,ZERO,S1,S0, GATE_NUM[457], FAULT_SEL);
 QY1MX4A X084 (N031,N119,N029,ZERO,N126,ZERO,S1,S0, GATE_NUM[458], FAULT_SEL);
 QY1MX4A X085 (N032,N121,N024,ZERO,TGICDIAGM_bit4,ZERO,S1,S0,
                              GATE_NUM[459], FAULT_SEL);
 QY1MX4A X086 (N033,N100,N025,N140,TGICDIAGM_bit3,N124,S1,S0, GATE_NUM[460],
                              FAULT_SEL);
 QY1MX4A X087 (N034,N123,N026,ZERO,N125,N141,S1,S0, GATE_NUM[461], FAULT_SEL);

endmodule

module strobe_logic_fault (STFLA,STFLB,STFLC,STFLD,
                              CVOUTA,CVOUTB,CVOUTC,CVOUTD,
                              CVINA,CVINB,CVINC,CVIND,
                              TYPEA,TYPEB,TYPEC,TYPED,
                              ACH,BCL,CLK,
                              STFLAB,STFLBB,STFLCB,STFLDB,
                              CVOUTAB,CVOUTBB,CVOUTCB,CVOUTDB,
                              CVINAB,CVINBB,CVINCB,CVINDB,
                              TYPEAB,TYPEBB,TYPECB,TYPEDB,
                              ACHB,BCLB,CLKB,ESM, FAULT_SEL, GATE_SEL);

 output STFLA,STFLB,STFLC,STFLD;
 output STFLAB,STFLBB,STFLCB,STFLDB;
 input CVOUTA,CVOUTB,CVOUTC,CVOUTD;
 input CVOUTAB,CVOUTBB,CVOUTCB,CVOUTDB;
 input CVINA,CVINB,CVINC,CVIND;
 input CVINAB,CVINBB,CVINCB,CVINDB;
 input [1:0] TYPEA,TYPEB,TYPEC,TYPED;
 input [1:0] TYPEAB,TYPEBB,TYPECB,TYPEDB;
 input ACH,BCL;
 input ACHB,BCLB,CLKB;
 input CLK,ESM; // ESM is bit 7 of EVENTMODE
 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

wire [1023:0] GATE_NUM;
assign GATE_NUM = 1 << GATE_SEL;

 QY1ONEDA X122 (ONE,ONEB,GATE_NUM[4],FAULT_SEL);// 0 input

 QY1S2DA X123 (ESMI,ESMBI,ESM,GATE_NUM[23],FAULT_SEL);// 1 input

 QY1CP1DA X81 (N123,N124,STB0,STB0B,STB0SET,STB0SETB,GATE_NUM[348],
                              FAULT_SEL);// 4 input
 QY1CP1DA X82 (N125,N126,STB0B,STB0,RSTSTBS,RSTSTBSB,GATE_NUM[348],FAULT_SEL);
 QY1CP1DA X83 (N129,N130,STB1,STB1B,STB1SET,STB1SETB,GATE_NUM[349],FAULT_SEL);
 QY1CP1DA X84 (N131,N132,STB1B,STB1,RSTSTBS,RSTSTBSB,GATE_NUM[350],FAULT_SEL);
 QY1CP1DA X85 (N135,N136,STBZ,STBZB,STBZSET,STBZSETB,GATE_NUM[351],FAULT_SEL);
 QY1CP1DA X86 (N137,N138,STBZB,STBZ,RSTSTBS,RSTSTBSB,GATE_NUM[352],FAULT_SEL);
 QY1CP1DA X102 (N145,N146,N140,N139,CVINAB,CVINA,GATE_NUM[353],FAULT_SEL);
 QY1CP1DA X107 (N149,N150,N148,N147,CVINBB,CVINB,GATE_NUM[354],FAULT_SEL);
 QY1CP1DA X112 (N159,N160,N158,N157,CVINCB,CVINC,GATE_NUM[355],FAULT_SEL);
 QY1CP1DA X117 (N167,N168,N166,N165,CVINDB,CVIND,GATE_NUM[356],FAULT_SEL);
 QY1DFDA X34 (EFAILB,EFAILBB,ELOGB,ELOGBB,EDGSTBB,
                              EDGSTBBB,GATE_NUM[357],FAULT_SEL);
 QY1DFDA X33 (EFAILA,EFAILAB,ELOGA,ELOGAB,EDGSTBA,
                              EDGSTBAB,GATE_NUM[358],FAULT_SEL);
 QY1DFDA X35 (EFAILC,EFAILCB,ELOGC,ELOGCB,EDGSTBC,
                              EDGSTBCB,GATE_NUM[359],FAULT_SEL);
 QY1DFDA X36 (EFAILD,EFAILDB,ELOGD,ELOGDB,EDGSTDB,
                              EDGSTBDB,GATE_NUM[360],FAULT_SEL);
 QY1DFDA X93 (WFAILA,WFAILAB,WFAIL,NWFAIL,RSTSTBA,
                              RSTSTBAB,GATE_NUM[361],FAULT_SEL);
```

```
QY1DFDA X94 (WFAILB,WFAILBB,WFAIL,NWFAIL,RSTSTBB,
                    RSTSTBBB,GATE_NUM[362],FAULT_SEL);
QY1DFDA X95 (WFAILC,WFAILCB,WFAIL,NWFAIL,RSTSTBC,
                    RSTSTBCB,GATE_NUM[363],FAULT_SEL);
QY1DFDA X96 (WFAILD,WFAILDB,WFAIL,NWFAIL,RSTSTBD,
                    RSTSTBDB,GATE_NUM[364],FAULT_SEL);
QY1DFDA X103 (N140,N139,N145,N146,CLK,CLKB,GATE_NUM[365],FAULT_SEL);
QY1DFDA X105 (N142,N141,N143,N144,CLK,CLKB,GATE_NUM[366],FAULT_SEL);
QY1DFDA X106 (STFLA,STFLAB,N142,N141,CLK,CLKB,GATE_NUM[367],FAULT_SEL);
QY1DFDA X111 (STFLB,STFLBB,N154,N153,CLK,CLKB,GATE_NUM[368],FAULT_SEL);
QY1DFDA X110 (N154,N153,N155,N156,CLK,CLKB,GATE_NUM[369],FAULT_SEL);
QY1DFDA X108 (N148,N147,N149,N150,CLK,CLKB,GATE_NUM[370],FAULT_SEL);
QY1DFDA X116 (STFLC,STFLCB,N162,N161,CLK,CLKB,GATE_NUM[371],FAULT_SEL);
QY1DFDA X115 (N162,N161,N163,N164,CLK,CLKB,GATE_NUM[372],FAULT_SEL);
QY1DFDA X113 (N158,N157,N159,N160,CLK,CLKB,GATE_NUM[373],FAULT_SEL);
QY1DFDA X121 (STFLD,STFLDB,N170,N169,CLK,CLKB,GATE_NUM[374],FAULT_SEL);
QY1DFDA X120 (N170,N169,N151,N152,CLK,CLKB,GATE_NUM[375],FAULT_SEL);
QY1DFDA X118 (N166,N165,N167,N168,CLK,CLKB,GATE_NUM[376],FAULT_SEL);
QY1OR2DA X21 (N033,N034,N002,N001,N004,N003,GATE_NUM[377],FAULT_SEL);
QY1OR2DA X22 (N035,N036,N006,N005,N008,N007,GATE_NUM[378],FAULT_SEL);
QY1OR2DA X29 (ELOGA,ELOGAB,N033,N034,N035,N036,GATE_NUM[379],FAULT_SEL);
QY1OR2DA X30 (ELOGB,ELOGBB,N037,N038,N039,N040,GATE_NUM[380],FAULT_SEL);
QY1OR2DA X24 (N039,N040,N014,N013,N016,N015,GATE_NUM[381],FAULT_SEL);
QY1OR2DA X23 (N037,N038,N010,N009,N012,N011,GATE_NUM[382],FAULT_SEL);
QY1OR2DA X27 (N045,N046,N026,N025,N028,N027,GATE_NUM[383],FAULT_SEL);
QY1OR2DA X28 (N047,N048,N030,N029,N032,N031,GATE_NUM[384],FAULT_SEL);
QY1OR2DA X32 (ELOGD,ELOGDB,N045,N046,N047,N048,GATE_NUM[385],FAULT_SEL);
QY1OR2DA X31 (ELOGC,ELOGCB,N041,N042,N043,N044,GATE_NUM[386],FAULT_SEL);
QY1OR2DA X26 (N043,N044,N022,N021,N024,N023,GATE_NUM[387],FAULT_SEL);
QY1OR2DA X25 (N041,N042,N018,N017,N020,N019,GATE_NUM[388],FAULT_SEL);
QY1OR2DA X10 (EDGSTBBB,EDGSTBB,CVOUTBB,CVOUTB,
                    ESMBI,ESMI,GATE_NUM[389],FAULT_SEL);
QY1OR2DA X05 (EDGSTBAB,EDGSTBA,CVOUTAB,CVOUTA,
                    ESMBI,ESMI,GATE_NUM[390],FAULT_SEL);
QY1OR2DA X15 (EDGSTBCB,EDGSTBC,CVOUTCB,CVOUTC,
                    ESMBI,ESMI,GATE_NUM[391],FAULT_SEL);
QY1OR2DA X20 (EDGSTBDB,EDGSTDB,CVOUTDB,CVOUTD,
                    ESMBI,ESMI,GATE_NUM[392],FAULT_SEL);
QY1OR2DA X53 (N081,N082,CVOUTAB,CVOUTA,N049,N050,GATE_NUM[393],FAULT_SEL);
QY1OR2DA X54 (N083,N084,CVOUTBB,CVOUTB,N051,N052,GATE_NUM[394],FAULT_SEL);
QY1OR2DA X55 (N085,N086,CVOUTCB,CVOUTC,N053,N054,GATE_NUM[395],FAULT_SEL);
QY1OR2DA X56 (N087,N088,CVOUTDB,CVOUTD,N055,N056,GATE_NUM[396],FAULT_SEL);
QY1OR2DA X69 (N105,N106,N082,N081,N084,N083,GATE_NUM[397],FAULT_SEL);
QY1OR2DA X70 (N107,N108,N086,N085,N088,N087,GATE_NUM[398],FAULT_SEL);
QY1OR2DA X77 (STB0SET,STB0SETB,N105,N106,N107,N108,GATE_NUM[399],FAULT_SEL);
QY1OR2DA X57 (N089,N090,CVOUTAB,CVOUTA,N057,N058,GATE_NUM[400],FAULT_SEL);
QY1OR2DA X58 (N091,N092,CVOUTBB,CVOUTB,N059,N060,GATE_NUM[401],FAULT_SEL);
QY1OR2DA X59 (N093,N094,CVOUTCB,CVOUTC,N061,N062,GATE_NUM[402],FAULT_SEL);
QY1OR2DA X60 (N095,N096,CVOUTDB,CVOUTD,N063,N064,GATE_NUM[403],FAULT_SEL);
QY1OR2DA X71 (N109,N110,N090,N089,N092,N091,GATE_NUM[404],FAULT_SEL);
QY1OR2DA X72 (N111,N112,N094,N093,N096,N095,GATE_NUM[405],FAULT_SEL);
QY1OR2DA X78 (STB1SET,STB1SETB,N109,N110,N111,N112,GATE_NUM[406],FAULT_SEL);
QY1OR2DA X79 (STBZSET,STBZSETB,N113,N114,N115,N116,GATE_NUM[407],FAULT_SEL);
QY1OR2DA X74 (N115,N116,N102,N101,N104,N103,GATE_NUM[408],FAULT_SEL);
QY1OR2DA X73 (N113,N114,N098,N097,N100,N099,GATE_NUM[409],FAULT_SEL);
QY1OR2DA X64 (N103,N104,CVOUTDB,CVOUTD,N071,N072,GATE_NUM[410],FAULT_SEL);
QY1OR2DA X63 (N101,N102,CVOUTCB,CVOUTC,N069,N070,GATE_NUM[411],FAULT_SEL);
QY1OR2DA X62 (N099,N100,CVOUTBB,CVOUTB,N067,N068,GATE_NUM[412],FAULT_SEL);
QY1OR2DA X61 (N097,N098,CVOUTAB,CVOUTA,N065,N066,GATE_NUM[413],FAULT_SEL);
QY1OR2DA X65 (RSTSTBAB,RSTSTBA,CVOUTAB,CVOUTA,N073,N074,
                    GATE_NUM[414],FAULT_SEL);
QY1OR2DA X66 (RSTSTBBB,RSTSTBB,CVOUTBB,CVOUTB,N075,N076,
                    GATE_NUM[415],FAULT_SEL);
QY1OR2DA X67 (RSTSTBCB,RSTSTBC,CVOUTCB,CVOUTC,N077,N078,
                    GATE_NUM[416],FAULT_SEL);
QY1OR2DA X68 (RSTSTBDB,RSTSTBD,CVOUTDB,CVOUTD,N079,N080,
                    GATE_NUM[417],FAULT_SEL);
QY1OR2DA X75 (N117,N118,RSTSTBA,RSTSTBAB,RSTSTBB,RSTSTBBB,
                    GATE_NUM[418],FAULT_SEL);
QY1OR2DA X76 (N119,N120,RSTSTBC,RSTSTBCB,RSTSTBD,RSTSTBDB,
                    GATE_NUM[419],FAULT_SEL);
QY1OR2DA X80 (RSTSTBS,RSTSTBSB,N117,N118,N119,N120,GATE_NUM[420],FAULT_SEL);
QY1OR2DA X89 (N127,N128,ACH,ACHB,ACH,ACHB,GATE_NUM[421],FAULT_SEL);
QY1OR2DA X87 (N121,N122,BCL,BCLB,BCL,BCLB,GATE_NUM[422],FAULT_SEL);
QY1OR2DA X91 (N134,N133,BCL,BCLB,ACH,ACHB,GATE_NUM[423],FAULT_SEL);
QY1RSDA X88 (STB0,STB0B,N123,N124,N125,N126,GATE_NUM[424],FAULT_SEL);
QY1RSDA X90 (STB1,STB1B,N129,N130,N131,N132,GATE_NUM[425],FAULT_SEL);
QY1RSDA X92 (STBZ,STBZB,N135,N136,N137,N138,GATE_NUM[426],FAULT_SEL);
QY1RSDA X101 (WFAIL,NWFAIL,STWFAIL,STWFAILB,
                    RSTSTBS,RSTSTBSB,GATE_NUM[427],FAULT_SEL);

QY1OR3DA X01 (N001,N002,ACH,ACHB,BCL,BCLB,TYPEAB[1],
                    TYPEA[1],GATE_NUM[462],FAULT_SEL);// 6 input
QY1OR3DA X02 (N003,N004,ACHB,ACH,TYPEA[1],TYPEAB[1],
                    TYPEAB[0],TYPEA[0],GATE_NUM[463],FAULT_SEL);
QY1OR3DA X03 (N005,N006,ACH,ACHB,BCLB,BCL,TYPEAB[0],
                    TYPEA[0],GATE_NUM[464],FAULT_SEL);
QY1OR3DA X04 (N007,N008,TYPEAB[1],TYPEA[1],BCL,BCLB,TYPEA[0],
                    TYPEAB[0],GATE_NUM[465],FAULT_SEL);
QY1OR3DA X09 (N015,N016,TYPEBB[1],TYPEB[1],BCL,BCLB,TYPEB[0],
                    TYPEBB[0],GATE_NUM[466],FAULT_SEL);
QY1OR3DA X08 (N013,N014,ACH,ACHB,BCLB,BCL,TYPEBB[0],TYPEB[0],
                    GATE_NUM[467],FAULT_SEL);
QY1OR3DA X07 (N011,N012,ACHB,ACH,TYPEB[1],TYPEBB[1],TYPEBB[0],
                    TYPEB[0],GATE_NUM[468],FAULT_SEL);
QY1OR3DA X06 (N009,N010,ACH,ACHB,BCL,BCLB,TYPEBB[1],TYPEB[1],
                    GATE_NUM[469],FAULT_SEL);
QY1OR3DA X16 (N025,N026,ACH,ACHB,BCL,BCLB,TYPEDB[1],TYPED[1],
                    GATE_NUM[470],FAULT_SEL);
QY1OR3DA X17 (N027,N028,ACHB,ACH,TYPED[1],TYPEDB[1],TYPEDB[0],TYPED[0],
                    GATE_NUM[471],FAULT_SEL);
QY1OR3DA X18 (N029,N030,ACH,ACHB,BCLB,BCL,TYPEDB[0],TYPED[0],
                    GATE_NUM[472],FAULT_SEL);
QY1OR3DA X19 (N031,N032,TYPEDB[1],TYPED[1],BCL,BCLB,TYPED[0],TYPEDB[0],
                    GATE_NUM[473],FAULT_SEL);
QY1OR3DA X14 (N023,N024,TYPECB[1],TYPEC[1],BCL,BCLB,TYPEC[0],TYPECB[0],
                    GATE_NUM[474],FAULT_SEL);
QY1OR3DA X13 (N021,N022,ACH,ACHB,BCLB,BCL,TYPECB[0],TYPEC[0],
                    GATE_NUM[475],FAULT_SEL);
QY1OR3DA X12 (N019,N020,ACHB,ACH,TYPEC[1],TYPECB[1],TYPECB[0],TYPEC[0],
                    GATE_NUM[476],FAULT_SEL);
QY1OR3DA X11 (N017,N018,ACH,ACHB,BCL,BCLB,TYPECB[1],TYPEC[1],
                    GATE_NUM[477],FAULT_SEL);
QY1OR3DA X37 (N049,N050,TYPEAB[1],TYPEA[1],TYPEA[0],TYPEAB[0],ESMI,ESMBI,
                    GATE_NUM[478],FAULT_SEL);
QY1OR3DA X38 (N051,N052,TYPEBB[1],TYPEB[1],TYPEB[0],TYPEBB[0],ESMI,ESMBI,
                    GATE_NUM[479],FAULT_SEL);
QY1OR3DA X39 (N053,N054,TYPECB[1],TYPEC[1],TYPEC[0],TYPECB[0],ESMI,ESMBI,
                    GATE_NUM[480],FAULT_SEL);
QY1OR3DA X40 (N055,N056,TYPEDB[1],TYPED[1],TYPED[0],TYPEDB[0],ESMI,ESMBI,
                    GATE_NUM[481],FAULT_SEL);
QY1OR3DA X41 (N057,N058,TYPEAB[1],TYPEA[1],TYPEAB[0],TYPEA[0],ESMI,ESMBI,
                    GATE_NUM[482],FAULT_SEL);
QY1OR3DA X42 (N059,N060,TYPEBB[1],TYPEB[1],TYPEBB[0],TYPEB[0],ESMI,ESMBI,
                    GATE_NUM[483],FAULT_SEL);
QY1OR3DA X43 (N061,N062,TYPECB[1],TYPEC[1],TYPECB[0],TYPEC[0],ESMI,ESMBI,
                    GATE_NUM[484],FAULT_SEL);
QY1OR3DA X44 (N063,N064,TYPEDB[1],TYPED[1],TYPEDB[0],TYPED[0],ESMI,ESMBI,
                              GATE_NUM[485],FAULT_SEL);
 QY1OR3DA X48 (N071,N072,TYPED[1],TYPEDB[1],TYPEDB[0],TYPED[0],ESMI,ESMBI,
                              GATE_NUM[486],FAULT_SEL);
 QY1OR3DA X47 (N069,N070,TYPEC[1],TYPECB[1],TYPECB[0],TYPEC[0],ESMI,ESMBI,
                              GATE_NUM[487],FAULT_SEL);
 QY1OR3DA X46 (N067,N068,TYPEB[1],TYPEBB[1],TYPEBB[0],TYPEB[0],ESMI,ESMBI,
                              GATE_NUM[488],FAULT_SEL);
 QY1OR3DA X45 (N065,N066,TYPEA[1],TYPEAB[1],TYPEAB[0],TYPEA[0],ESMI,ESMBI,
                              GATE_NUM[489],FAULT_SEL);
 QY1OR3DA X49 (N073,N074,TYPEA[1],TYPEAB[1],TYPEA[0],TYPEAB[0],ESMI,ESMBI,
                              GATE_NUM[490],FAULT_SEL);
 QY1OR3DA X50 (N075,N076,TYPEB[1],TYPEBB[1],TYPEB[0],TYPEBB[0],ESMI,ESMBI,
                              GATE_NUM[491],FAULT_SEL);
 QY1OR3DA X51 (N077,N078,TYPEC[1],TYPECB[1],TYPEC[0],TYPECB[0],ESMI,ESMBI,
                              GATE_NUM[492],FAULT_SEL);
 QY1OR3DA X52 (N079,N080,TYPED[1],TYPEDB[1],TYPED[0],TYPEDB[0],ESMI,ESMBI,
                              GATE_NUM[493],FAULT_SEL);
 QY1OR3DA X98 (FSTB1B,FSTB1,N127,N128,STB1B,STB1,ESMI,ESMBI,
                              GATE_NUM[494],FAULT_SEL);
 QY1OR3DA X97 (FSTB0B,FSTB0,N121,N122,STB0B,STB0,ESMI,ESMBI,
                              GATE_NUM[495],FAULT_SEL);
 QY1OR3DA X99 (FSTBZB,FSTBZ,N133,N134,STBZB,STBZ,ESMI,ESMBI,
                              GATE_NUM[496],FAULT_SEL);
 QY1OR3DA X100 (STWFAIL,STWFAILB,FSTB0,FSTB0B,FSTB1,FSTB1B,FSTBZ,FSTBZB,
                              GATE_NUM[497],FAULT_SEL);

 QY1MX4DA X104 (N143,N144,N142,N141,N142,N141,WFAILA,WFAILAB,EFAILA,EFAILAB,
                              N140,N139,ESMI,ESMBI,GATE_NUM[506],FAULT_SEL);// 12 input
 QY1MX4DA X109 (N155,N156,N154,N153,N154,N153,WFAILB,WFAILBB,EFAILB,EFAILBB,
                              N148,N147,ESMI,ESMBI,GATE_NUM[507],FAULT_SEL);
 QY1MX4DA X114 (N163,N164,N162,N161,N162,N161,WFAILC,WFAILCB,EFAILC,EFAILCB,
                              N158,N157,ESMI,ESMBI,GATE_NUM[508],FAULT_SEL);
 QY1MX4DA X119 (N151,N152,N170,N169,N170,N169,WFAILD,WFAILDB,EFAILD,EFAILDB,
                              N166,N165,ESMI,ESMBI,GATE_NUM[509],FAULT_SEL);

endmodule

module response_TMU_fault(TMUA,TMUB,
                              ACH,BCL,
                              TMUPA,TMUPB,CVO,CVI,
                              ACHA,BCLA,ACHB,BCLB,
                              TMUAB,TMUBB,
                              NACH,NBCL,
                              TMUPAB,TMUPBB,CVOB,CVIB,
                              ACHAB,BCLAB,ACHBB,BCLBB,
                              ACHS,BCLS,
                              THBSEL,HSPATHASEL,HSPATHBSEL,LPBK,TSTPULSE,
                              FAULT_SEL, GATE_SEL);
 output TMUA,TMUB;
 output TMUAB,TMUBB;
 output ACH,BCL; // to STROBE LOGIC
 output NACH,NBCL; // to STROBE LOGIC
 output ACHS,BCLS; // to register section (single ended)
 input TMUPA,TMUPB; // from DRIVE CONTROL's TMU Mux
 input TMUPAB,TMUPBB; // from DRIVE CONTROL's TMU Mux
 input [3:0] CVO,CVI; // from LDLs
 input [3:0] CVOB,CVIB; // from LDLs
 input ACHA,BCLA,ACHB,BCLB;
 input ACHAB,BCLAB,ACHBB,BCLBB;
 input THBSEL;
 input [2:1] HSPATHASEL;
 input [4:0] HSPATHBSEL;
 input LPBK,TSTPULSE;
 input [4:0] FAULT_SEL;
 input [9:0] GATE_SEL;

wire [1023:0] GATE_NUM;
assign GATE_NUM = 1 << GATE_SEL;

 QY1ONEDA X15 (N034,N035,GATE_NUM[5],FAULT_SEL);// 0 input

 QY1S2DA X01 (N001,N002,LPBK,GATE_NUM[24],FAULT_SEL);// 1 input
 QY1S2DA X02 (N003,N004,THBSEL,GATE_NUM[25],FAULT_SEL);
 QY1S2DA X03 (N009,N010,HSPATHBSEL[3],GATE_NUM[26],FAULT_SEL);
 QY1S2DA X04 (N013,N014,HSPATHBSEL[1],GATE_NUM[27],FAULT_SEL);
 QY1S2DA X05 (N011,N012,HSPATHBSEL[2],GATE_NUM[28],FAULT_SEL);
 QY1S2DA X06 (N015,N016,HSPATHBSEL[0],GATE_NUM[29],FAULT_SEL);
 QY1S2DA X07 (N017,N018,HSPATHBSEL[4],GATE_NUM[30],FAULT_SEL);
 QY1S2DA X08 (N019,N020,TSTPULSE,GATE_NUM[31],FAULT_SEL);
 QY1S2DA X09 (N005,N006,HSPATHASEL[2],GATE_NUM[32],FAULT_SEL);
 QY1S2DA X10 (N007,N008,HSPATHASEL[1],GATE_NUM[33],FAULT_SEL);

 QY1D2SDA X23 (ACHS,ACH,NACH,GATE_NUM[144],FAULT_SEL);// 2 input
 QY1D2SDA X24 (BCLS,BCL,NBCL,GATE_NUM[145],FAULT_SEL);

 QY1MX2DA X17 (N032,N029,N036,N027,N028,N030,N015,N016,
                        GATE_NUM[498],FAULT_SEL);// 6 input
 QY1DFRDA X18 (N031,N033,N033,N031,N032,N029,N020,N019,GATE_NUM[499],FAULT_SEL);
 QY1MX2DA X19 (N023,N024,N032,N029,N031,N033,N019,N020,GATE_NUM[500],FAULT_SEL);
 QY1MX2DA X22 (TMUB,TMUBB,_S141,_S142,N021,N022,N017,N018,
                        GATE_NUM[501],FAULT_SEL);

 QY1MX4DA X11 (ACH,NACH,ACHA,ACHAB,ACHB,ACHBB,TMUPA,TMUPAB,TMUPA,
                        TMUPAB,N001,N002,N003,N004,GATE_NUM[510],FAULT_SEL);// 12 input
 QY1MX4DA X12 (BCL,NBCL,BCLA,BCLAB,BCLB,BCLBB,TMUPB,TMUPBB,TMUPB,
                        TMUPBB,N001,N002,N003,N004,GATE_NUM[511],FAULT_SEL);
 QY1MX4DA X13 (N036,N027,CVI[0],CVIB[0],CVI[1],CVIB[1],CVI[2],CVIB[2],CVI[3],CVIB[3],
                        N009,N010,N011,N012,GATE_NUM[512],FAULT_SEL);
 QY1MX4DA X14 (N028,N030,CVO[0],CVOB[0],CVO[1],CVOB[1],CVO[2],CVOB[2],-
CVO[3],CVOB[3],
                        N009,N010,N011,N012,GATE_NUM[513],FAULT_SEL);
 QY1MX4DA X16 (TMUA,TMUAB,ACH,NACH,NBCL,BCL,TMUPA,TMUPAB,N036,N027,
                        N005,N006,N007,N008,GATE_NUM[514],FAULT_SEL);
 QY1MX4DA X20 (_S141,_S142,ACH,NACH,NBCL,BCL,N034,N035,N034,N035,
                        N011,N012,N013,N014,GATE_NUM[515],FAULT_SEL);
 QY1MX4DA X21 (N021,N022,TMUPA,TMUPAB,TMUPB,TMUPBB,N023,N024,N023,
                        N024,N013,N014,N015,N016,GATE_NUM[516],FAULT_SEL);

endmodule
```

Appendix C

Environment Model Specifications

C.1 Development Structures

The environment model implemented for the formatter subsystem is organized in the following manner: it is based upon a system schematic defined using the Electric design tool, which consists of modules, submodules and interconnects. Each of the lowest level submodules in the design is treated as a black box and has an associated Verilog behaviour file. Electric automatically integrates these behaviour files using the schematic description to create one monolithic Verilog simulation file, which captures all of the environment's behaviour and internal netlists. The modules are as follows:

- environment_control
- reg_control
- barrel
- mux_control
- dut_control

The env_control module is used to coordinate the various functions of the environment: applying register setup events, pattern events, mux events and dut events. The env_-control module activates each of the other submodules and handles their termination behaviour. Each of the other submodules is responsible for one event type. Because of the design of the formatters and the overall Timing Generator Module, the environment uses four instances of the barrel module in parallel to implement pattern events, rather than one pattern control module. The schematic and modules are given below:

- ~/electric/eets.lib

- ~/verilog/env_control/env_control
- ~/verilog/barrel/barrel
- ~/verilog/mux_control/mux_control
- ~/verilog/dut_control/dut_control
- ~/electric/system.ver

The Electric schematic description and the Verilog behaviour models are considered static, and are only modified during development of the environment model. When the model is locked to a particular revision, these files are integrated in Electric to produce a system.ver file which should not be changed by the runtime user.

C.2 Runtime Structures

The structures created and modified by the runtime users of the environment model consist of the memories that describe event streams, and one runtime control file which directs program flow.

The memories used in the Formatter environment are the time and type files for pattern, dut and mux events, as well as three register setup files, for register address, data (on a register write), and register access function (read/write). Each memory file is a column of entries indexed from 0. Each entry is a dataword which must be in a parsable format.

Each memory file is accessed by a 16-bit binary address, starting at 0. The maximum number of entries any file can have is 65,536. These files are described below:

C.3 Register Setup Files

A register setup event is stored as an address, operation, and data word. Verilog's syntax does not permit these pieces of information to be stored in the same file, so there is a separate file for each item describing an event. The register setup files are:

- ~<local>/memory/rega.mem
- ~<local>/memory/regd.mem
- ~<local>/memory/regf.mem

Rega.mem contains 4-digit hexadecimal VTI internal machine addresses used in the 9000 tester. The following addresses correspond to valid registers within the formatter subsystem:

VTI Address	Description
1800, 1801	PE Control (force f'n for PE Driver)
1802	Pin Status (readback of compare - achi, bclo)
1804, 1805	Event Mode
1840	TMU Path A Select
1841	TMU Path B Select
D808	TGIC Master Reset
D80A, D80B	TGIC Diags
D820, D821, D822, D823	LDL Driver Gains for 4 barrels
D830, D831, D832, D833	LDL Strobe Gains for 4 barrels
D860, D861, D862, D863	LDL Driver Delays for 4 barrels
D870, D871, D872, D873	LDL Strobe Delays for 4 barrels

Regf.mem contains 2-bit entries [1:0] describing the corresponding access operation required for each address:

Access Code	Operation
00	Register read
01	Register write

Regd.mem contains 4-digit hexadecimal data words to be written to the addresses specified in the corresponding entries of rega.mem in the event of a write operation. Unless the register operation is write, this entry will be ignored by the model when it accesses those registers. The entries cannot be accessed independently, so that regd.-mem[12] cannot be written to rega.mem[34].

There are two user-definable parameters associated with register setup events in the system model: REG_CLK_PERIOD and RTXC_DELAY. Register setup is done by clocking data back and forth to the formatter subsystem across an 8-bit bus - this activity can be asynchronous with the global clock that controls pattern event streams. REG_CLK_PERIOD specifies the clockspeed used for register setup, and RTXC_DELAY determines the setup time for putting data on the 8-bit bus.

The following table shows the first four lines of a sample set of register memories:

Rega.mem	Regf.mem	Regd.mem
D820	01	0230
D821	01	0230
D822	01	0230
D823	01	0230

Setting the LDL driver B and C gains would be done by accessing entries [1:2] in these memories. Accessing entries [0:3] would set all four driver gains.

C.4 Event Memory Files

Every pattern, dut and mux event is described by two entries: an event time and an event type. As with register setup events, these items have to be stored in separate files.

For example, the mux events are stored in muxtime.mem and muxtype.mem. These two files are always addressed in parallel, so that applying mux event 50 means applying muxtime.mem[50] and muxtype.mem[50] to the environment. The pattern, mux and dut memory files are described below:

C.5 Pattern Events

Pattern events are described in the following files:

- ~<local>/memory/time.mem
- ~<local>/memory/type.mem

Time.mem contains 32-bit binary words [31:0] which specify event times as follows:

- bits [7:0] have a fixed resolution of 12.5 ps (in fact, 12.5 ps is the desired operating resolution of the formatters - by adjusting the driver gain registers described in section 2.3, the actual resolution can be calibrated within a range of approximately 8 - 20 ps)
- bits [31:8] are controlled differently: the least significant bit of this set, bit [8], is determined by the user-definable parameter GLOBAL_CLK_PERIOD which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any even value >= 50 ps. Whatever the value, bits [31:8] of each word in time.mem will be read in units of that value, and bits [7:0] will be read in units of 12.5 ps.
- bits [7:0] are referred to as the time vernier. When GLOBAL_CLK_PERIOD = 3200 ps, these bits resolve the time to within one 3.2 ns period (this is the normal operating mode for the model). Otherwise, they can add anywhere from 0 ps to 3187.5 ps to whatever unit is specified for bit [8] (i.e., whatever the value of GLOBAL_CLK_-PERIOD).

- all times are given relative to a global starting point, and must be well ordered, ie.for any index <n>, time.mem[n-1] <= time.mem[n] <= time.mem[n+1]
- for any index <n>, time.mem[n], time.mem[n+4] and time.mem[n-4] must fall within different periods of size GLOBAL_CLK_PERIOD, i.e. they must differ by at least one of bits [31:8]

Type.mem contains 3-bit entries [2:0] which describe event types as follows:

Event Type Code	Function
000	Drive Z
001	Drive On
010	Drive LO
011	Drive HI
100	Strobe Off
101	Strobe Z
110	Strobe LO
111	Strobe HI

The following table shows the first four lines of a sample set of pattern memories:

Time.mem	Type.mem
00000000000000000000000100000000	010
00000000000000000000001000000000	011
00000000000000000000001000000001	010
00000000000000000000001100000011	011

When GLOBAL_CLK_PERIOD = 3200 ps, applying entries [1:2] in these memories would result in: Drive HI @ 6.4 ns, Drive LO @ 6.4125 ns. Applying entries [0:3] would result in: Drive LO @3.2 ns, Drive HI @ 6.4 ns, Drive LO @ 6.4125 ns, Drive HI @ 9.625 ns.

When GLOBAL_CLK_PERIOD = 10000 ps, applying entries [1:2] in these memories would result in: Drive HI @ 20.0 ns, Drive LO @ 20.0125 ns. Applying entries [0:3] would result in: Drive LO @ 10.0 ns, Drive HI @ 20.0 ns, Drive LO @ 20.0125 ns, Drive HI @ 30.025 ns.

C.6 Mux Events

Mux events are described in the following files:

- ~<local>/memory/muxtime.mem
- ~<local>/memory/muxtype.mem

Muxtime.mem contains 36-bit binary words [35:0] which specify event times as follows:

- the least significant bit of each word is set by the user-definable parameter MUX_RESOLUTION which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any value >= 1 ps. Whatever the value, each word in muxtime.mem will be read in units of that value.
- all times are given relative to a global starting point, and must be strictly ordered, ie.for any index <n>, muxtime.mem[n-1] < muxtime.mem[n] < muxtime.mem[n+1]

Muxtype.mem contains 3-bit entries [2:0] which describe event types as follows:

- bit [2] of each type word is ignored by the model, since mux events can only be drive events

Event Type Code	Function
000/100	Drive Z
001/101	Drive On
010/110	Drive LO
011/111	Drive HI

Mux events generate pulses that are sent to the formatter subsystem. The width of these pulses is set by the user-definable parameter MUX_PULSE_WIDTH which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any value >= 1 ps.

Because pattern events and mux events pass through different logic within the formatter subsystem, they may not be synchronized when they appear at the outputs. A pattern event and mux event which have the same global timing may not appear at the outputs simultaneously. In order to calibrate this difference, the user may delay the beginning of the mux events by manipulating the parameter MUX_DELAY which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any value >= 0 ps.

The following table shows the first four lines of a sample set of pattern memories:

Muxtime.mem	Muxtype.mem
00000000000000000000000100000000	010
00000000000000000000001000000000	011
00000000000000000000001000000001	010
00000000000000000000001100000011	011

When MUX_RESOLUTION = 1 ps, applying entries [1:2] in these memories would result in: Drive HI @ 512 ps, Drive LO @ 513 ps. Applying entries [0:3] would result in: Drive LO @ 256 ps, Drive HI @ 512 ps, Drive LO @ 513 ps, Drive HI @ 1031 ps.

When MUX_RESOLUTION = 10 ps, applying entries [1:2] in these memories would result in: Drive HI @ 5120 ps, Drive LO @ 5130 ps. Applying entries [0:3] would result in: Drive LO @ 2560 ps, Drive HI @ 5120 ps, Drive LO @ 5130 ps, Drive HI @ 10310 ps.

C.7 Dut Events

Dut events are described in the following files:

- ~<local>/memory/duttime.mem
- ~<local>/memory/duttype.mem

Duttime.mem contains 36-bit binary words [35:0] which specify event times as follows:

- the least significant bit of each word is set by the user-definable parameter DUT_RESOLUTION which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any value >= 1 ps. Whatever the value, each word in duttime.mem will be read in units of that value.
- all times are given relative to a global starting point, and must be strictly ordered, ie.for any index <n>, duttime.mem[n-1] < duttime.mem[n] < duttime.mem[n+1]

Duttype.mem contains 2-bit entries [1:0] which describe event types as follows:

Event Type Code	Function
00	ACHI = 0, BCLO = 0
01	ACHI = 0, BCLO = 1

Event Type Code	Function
10	ACHI = 1, BCLO = 0
11	ACHI = 1, BCLO = 1

Dut events describe comparator inputs to the formatter subsystem that simulate the results which the formatters would see coming from the test head in the course of operation. These values when strobed by the formatters determine pass/fail results that are outputted by the formatter subsystem.

Because pattern events and dut events pass through different logic within the model, they may not be synchronized when they appear at the outputs. A pattern event and dut event which have the same global timing may not appear to the formatters simultaneously. In order to calibrate this difference, the user may delay the beginning of the dut events by manipulating the parameter DUT_DELAY which is specified in the runtime control file envTest.v. This parameter is given in picoseconds and can be any value >= 0 ps.

The following table shows the first four lines of a sample set of pattern memories:

Duttime.mem	Duttype.mem
00000000000000000000000100000000	00
00000000000000000000001000000000	01
00000000000000000000001000000001	10
00000000000000000000001100000011	11

When DUT_RESOLUTION = 1 ps, applying entries [0:1] in these memories would result in: ACHI = 0, BCLO = 0 @ 256 ps, ACHI = 0, BCLO = 1 @ 512 ps. Applying entries [0:3] would result in: ACHI = 0, BCLO = 0 @ 256 ps, ACHI = 0, BCLO = 1 @ 512 ps, ACHI = 1, BCLO = 0 @ 513 ps, ACHI = 1, BCLO = 1 @ 1031 ps.

When DUT_RESOLUTION = 5 ps, applying entries [0:1] in these memories would result in: ACHI = 0, BCLO = 0 @ 1280 ps, ACHI = 0, BCLO = 1 @ 2560 ps. Applying entries [0:3] would result in: ACHI = 0, BCLO = 0 @ 1280 ps, ACHI = 0, BCLO = 1 @ 2560 ps, ACHI = 1, BCLO = 0 @ 2565 ps, ACHI = 1, BCLO = 1 @ 5155 ps.

C.8 Test Sequence Control

The event generation sequence is controlled by the file:

- ~/verilog/run/envTest.v

This file contains all the information required to compose a customized multi-stage event generation program for the formatter environment model. It determines what event streams are processed, what streams are ignored, and how they are combined. The runtime user can build an event program by specifying blocks of register, pattern, dut and mux events to occur alone or together, in any order, and for any period of time. Each block of events is specified by a start and end address. All of the events the user wishes to apply must be in the memory files specified earlier. However, by addressing different blocks in those files at different times, the user can design arbitrarily complex sequences of register setup, pattern, dut and mux streams, together or independently, with all of the control mechanisms available in the Verilog definition language, including while, for and if-then constructs.

The test sequence control has access to all of the I/O signals in the interface to the environment/formatter system shown in Figure 1. In the control file, the user can set and vary input signals as well as a group of special parameters that customize the model to the user's preference.

By initializing all of the interface inputs and parameters, the user effectively creates one test sequence including one or more of register, pattern, mux and dut events. By using the interface's outputs to trigger changes in the inputs and parameters, the user starts other test sequences. In this manner, using all of the control syntax available in Verilog's definition language, the user can create arbitrarily complex and long sets of test sequences.

C.9 System Inputs

The user can specify the following inputs to the system, which format the test sequence to the user's preference:

- THBSEL [0]

Because of the design of the ITS 9000 Tester, outputs from the formatters can be sent to one of two test head modules; this toggle specifies which lines the formatter outputs follow. The resulting driver outputs will be either DHIA and DINHA or DHIB and DINHB, correspondingly, the comparator signals strobed will be either ACHA and BCLA or ACHB and BCLB. When THBSEL = 0, the test head is A; when THBSEL = 1, the test head is B.

- PATTERN_BEG [15:0]
- PATTERN_END [15:0]

These 16-bit addresses specify the pattern event block to be applied from the files time.mem and type.mem on the current test sequence. To apply the first four events, for example, the user would set PATTERN_BEG = 16'd0, PATTERN_END = 16'd3. The maximum number of entries in the memory files is limited by the 16-bit address to 65536.

- REG_BEG [15:0]
- REG_END [15:0]

These 16-bit addresses specify the register setup event block to be applied from the files rega.mem, regf.mem and regd.mem on the current test sequence.

- MUX_BEG [15:0]

• MUX_END [15:0]

These 16-bit addresses specify the pattern event block to be applied from the files muxtime.mem and muxtype.mem on the current test sequence.

• DUT_BEG [15:0]REG_END

• DUT_END [15:0]

These 16-bit addresses specify the pattern event block to be applied from the files dut-time.mem and duttype.mem on the current test sequence.

• TEST_MODE [2:0]

This 3-bit code specifies which of the three event streams are activated on the current test sequence. TEST_MODE is coded in the following way:

• bit [0] enables and disables pattern event streams

• bit [1] enables and disables mux event streams

• bit [2] enables and disables dut event streams

• on a given test run, any combination of these three event streams can be enabled to allow the user to control access to all of the interfaces to the system

• when one of these event types is enabled, it references the start and end addresses given above; when it is disabled, those addresses are ignored and do not have to be reset

• REG_MODE [0]

The formatters are most frequently exercised in the following context: the registers are initialized asynchronously for the current task, and then pattern, mux and dut events are applied synchronously to the system. Since register events cannot occur at the same time as the other event types, the control of a test sequence is handled by REG_MODE and TEST_MODE in the following way:

• as soon as REG_MODE is set to 1, the register block specified by REG_BEG and REG_END is executed. While REG_MODE = 1, pattern, mux and dut events cannot be executed.

• as soon as REG_MODE is set to 0, REG_BEG and REG_END are ignored, and the event streams specified by TEST_MODE are applied using the start and end addresses chosen for them.

The end of event streams is signaled by the following outputs:

• REG_DONE [0]

• PATTERN_DONE [0]

• DUT_DONE [0]

Pattern and mux events are very closely tied within the model. If the user chooses to apply both pattern and mux events, it is to exercise the formatters in a special (PINMUX) mode of operation in which two identical formatter systems feed the same pin on the 9000 test head, in order to double the effective event speed of the tester. As a result, when both pattern and mux events are applied, the test sequence is not considered finished until both event streams have been completed. The environment model incorporates this internal control mechanism, and outputs only one signal to indicate that pattern and mux events have finished: PATTERN_DONE. Since dut event streams may be longer or shorter than pattern event streams during the course of a test, the user is given control over responding to the signal DUT_DONE.

The user can specify test sequences in the following way:

• set REG_MODE to 1, starting a register setup sequence

• when REG_DONE changes from 0 to 1 (at the positive edge), set REG_MODE to 0, starting the pattern, mux and dut streams

- when both or either of PATTERN_DONE and DUT_DONE change from 0 to 1, the sequence has completed; this can trigger changing the start and end addresses for all the events, and changing REG_MODE back to 1, to start the new register setup sequence

In this way, the user can write test programs that manipulate blocks of events and control flow between execution of those events to produce arbitrarily complex behaviour in the environment/formatter system.

C.10 Control Parameters

The user can customize the test sequence by specifying the following parameters:

- GLOBAL_CLK_PERIOD

GLOBAL_CLK_PERIOD specifies the speed at which the entire system is operated, although certain modules in the environment model have different internal clocks. This is also the clock which drives the formatter chips themselves. Finally, it specifies the resolution of pattern event timing words, as described in section C.5.

- CLKOUT_DELAY

Although the period of the formatter clock is given by GLOBAL_CLK_PERIOD, there are certain situations in which the user may wish to delay the formatter clock from the environment clock, in order to ensure outputs from the environment are stable before they are clocked into the formatters. CLKOUT_DELAY specifies this delay value; when CLKOUT_DELAY = 0 ps, the formatters and the environment are synchronized.

- REG_CLK_PERIOD
- RTXC_DELAY

These parameters determine the clock rate for register setup (which is asynchronous from the other event streams), and the strobing rate on the data bus which is used to initialize registers. They are described in section C.4.

- MUX_RESOLUTION
- MUX_DELAY
- MUX_PULSE_WIDTH

These parameters determine the specific characteristics of mux event streams, and are described in section C.6.

- DUT_RESOLUTION
- DUT_DELAY

These parameters determine the specific characteristics of dut event streams, and are described in section C.7.

C.11 Memory File Converter

In order to make the interface to the runtime user more intuitive, a program has been written which accepts as inputs files of the form:

- D0@6.4
- D1@9.6125

and produces time and type files of the form required by the formatter/environment model.

Syntax

Synopsis:

```
convert [ -mbf ] [ parameter values ] < [ input file ]
```

Options

[-mbf]	Parameter declaration, specifies which parameters are explicitly to be set, and the order in which they follow:
-m	Mode of operation, specifies whether output files are for pattern events or for mux events: p - pattern events m - mux events d - dut events Default is pattern events.
-b	Bit-length of timing words. Can be any integer value, but must be large enough to accomodate largest timing value to be produced. Default is 32. If mux/dut mode is specified, but bit-length is not, then the default is 36.
-f	File output command. Specifies output to two files: <time.file> <type.file> Both filenames must be given. If -f option is not used, output is sent to tty.
[parameter values]	When the parameters have been declared in the first section, the values to which they are assigned follow, in the order specified in the declaration:

```
convert -mbf m 36 ~/time.mem ~/type.mem < test

convert -fmb ~/time.mem ~/type.mem m 36 < test
```

When the declaration order changes, so does the order of the values assigned.

Modes and Input FIles

The convert program operates in two modes that require different input file formats. In pattern mode, each line of the input file has the following syntax:<event type><value>@<time in ns>

Event Types

This input string is coded in the following way for pattern and mux events:

Event Type Code	Function
D,d	Drive
S,s,T,t	Strobe

Value Code	Function
0	Lo
1	Hi
Z,z	Z
X.x	On (with Drive), Off (with Strobe)
ON	On (only valid with Drive)

There are four special cases in specifying event types:

• DX (case insensitive) - Drive On

• DON (case insensitive) - Drive On

• TX (caseinsensitive) - Strobe Off

• X (case insensitive) - Strobe Off

• NOP (case insensitive) - No Operation (skip current barrel)

For dut events, the input string has the following format:

Event Type Code	Function
00	ACHI = 0, BCLO = 0
01	ACHI = 0, BCLO = 1
10	ACHI = 1, BCLO = 0
11	ACHI = 1, BCLO = 1

Event Times

TIming values are always specified in units of 1 ns. Although the environment model allows pattern timing to be specified with greater flexibility, the convert program fixes pattern event resolution to 12.5 ps, and requires that the clock period for pattern events be set to 3200 ps. The resolution of mux and dut events can be set by the user, as explained in the following sections. All events in the pattern, mux and dut files must meet the following requirements:

• all times are given relative to a global starting point, and must be well ordered, ie.for any index <n>, time[n-1] <= time[n] <= time[n+1]

• for any index <n>, time[n], time[n+4] and time[n-4] must fall within different clock periods, i.e. they must differ by at least 3.2 ns

Sample Files

Pattern Events

The following is a sample pattern file that is used by convert to create inputs to the model:

patterntest

```
D0@3.2

d1@6.4

Dz@6.45

DOn@9.6

S0@9.6

s1@9.6125

tZ@12.8

X@16
```

This can be used with the command:

```
convert -mbf p 32 ./time.mem ./type.mem < patterntest
```

The output files produced are:

time.mem	type.mem
00000000000000000000000100000000	010
00000000000000000000001000000000	011
00000000000000000000001000000100	001
00000000000000000000001100000000	000

time.mem	type.mem
00000000000000000000001100000000	110
00000000000000000000001100000001	111
00000000000000000000010000000000	101
00000000000000000000010100000000	100

Mux Events

When convert is used in mux/dut event mode, it requires a slightly different input file. This file consists of entries with the same format as those for pattern events. However, the first line this file must be of the following format:resolution=<resolution value in ps>

If this is line is missing from the top of a mux/dut input file, the convert program will signal an error. The resolution value can be any integer >= 1 ps. After this line, the mux/dut file has the same format as the pattern file.

The following is a sample mux file that is used by convert to create inputs to the model:

muxtest

```
resolution=1
D0@3.2
d1@6.4
Dz@6.45
DOn@9.6
S0@9.6
```

s1@9.6125

tZ@12.8

X@16

This can be used with the command:

convert -mbf m 36 ./muxtime.mem ./muxtype.mem < muxtest

The output files produced are:

muxtime.mem	muxtype.mem
000000000000000000000000110010000000	010
000000000000000000000001100100000000	011
000000000000000000000001100100110010	001
000000000000000000000010010110000000	000
000000000000000000000010010110000000	110
000000000000000000000010010110001100	111
000000000000000000000011001000000000	101
000000000000000000000011111010000000	100

Dut Events

The following is a sample dut file that is used by convert to create inputs to the model:duttest

resolution=12.5

00@3.2

01@6.4

10@6.45

11@9.6

10@9.6

01@9.6125

00@12.8

11@16

This can be used with the command:

convert -mbf d 36 ./duttime.mem ./duttype.mem < duttest

The output files produced are:

duttime.mem	duttype.mem
00000000000000000000000100000000	00
00000000000000000000001000000000	01
00000000000000000000001000000100	10
00000000000000000000001100000000	11
00000000000000000000001100000000	10
00000000000000000000001100000001	01
00000000000000000000010000000000	00
00000000000000000000010100000000	11

Appendix D

The Environment Model

```
module environment(PATTERN_BEG, DA, DB, DC, DD, R, RS, CA, CB, CC, CD, REG_BEG,
 REG_END, PATTERN_END, MUX_BEG, MUX_END, TEST_MODE, DUT_BEG, DUT_END,
 R_G, THBSELOUT, CLK, SETLOOUT, SETZOUT, SETHIOUT, SETONOUT, RTXC,
 CLKOUT, REG_MODE, REG_DONE, PATTERN_DONE, ACHA, BCLA, ACHB, BCLB,
 DUT_DONE);
 input [15:0] PATTERN_BEG;
 input [15:0] PATTERN_END;
 output [5:0] DB;
 output [5:0] DA;
 output [5:0] CB;
 output [5:0] CD;
 output [5:0] CC;
 output [5:0] DC;
 output [5:0] DD;
 output [5:0] CA;
 input [15:0] REG_BEG;
 input [15:0] REG_END;
 inout [9:2] R;
 output [1:0] RS;
 input [15:0] MUX_BEG;
 input [15:0] MUX_END;
 input [2:0] TEST_MODE;
 input [15:0] DUT_BEG;
 input [15:0] DUT_END;
 inout [9:2] R_G;
 input THBSELOUT, CLK, REG_MODE;
 output SETLOOUT, SETZOUT, SETHIOUT, SETONOUT, RTXC, CLKOUT, REG_DONE,
 PATTERN_DONE, ACHA, BCLA, ACHB, BCLB, DUT_DONE;
 wire [1:0] INDEXB;
 wire [1:0] INDEXC;
 wire [1:0] INDEXD;
 wire [1:0] INDEXA;
 dut_control U1 (.DUT_BEG(DUT_BEG), .DUT_END(DUT_END), .CLK(CLK),
 .START_DUT_RUN(NET1), .ACHA(ACHA), .BCLA(BCLA), .ACHB(ACHB), .BCLB(BCLB),
 .DONE(DUT_DONE), .THBSEL(THBSELOUT));
 mux_control U2 (.MUX_BEG(MUX_BEG), .MUX_END(MUX_END), .CLK(CLK),
 .START_MUX_RUN(START_MUX_RUN), .SETLOOUT(SETLOOUT), .SETHIOUT(SETHIOUT),
 .SETZOUT(SETZOUT), .SETONOUT(SETONOUT), .DONE(PATTERN_DONEM));
 env_control U3 (.CLK(CLK), .REG_MODE(REG_MODE), .PATTERN_DONEA(PATTERN_DONEA),
 .PATTERN_DONEB(PATTERN_DONEB), .PATTERN_DONEC(PATTERN_DONEC),
 .PATTERN_DONED(PATTERN_DONED), .PATTERN_DONE(PATTERN_DONE),
 .START_PATTERN_RUN(START_PATTERN_RUN), .INDEXA(INDEXA),
 .START_REG_SETUP(START_REG_SETUP), .CLKOUT(CLKOUT), .INDEXB(INDEXB),
 .INDEXC(INDEXC), .INDEXD(INDEXD), .PATTERN_DONEM(PATTERN_DONEM),
 .START_MUX_RUN(START_MUX_RUN), .TEST_MODE(TEST_MODE),
 .START_DUT_RUN(NET1));
 reg_control U4 (.REG_BEG(REG_BEG), .REG_END(REG_END), .CLK(CLK),
 .START_REG_SETUP(START_REG_SETUP),
 .R(R), .RS(RS), .RTXC(RTXC), .DONE(REG_DONE), .R_G(R_G));
 barrel U5 (.CLK(CLK), .PATTERN_BEG(PATTERN_BEG), .PATTERN_END(PATTERN_END),
 .DRIVE(DA), .DONE(PATTERN_DONEA), .STROBE(CA), .INDEX(INDEXA),
 .START_PATTERN_RUN(START_PATTERN_RUN));
 barrel U6 (.CLK(CLK), .PATTERN_BEG(PATTERN_BEG), .PATTERN_END(PATTERN_END),
 .DRIVE(DC), .DONE(PATTERN_DONEC), .STROBE(CC), .INDEX(INDEXC),
 .START_PATTERN_RUN(START_PATTERN_RUN));
 barrel U7 (.CLK(CLK), .PATTERN_BEG(PATTERN_BEG), .PATTERN_END(PATTERN_END),
 .DRIVE(DB), .DONE(PATTERN_DONEB), .STROBE(CB), .INDEX(INDEXB),
 .START_PATTERN_RUN(START_PATTERN_RUN));
 barrel U8 (.CLK(CLK), .PATTERN_BEG(PATTERN_BEG), .PATTERN_END(PATTERN_END),
 .DRIVE(DD), .DONE(PATTERN_DONED), .STROBE(CD), .INDEX(INDEXD),
 .START_PATTERN_RUN(START_PATTERN_RUN));

endmodule

// IMPORTED EXTERNAL FILE /usr/users/guru/thesis/verilog/dut_control/dut_control
'timescale 1 ps / 1 ps

/***************************** DUT_CONTROL ********************************/

module dut_control(DUT_BEG, DUT_END, CLK,
                    START_DUT_RUN, THBSEL, ACHA, BCLA, ACHB, BCLB, DONE);

 input [15:0] DUT_END;
 input [15:0] DUT_BEG;
 input CLK, START_DUT_RUN, THBSEL;
 output ACHA, BCLA, ACHB, BCLB, DONE;

reg [35:0] DUTTIME_MEM [0:65535]; // cache for DUT pattern times
reg [1:0] DUTTYPE_MEM [0:65535]; // cache for DUT pattern types
reg CLK2; // internal clock, 1ps period
reg [15:0] MCOUNT; // pointer to memory addresses
reg [35:0] TCOUNT; // 24 msb counter for 3.2ns resolution
reg [35:0] DTIME, LASTTIME, DELAY; // current 32-bit time
reg [1:0] TYPE; // current type
reg FIRE; // fires DUT event
reg ACHA, BCLA, ACHB, BCLB; // outputs to ric and dic
reg DONE; // file has been finished
wire [35:0] DEBUG;

initial begin
 ACHA = 1'b0;
 BCLA = 1'b0;
 ACHB = 1'b0;
 BCLB = 1'b0;
end

always @(posedge faultdrive.start_run) begin
 ACHA = 1'b0;
 BCLA = 1'b0;
 ACHB = 1'b0;
 BCLB = 1'b0;
end

always @(posedge START_DUT_RUN) begin
@(posedge CLK) begin
init_run;
end
end

always @(posedge CLK) if ((START_DUT_RUN===1'b0)&&(faultdrive.e1.reg_mode===1'b0))
TCOUNT = TCOUNT+1;
always @(posedge FIRE) #1 FIRE = 1'b0;

always @(TCOUNT) if (DONE===1'b0) begin
if (TCOUNT === DTIME+1) FIRE = 1'b1;
#0.1 dut_run;
end

always @(negedge FIRE) if (MCOUNT > DUT_END) #5 DONE = 1'b1;

/*---------------------------- End of model ---------------------------*/
```

```
/********************************* task **********************************/

task init_run;
begin
// loading of caches from files
$readmemb("../memory/duttype.mem",
                    DUTTYPE_MEM);
$readmemb("../memory/duttime.mem",
                    DUTTIME_MEM);
MCOUNT [15:0] = DUT_BEG [15:0];
DTIME = DUTTIME_MEM[MCOUNT]+(envdrive.DUT_DELAY/envdrive.DUT_RESOLUTION);
TYPE = DUTTYPE_MEM[MCOUNT];
TCOUNT = 36'b0;
LASTTIME = 36'b0;
FIRE = 1'b0;
DONE = 1'b0;
end
endtask

task dut_run;
begin
if (FIRE===1'b1) begin
case (THBSEL)
1'b0: begin
ACHA = TYPE[1];
BCLA = TYPE[0];
end
1'b1: begin
ACHB = TYPE[1];
BCLB = TYPE[0];
end
endcase
LASTTIME = DTIME;
MCOUNT = MCOUNT+1;
DTIME = DUTTIME_MEM[MCOUNT]+(envdrive.DUT_DELAY/envdrive.DUT_RESOLUTION);
if (MCOUNT <= DUT_END) TYPE = DUTTYPE_MEM[MCOUNT];
end
end
endtask

endmodule

// IMPORTED EXTERNAL FILE /usr/users/guru/thesis/verilog/mux_control/mux_control
'timescale 1 ps / 1 ps

/***************************** MUX_CONTROL ********************************/

module mux_control(MUX_BEG, MUX_END, CLK, START_MUX_RUN,
                        SETLOOUT, SETHIOUT, SETZOUT, SETONOUT, DONE);

 input [15:0] MUX_END;
 input [15:0] MUX_BEG;
 input CLK, START_MUX_RUN;
 output SETLOOUT, SETHIOUT, SETZOUT, SETONOUT, DONE;

reg [35:0] MUXTIME_MEM [0:65535]; // cache for mux pattern times
reg [2:0] MUXTYPE_MEM [0:65535]; // cache for mux pattern types
reg [15:0] MCOUNT; // pointer to memory addresses
reg [35:0] TCOUNT; // 24 msb counter for 3.2ns resolution
reg [35:0] MTIME, LASTTIME, DELAY; // current 32-bit time
reg [2:0] TYPE; // current type
reg FIRE; // fires mux event
reg SETLOOUT, SETHIOUT, SETZOUT, SETONOUT; // outputs to ric and dic
reg DONE; // file has been finished

initial begin
SETLOOUT = 1'b0;
SETHIOUT = 1'b0;
SETZOUT = 1'b0;
SETONOUT = 1'b0;
end

always @(posedge faultdrive.start_run) begin
SETLOOUT = 1'b0;
SETHIOUT = 1'b0;
SETZOUT = 1'b0;
SETONOUT = 1'b0;
end

always @(posedge START_MUX_RUN) begin
@(posedge CLK) begin
init_run;
end
end

always @(posedge CLK) if ((START_MUX_RUN===1'b0)&&(faultdrive.e1.reg_mode===1'b0))
TCOUNT = TCOUNT+1;

always @(posedge SETLOOUT) #(envdrive.MUX_PULSE_WIDTH) SETLOOUT = 1'b0;
always @(posedge SETHIOUT) #(envdrive.MUX_PULSE_WIDTH) SETHIOUT = 1'b0;
always @(posedge SETZOUT) #(envdrive.MUX_PULSE_WIDTH) SETZOUT = 1'b0;
always @(posedge SETONOUT) #(envdrive.MUX_PULSE_WIDTH) SETONOUT = 1'b0;
always @(posedge FIRE) #1 FIRE = 1'b0;

always @(TCOUNT) if (DONE===1'b0) begin
if (TCOUNT === MTIME+1) FIRE = 1'b1;
#0.1 mux_run;
end

always @(negedge FIRE) if (MCOUNT > MUX_END) #5 DONE = 1'b1;

/*---------------------------- End of model ----------------------------*/

/********************************* task *********************************/

task init_run;
begin
// loading of caches from files
$readmemb("../memory/muxtype.mem",
                    MUXTYPE_MEM);
$readmemb("../memory/muxtime.mem",
                    MUXTIME_MEM);
MCOUNT [15:0] = MUX_BEG [15:0];
MTIME = MUXTIME_MEM[MCOUNT]+(envdrive.MUX_DELAY/envdrive.MUX_RESOLUTION);
TYPE = MUXTYPE_MEM[MCOUNT];
LASTTIME = 36'b0;
TCOUNT = 36'b0;
SETLOOUT = 1'b0;
SETHIOUT = 1'b0;
SETZOUT = 1'b0;
SETONOUT = 1'b0;
FIRE = 1'b0;
DONE = 1'b0;
end
endtask

task mux_run;
```

```
begin
if (FIRE===1'b1) begin
case (TYPE)
3'b000: begin
SETZOUT = 1'b1;
end
3'b001: begin
SETONOUT = 1'b1;
end
3'b010: begin
SETLOOUT = 1'b1;
SETONOUT = 1'b1;
end
3'b011: begin
SETHIOUT = 1'b1;
SETONOUT = 1'b1;
end
endcase
LASTTIME = MTIME;
MCOUNT = MCOUNT+1;
MTIME = MUXTIME_MEM[MCOUNT]+(envdrive.MUX_DELAY/envdrive.MUX_RESOLUTION);
if (MCOUNT <= MUX_END) TYPE = MUXTYPE_MEM[MCOUNT];
end
end
endtask

endmodule

// IMPORTED EXTERNAL FILE /usr/users/guru/thesis/verilog/env_control/env_control
'timescale 1 ps / 1 ps

/**************************** ENV_CONTROL ********************************/

module env_control(INDEXA, INDEXB, INDEXC, INDEXD, CLK, REG_MODE,
                   PATTERN_DONEA, PATTERN_DONEB, PATTERN_DONEC,
                   PATTERN_DONED, PATTERN_DONEM, PATTERN_DONE,
                   START_PATTERN_RUN, START_REG_SETUP, CLKOUT,
                   TEST_MODE, START_MUX_RUN, START_DUT_RUN);

 output [1:0] INDEXD;
 output [1:0] INDEXC;
 output [1:0] INDEXB;
 output [1:0] INDEXA;
 input CLK, REG_MODE, PATTERN_DONEA, PATTERN_DONEB,
       PATTERN_DONEC, PATTERN_DONED, PATTERN_DONEM;
 input [2:0] TEST_MODE;
 output PATTERN_DONE, START_PATTERN_RUN, START_REG_SETUP,
       CLKOUT, START_MUX_RUN, START_DUT_RUN;

reg [1:0] INDEXA, INDEXB, INDEXC, INDEXD;
reg PATTERN_DONE, START_PATTERN_RUN, START_REG_SETUP,
      START_MUX_RUN, START_DUT_RUN;
reg PDONE;
reg [5:0] DONE_COUNT;
wire CLKOUT;

assign #(envdrive.CLKOUT_DELAY) CLKOUT = CLK;

initial begin
fork
DONE_COUNT = 6'b0;
case (REG_MODE)
1'b1: begin
init_reg_setup;
```

```
 end
1'b0: begin
init_pattern_run;
 end
endcase
join
end

// switching between register setup and pattern run modes
always @(negedge REG_MODE) begin
 @(posedge CLK) begin
init_pattern_run;
 end
end

always @(posedge REG_MODE) begin
 @(posedge CLK) begin
init_reg_setup;
 end
end

always @(posedge PATTERN_DONEA) DONE_COUNT[4] = 1;
always @(posedge PATTERN_DONEB) DONE_COUNT[3] = 1;
always @(posedge PATTERN_DONEC) DONE_COUNT[2] = 1;
always @(posedge PATTERN_DONED) DONE_COUNT[1] = 1;
always @(posedge PATTERN_DONEM) DONE_COUNT[0] = 1;
always @(posedge PDONE) DONE_COUNT[5] = 1;

always @(posedge CLK) #1 if ((REG_MODE===1'b0)&&(START_PATTERN_RUN===1'b0))
begin
 if (TEST_MODE[0]===1'b1) begin
 if (DONE_COUNT[4:1]===4'b1111) begin
 fork
 PDONE = 1'b1;
 #(envdrive.GLOBAL_CLK_PERIOD/2) PDONE = 0;
 #(envdrive.GLOBAL_CLK_PERIOD/2) DONE_COUNT[4:1] = 4'b0;
 join
 end
 end
 else begin
 fork
 PDONE = 1'b1;
 #(envdrive.GLOBAL_CLK_PERIOD/2) PDONE = 0;
 #(envdrive.GLOBAL_CLK_PERIOD/2) DONE_COUNT[4:1] = 4'b0;
 join
 end
end

always @(posedge CLK) #1 if ((REG_MODE ===1'b0)&&(START_PATTERN_RUN===1'b0)) begin
if (TEST_MODE[1]===1'b1) begin
if ((DONE_COUNT[5]===1'b1)&&(DONE_COUNT[0]===1'b1)) begin
 fork
 PATTERN_DONE = 1'b1;
 #(envdrive.GLOBAL_CLK_PERIOD/2) PATTERN_DONE = 1'b0;
 #(envdrive.GLOBAL_CLK_PERIOD/2) DONE_COUNT = 6'b0;
 join
end
end
else if (DONE_COUNT[5]===1'b1) begin
 fork
 PATTERN_DONE = 1'b1;
 #(envdrive.GLOBAL_CLK_PERIOD/2) PATTERN_DONE = 1'b0;
 #(envdrive.GLOBAL_CLK_PERIOD/2) DONE_COUNT = 6'b0;
 join
```

```
end
end
always @(posedge CLK) #1 if ((REG_MODE ===1'b0)&&(TEST_MODE===3'b0)) begin
 fork
 PATTERN_DONE = 1'b1;
 #(envdrive.GLOBAL_CLK_PERIOD/2) PATTERN_DONE = 1'b0;
 #(envdrive.GLOBAL_CLK_PERIOD/2) DONE_COUNT = 6'b0;
 join
end

/*---------------------------- End of model ---------------------------*/

/******************************** task *********************************/

task init_pattern_run;

begin
PATTERN_DONE = 1'b0;
DONE_COUNT = 6'b0;
fork
INDEXA = 2'b00;
INDEXB = 2'b01;
INDEXC = 2'b10;
INDEXD = 2'b11;
PDONE = 1'b0;
case (TEST_MODE[0])
1'b1: begin
START_PATTERN_RUN = 1'b1;
#(3*(envdrive.GLOBAL_CLK_PERIOD/2)) START_PATTERN_RUN = 1'b0;
end
1'b0: START_PATTERN_RUN=1'b0;
endcase
case (TEST_MODE[1])
1'b1: begin
START_MUX_RUN = 1'b1;
#(3*(envdrive.GLOBAL_CLK_PERIOD/2)) START_MUX_RUN = 1'b0;
end
1'b0: START_MUX_RUN = 1'b0;
endcase
case (TEST_MODE[2])
1'b1: begin
START_DUT_RUN = 1'b1;
#(3*(envdrive.GLOBAL_CLK_PERIOD/2)) START_DUT_RUN = 1'b0;
end
1'b0: START_DUT_RUN = 1'b0;
endcase
join
end

endtask

task init_reg_setup;
begin
PDONE = 1'b0;
PATTERN_DONE = 1'b0;
DONE_COUNT = 6'b0;
START_REG_SETUP = 1'b1;
#(envdrive.GLOBAL_CLK_PERIOD-1) START_REG_SETUP = 1'b0;
end
endtask

/*************************** end of task *********************************/

endmodule
```

```
// IMPORTED EXTERNAL FILE /usr/users/guru/thesis/verilog/reg_control/reg_control
'timescale 1 ps / 1 ps

/***************************** REG_CONTROL *******************************/

module reg_control(REG_BEG, REG_END, R, RS, CLK, START_REG_SETUP, RTXC,
                              DONE, R_G);

 output [1:0] RS;
 inout [9:2] R, R_G;
 input [15:0] REG_END;
 input [15:0] REG_BEG;
 input CLK, START_REG_SETUP;
 output RTXC, DONE;

reg [15:0] REGAMEM [0:65535]; // cache for setup register addresses
reg [15:0] REGDMEM [0:65535]; // cache for setup register data
reg [1:0] REGFMEM [0:65535]; // cache for register functions
reg [15:0] REGADDR; // current register address
reg [15:0] REGDATA; // current register data
reg [1:0] REGFUNC; // current register function
reg [15:0] MCOUNT; // pointer to memory addresses
reg CLK2; // internal clock, 4800ps
reg [1:0] RS;
wor [9:2] R, R_G;
reg [9:2] RTMP;
reg RTXC, DONE;

assign R = (RS!=2'b00) ? RTMP : 8'b0;
assign R_G = (RS!=2'b00) ? RTMP : 8'b0;

initial begin
CLK2 = 1'b0;
#(envdrive.GLOBAL_CLK_PERIOD/2) CLK2= 1'b1;
end

always @(posedge CLK2) if (faultdrive.e1.reg_mode===1'b1) #(envdrive.REG_CLK_PERIOD/2) CLK2 =
1'b0;
always @(negedge CLK2) if (faultdrive.e1.reg_mode===1'b1) #(envdrive.REG_CLK_PERIOD/2) CLK2
= 1'b1;

always @(posedge CLK) #1 if (START_REG_SETUP===1'b1) begin
init_run;
end

always @(posedge faultdrive.e1.reg_mode) begin
CLK2 = 1'b0;
@(posedge CLK) CLK2=1'b1;
end

// incrementing mcount for pattern run mode
always @(posedge CLK2) #1 begin
if ((START_REG_SETUP===1'b0)&&(MCOUNT<REG_END))
begin
FMEM[MCOUNT]);
MCOUNT = MCOUNT+1;
REGADDR = REGAMEM[MCOUNT];
REGDATA = REGDMEM[MCOUNT];
REGFUNC = REGFMEM[MCOUNT];
end
else if (START_REG_SETUP===1'b1)
begin
FMEM[MCOUNT]);
MCOUNT = MCOUNT;
```

```
REGADDR = REGAMEM[MCOUNT];
REGDATA = REGDMEM[MCOUNT];
REGFUNC = REGFMEM[MCOUNT];
end
else if ((MCOUNT>=REG_END)) begin
RS = 2'b10;
RTMP = 8'b11111111;
DONE = 1'b1;
end
end

always @(posedge CLK2) #2 if (DONE===1'b0) reg_run;
always @(posedge CLK2) #2 if (DONE===1'b1) RS = 2'b10;
always @(posedge RTXC) #envdrive.RTXC_DELAY RTXC = 1'b0;

/*---------------------------- End of model ----------------------------*/

/********************************* task **********************************/

task init_run;

begin
// loading of caches from files
$readmemh("../memory/rega.mem",
                    REGAMEM);
$readmemh("../memory/regd.mem",
                    REGDMEM);
$readmemb("../memory/regf.mem",
                    REGFMEM);
MCOUNT [15:0] = REG_BEG [15:0];
RTXC = 1'b0;
DONE = 1'b0;
//$display("TIME = %t, MCOUNT = %b, REGF_MEM[MCOUNT] = %b\n", $time, MCOUNT, REG-
FMEM[MCOUNT]);
REGADDR = REGAMEM[MCOUNT];
REGDATA = REGDMEM[MCOUNT];
REGFUNC = REGFMEM[MCOUNT];
end

endtask

task reg_run;

begin
if (DONE===1'b0) begin
RS = 2'b11;
case (REGADDR)
16'h1800: RTMP = 8'h0;
16'h1801: RTMP = 8'h1;
16'h1802: RTMP = 8'h2;
16'h1804: RTMP = 8'h4;
16'h1805: RTMP = 8'h5;
16'h1840: RTMP = 8'h6;
16'h1841: RTMP = 8'h7;
16'hD808: RTMP = 8'h8;
16'hD80A: RTMP = 8'hA;
16'hD80B: RTMP = 8'hB;
16'hD820: RTMP = 8'h20;
16'hD821: RTMP = 8'h21;
16'hD822: RTMP = 8'h22;
16'hD823: RTMP = 8'h23;
16'hD830: RTMP = 8'h30;
16'hD831: RTMP = 8'h31;
16'hD832: RTMP = 8'h32;
```

```
16'hD833: RTMP = 8'h33;
16'hD860: RTMP = 8'h60;
16'hD861: RTMP = 8'h61;
16'hD862: RTMP = 8'h62;
16'hD863: RTMP = 8'h63;
16'hD870: RTMP = 8'h70;
16'hD871: RTMP = 8'h71;
16'hD872: RTMP = 8'h72;
16'hD873: RTMP = 8'h73;
endcase
#(envdrive.RTXC_DELAY) RTXC = 1'b1;
@(negedge CLK2) begin
RS = REGFUNC;
case (REGADDR)
16'h1800: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1801: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1802: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1804: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1805: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1840: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'h1841: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'hD808: RTMP = {REGDATA [1:0], REGDATA[7:2]};
16'hD80A: RTMP = {REGDATA [1:0], REGDATA[15], REGDATA[6:2]};
16'hD80B: RTMP = {REGDATA [1:0], REGDATA[15], REGDATA[6:2]};
16'hD820: RTMP = REGDATA [9:2];
16'hD821: RTMP = REGDATA [9:2];
16'hD822: RTMP = REGDATA [9:2];
16'hD823: RTMP = REGDATA [9:2];
16'hD830: RTMP = REGDATA [9:2];
16'hD831: RTMP = REGDATA [9:2];
16'hD832: RTMP = REGDATA [9:2];
16'hD833: RTMP = REGDATA [9:2];
16'hD860: RTMP = REGDATA [9:2];
16'hD861: RTMP = REGDATA [9:2];
16'hD862: RTMP = REGDATA [9:2];
16'hD863: RTMP = REGDATA [9:2];
16'hD870: RTMP = REGDATA [9:2];
16'hD871: RTMP = REGDATA [9:2];
16'hD872: RTMP = REGDATA [9:2];
16'hD873: RTMP = REGDATA [9:2];
endcase
#(envdrive.RTXC_DELAY) RTXC = 1'b1;
end
end
end

endtask

endmodule

// IMPORTED EXTERNAL FILE /usr/users/guru/thesis/verilog/barrel/barrel
'timescale 1 ps / 1 ps

/******************************** BARREL **********************************/

module barrel(PATTERN_BEG, PATTERN_END, DRIVE, STROBE, INDEX, CLK,
                    START_PATTERN_RUN, DONE);

 input [1:0] INDEX;
 output [5:0] STROBE;
 output [5:0] DRIVE;
 input [15:0] PATTERN_END;
 input [15:0] PATTERN_BEG;
 input CLK, START_PATTERN_RUN;
 output DONE;

reg [31:0] TIME_MEM [0:65535]; // cache for pattern times
reg [2:0] TYPE_MEM [0:65535]; // cache for pattern types
reg [15:0] MCOUNT; // pointer to memory addresses
reg [23:0] TCOUNT; // 24 msb counter for 3.2ns resolution
reg [31:0] ETIME; // current 32-bit time
reg [2:0] TYPE; // current type
reg [31:8] MSB; // 24 msbs of time
reg [7:0] VERNIER; // 8 lsbs of time
reg FIRE_TAG; // fires tag bit to formatters
reg [5:0] TMP; // temporary word register
reg DRIVE_MODE; // 1 for drive event, 0 for strobe
reg [5:0] DRIVE, STROBE; // outputs to ric and dic
reg DONE; // file has been finished
wire [31:0] DEBUG;

event END_RUN;

assign DEBUG = TIME_MEM[MCOUNT];

initial begin
 DRIVE = 6'b0;
 STROBE = 6'b0;
end

always @(posedge CLK) if (START_PATTERN_RUN===1'b1)
begin
 init_run;
end

always @(posedge CLK) TCOUNT = TCOUNT+1;

// incrementing mcount for pattern run mode
always @(posedge CLK) #3
begin
 if ((START_PATTERN_RUN===1'b0)&&(FIRE_TAG===1'b1)&&(MCOUNT<=PATTERN_END+4))
 begin
 while (TIME_MEM[MCOUNT+4]===32'hffffffff)
 begin
 MCOUNT = MCOUNT+4;
 end
 MCOUNT = MCOUNT +4;
 #1 if (TIME_MEM[MCOUNT]!==32'bx) ETIME = TIME_MEM[MCOUNT];
 if (TYPE_MEM[MCOUNT]!==3'bx) TYPE = TYPE_MEM[MCOUNT];
 #1 MSB = ETIME[31:8];
 VERNIER = ETIME[7:0];
 DRIVE_MODE = (TYPE[2]===0) ? 1'b1 : 1'b0;
 end
 else if (START_PATTERN_RUN===1'b1)
 begin
 while (TIME_MEM[MCOUNT]===32'hffffffff)
 begin
 MCOUNT = MCOUNT+4;
 end
 #1 if (TIME_MEM[MCOUNT]!==32'bx) ETIME = TIME_MEM[MCOUNT];
 if (TYPE_MEM[MCOUNT]!==3'bx) TYPE = TYPE_MEM[MCOUNT];
 #1 MSB = ETIME[31:8];
 VERNIER = ETIME[7:0];
 DRIVE_MODE = (TYPE[2]===0) ? 1'b1 : 1'b0;
 end
 else if ((MCOUNT>PATTERN_END+4))
 begin
 DRIVE = 6'b0;
 STROBE = 6'b0;
 DONE = 1'b1;
 end
end

always @(posedge CLK) #2 count;
always @(posedge CLK) #2 pattern_run;
always @(posedge CLK) #2 if (MCOUNT>PATTERN_END) ->END_RUN;

always @(END_RUN) @(posedge CLK)
begin
 DRIVE = 6'b0;
 STROBE = 6'b0;
 DONE = 1'b1;
end

always @(posedge FIRE_TAG)
begin
 #((envdrive.GLOBAL_CLK_PERIOD/2)+10) FIRE_TAG = 1'b0;
end

always @(posedge DRIVE[5]) #(envdrive.GLOBAL_CLK_PERIOD) DRIVE[5] = 1'b0;
always @(posedge STROBE[5]) #(envdrive.GLOBAL_CLK_PERIOD) STROBE[5] = 1'b0;

/*---------------------------- End of model ----------------------------*/

/********************************* task **********************************/

task init_run;
begin
 // loading of caches from files
 $readmemb("../memory/time.mem",
                TIME_MEM);
 $readmemb("../memory/type.mem",
                TYPE_MEM);
 case (INDEX)
 2'b00: MCOUNT [15:0] = PATTERN_BEG [15:0];
 2'b01: MCOUNT [15:0] = PATTERN_BEG [15:0] + 1;
 2'b10: MCOUNT [15:0] = PATTERN_BEG [15:0] + 2;
 2'b11: MCOUNT [15:0] = PATTERN_BEG [15:0] + 3;
 endcase

 FIRE_TAG = 1'b0;
 TMP [5] = 1'b0;
 DRIVE = 6'b0;
 STROBE = 6'b0;
 #0.1 DONE = 1'b0;
 TCOUNT = 24'b1;
end
endtask

task count;

begin
 if (START_PATTERN_RUN===1'b0)
 begin
 if (TCOUNT===MSB+2) @(negedge CLK)
 fork
 FIRE_TAG = 1'b1;
 join
 end
end

endtask
```

```
task pattern_run;

begin
 if (DONE ===1'b0)
 begin
 case (DRIVE_MODE)
 1'b1:if ((ETIME !== 32'bx)&&(ETIME!==32'hffffffff))
 begin
 DRIVE [3:0] = (FIRE_TAG===1'b1) ? VERNIER[3:0] : VERNIER[7:4];
 // drive mode: 000->00, 001->01, 010->10, 011->11
 // strobe mode: 100->00, 101->01, 110->10, 111->11
 case (TYPE)
 3'b000: DRIVE [4] = 0;
 3'b001: DRIVE [4] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 3'b010: DRIVE [4] = (FIRE_TAG===1'b1) ? 1'b0 : 1'b1;
 3'b011: DRIVE [4] = 1'b1;
 3'b100: DRIVE [4] = 1'b0;
 3'b101: DRIVE [4] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 3'b110: DRIVE [4] = (FIRE_TAG===1'b1) ? 1'b0 : 1'b1;
 3'b111: DRIVE [4] = 1'b1;
 endcase
 DRIVE [5] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 end
 1'b0:if ((ETIME !== 32'bx)&&(ETIME!==32'hffffffff))
 begin
 STROBE [3:0] = (FIRE_TAG===1'b1) ? VERNIER[3:0] : VERNIER[7:4];
 // STROBE mode: 000->00, 001->01, 010->10, 011->11
 // strobe mode: 100->00, 101->01, 110->10, 111->11
 case (TYPE)
 3'b000: STROBE [4] = 0;
 3'b001: STROBE [4] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 3'b010: STROBE [4] = (FIRE_TAG===1'b1) ? 1'b0 : 1'b1;
 3'b011: STROBE [4] = 1'b1;
 3'b100: STROBE [4] = 1'b0;
 3'b101: STROBE [4] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 3'b110: STROBE [4] = (FIRE_TAG===1'b1) ? 1'b0 : 1'b1;
 3'b111: STROBE [4] = 1'b1;
 endcase
 STROBE [5] = (FIRE_TAG===1'b1) ? 1'b1 : 1'b0;
 end
 endcase
 end
end
endtask

endmodule
```

Appendix E

Control Modules

E.1 Test Set Control Module

```
'include "/cadusr2/tg2/verilog/formatter/reg_chip_addr.def"

'timescale 1 ps / 1 ps

module envdrive (SEQUENCE, SEQUENCE_INDEX, START_RUN, GATE, FAULT, DONE, FAILTEST,
                 FAILINDEX, FAILCLOCK, EXPECTPINS, FAILPINS, XPINS);

 input [15:0] SEQUENCE, SEQUENCE_INDEX;
 input [4:0] FAULT;
 input [9:0] GATE;
 input START_RUN;
 output DONE;
 output [15:0] FAILTEST, FAILINDEX, FAILCLOCK;
 output [21:0] EXPECTPINS, FAILPINS, XPINS;

 reg[15:0] test_number, test_index, OLD_SEQUENCE_INDEX, clock_count;
 reg[15:0] FAILTEST, FAILINDEX, FAILCLOCK;
 reg [21:0] EXPECTPINS, FAILPINS, XPINS;
 reg DONE;

 reg clock, thbsel, reg_mode;
 reg [2:0] test_mode;
 reg [15:0] pattern_beg, reg_beg, reg_end, pattern_end, mux_beg, mux_end,
 dut_beg, dut_end;
 reg [4:0] fault_sel;
 reg [9:0] gate_sel;
 reg strobe_enable;
 wire reg_done, pattern_done, dut_done;

 system s1(pattern_beg, reg_beg, reg_end, pattern_end, mux_beg, mux_end,
 test_mode, dut_beg, dut_end, FAULT, GATE, thbsel, clock,
 DHIA, DINHA, DHIB, DINHB, setloout, sethiout,
 setzout, setonout, STFLA, STFLB, STFLC, STFLD, tmua, tmub, reg_mode,
 reg_done, pattern_done, dut_done, DHIA_G, DINHA_G, DHIB_G,
 DINHB_G, setloout_g, sethiout_g, setzout_g, setonout_g,
 STFLA_G, STFLB_G, STFLC_G, STFLD_G, tmua_g, tmub_g);

 parameter GLOBAL_CLK_PERIOD=10000;
 parameter CLKOUT_DELAY = 100;
 parameter REG_CLK_PERIOD = GLOBAL_CLK_PERIOD*6;
 parameter RTXC_DELAY = GLOBAL_CLK_PERIOD;
 parameter MUX_RESOLUTION = 10000;
 parameter MUX_PULSE_WIDTH = 10000;
 parameter MUX_DELAY = 3*GLOBAL_CLK_PERIOD+2850;
 parameter DUT_RESOLUTION = 10000;
 parameter DUT_DELAY = 3*GLOBAL_CLK_PERIOD+2850;
 parameter PATTERN_DELAY = 4*GLOBAL_CLK_PERIOD+2850;
 parameter GATE_LEAD = 2490;

 integer e_messages, e_broadcast, e_broadcast2, e_init_state, e_init_io, e_dictionary;

 initial begin
 e_messages = $fopen("errors.dat"); if (e_messages == 0) $finish;
 e_init_state = $fopen("init_state.dat"); if (e_init_state == 0) $finish;
 e_init_io = $fopen("init_io.dat"); if (e_init_io == 0) $finish;
 e_dictionary = $fopen("dictionary.dat"); if (e_dictionary == 0) $finish;
// stdout = $fopen("stdout.dat"); if (stdout == 0) $finish;
 e_broadcast = 1 | e_messages;
// e_broadcast2 = 1 | stdout;
 clock = 1'b0;
 init_test;
 $fstrobe ( e_init_io, " Seq G Flt Clk ||DHIA |DINHA|DHIB |DINHB|setlo|sethi|setz |seton|STFLA|STFLB|
                    STFLC|STFLD|tmua |tmub | R\n");
 $fstrobe ( e_init_state, " Seq G Flt Clk ||WFAIL|WFLA |WFLB |WFLC |WFLD |PINSTAT|EMOD7|
                    HSPATHA| HSPATHBSEL |TGICR|TGICF|TGIC6|TGIC4|TGIC3\n");
 $fstrobe ( e_dictionary, " Sequence Gate Fault Clock || Failing Test | Failing Index | Failing Pin\n");
 end

task init_test;
 begin
 clock_count = 16'b0;
 strobe_enable = 1'b0;
 FAILTEST = 16'b0;
 FAILPINS = 22'b0;
 XPINS = 22'b0;
 EXPECTPINS = 22'b0;
 test_number = 16'd0;
 test_index = 16'd1;
 thbsel = 1'b0;
 mux_beg = 16'd00;
 mux_end = 16'd00;
 dut_beg = 16'd00;
 dut_end = 16'd00;
 reg_mode = 1'b1;
 test_mode = 3'b001; //DUT-MUX-PAT
 reg_beg = 16'd00;
 reg_end = 16'd12;
 pattern_beg = 16'd00;
 pattern_end = 16'd11;
 @(posedge envdrive.pattern_done) #GLOBAL_CLK_PERIOD strobe_enable = 1'b1;
 end
endtask

task end_test;
 begin
 FAILTEST = test_number;
 FAILINDEX = test_index;
 FAILCLOCK = clock_count;
 #1 strobe_enable = 1'b0;
 if (test_number !== faultdrive.LAST_TEST) begin
 if ((test_mode[0]===0)&&(test_mode[1]==0)) @(posedge dut_done) begin
          #(GLOBAL_CLK_PERIOD) DONE = 1'b1;
 end
 else if ((test_mode[0]===1)||(test_mode[1]===1)) @(posedge pattern_done) #(GLOBAL_CLK_PERIOD)
DONE = 1'b1;
 end
 else if (test_number === faultdrive.LAST_TEST) @(negedge clock) DONE = 1'b1;
 end
endtask

task next_test;
 begin
 OLD_SEQUENCE_INDEX = SEQUENCE_INDEX;
 #(GLOBAL_CLK_PERIOD*3/2) if (SEQUENCE_INDEX===OLD_SEQUENCE_INDEX) reg_mode =
1'b1;
 end
endtask
```

```
task fault_strobe;
begin
 fork
          if (DHIA !== DHIA_G) begin
           if (DHIA_G ===1'bX) XPINS[21] = 1'b1;
           FAILPINS[21] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\n
                    Faulted gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail
                    on clock cycle: %d\nError at time %d: DHIA = %d \t DHIA_G = %d\n", SEQUENCE,
                    GATE, FAULT, test_number, test_index, clock_count, $time, DHIA, DHIA_G);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | DHIA = %d \t DHIA_G = %d", SEQUENCE,
                    GATE, FAULT, clock_count, test_number, test_index, DHIA, DHIA_G);
           end
          if (DINHA !== DINHA_G) begin
           if (DINHA_G ===1'bX) XPINS[20] = 1'b1;
           FAILPINS[20] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
                    gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
                    %d\nError at time %d: DINHA = %d \t DINHA_G = %d\n", SEQUENCE, GATE,
                    FAULT, test_number, test_index, clock_count, $time, DINHA, DINHA_G);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | DINHA = %d \t DINHA_G = %d",
                    SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, DINHA,
                     DINHA_G);
           end
          if (DHIB !== DHIB_G) begin
           if (DHIB_G ===1'bX) XPINS[19] = 1'b1;
           FAILPINS[19] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
                    gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
                    %d\nError at time %d: DHIB = %d \t DHIB_G = %d\n", SEQUENCE, GATE, FAULT,
                    test_number, test_index, clock_count, $time, DHIB, DHIB_G);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | DHIB = %d \t DHIB_G = %d", SEQUENCE,
                    GATE, FAULT, clock_count, test_number, test_index, DHIB, DHIB_G);
           end
          if (DINHB !== DINHB_G) begin
           if (DINHB_G ===1'bX) XPINS[18] = 1'b1;
           FAILPINS[18] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
                    gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
                    %d\nError at time %d: DINHB = %d \t DINHB_G = %d\n", SEQUENCE, GATE,
                    FAULT, test_number, test_index, clock_count, $time, DINHB, DINHB_G);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | DINHB = %d \t DINHB_G = %d",
                    SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, DINHB,
                     DINHB_G);
           end
          if (setloout !== setloout_g) begin
           if (setloout_g ===1'bX) XPINS[17] = 1'b1;
           FAILPINS[17] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
                    gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
                    %d\nError at time %d: setloout = %d \t setloout_g = %d\n", SEQUENCE, GATE, FAULT,
                    test_number, test_index, clock_count, $time, setloout, setloout_g);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | setloout = %d \t setloout_g = %d",
                    SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, setloout, setloout_g);
           end
          if (sethiout !== sethiout_g) begin
           if (sethiout_g ===1'bX) XPINS[16] = 1'b1;
           FAILPINS[16] = 1'b1;
           $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
                    gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
                    %d\nError at time %d: sethiout = %d \t sethiout_g = %d\n", SEQUENCE, GATE, FAULT,
                    test_number, test_index, clock_count, $time, sethiout, sethiout_g);
           $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | sethiout = %d \t sethiout_g = %d",
                    SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, sethiout, sethiout_g);
 end
if (setzout !== setzout_g) begin
 if (setzout_g ===1'bX) XPINS[15] = 1'b1;
 FAILPINS[15] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: setzout = %d \t setzout_g = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, setzout, setzout_g);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | setzout = %d \t setzout_g = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, setzout, setzout_g);
 end
if (setonout !== setonout_g) begin
 if (setonout_g ===1'bX) XPINS[14] = 1'b1;
 FAILPINS[14] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: setonout = %d \t setonout_g = %d\n", SEQUENCE, GATE,
          FAULT, test_number, test_index, clock_count, $time, setonout, setonout_g);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | setonout = %d \t setonout_g = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, setonout,
           setonout_g);
 end
if (STFLA !== STFLA_G) begin
 if (STFLA_G ===1'bX) XPINS[13] = 1'b1;
 FAILPINS[13] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: STFLA = %d \t STFLA_G = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, STFLA, STFLA_G);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | STFLA = %d \t STFLA_G = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, STFLA, STFLA_G);
 end
if (STFLB !== STFLB_G) begin
 if (STFLB_G ===1'bX) XPINS[12] = 1'b1;
 FAILPINS[12] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: STFLB = %d \t STFLB_G = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, STFLB, STFLB_G);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | STFLB = %d \t STFLB_G = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, STFLB, STFLB_G);
 end
if (STFLC !== STFLC_G) begin
 if (STFLC_G ===1'bX) XPINS[11] = 1'b1;
 FAILPINS[11] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nEFail on clock cycle:
          %d\nError at time %d: STFLC = %d \t STFLC_G = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, STFLC, STFLC_G);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | STFLC = %d \t STFLC_G = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, STFLC, STFLC_G);
 end
if (STFLD !== STFLD_G) begin
 if (STFLD_G ===1'bX) XPINS[10] = 1'b1;
 FAILPINS[10] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: STFLD = %d \t STFLD_G = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, STFLD, STFLD_G);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | STFLD = %d \t STFLD_G = %d",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, STFLD, STFLD_G);
 end
if (tmua !== tmua_g) begin
 if (tmua_g ===1'bX) XPINS[9] = 1'b1;
 FAILPINS[9] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: tmua = %d \t tmua_g = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, tmua, tmua_g);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | tmua = %d \t tmua_g = %d", SEQUENCE,
          GATE, FAULT, clock_count, test_number, test_index, tmua, tmua_g);
 end
if (tmub !== tmub_g) begin
 if (tmub_g ===1'bX) XPINS[8] = 1'b1;
 FAILPINS[8] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: tmub = %d \t tmub_g = %d\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, tmub, tmub_g);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | tmub = %d \t tmub_g = %d", SEQUENCE,
          GATE, FAULT, clock_count, test_number, test_index, tmub, tmub_g);
 end
if (s1.R !== s1.R_G) begin
 if (s1.R_G[9] ===1'bX) XPINS[7] = 1'b1;
 if (s1.R_G[8] ===1'bX) XPINS[6] = 1'b1;
 if (s1.R_G[7] ===1'bX) XPINS[5] = 1'b1;
 if (s1.R_G[6] ===1'bX) XPINS[4] = 1'b1;
 if (s1.R_G[5] ===1'bX) XPINS[3] = 1'b1;
 if (s1.R_G[4] ===1'bX) XPINS[2] = 1'b1;
 if (s1.R_G[3] ===1'bX) XPINS[1] = 1'b1;
 if (s1.R_G[2] ===1'bX) XPINS[0] = 1'b1;
 if (s1.R[9] !== s1.R_G[9]) FAILPINS[7] = 1'b1;
 if (s1.R[8] !== s1.R_G[8]) FAILPINS[6] = 1'b1;
 if (s1.R[7] !== s1.R_G[7]) FAILPINS[5] = 1'b1;
 if (s1.R[6] !== s1.R_G[6]) FAILPINS[4] = 1'b1;
 if (s1.R[5] !== s1.R_G[5]) FAILPINS[3] = 1'b1;
 if (s1.R[4] !== s1.R_G[4]) FAILPINS[2] = 1'b1;
 if (s1.R[3] !== s1.R_G[3]) FAILPINS[1] = 1'b1;
 if (s1.R[2] !== s1.R_G[2]) FAILPINS[0] = 1'b1;
 $fstrobe ( e_broadcast, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted
          gate: %d\nFault injected: %d\nFailing test: %d\nFailing index: %d\nFail on clock cycle:
          %d\nError at time %d: R[9:2] = %b \t R_G[9:2] = %b\n", SEQUENCE, GATE, FAULT,
          test_number, test_index, clock_count, $time, s1.R, s1.R_G);
 $fstrobe ( e_dictionary, " %d %d %d %d || %d | %d | R[9:2] = %b \t R_G[9:2] = %b",
          SEQUENCE, GATE, FAULT, clock_count, test_number, test_index, s1.R, s1.R_G);
 end
 EXPECTPINS[21] = DHIA;
 EXPECTPINS[20] = DINHA;
 EXPECTPINS[19] = DHIB;
 EXPECTPINS[18] = DINHB;
 EXPECTPINS[17] = setloout;
 EXPECTPINS[16] = sethiout;
 EXPECTPINS[15] = setzout;
 EXPECTPINS[14] = setonout;
 EXPECTPINS[13] = STFLA;
 EXPECTPINS[12] = STFLB;
 EXPECTPINS[11] = STFLC;
 EXPECTPINS[10] = STFLD;
 EXPECTPINS[9] = tmua;
 EXPECTPINS[8] = tmub;
 EXPECTPINS[7] = s1.R[9];
 EXPECTPINS[6] = s1.R[8];
 EXPECTPINS[5] = s1.R[7];
 EXPECTPINS[4] = s1.R[6];
 EXPECTPINS[3] = s1.R[5];
 EXPECTPINS[2] = s1.R[4];
 EXPECTPINS[1] = s1.R[3];
 EXPECTPINS[0] = s1.R[2];
```

```
join
#1 if (FAILPINS !==22'b0) begin
#1 end_test;
end
end
endtask

task strobe_init;
begin
$fstrobe ( e_init_io, "%d %d %d %d || %d %d | %d %d | %d %d | %d %d | %d %d | %d %d | %d %d | %d
                    %d | %d %d | %d %d | %d %d | %d %d | %d %d | %d %d | %b %b", SEQUENCE, GATE,
                    FAULT, clock_count, DHIA, DHIA_G, DINHA, DINHA_G, DHIB, DHIB_G, DINHB,
                    DINHB_G, setloout, setloout_g, sethiout, sethiout_g, setzout, setzout_g, setonout,
                    setonout_g, STFLA, STFLA_G, STFLB, STFLB_G, STFLC, STFLC_G, STFLD,
                    STFLD_G, tmua, tmua_g, tmub, tmub_g, s1.R, s1.R_G);
$fstrobe ( e_init_state, "%d %d %d %d || %b %b | %b %b | %b %b | %b %b | %b %b | %b %b | %b %b | %b
                    %b | %b %b | %b %b | %b %b | %b %b | %b %b | %b %b", SEQUENCE, GATE, FAULT,
                    clock_count, s1.U3.U2.strobe_logic.WFAIL, s1.U1.U1.strobe_logic_fault.WFAIL,
                    s1.U3.U2.strobe_logic.WFAIL_A, s1.U1.U1.strobe_logic_fault.WFAILA,
                    s1.U3.U2.strobe_logic.WFAIL_B, s1.U1.U1.strobe_logic_fault.WFAILB,
                    s1.U3.U2.strobe_logic.WFAIL_C, s1.U1.U1.strobe_logic_fault.WFAILC,
                    s1.U3.U2.strobe_logic.WFAIL_D, s1.U1.U1.strobe_logic_fault.WFAILD,
                    s1.U3.U2.PINSTATUS, s1.U1.U1.PINSTATUS, s1.U3.U2.EVENTMODE_bit7,
                    s1.U1.U1.EVENTMODE_bit7, s1.U3.U2.HSPATHASEL, s1.U1.U1.HSPATHASEL,
                    s1.U3.U2.HSPATHBSEL, s1.U1.U1.HSPATHBSEL, s1.U3.U2.TGICRESET,
                    s1.U1.U1.TGICRESET, s1.U3.U2.TGICDIAGM_bitF, s1.U1.U1.TGICDIAGM_bitF,
                    s1.U3.U2.TGICDIAGM_bit6, s1.U1.U1.TGICDIAGM_bit6, s1.U3.U2.
                    TGICDIAGM_bit4, s1.U1.U1.TGICDIAGM_bit4, s1.U3.U2.TGICDIAGM_bit3,
                    s1.U1.U1.TGICDIAGM_bit3);
end
endtask

always @(negedge START_RUN) begin
if (SEQUENCE===0) begin
if (SEQUENCE_INDEX!==1) init_test;
end
else if (SEQUENCE!==0) init_test;
end

always #(GLOBAL_CLK_PERIOD/2) clock = ~clock;
always @(posedge clock) clock_count = clock_count+1;
always @(posedge reg_done) reg_mode = 1'b0;
always @(posedge DONE) #(GLOBAL_CLK_PERIOD/2) DONE = 1'b0;
always @(posedge strobe_enable) #(GLOBAL_CLK_PERIOD/2) strobe_init;

always @(posedge clock) #(0.95*GLOBAL_CLK_PERIOD) if ((strobe_enable===1'b1)&&(faultdrive.-
FAIL_ENABLE===1'b1)) fault_strobe;

always @(test_number) begin
$strobe("Clock: %d\t Test: %d", clock_count, test_number);
end

always @(posedge pattern_done) begin
if ((test_number===0)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 16'd13;
reg_end = 16'd22;
pattern_beg = 16'd12;
pattern_end = 16'd12;
next_test;
#1 test_number = test_number +1;
end
end
```

```
always @(posedge pattern_done) begin
if ((test_number>=1)&&(test_number<=23)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:   pattern_end = pattern_beg+1; //2,24,13-14; 6,28,23-24; 10,32,33-34; 14,36,43-44;
                                       18,40,53-54; 22,44,63-64;
16'd2:   pattern_end = pattern_beg+2; //3,25,15-17; 7,29,25-27; 11,33,35-37; 15,37,45-47;
                                       19,41,55-57; 23,45,65-67;
16'd3:   pattern_end = pattern_beg+3; //4,26,18-21; 8,30,28-31; 12,34,38-41; 16,38,48-51; 20,42,58-61
16'd4:   pattern_end = pattern_beg; //5,27,22; 9.31,32; 13,35,42; 17,39,52; 21,43,62;
endcase
if (test_number===23) pattern_end = 75; // 24,46,68-75;
end
next_test;
#1 begin
test_number = test_number +1;
if ((test_index===0)||(test_index===4)) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if ((test_number===24)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 16'd46;
reg_end = 16'd344;
pattern_beg = 16'd76;
pattern_end = 16'd76;
next_test;
#1 test_number = test_number +1;
test_index=16'b1;
end

always @(posedge pattern_done) if ((test_number===25)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b011; //DUT-MUX-PAT
reg_beg = 16'd345;
reg_end = 16'd345;
pattern_beg = 16'd77;
pattern_end = 16'd115;
mux_beg = 16'd0;
mux_end = 16'd3;
next_test;
#1 test_number = test_number +1;
test_index=16'b1;
end

always @(posedge pattern_done) if ((test_number===26)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b011; //DUT-MUX-PAT
reg_beg = 16'd353;
reg_end = 16'd354;
mux_beg = 16'd4;
mux_end = 16'd7;
pattern_beg = 16'd144;
pattern_end = 16'd210;
next_test;
#1 test_number = test_number +1;
end
```

```
always @(posedge pattern_done) if ((test_number===27)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b011; //DUT-MUX-PAT
reg_beg = 16'd358;
reg_end = 16'd359;
mux_beg = 16'd8;
mux_end = 16'd9;
pattern_beg = 16'd213;
pattern_end = 16'd215;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===28)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b011; //DUT-MUX-PAT
reg_beg = 16'd360;
reg_end = 16'd360;
mux_beg = 16'd10;
mux_end = 16'd11;
pattern_beg = 16'd216;
pattern_end = 16'd219;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===29)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 16'd365;
reg_end = 16'd365;
pattern_beg = 16'd230;
pattern_end = 16'd237;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===30)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 366;
#1 reg_end = 384;
pattern_beg = 16'd238;
pattern_end = 16'd245;
next_test;
#1 test_number = test_number +1;
test_index = 16'd1;
end

always @(posedge pattern_done) begin
if ((test_number>=31)&&(test_number<=37)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:  pattern_end = pattern_beg+7; //32,385,246-253
16'd2:  pattern_end = pattern_beg+7; //33,386,254-261
16'd3:  pattern_end = pattern_beg+7; //34,387,262-269
16'd4:  pattern_end = pattern_beg+4; //35,388,270-274
16'd5:  pattern_end = pattern_beg+5; //36,389,275-280
16'd6:  pattern_end = pattern_beg+6; //37,390,281-287
```

```
16'd7:   pattern_end = pattern_beg+7; //38,391,288-295
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==7) test_index = 16'b1;
else test_index = test_index+1;
end
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = ~thbsel;
end
end

always @(posedge pattern_done) if ((test_number===38)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 396;
#1 reg_end = 398;
pattern_beg = 16'd330;
pattern_end = 16'd337;
next_test;
#1 test_number = test_number +1;
test_index = 16'd1;
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = 1'b0;
end

always @(posedge pattern_done) begin
if ((test_number>=39)&&(test_number<=44)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:   pattern_end = pattern_beg+7; //40,399,338-345
16'd2:   pattern_end = pattern_beg+7; //41,400,346-353
16'd3:   pattern_end = pattern_beg+4; //42,401,354-358
16'd4:   pattern_end = pattern_beg+5; //43,402,359-364
16'd5:   pattern_end = pattern_beg+6; //44,403,365-371
16'd6:   pattern_end = pattern_beg+7; //45,404,372-379
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==6) test_index = 16'b1;
else test_index = test_index+1;
end
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = ~thbsel;
end
end

always @(posedge pattern_done) if ((test_number===45)&&(test_number!==faultdrive.LAST_TEST))
begin
thbsel = 1'b0;
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 409;
reg_end = 410;
pattern_beg = 406;
pattern_end = 413;
next_test;
#1 test_number = test_number +1;
test_index = 16'd1;
end
```

```
always @(posedge pattern_done) begin
if ((test_number===46)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 411;
#1 reg_end = 411;
pattern_beg = 414; //47,411,414-417
#1 pattern_end = 417;
next_test;
#1 test_number = test_number +1;
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = 1'b1;
end
end

always @(posedge pattern_done) begin
if ((test_number>=47)&&(test_number<=49)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
case (test_number)
16'd75: begin
reg_beg = 418;
#1 reg_end = 418;
pattern_beg = 442; //48,418,442-445
pattern_end = 445;
end
16'd76: begin
reg_beg = 419;
#1 reg_end = 419;
pattern_beg = 446; //49,419,446-449
pattern_end = 449;
end
16'd77: begin
reg_beg = 420;
#1 reg_end = 420;
pattern_beg = 450; //50,420,450-453
pattern_end = 453;
end
endcase
next_test;
#1 test_number = test_number +1;
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = ~thbsel;
end
end

always @(posedge pattern_done) if ((test_number===50)&&(test_number!==faultdrive.LAST_TEST))
begin
thbsel = 1'b0;
test_mode = 3'b101; //DUT-MUX-PAT
reg_beg = 421;
reg_end = 427;
dut_beg = 0;
dut_end = 0;
pattern_beg = 454;
pattern_end = 458;
next_test;
#1 test_number = test_number +1;
@(posedge envdrive.s1.U2.START_PATTERN_RUN) thbsel = 1;
end

always @(posedge pattern_done) if ((test_number===51)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 428;
reg_end = 676;
dut_beg = 1;
```

```
dut_end = 1;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) if ((test_number===52)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 677;
reg_end = 677;
dut_beg = 2;
dut_end = 2;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) if ((test_number===53)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 678;
reg_end = 678;
dut_beg = 3;
dut_end = 3;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) if ((test_number===54)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 679;
reg_end = 679;
dut_beg = 4;
dut_end = 4;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) if ((test_number===55)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 680;
reg_end = 680;
dut_beg = 5;
dut_end = 5;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) if ((test_number===56)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 681;
reg_end = 681;
dut_beg = 6;
dut_end = 6;
next_test;
#1 test_number = test_number +1;
 @(posedge envdrive.s1.U2.START_MUX_RUN) thbsel = 1;
end

always @(posedge dut_done) begin
if ((test_number>=16'd52)&&(test_number<=16'd56)&&(test_number!==faultdrive.LAST_TEST)) begin
 reg_beg = 682;
```

```
#1 reg_end = 683;
pattern_beg = 459;
#1 pattern_end = 462;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) begin
if ((test_number>=58)&&(test_number<=60)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg+3;
pattern_beg = pattern_end+1; //59,684-687,463-466; 60,688-691,467-470; 61,692-695,471-474
#1 pattern_end = pattern_end+3;
next_test;
#1 test_number = test_number +1;
end
end

always @(posedge pattern_done) if ((test_number===61)&&(test_number!==faultdrive.LAST_TEST))
begin
thbsel = 0;
test_mode = 3'b101; //DUT-MUX-PAT
reg_beg = 696;
reg_end = 706;
dut_beg = 7;
dut_end = 7;
pattern_beg = 475;
#1 pattern_end = 475;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===62)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 707;
#1 reg_end = 707;
pattern_beg = 476;
pattern_end = 479;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===63)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b101; //DUT-MUX-PAT
reg_beg = 708;
#1 reg_end = 710;
dut_beg = 8;
dut_end = 16;
pattern_beg = 480;
pattern_end = 651;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===64)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b101; //DUT-MUX-PAT
reg_beg = 711;
#1 reg_end = 713;
dut_beg = 17;
dut_end = 19;
```

```
pattern_beg = 652;
pattern_end = 699;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) if ((test_number===65)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 714;
#1 reg_end = 717;
pattern_beg = 700;
pattern_end = 700;
next_test;
#1 test_number = test_number +1;
end

always @(posedge pattern_done) begin
if ((test_number>=66)&&(test_number<=68)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:  pattern_end = pattern_beg+1; //67,718,701-702
16'd2:  pattern_end = pattern_beg+2; //68,719,703-705
16'd3:  pattern_end = pattern_beg+3; //69,720,706-709
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==3) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if ((test_number===69)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 721;
#1 reg_end = 721;
dut_beg = 20;
#1 dut_end = 20;
next_test;
#1 test_number = test_number +1;
test_index = 1;
end

always @(posedge dut_done) begin
if ((test_number>=70)&&(test_number<=77)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
dut_beg = dut_end+1;
#1 dut_end = dut_beg+8;
#1 begin
case (test_index)
16'd1:  dut_end = dut_beg+7; //71,722, 21-28
16'd2:  thbsel = 1; //72,723, 29-37
16'd3:  begin
dut_end = dut_beg; //73,724, 38-38
```

```
thbsel = 0;
end
16'd4:  dut_end = dut_beg+7; //74,725, 39-46
16'd5:  thbsel = ~thbsel; //75,726, 47-55
16'd6:  thbsel = ~thbsel; //76,727, 56-64
16'd7:  thbsel = ~thbsel; //77,728, 65-73
16'd8:  begin
reg_beg = 738; //78,730,75
reg_end = 738;
dut_beg = 75;
dut_end = 75;
thbsel = 0;
end
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==8) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge dut_done) if ((test_number===78)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 739;
#1 reg_end = 739;
pattern_beg = 710;
#1 pattern_end = 710;
next_test;
#1 test_number = test_number +1;
test_index = 1;
end

always @(posedge pattern_done) begin
if ((test_number>=79)&&(test_number<=85)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:  pattern_end = pattern_beg; //80,740,711
16'd2:  pattern_end = pattern_beg+1; //81,741,712-713
16'd3:  pattern_end = pattern_beg+1; //82,742,714-715
16'd4:  pattern_end = pattern_beg+2; //83,743,716-718
16'd5:  pattern_end = pattern_beg+2; //84,744,719-721
16'd6:  pattern_end = pattern_beg+3; //85,745,722-725
16'd7:  pattern_end = pattern_beg+3; //86,746,726-729
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==7) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if ((test_number===86)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b100; //DUT-MUX-PAT
```

```
reg_beg = 755;
#1 reg_end = 756;
dut_beg = 76;
#1 dut_end = 76;
next_test;
#1 test_number = test_number +1;
end

always @(posedge dut_done) if ((test_number===87)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b101; //DUT-MUX-PAT
reg_beg = 771;
#1 reg_end = 773;
dut_beg = 77;
#1 dut_end = 77;
pattern_beg = 762;
#1 pattern_end = 762;
next_test;
#1 test_number = test_number +1;
test_index = 1;
end

always @(posedge pattern_done) begin
if ((test_number>=88)&&(test_number<=102)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:  pattern_end = pattern_beg; //89,774, 763
16'd2:  pattern_end = pattern_beg+1; //90,775, 764-765
16'd3:  pattern_end = pattern_beg+1; //91,776, 766-767
16'd4:  pattern_end = pattern_beg+2; //92,777, 768-770
16'd5:  pattern_end = pattern_beg+2; //93,778, 771-773
16'd6:  pattern_end = pattern_beg+3; //94,779, 774-777
16'd7:  pattern_end = pattern_beg+3; //95,780, 778-781
16'd8:  begin
pattern_beg = 814;
pattern_end = 818; //96,789, 814-818
end
16'd9:  pattern_end = pattern_beg+4; //97,790, 819-823
16'd10: pattern_end = pattern_beg+5; //98,791, 824-829
16'd11: pattern_end = pattern_beg+5; //99,792, 830-835
16'd12: pattern_end = pattern_beg+6; //100,793, 836-842
16'd13: pattern_end = pattern_beg+6; //101,794, 843-849
16'd14: pattern_end = pattern_beg+7; //102,795, 850-857
16'd15: pattern_end = pattern_beg+7; //103,796, 858-865
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==15) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if ((test_number===103)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 805;
#1 reg_end = 805;
dut_beg = 78;
```

```
#1 dut_end = 78;
next_test;
#1 test_number = test_number +1;
end

always @(posedge dut_done) if ((test_number===104)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 806;
#1 reg_end = 806;
pattern_beg = 898;
#1 pattern_end = 902;
next_test;
#1 test_number = test_number +1;
test_index = 1;
end

always @(posedge pattern_done) begin
if ((test_number>=105)&&(test_number<=111)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:   pattern_end = pattern_beg+4; //106, 807, 903-907
16'd2:   pattern_end = pattern_beg+5; //107, 808, 908-913
16'd3:   pattern_end = pattern_beg+5; //108, 809, 914-919
16'd4:   pattern_end = pattern_beg+6; //109, 810, 920-926
16'd5:   pattern_end = pattern_beg+6; //110, 811, 927-933
16'd6:   pattern_end = pattern_beg+7; //111, 812, 934-941
16'd7:   pattern_end = pattern_beg+7; //112, 813, 942-949
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==7) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if ((test_number===112)&&(test_number!==faultdrive.LAST_TEST))
begin
test_mode = 3'b100; //DUT-MUX-PAT
reg_beg = 822;
#1 reg_end = 822;
dut_beg = 79;
#1 dut_end = 79;
next_test;
#1 test_number = test_number +1;
end

always @(posedge dut_done) if ((test_number===113)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = 823;
#1 reg_end = 823;
pattern_beg = 982;
#1 pattern_end = 986;
next_test;
#1 test_number = test_number +1;
test_index = 1;
end
```

```
always @(posedge pattern_done) begin
if ((test_number>=114)&&(test_number<=120)&&(test_number!==faultdrive.LAST_TEST)) begin
test_mode = 3'b001; //DUT-MUX-PAT
reg_beg = reg_end+1;
#1 reg_end = reg_beg;
pattern_beg = pattern_end+1;
#1 begin
case (test_index)
16'd1:   pattern_end = pattern_beg+4; //115, 824, 987-991
16'd2:   pattern_end = pattern_beg+5; //116, 825, 992-997
16'd3:   pattern_end = pattern_beg+5; //117, 826, 998-1003
16'd4:   pattern_end = pattern_beg+6; //118, 827, 1004-1010
16'd5:   pattern_end = pattern_beg+6; //119, 828, 1011-1017
16'd6:   pattern_end = pattern_beg+7; //120, 829, 1018-1025
16'd7:   pattern_end = pattern_beg+7; //121, 830, 1026-1033
endcase
end
next_test;
#1 begin
test_number = test_number +1;
if (test_index==7) test_index = 16'b1;
else test_index = test_index+1;
end
end
end

always @(posedge pattern_done) if (test_number===faultdrive.LAST_TEST) begin
end_test;
end

endmodule
```

E.2 Fault Injection Control Module

```
'include "/usr/users/guru/thesis/formatter/dic/reg_chip_addr.def"

'timescale 1 ps / 1 ps
'define f faultdrive.end_simulation;.
'define u faultdrive.update_status;.
'define c $strobe(faultdrive.e1.clock_count);.

module faultdrive;

 reg [15:0] sequence, passes, fails, sequence_index, sequence_index_max;
 reg [4:0] fault;
 reg [9:0] gate;
 reg start_run;

 wire done;
 wire [15:0] fail_no, fail_index, fail_clock;
 wire [21:0] expect_pins, fail_pins, x_pins;

 integer messages, broadcast, summary, cnt;
 real percent;
 reg[8*34:1] summary_pins;

 parameter TESTCASE="fmt_tst_0";
 parameter START_GATE = 6;
 parameter END_GATE = 54;
 parameter START_FAULT = 1;
 parameter END_FAULT = 1;
 parameter FAULT_ENABLE = 1;
 parameter END_FAULT_ENABLE = 0;
 parameter FAIL_ENABLE = 1;
 parameter LAST_TEST = 121;

 envdrive e1 (sequence, sequence_index, start_run, gate, fault, done, fail_no, fail_index, fail_clock, expect_-
pins, fail_pins, x_pins);

 initial begin
 messages = $fopen("messages.dat"); if (messages == 0) $finish;
 summary = $fopen("summary.dat"); if (summary == 0) $finish;
 broadcast = 1 | messages;
 end

 initial begin
 fork
 if ($test$plusargs("dumpall")) $dumpvars;
 if ($test$plusargs("shortdumpall")) begin
 $dumpvars;
 wait (e1.test_number===26) $dumpoff;
 end
 if ($test$plusargs("profile")) $startprofile;
 sequence = 16'b0;
 fails = 16'b0;
 passes = 16'b0;
 start_run = 1'b0;
 init_sequence;
 $fstrobe ( broadcast, "***************************FAULT GRADING TEST*************
                  ************\n\nNo. of Sequences: %d\nGate Numbers: %d to %d\nFault Model:
                  Single Stuck-At Fault\n", END_GATE-START_GATE+1, gate, END_GATE);
 $fstrobe ( summary, " Sequence Gate No.Inputs Fault Clock || Pass/Fail || Failing test | Failing index |
                  Drive Setout STFL TMU R\n");
 join
 end
 always @(posedge done) increment;
 always @(posedge start_run) #7500 start_run = 1'b0;

task init_sequence;
 begin
 gate = START_GATE;
 if ((gate>=6)&&(gate<=516)) sequence_index = START_FAULT;
 if (FAULT_ENABLE===0) sequence_index = 0;
 #1 sequence_index_max = sequence_index;
 if ((gate>=6)&&(gate<=33)) sequence_index_max = 2;
 else if ((gate>=34)&&(gate<=145)) sequence_index_max = 4;
 else if ((gate>=146)&&(gate<=249)) sequence_index_max = 6;
 else if ((gate>=250)&&(gate<=427)) sequence_index_max = 8;
 else if ((gate>=428)&&(gate<=435)) sequence_index_max = 10;
 else if ((gate>=436)&&(gate<=501)) sequence_index_max = 12;
 else if ((gate>=502)&&(gate<=516)) sequence_index_max = 24;
 if ((gate===326)||(gate===329)||(gate===332)||(gate===335)) sequence_index_max = 4;
 if (END_FAULT_ENABLE===1) sequence_index_max = END_FAULT;
 fault = sequence_index;
 end
endtask

task increment;
 begin
 if (fail_no!==LAST_TEST) begin
 $strobe("expect: %b\nfails: %b\nundetermined_gate: %b", expect_pins, fail_pins, x_pins);
 for (cnt = 21; cnt>=0;cnt = cnt-1) begin
 summary_pins = (fail_pins[cnt]===1) ? ((x_pins[cnt]===1) ? ((expect_pins[cnt]===1) ? {summary_pins,
"x"}:{summary_pins, "X"}) :
 ((expect_pins[cnt]===1) ? {summary_pins, "L"}:{summary_pins, "H"})) :
 ((expect_pins[cnt]===1) ? {summary_pins, "1"}:{summary_pins, "0"}) ;
 if ((cnt===18)||(cnt===14)||(cnt===10)||(cnt===8)) summary_pins = {summary_pins, " | "};
 end
 $fstrobe ( messages, "\t\t\t<==========FAIL==========>\n\nTest Sequence: %d\nFaulted gate:
                     %d\nNumber of inputs tested (stuck_1&stuck_0): %d\nFault injected: %d\nFailing test:
                     %d\nFailing index: %d\nFail on clock cycle: %d\n", sequence, gate,
                     sequence_index_max, fault, fail_no, fail_index, fail_clock);
 Sfstrobe ( summary, " %d %d %d %d %d || fail || %d | %d | %s", sequence, gate, sequence_index_max,
                     fault, fail_clock, fail_no, fail_index, summary_pins);
 #1 fails = fails +1;
 end
 else if (fail_no===LAST_TEST) begin
 $fstrobe ( broadcast, "\t\t\t<==========PASS==========>\n\nTest Sequence: %d\nFaulted gate:
                     %d\nNumber of inputs tested (stuck_1&stuck_0): %d\nFault injected: %d\nClock cycle:
                     %d\n", sequence, gate, sequence_index_max, fault, fail_clock);
 $fstrobe ( summary, " %d %d %d %d %d || pass", sequence, gate, sequence_index_max, fault, fail_clock);
 #1 passes = passes +1;
 end
 if ((sequence_index >= sequence_index_max)||(FAULT_ENABLE===0)) begin
 #1 gate = gate+1;
 #1 sequence_index = 1;
 if ((gate>=6)&&(gate<=33)) sequence_index_max = 2;
 else if ((gate>=34)&&(gate<=145)) sequence_index_max = 4;
 else if ((gate>=146)&&(gate<=249)) sequence_index_max = 6;
 else if ((gate>=250)&&(gate<=427)) sequence_index_max = 8;
 else if ((gate>=428)&&(gate<=435)) sequence_index_max = 10;
 else if ((gate>=436)&&(gate<=501)) sequence_index_max = 12;
 else if ((gate>=502)&&(gate<=516)) sequence_index_max = 24;
 fault = sequence_index;
 #1 if (sequence < END_GATE-START_GATE) begin
 start_run = 1'b1;
 #1 sequence = sequence+1;
 end
 else if (sequence >= END_GATE-START_GATE) analysis;
 end
 else if (sequence_index < sequence_index_max) begin
 if (FAULT_ENABLE===1) sequence_index = sequence_index+1;
 #1 fault = sequence_index;
 start_run = 1'b1;
 end
 end
endtask

task analysis;
 begin
 percent = 100*fails/(passes+fails);
 $fstrobe ( broadcast, "\n\n********************************SUMMARY
                    ********************************\n\nNo. of Sequences: %d\nNo. of faults applied:
                    %d\nPasses: %d\nFails: %d\nPercent Coverage: %f\n\n\nSimulation Time: %d
                    picoseconds\nSimulation Cycles: %d\nEquivalent Emulation Time: %d nanoseconds",
                    sequence+1, passes+fails, passes, fails, percent, $time, $time/10000, $time/10000/5);
 $fstrobe ( summary, "\n\n********************************SUMMARY
                    ********************************\n\nNo. of Sequences: %d\nNo. of faults applied:
                    %d\nPasses: %d\nFails: %d\nPercent Coverage: %f\n\n\nSimulation Time: %d
                    picoseconds\nSimulation Cycles: %d\nEquivalent Emulation Time: %d nanoseconds",
                    sequence+1, passes+fails, passes, fails, percent, $time, $time/10000, $time/10000/5);
 #10000 $finish;
 end
endtask

task end_simulation;

 begin
 analysis;
 $finish;
 $strobe ("\n\n********************************UPDATE
                    ********************************\n\nNo. of Sequences: %d\nNo. of faults applied:
                    %d\nPasses: %d\nFails: %d\nPercent Coverage: %f\n\n\nSimulation Time: %d
                    picoseconds\nSimulation Cycles: %d\nEquivalent Emulation Time: %d nanoseconds",
                    sequence+1, passes+fails, passes, fails, percent, $time, $time/10000, $time/10000/5);
 end
endtask

task update_status;
 begin
 percent = 100*fails/(passes+fails);
 $strobe ("\n\n********************************UPDATE
                    ********************************\n\nNo. of Sequences: %d\nNo. of faults applied:
                    %d\nPasses: %d\nFails: %d\nPercent Coverage: %f\n\n\nSimulation Time: %d
                    picoseconds\nSimulation Cycles: %d\nEquivalent Emulation Time: %d nanoseconds",
                    sequence+1, passes+fails, passes, fails, percent, $time, $time/10000, $time/10000/5);
 end
endtask

'include "/home/nfs/uhi27/guru/thesis/asap/simul/fmt.dbg.gate"
endmodule
```

Appendix F

The Functional Test Set

F.1 Sample Event Stream Memory File - Pattern Events

```
d0@0            # initialisation
d0@0
d0@0
d0@0
x@6.4
x@6.4
x@6.4
x@6.4
x@12.8
x@12.8
x@12.8
x@12.8
d1@.2125        # **DIC PATTERNS** program delay DAC in: A-11 B-22 C-44 D-88
nop
d0@.425
nop
nop
dz@.850
nop
nop
nop
don@1.7
d1@.425         # program delay DAC in: A-22 B-44 C-88 D-0f
nop
d0@.850
nop
nop
dz@1.7
nop
nop
nop
don@.1875
d1@.850         # program delay DAC in: A-44 B-88 C-0f D-f0
nop
d0@1.7
nop
nop
dz@.1875
nop
nop
nop
don@3
d1@1.7          # program delay DAC in: A-88 B-0f C-f0 D-11
nop
d0@.1875
nop
nop
dz@3
nop
nop
nop
don@.2125
d1@.1875        # program delay DAC in: A-0f B-f0 C-11 D-22
nop
```

```
d0@3
nop
nop
dz@.2125
nop
nop
nop
don@.425
d1@3            # program delay DAC in: A-f0 B-11 C-22 D-44
nop
d0@.2125
nop
nop
dz@.425
nop
nop
nop
don@.850
d1@.2125        # program delay DAC in: A-11 B-22 C-44 D-88
d0@3.8375       # .2125+.425+3.2
dz@7.8875       # 3.8375+.850+3.2
don@12.7875     # 7.8875+1.7+3.2
d0@0            # Set TSTPULSE in order to initialise PMXDHI/PMXDINH
d1@16           # Check every LDL can produce all SET_OUT pulses
dz@19.2
don@22.4
d0@25.6
nop             #
d1@28.8
dz@32
don@35.2
d0@38.4         #
nop
d1@41.6
dz@44.8
don@48          #
d0@51.2
nop
d1@54.4
dz@57.6         #
don@60.8
d0@64
nop
dz@67.2         # Check D1 and D0 of each LDL can also set on driver
d1@70.4
dz@73.6
d0@76.8
nop             #
dz@80
d1@83.2
dz@86.4
d1@89.6         #
nop
dz@92.8
d1@96
dz@99.2         #
d0@102.4
nop
dz@105.6
d1@108.8#
dz@112
d0@115.2
don@3.2         # Reset TSTPULSE
d0@0.1
```

```
dz@0.2
d1@0
d1@0            # Set TSTPMXVOH
d0@0.1
dz@0.2
don@3.2
dz@0.2          # Reset PMM
d1@0
don@3.2
d0@0.1
dz@0.2          # Reset TSTPMXVOH
don@3.2
d1@0
d0@0.1
don@3.2         # Set TSTPULSE
d0@0.1
dz@0.2
d1@0
d1@0            # Set PMM
d0@0.1
dz@0.2
don@3.2
dz@0.2          # Reset TSTPULSE
d1@0
don@3.2
d0@0.1
d1@16           # Event Sequence Test(a) D1 D0 D1 D0 D1 ...(check DHI toggles)
d0@19.2
d1@22.4
d0@25.6
nop
d1@28.8
d0@32
d1@35.2
d0@38.4
nop
dz@41.6         # (b) DZ DON DZ DON ...(check if DINH toggles)
don@44.8
dz@48
don@51.2
nop
dz@54.4
don@57.6
dz@60.8
don@64
dz@67.2
d1@70.4         # (c) D1 DZ D0 DZ ...(check if D1/D0 will reset DINH)
dz@73.6
d0@76.8
dz@80
nop
d1@83.2
dz@86.4
d0@89.6
dz@92.8
nop
d1@96
dz@99.2
d0@102.4
dz@105.6
nop
d1@108.8
dz@112
d0@115.2
```

```
dz@118.4
nop
don@121.6        # (d) DON DZ D1 D0 D1 DZ
dz@124.8
d1@128
d0@131.2
d1@134.4
nop
nop
dz@137.6
d1@140.8         # (e) D1 D0 DZ DON ...
d0@144
dz@147.2
don@150.4
nop
d1@153.6
d0@156.8
dz@160
don@163.2
nop
d1@166.4
d0@169.6
dz@172.8
don@176
nop
d1@179.2
d0@182.4
dz@185.6
don@188.8
d1@0             # Set TMUPA/B to PMXDHI/PMXDINH
dz@0
dz@0             # check no event sequence or SET_IN pulses can set DHI/DINH
nop
d1@3.2
nop              # check no event sequence or SET_IN pulses can reset DHI/DINH
don@0
nop
d0@3.2
nop              # check no SETIN pulse or drive events can change DHI/DINH
nop
d1@0
nop              # check no SETIN pulse or drive events can change DHI/DINH
nop
nop
dz@3.2
d0@0             # check no SETIN pulse or drive events can change DHI/DINH
nop              # check no SETIN pulse or drive events can change DHI/DINH
don@3.2
nop              # check event sequence to change DHI/DINH
nop
don@0
dz@3.2
d0@6.4
d1@9.6
dz@12.8
d0@16
don@6.0125       # Check TMU Mux and minimum DHI/DINH pulse width - barrel A
d1@0
d0@2
dz@4.0125
d1@16
d0@18
dz@20.0125
don@22.0125
```

```
d1@0            # barrel B
don@2.8125
d0@2
dz@.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0            # barrel C
d0@2
don@2.8125
dz@.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0            # barrel D
d0@2
dz@.8125
don@2.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0            # Check TMUPA/B pulses with only its events - barrel A
nop
nop
nop
d0@6.4
nop             # barrel B
dz@0
nop
nop
nop
don@6.4
nop             # barrel C
nop
dz@0
nop
nop
nop
don@6.4
nop             # barrel D
nop
nop
d1@0
nop
nop
nop
d0@6.4
nop             # Check TMUPA/B output no pulses for other barrels' events - A
d1@0
d0@0.1
dz@0.2
nop
don@6.4
dz@0.2          # barrel B (THBSEL = 1)
nop
d1@0
d0@0.1
nop
nop
don@6.4
d0@0.1          # barrel C (THBSEL = 0)
```

```
dz@0.2
nop
d1@0
nop
nop
nop
don@6.4
d1@0              # barrel D (THBSEL = 1)
d0@0.1
dz@0.2
nop
don@6.4
don@2.8125        # Check TMUPA/B toggle for the selected DVI/O pulse - A
d1@0
d0@2
dz@.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0              # barrel B
don@2.8125
d0@2
dz@.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0              # barrel C
d0@2
don@2.8125
dz@.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0              # barrel D
d0@2
dz@.8125
don@2.8125
d1@9.6
d0@11.6
dz@13.6125
don@15.6125
d1@0              # Check TMUPA/B toggle with only its events - barrel A
nop
nop
nop
d0@6.4
nop               # barrel B
dz@0
nop
nop
nop
don@6.4
nop               # barrel C
nop
dz@0
nop
nop
nop
don@6.4
nop               # barrel D
nop
```

```
nop
d1@0
nop
nop
nop
d0@6.4
nop                # Check TMUPA/B not toggle for other barrels' events - A
d1@0
d0@0.1
dz@0.2
nop
don@6.4
dz@0.2             # barrel B (THBSEL = 1)
nop
d1@0
d0@0.1
nop
nop
don@6.4
d0@0.1             # barrel C (THBSEL = 0)
dz@0.2
nop
d1@0
nop
nop
nop
don@6.4
d1@0               # barrel D (THBSEL = 1)
d0@0.1
dz@0.2
nop
don@6.4
d1@0               # Set LPBK {DRSTAT,STRBZ} = 00 TMUPA/B select DHI/DINH
dz@3.2
don@6.4
d0@9.6
d1@28.8
dz@32
don@35.2
d0@38.4
d1@0               # {DRSTAT,STRBZ} = 01 TMUPA/B sel PMXDHI/PMXDINH(TSTPMXVOH = 0)
dz@3.2
don@6.4
d0@9.6
d1@0               # Set PMM (PMXDHI/PMXDINH changes as DHI/DINH changes)
dz@.8
don@1.6
d0@2.4
d1@0               # Reset TSTPULSE (PMXDHI/PMXDINH stay the same)
dz@.8
don@1.6
d0@2.4
d1@0               # Set TSTVOH (PMXDHI/PMXDINH stay the same)
dz@.8
don@1.6
d0@2.4
d1@0               # Reset TSTVOH (PMXDHI/PMXDINH stay the same)
dz@0.8
don@1.6
d0@2.4
d1@0               # Set TSTPULSE (PMXDHI/PMXDINH toggle)
dz@0.8
don@1.6
d0@2.4
```

```
d1@0            # Reset PMM (PMXDHI/PMXDINH stay the same)
dz@0.8
don@1.6
d0@2.4
d1@0            # TMUPA/B select DINH_/DINH_ (THBSEL = 0)
dz@3.2
don@6.4
d0@9.6
d1@0            # TMUPA/B select DHI/DHI_ (THBSEL = 1)
dz@3.2
don@6.4
d0@9.6
d1@0            # TMUPA/B select DHI/DINH (THBSEL = 0)
dz@3.2
don@6.4
d0@9.6
t0@3.1875       # **RIC ** # Initialize all delay registers by sending TX(FF)
t0@6.3875
t0@9.5875
t0@12.7875
t0@15.9875
t0@2.0625       # program delay DAC in: A-A5 B-5A C-96 D-69
t0@4.325
t0@8.275
t0@10.9125
t0@1.3125       # program delay DAC in: A-69 B-A5 C-5A D-96g
t0@5.2625
t0@7.525
t0@11.475
t0@1.875        # program delay DAC in: A-96 B-69 C-A5 D-5A
t0@4.5125
t0@8.4625
t0@10.725
t0@1.125        # program delay DAC in: A-5A B-96 C-69 D-A5
t0@5.075
t0@7.7125
t0@11.6625
x@0             # Master reset
t1@0            # Clear all fail buckets
t1@0
t1@0
t1@0
x@0             # enable window strobe and clear fail buckets
x@0
x@0
x@0
tz@9.6          # strobe events (short window strobe) - fails
tz@12.8
x@12.8
x@16
x@24.4
x@27.6
tz@22.4
tz@25.6
t1@35.2         # passes
t1@38.4
x@37.2
x@40.4
x@50
x@53.2
t1@48
t1@51.2
t0@60.8         # fails
t0@64
```

```
x@62.8
x@66
x@75.6
x@78.8
t0@73.6
t0@76.8
t1@86.4         # 6.4ns barrel reuse (8ns marker separation) - passes
nop
nop
nop
x@94.4
nop
nop
nop
t0@102.4        # fails
nop
nop
nop
x@110.4
t1@112          # passes
nop
nop
nop
x@120
nop
nop
nop
t0@128          # fails
nop
nop
nop
t0@136
t1@137.6        # passes
nop
nop
nop
x@145.6
nop
nop
nop
t0@153.6        # fails
nop
nop
nop
x@161.6
t1@163.2        # passes
nop
nop
nop
x@171.2
nop
nop
nop
t0@179.2        # fails
nop
nop
nop
x@187.2
t1@188.8        # (intermediate window strobe) - passes
x@192
t1@195.2
x@198.4
x@204.8
t1@201.6
```

```
x@211.2
t1@208
t0@214.4        # fails
x@217.6
t0@220.8
x@224
x@230.4
t0@227.2
x@236.8
t0@233.6
t1@240          # passes
x@243.2
t1@246.4
x@249.6
x@256
t1@252.8
x@262.4
t1@259.2
tz@265.6        # fails
x@268.8
tz@272
x@275.2
x@281.6
tz@278.4
x@288
tz@284.8
t1@291.2        # (long window strobes) - passes
t1@300.8
t1@310.4
x@297.6
x@307.2
x@316.8
x@326.4
t1@320
t0@329.6        # fails
t0@339.2
t0@348.8
x@336
x@345.6
x@355.2
x@364.8
t0@358.4
t1@368          # (really long window strobes) - passes
t1@380.8
t1@393.6
t1@406.4
x@377.6
x@390.4
x@403.2
x@416
tz@419.2        # fails
t1@432
t1@444.8
t1@457.6
x@428.8
x@441.6
x@454.4
x@467.2
tz@470.4        # passes
t1@483.2
t1@496
t1@508.8
x@480
x@492.8
```

```
x@505.6
x@518.4
t1@537.6          # window strobes catch excursions of compare data
nop
nop
nop
x@560
t1@572.8
nop
nop
nop
x@595.2
t1@608
nop
nop
nop
x@630.4
t1@643.2
nop
nop
nop
x@665.6
t1@3.2            # Input ONE from Test Station A
t1@6.4
t1@9.6
t1@12.8
t0@16
t0@19.2
t0@22.4
t0@25.6
t1@28.8
t1@32
t1@35.2
t1@38.4
tz@41.6
tz@44.8
tz@48
tz@51.2
t0@70.4           # Input ZERO from Test Station A
t0@73.6
t0@76.8
t0@80
t1@83.2
t1@86.4
t1@89.6
t1@92.8
t0@96
t0@99.2
t0@102.4
t0@105.6
tz@108.8
tz@112
tz@115.2
tz@118.4
tz@137.6          # Input Z from Test Station A
tz@140.8
tz@144
tz@147.2
t0@150.4
t0@153.6
t0@156.8
t0@160
tz@163.2
tz@166.4
```

```
tz@169.6
tz@172.8
t1@176
t1@179.2
t1@182.4
t1@185.6
x@0             # send TX to each barrel to clear WFAIL latches
nop             # send TX to each barrel to clear WFAIL latches
x@0
x@0             # send TX to each barrel to clear WFAIL latches
nop
x@0
nop             # send TX to each barrel to clear WFAIL latches
nop
nop
x@0
t1@1.6          # TMUA/TMUB select CVIA/CVIA
x@1.6           # TMUA/TMUB select CVIA/CVOA
nop             # TMUA/TMUB select CVIB/CVIB
t0@1.6
nop             # TMUA/TMUB select CVIB/CVOB
x@1.6
nop             # TMUA/TMUB select CVIC/CVIC
nop
tz@1.6
nop             # TMUA/TMUB select CVIC/CVOC
nop
x@1.6
nop             # TMUA/TMUB select CVID/CVID
nop
nop
x@1.6
nop             # TMUA/TMUB select CVID/CVOD
nop
nop
x@1.6
nop             # TMUA/TMUB select CVIA/CVIA
t1@1.6
t0@4.8
tz@8
nop             # TMUA/TMUB select CVIA/CVOA
t1@1.6
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIB/CVIB
nop
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIB/CVOB
nop
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIC/CVIC
t0@4.8
nop
tz@8
t1@1.6          # TMUA/TMUB select CVIC/CVOC
t0@4.8
nop
tz@8
t1@1.6          # TMUA/TMUB select CVID/CVID
t0@4.8
tz@8
nop
```

```
t1@1.6          # TMUA/TMUB select CVID/CVOD
t0@4.8
tz@8
nop
t1@1.6          # TMUA/TMUB select CVIA/CVIA
x@1.6           # TMUA/TMUB select CVIA/CVOA
nop             # TMUA/TMUB select CVIB/CVIB
x@1.6
nop             # TMUA/TMUB select CVIB/CVOB
x@1.6
nop             # TMUA/TMUB select CVIC/CVIC
nop
t0@1.6
nop             # TMUA/TMUB select CVIC/CVOC
nop
x@1.6
nop             # TMUA/TMUB select CVID/CVID
nop
nop
tz@1.6
nop             # TMUA/TMUB select CVID/CVOD
nop
nop
x@1.6
nop             # TMUA/TMUB select CVIA/CVIA
t1@1.6
t0@1.6
tz@1.6
nop             # TMUA/TMUB select CVIA/CVOA
t1@1.6
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIB/CVIB
nop
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIB/CVOB
nop
t0@4.8
tz@8
t1@1.6          # TMUA/TMUB select CVIC/CVIC
t0@4.8
nop
tz@8
t1@1.6          # TMUA/TMUB select CVIC/CVOC
t0@4.8
nop
tz@8
t1@1.6          # TMUA/TMUB select CVID/CVID
t0@4.8
tz@8
nop
t1@1.6          # TMUA/TMUB select CVID/CVOD
t0@4.8
tz@8
nop
t1@1.6          # TMUA/TMUB select CVIA/CVIA
nop
nop
nop
t1@9.5875
t1@1.6          # TMUA/TMUB select CVIA/CVOA
nop
nop
```

```
nop
t1@9.5875
nop                # TMUA/TMUB select CVIB/CVIB
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIB/CVOB
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIC/CVIC
nop
t0@1.6
nop
nop
nop
t0@9.5875
nop                # TMUA/TMUB select CVIC/CVOC
nop
t0@1.6
nop
nop
nop
t0@9.5875
nop                # TMUA/TMUB select CVID/CVID
nop
nop
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVID/CVOD
nop
nop
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIA/CVIA
t1@0
t0@3.2
tz@6.4
nop                # TMUA/TMUB select CVIA/CVOA
t1@0
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVIB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVOB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIC/CVIC
t0@3.2
nop
tz@6.4
```

```
t1@0            # TMUA/TMUB select CVIC/CVOC
t0@3.2
nop
tz@6.4
t1@0            # TMUA/TMUB select CVID/CVID
t0@3.2
tz@6.4
nop
t1@0            # TMUA/TMUB select CVID/CVOD
t0@3.2
tz@6.4
nop
t1@1.6          # TMUA/TMUB select CVIA/CVIA
nop
nop
nop
t1@9.5875
t1@1.6          # TMUA/TMUB select CVIA/CVOA
nop
nop
nop
t1@9.5875
nop             # TMUA/TMUB select CVIB/CVIB
x@1.6
nop
nop
nop
x@9.5875
nop             # TMUA/TMUB select CVIB/CVOB
x@1.6
nop
nop
nop
x@9.5875
nop             # TMUA/TMUB select CVIC/CVIC
nop
t0@1.6
nop
nop
nop
t0@9.5875
nop             # TMUA/TMUB select CVIC/CVOC
nop
t0@1.6
nop
nop
nop
t0@9.5875
nop             # TMUA/TMUB select CVID/CVID
nop
nop
x@1.6
nop
nop
nop
x@9.5875
nop             # TMUA/TMUB select CVID/CVOD
nop
nop
x@1.6
nop
nop
nop
x@9.5875
```

```
nop                # TMUA/TMUB select CVIA/CVIA
t1@0
t0@3.2
tz@6.4
nop                # TMUA/TMUB select CVIA/CVOA
t1@0
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVIB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVOB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIC/CVIC
t0@3.2
nop
tz@6.4
t1@0               # TMUA/TMUB select CVIC/CVOC
t0@3.2
nop
tz@6.4
t1@0               # TMUA/TMUB select CVID/CVID
t0@3.2
tz@6.4
nop
t1@0               # TMUA/TMUB select CVID/CVOD
t0@3.2
tz@6.4
nop
t1@1.6             # TMUA/TMUB select CVIA/CVIA
nop
nop
nop
t1@9.5875
t1@1.6             # TMUA/TMUB select CVIA/CVOA
nop
nop
nop
t1@9.5875
nop                # TMUA/TMUB select CVIB/CVIB
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIB/CVOB
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIC/CVIC
nop
t0@1.6
nop
nop
nop
t0@9.5875
nop                # TMUA/TMUB select CVIC/CVOC
nop
t0@1.6
```

```
nop
nop
nop
t0@9.5875
nop                # TMUA/TMUB select CVID/CVID
nop
nop
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVID/CVOD
nop
nop
x@1.6
nop
nop
nop
x@9.5875
nop                # TMUA/TMUB select CVIA/CVIA
t1@0
t0@3.2
tz@6.4
nop                # TMUA/TMUB select CVIA/CVOA
t1@0
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVIB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIB/CVOB
nop
t0@3.2
tz@6.4
t1@0               # TMUA/TMUB select CVIC/CVIC
t0@3.2
nop
tz@6.4
t1@0               # TMUA/TMUB select CVIC/CVOC
t0@3.2
nop
tz@6.4
t1@0               # TMUA/TMUB select CVID/CVID
t0@3.2
tz@6.4
nop
t1@0               # TMUA/TMUB select CVID/CVOD
t0@3.2
tz@6.4
nop
```

Appendix G

Utility Programs

G.1 Event Stream Parser

```
/*
 *
 */
#include <stdio.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0

main(argc, argv)
  int argc;
  char **argv;
{
  char linebuf[1024];
  char *gets();
  char *fgets();
  char dorc, dorc2, state;
  float time, res;
  float lsb, lsblast;
  char *bin;
  char type [3];
  char duttype [2];
  char *dec_to_bin();
  char *all_ones();
  FILE *ftime, *ftype, *fopen();
  int MODE, BIT, FILEON, FILEARG, lcount;

  lcount = 1;
  MODE = 0;
  BIT = 32;
  FILEON = 0;
  FILEARG = 0;

  if (argc>1){
  switch (argv[1][1]){
  case 'm': {
/*    (void)printf("got to argv[1][1]= m\n");*/
    if (argv[2][0]=='m') {
      MODE = 1;
      BIT = 36;
    }
    if (argv[2][0]=='d') {
      MODE = 2;
      BIT = 36;
    }
```

```
      break;}
    case 'b': {
/*      (void)printf("got to argv[1][1]= b\n"); */
      BIT = atoi(argv[2]);
      break;}
    case 'f': {
/*      (void)printf("got to argv[1][1]= f\n");*/
      FILEON = 1;
      FILEARG = 2;
      break;}}
    switch (argv[1][2]){
    case 'm': {
/*      (void)printf("got to argv[1][2]= m\n");*/
      if (FILEARG!=0){
        if (argv[FILEARG+2][0]=='m') {
        MODE = 1;
        BIT = 36;
        }
        if (argv[FILEARG+2][0]=='d') {
        MODE = 2;
        BIT = 36;
        }}
      else {
        if (argv[3][0]=='m') MODE = 1;
        if (argv[3][0]=='d') MODE = 2;}
      break;}
    case 'b': {
/*      (void)printf("got to argv[1][2]= b\n");*/
      if (FILEARG!=0){
        BIT = atoi (argv[FILEARG+2]);}
      else BIT = atoi (argv[3]);
      break;}
    case 'f': {
/*      (void)printf("got to argv[1][2]= f\n"); */
      FILEON = 1;
      FILEARG = 3;
      break;}}
    switch (argv[1][3]){
    case 'm': {
      /*    (void)printf("got to argv[1][3]= m\n");*/
      if (argv[FILEARG+2][0]=='m') MODE = 1;
      if (argv[FILEARG+2][0]=='d') MODE = 2;
      break;}
    case 'b': {
/*      (void)printf("got to argv[1][3]= b\n");*/
      BIT = atoi (argv[FILEARG+2]);
      break;}
    case 'f': {
/*      (void)printf("got to argv[1][3]= f\n");*/
      FILEON = 1;
      FILEARG = 4;
      break;}}
}
```

```
/
*  (void)printf("m: %i\tb: %i\tf: %i,%i\n", MODE, BIT, FILEON, FILEARG);
*/

    if (FILEON == 1)
    {
      if ((ftime = fopen(argv[FILEARG], "w")) == NULL)
      fprintf (stderr, "Cannot open %s\n", argv[1]);
      if ((ftype = fopen(argv[FILEARG+1], "w")) == NULL)
      fprintf (stderr, "Cannot open %s\n", argv[2]);
    }

  lsblast = 0;
  if ((MODE == 1)||(MODE == 2)){
    if (gets(linebuf) == NULL)fprintf (stderr, "%i: %s: invalid event de
scription format\n", lcount,linebuf);
    if (strlen(linebuf) == 0)fprintf (stderr, "%i: %s: invalid event des
cription format\n", lcount,linebuf);
    if (sscanf (linebuf, "resolution=%f", &res) == 1) {
      if (res >= 1) (void)printf("resolution = %fps\n", res);
      else fprintf (stderr, "%i: %s: invalid event description format\n"
, lcount,linebuf);
}
    else fprintf (stderr, "%i: %s: invalid event description format\n",
lcount,linebuf);}
  for (;;)
    {
      if (gets(linebuf) == NULL) break;
      if (strlen(linebuf) == 0) break;
      if (sscanf (linebuf, "%c@%f", &state, &time) == 2) {
      if ((state == 'X')||(state == 'x')) {
        if (MODE == 0){
          lsb = (time/0.0125);

    if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
          lsblast = lsb;
          bin = dec_to_bin(lsb,BIT);}
        if (MODE == 1){
          lsb = ((time/res)*1000);
/*        (void)printf("lsb = %f\n", lsb);*/

    if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
          lsblast = lsb;
          bin = dec_to_bin(lsb,BIT);}
        type[0] = '1';
        type[1] = '0';
        type[2] = '0';}

else fprintf (stderr, "%i: %s: invalid event description format\n", lcou
nt,linebuf);}
      else if (sscanf (linebuf, "%c%c%c@%f",&dorc,&dorc2,&state, &time)=
=4) {
```

```
if (((dorc == 'D')||(dorc == 'd'))&&((dorc2 == 'O')||(dorc2 == 'o'))&&((
state == 'N')||(state == 'n'))) {
        if (MODE == 0){
          lsb = (time/0.0125);

    if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
          lsblast = lsb;
          bin = dec_to_bin(lsb,BIT);}
        if (MODE == 1){
          lsb = ((time/res)*1000);
/*        (void)printf("lsb = %f\n", lsb);*/

    if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
          lsblast = lsb;
          bin = dec_to_bin(lsb,BIT);}
/*      (void)printf ("drive on\n");*/
        type[0] = '0';
        type[1] = '0';
        type[2] = '1';}

else fprintf (stderr, "%i: %s: invalid event description format\n", lcou
nt,linebuf);}
      else if (sscanf (linebuf, "%c%c@%f", &dorc, &state, &time) == 3) {
/*      (void)printf ("time: %f\n", time);*/
      if (MODE == 0){
        lsb = (time/0.0125);

  if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
        lsblast = lsb;
        bin = dec_to_bin(lsb,BIT);
      }
      if ((MODE == 1)||(MODE == 2)){
        lsb = ((time/res)*1000);
/*      (void)printf("lsb = %f\n", lsb);*/

  if (lsblast > lsb) printf ("%i: warning: event times are not well-
ordered\n", lcount);
        lsblast = lsb;
        bin = dec_to_bin(lsb,BIT);
      }
      if (MODE == 2){
        if (dorc == '0') duttype[0] = '0';
        else if (dorc == '1') duttype[0] = '1';
        if (state == '0') duttype[1] = '0';
        else if (state == '1') duttype[1] = '1';

  else fprintf (stderr, "%i: %s: invalid event description format\n", lc
ount,linebuf);}
      if ((MODE == 0)||(MODE == 1)){
        if ((dorc == 'D')||(dorc == 'd'))
```

```
          type[0] = '0';

  else if ((dorc == 'S')||(dorc == 's')||(dorc == 'T')||(dorc == 't'))
          type[0] = '1';

  else fprintf (stderr, "%i: %s: invalid event description format\n", lc
ount,linebuf);
        if (state == '0'){
          type[1] = '1';
          type[2] = '0';
        }
        else if (state == '1'){
          type[1] = '1';
          type[2] = '1';
        }
        else if ((state == 'Z')||(state == 'z')){
          type[1] = '0';
          if ((dorc == 'D')||(dorc == 'd')) {
/*            (void)printf ("drive z\n");*/
            type[2] = '0';}
          else if ((dorc != 'D')&&(dorc != 'd')) type[2] = '1';
        }
        else if ((state == 'X')||(state == 'x')){
          type[1] = '0';
          if ((dorc == 'D')||(dorc == 'd')) {
/*            (void)printf ("drive x\n");*/
            type[2] = '1';}
          else if ((dorc != 'D')&&(dorc != 'd')) type[2] = '0';
        }

  else fprintf (stderr, "%i: %s: invalid event description format\n", lc
ount,linebuf);}
      }
      else if (sscanf (linebuf, "%c%c%c", &dorc, &dorc2, &state) == 3) {

if (((dorc=='N')||(dorc=='n'))&&((dorc2=='O')||(dorc2=='o'))&&((state=='
P')||(state=='p'))){
        bin = all_ones(BIT);
        type[0] = '0';
        type[1] = '0';
        type[2] = '0';}

else fprintf (stderr, "%i: %s: invalid event description format\n", lcou
nt,linebuf);}
      else fprintf (stderr, "%i: %s: 0 invalid event description format\
n", lcount,linebuf);
      {
      if (FILEON == 1){
        fprintf (ftime, "%s\n", bin);
        if (MODE == 2) fprintf (ftype, "%s\n", duttype);
        else fprintf (ftype, "%s\n", type);
      }
      else {
        if (MODE == 2) (void)printf ("%s\t%s\n", bin, duttype);
        else (void)printf ("%s\t%s\n", bin, type);}
      }
      lcount = lcount+1;
    }
}


char *dec_to_bin(input, length)
  int length;
  float input;
{
  float divisor;
  int count;
  float power();
  char *output = (char *) malloc(sizeof(char)*(length+1));

  count = length;

  while (count > 0)
    {
      divisor = power(2, count-1);
/*       (void)printf ("dec: %f, divisor: %f  ", input, divisor); */
      if (input >= divisor)
      {
        input = input - divisor;
        output [length - count] = '1';
/*       (void)printf ("bit %d set to %c\n", (length-count), '1'); */
      }
      else {output [length - count] = '0';}
/*       (void)printf ("bit %d set to %c\n", (length-count), '0'); */
      count = count - 1;
    }
/*  (void)printf ("last dec: %f, divisor: %f\n", input, divisor); */

  if (count >= length) fprintf(stderr, "error in dec_to_bin function");
/*  (void)printf ("bin: %s\n", output); */
  return output;
}

char *all_ones(length)
  int length;
{
  char *output = (char *) malloc(sizeof(char)*(length+1));

  while (length > 0)
    {
      output [length-1] = '1';
      length = length - 1;
    }
  return output;
}

float power(base, n)
     int base, n;
```

```
{
  float p;

  if (n==0) return(1);
  for (p = 1; n > 0; --n)
    p = p* base;
  return p;
}
```

G.2 S9000 Test Program Generator

```
/*****************************************************************************/
/* verilog file for formatters characterization module test vectors generation */
/* works with Guru's environment */
/* Generates 4 .m9k vector files */
/* TESTCASE_0.m9k : 100Mhz test (1 clock per cycle)*/
/* TESTCASE_1.m9k, --                                    */
/* TESTCASE_2.m9k, |-> 300Mhz tests (3 clocks per cycle)*/
/* TESTCASE_3.m9k --                                     */
/*                                                              */
/* author :Didier Wimmers                                */
/* rev :  0.1                                                   */
/* creation :08-04-93                                    */
/* date : 08-20-93                                        */
/*****************************************************************************/

/******************************/
/* Pin list order in Pindef : */
/******************************/
/*****************
 clk          =
 clk_    =
 clkcal  =
 clkcal_ =
 cg0          =
 cg1          =
 cg2          =
 cg3          =
 tclk    =
 tclk_   =
 tclkcal =
 tclkcal_=
 tcg0    =
 tcg1    =
 tcg2    =
 tcg3    =

 dclk =
 dclk_ =

 rbus    =
 rs      =
 rtxc    =
 th2sel  =

 da      =
 db =
 dc =
 dd =
 ca =
 cb =
 cc =
 cd =

 achia   =
 achia_  =
 bcloa   =
 bcloa_  =
 achib   =
 achib_  =
 bclob   =
 bclob_  =
z_in     =
z_in_    =
on_in    =
on_in_   =
hi_in    =
hi_in_   =
lo_in    =
lo_in_   =

dhia     =
dhia_    =
dinha    =
dinha_   =
dhib     =
dhib_    =
dinhb    =
dinhb_   =

hspa     =
hspa_    =
hspb     =
hspb_    =

on_out   =
on_out_=
z_out    =
z_out_   =
lo_out   =
lo_out_ =
hi_out   =
hi_out_ =

stfla =
stflb =
stflc =
stfld =

ptemp =
ntemp =
tempad  =
tempbd =
tempar  =
tempbr  =

vbbd     =
vbbr     =
*********************/
'define FMTIO envdrive.s1.U2
wire [8*1:1] RBUS_CYCLE = (('FMTIO.R===8'hxx) ? "X" :(('FMTIO.RS[1]==0 && 'FMTIO.RS[0]==0
&& 'FMTIO.RTXC==1) ? "R" : "W"));

integer file_stat[0:3];
reg [3:0] f_no;
parameter size=1;

initial
begin
 file_stat[0] = $fopen({TESTCASE,"_0.m9k"}); if (file_stat[0] == 0) $finish;
end

initial
begin
 for(f_no=0; f_no<=size; f_no=f_no+1)
 begin
 $fdisplay(file_stat[f_no], "#include <fmt.pindef.h>\n");
 $fdisplay(file_stat[f_no], "#pattern %s_%h", TESTCASE, f_no);
 $fdisplay(file_stat[f_no], "PINDEF_TABLE = fmt_pindef");
 end

 $fdisplay(file_stat[0], "TIMING = tim200\n");

 for(f_no=1; f_no<=size; f_no=f_no+1)
 begin
 $fdisplay(file_stat[f_no], "TIMING = tim_fast1\n");
 end


 for(f_no=0; f_no<=0; f_no=f_no+1)
 begin
 $fdisplay(file_stat[f_no], "#include <fmt.vecdef.h>");
 $fdisplay(file_stat[f_no], "\n");
 $fdisplay(file_stat[f_no], ";Pindef Type : I=input O=output B=input/output");
 $fdisplay(file_stat[f_no], "; I I O O I I I I I I O O I I I I O O BBB I I I I I I I I I I I I I I I I I I I I I I I I I I O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O I I");
 $fdisplay(file_stat[f_no], "; ------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------");
 $fdisplay(file_stat[f_no], "; C C C C C C C C T T T T T T T T D D RRR R R T DDDDDD DDDDDD
DDDDDD DDDDDD CCCCCC CCCCCC CCCCCC CCCCCC A A B B A A B B Z Z O O H H L L H H I
I H H I I H H H H O O Z Z L L H H S S S S P N T T T T V V");
 $fdisplay(file_stat[f_no], "; L L K K G G G G L L K K C C C C C C BBB S T H AAAAAA BBBBBB
CCCCCC DDDDDD AAAAAA BBBBBB CCCCCC DDDDDD C C C C C C C C I I N N I I O O I I N N I
I N H S P S P N N O O O O I I T T T T T T P P P P B B");
 $fdisplay(file_stat[f_no], "; K K L L 0 1 2 3 K K L L G G G G L L UUU X 2 543210 543210 543210
543210 543210 543210 543210 543210 H H L L H H L L N N I I I I I I A A H A B B H B P A P B O O U
T O O O O F F F F M M A B A D B B");
 $fdisplay(file_stat[f_no], "; - - - - 0 1 2 3 - SSS C A - A - B - B - - N - N - N - - A - - B - A - B - T - T - T -
T - A B C D P P D D R R D R");
 $fdisplay(file_stat[f_no], "%s%h_start;",TESTCASE, f_no);
 end

end

/**********************************************************/
 event firstpart_captured;

 reg [31:0] count;
 reg clk;
 reg clk_;
 reg clkcal, clkcal1, clkcal2, clkcal3;
 reg clkcal_, clkcal_1, clkcal_2, clkcal_3;
 reg cg0;
 reg cg1;
 reg cg2;
 reg cg3;
 reg            tclk;
 reg            tclk_;
 reg tclkcal, tclkcal1, tclkcal2, tclkcal3;
 reg tclkcal_, tclkcal_1, tclkcal_2, tclkcal_3;
 reg tcg0;
 reg tcg1;
 reg tcg2;
 reg tcg3;
 reg             dclk, dclk1, dclk2, dclk3;
 reg             dclk_, dclk_1, dclk_2, dclk_3;

 reg [7:0]rbus, rbus1, rbus2, rbus3;
 reg[8*1:1]rbus_cycle, rbus_cycle1, rbus_cycle2, rbus_cycle3;
```

```
reg [1:0]rs, rs1, rs2, rs3;
reg                    rtxc, rtxc1, rtxc2, rtxc3;
reg th2sel, th2sel1, th2sel2, th2sel3;

reg [5:0]da, da1, da2, da3; //to be changed (Binary?)
reg [5:0]db, db1, db2, db3;
reg [5:0]dc, dc1, dc2, dc3;
reg [5:0]dd, dd1, dd2, dd3;
reg [5:0]ca, ca1, ca2, ca3;
reg [5:0]cb, cb1, cb2, cb3;
reg [5:0]cc, cc1, cc2, cc3;
reg [5:0]cd, cd1, cd2, cd3;

reg achia, achia1, achia2, achia3;
reg achia_, achia_1, achia_2, achia_3;
reg bcloa, bcloa1, bcloa2, bcloa3;
reg bcloa_, bcloa_1, bcloa_2, bcloa_3;
reg achib, achib1, achib2, achib3;
reg achib_, achib_1, achib_2, achib_3;
reg bclob, bclob1, bclob2, bclob3;
reg bclob_, bclob_1, bclob_2, bclob_3;

reg z_in, z_in1, z_in2, z_in3;
reg z_in_, z_in_1, z_in_2, z_in_3;
reg on_in, on_in1, on_in2, on_in3;
reg on_in_, on_in_1, on_in_2, on_in_3;
reg hi_in, hi_in1, hi_in2, hi_in3;
reg hi_in_, hi_in_1, hi_in_2, hi_in_3;
reg lo_in, lo_in1, lo_in2, lo_in3;
reg lo_in_, lo_in_1, lo_in_2, lo_in_3;

reg dhia, dhia1, dhia2, dhia3;
reg dhia_, dhia_1, dhia_2, dhia_3;
reg dinha, dinha1, dinha2, dinha3;
reg dinha_, dinha_1, dinha_2, dinha_3;
reg dhib, dhib1, dhib2, dhib3;
reg dhib_, dhib_1, dhib_2, dhib_3;
reg dinhb, dinhb1, dinhb2, dinhb3;
reg dinhb_, dinhb_1, dinhb_2, dinhb_3;

reg hspa, hspa1, hspa2, hspa3;
reg hspa_, hspa_1, hspa_2, hspa_3;
reg hspb, hspb1, hspb2, hspb3;
reg hspb_, hspb_1, hspb_2, hspb_3;

reg on_out, on_out1, on_out2, on_out3;
reg on_out_, on_out_1, on_out_2, on_out_3;
reg z_out, z_out1, z_out2, z_out3;
reg z_out_, z_out_1, z_out_2, z_out_3;
reg lo_out, lo_out1, lo_out2, lo_out3;
reg lo_out_, lo_out_1, lo_out_2, lo_out_3;
reg hi_out, hi_out1, hi_out2, hi_out3;
reg hi_out_, hi_out_1, hi_out_2, hi_out_3;

reg                    stfla, stfla1, stfla2, stfla3;
reg                    stflb, stflb1, stflb2, stflb3;
reg                    stflc, stflc1, stflc2, stflc3;
reg                    stfld, stfld1, stfld2, stfld3;

reg ptemp;
reg ntemp;
reg                    tempad;
reg                    tempbd;
reg                    tempar;
```

```
 reg                tempbr;

 reg                vbbd;
 reg                vbbr;

initial
begin
 count = 1;
end

always @(posedge 'FMTIO.CLK)
#(envdrive.GLOBAL_CLK_PERIOD*0.00) //strobe inputs @ CLK leading edge //strobe also dclk
begin

 clk ='FMTIO.CLK;
 clk_     = 1-clk;
 clkcal = 0;
 clkcal_ = 1;
 cg0 = 1;
 cg1 = 1;
 cg2 = 1;
 cg3 = 0;
 tclk     = clk;
 tclk_    = 1-clk;
 tclkcal = 0;
 tclkcal_ = 1;
 tcg0 = 1;
 tcg1 = 1;
 tcg2 = 1;
 tcg3 = 0;
 dclk     = clk;
 dclk_ = 1-clk;

// rbus = 'FMTIO.R;
 rbus     = (('FMTIO.R===8'hxx) ? 8'h00 :'FMTIO.R);
 rbus_cycle= RBUS_CYCLE;
 rs               = (('FMTIO.RS===2'bxx) ? 2'b10 : 'FMTIO.RS);
 rtxc     = (('FMTIO.RTXC===1'bx) ? 1'b0 : 'FMTIO.RTXC);
 th2sel = (('FMTIO.THBSEL===1'bx) ? 1'b0 : 'FMTIO.THBSEL);

 da                 = (('FMTIO.DA===6'bxxxxxx) ? 6'h00 : 'FMTIO.DA);
 db                 = (('FMTIO.DB===6'bxxxxxx) ? 6'h00 : 'FMTIO.DB);
 dc                 = (('FMTIO.DC===6'bxxxxxx) ? 6'h00 : 'FMTIO.DC);
 dd                 = (('FMTIO.DD===6'bxxxxxx) ? 6'h00 : 'FMTIO.DD);
 ca                 = (('FMTIO.CA===6'bxxxxxx) ? 6'h00 : 'FMTIO.CA);
 cb                 = (('FMTIO.CB===6'bxxxxxx) ? 6'h00 : 'FMTIO.CB);
 cc                 = (('FMTIO.CC===6'bxxxxxx) ? 6'h00 : 'FMTIO.CC);
 cd                 = (('FMTIO.CD===6'bxxxxxx) ? 6'h00 : 'FMTIO.CD);

 achia =(( 'FMTIO.ACHA===1'hx) ? 0 : 'FMTIO.ACHA);
 achia_ = 1 - achia;
 bcloa =(( 'FMTIO.BCLA===1'hx) ? 0 : 'FMTIO.BCLA);
 bcloa_ = 1 - bcloa;
 achib =(( 'FMTIO.ACHB===1'hx) ? 0 : 'FMTIO.ACHB);
 achib_ = 1 - achib;
 bclob =(( 'FMTIO.BCLB===1'hx) ? 0 : 'FMTIO.BCLB);
 bclob_ = 1 - bclob;
 z_in = (('FMTIO.SETZIN===1'bx) ? 0 : 'FMTIO.SETZIN);
 z_in_ = 1 - z_in;
 on_in = (('FMTIO.SETONIN===1'bx) ? 0 : 'FMTIO.SETONIN);
 on_in_ = 1 - on_in;
 hi_in = (('FMTIO.SETHIIN===1'bx) ? 0 : 'FMTIO.SETHIIN);
 hi_in_ = 1 - hi_in;
 lo_in = (('FMTIO.SETLOIN===1'bx) ? 0 : 'FMTIO.SETLOIN);
```

```
lo_in_ =1 - lo_in;

#(envdrive.GLOBAL_CLK_PERIOD*0.95); // time is now 95% of GLOBAL_CLK_PERIOD after CLK's
posedge. Strobe outputs
dhia = 'FMTIO.DHIA;
dhia_ = 1 - dhia;
dinha = 'FMTIO.DINHA;
dinha_ = 1 - dinha;
dhib = 'FMTIO.DHIB;
dhib_ = 1 - dhib;
dinhb = 'FMTIO.DINHB;
dinhb_ = 1 - dinhb;

hspa = 'FMTIO.TMUA;
hspa_ = 1 - hspa;
hspb = 'FMTIO.TMUB;
hspb_ = 1 - hspb;

on_out = 'FMTIO.SETONOUT;
on_out_ = 1 - on_out;
z_out = 'FMTIO.SETZOUT;
z_out_ = 1 - z_out;
lo_out = 'FMTIO.SETLOOUT;
lo_out_ = 1 - lo_out;
hi_out = 'FMTIO.SETHIOUT;
hi_out_ = 1 - hi_out;

stfla    = 'FMTIO.STFLA;
stflb    = 'FMTIO.STFLB;
stflc    = 'FMTIO.STFLC;
stfld    = 'FMTIO.STFLD;

ptemp = 0;
ntemp = 0;
tempad  = 0;
tempbd  = 0;
tempar  = 0;
tempbr  = 0;

vbbd    = 0;
vbbr    = 0;


#(envdrive.GLOBAL_CLK_PERIOD*0.01); // time is now 96% of GLOBAL_CLK_PERIOD after CLK's
posedge
 -> firstpart_captured;
end

always @(firstpart_captured)
begin
 #(envdrive.GLOBAL_CLK_PERIOD*0.0);

 $fstrobeh(file_stat[0],
         "INC ","TSNORM"," {",
         clk,,clk_,,clkcal,,clkcal_,,count[0],,count[0],,"0",,"0",,,
         tclk,,tclk_,,tclkcal,,tclkcal_,,count[0],,count[0],,"0",,"0",,,
 dclk,,dclk_,,
         "%s",rbus_cycle,rbus,,rs,,rtxc,,th2sel,,,
         "%b",da,,"%b",db,,"%b",dc,,"%b",dd,,"%b",ca,,"%b",cb,,"%b",cc,,"%b",cd,,,
         achia,,achia_,,bcloa,,bcloa_,,achib,,achib_,,bclob,,bclob_,,,
         z_in,,z_in_,,on_in,,on_in_,,hi_in,,hi_in_,,lo_in,,lo_in_,,,
         dhia,,dhia_,,dinha,,dinha_,,dhib,,dhib_,,dinhb,,dinhb_,,,
         hspa,,hspa_,,hspb,,hspb_,,,
         on_out,,on_out_,,z_out,,z_out_,,lo_out,,lo_out_,,hi_out,,hi_out_,,,
            stfla,,stflb,,stflc,,stfld,,,
            ptemp,,ntemp,,tempad,,tempbd,,tempar,,tempbr,,,
            vbbd,,vbbr,
            "}");
 count = count + 1;
end
```

G.3 Fault Dictionary Database Builder

```
#!/usr/local/bin/perl

dbmopen(HIST1, 'failclock.db', 0666) || die "can't dbmopen failclock.db";
dbmopen(HIST2, 'failsig.db', 0666) || die "can't dbmopen failsig.db";
dbmopen(HIST3, 'failtest.db', 0666) || die "can't dbmopen failtest.db";
dbmopen(HIST4, 'failcs.db', 0666) || die "can't dbmopen failcs.db";
dbmopen(HIST5, 'failgate.db', 0666) || die "can't dbmopen failgate.db";
dbmopen(HIST6, 'failct.db', 0666) || die "can't dbmopen failct.db";

while(<>){
 split;
 if ($_[7]=~/fail/){
 $clock = $_[5];
 $test = $_[9].":".$_[11];
 $sig = "$_[13]$_[15]$_[17]$_[19]$_[21]";
 if (($_[2]>=482)&&($sig=~/^00000000x000001111111$/)) {
 $sig=$sig."b";
 }
 $gate = "$_[2]:$_[4]";
 $content1 = "$_[2]:$_[4]";
 $content2 = "$_[2]:$_[4]";
 $content3 = "$_[2]:$_[4]";
 $content4 = "$_[2]:$_[4]";
 $content5 = "$test:$clock:$sig";
 $content6 = $test;
 if (($_[2]>=372)&&($clock==91)) {
 $clock=$clock."b";
 }
 if($HIST1{$clock}){
 $content1 = $HIST1{$clock}." ".$content1;
 }
 if($HIST2{$sig}){
 $content2 = $HIST2{$sig}." ".$content2;
 }
 if($HIST3{$test.":".$sig}){
 $content3 = $HIST3{$test.":".$sig}." ".$content3;
 }
 if($HIST4{"$clock:$sig"}){
 $content4 = $HIST4{"$clock:$sig"}." ".$content4;
 }
 if($HIST5{$gate}){
 $content5 = $HIST5{$gate}." ".$content5;
 }
 if($HIST6{$clock}){
 if ($HIST6{$clock}!=$content6){
 $content6 = $HIST6{$clock}." ".$content6;}
 }
# print "$clock = $content\n";
 $HIST1{$clock} = $content1;
 $HIST2{$sig} = $content2;
 $HIST3{$test.":".$sig} = $content3;
 $HIST4{"$clock:$sig"} = $content4;
 $HIST5{$gate} = $content5;
 $HIST6{$clock} = $content6;
}}

dbmclose(HIST1);
dbmclose(HIST2);
dbmclose(HIST3);
dbmclose(HIST4);
dbmclose(HIST5);
dbmclose(HIST6);
```

G.4 Fault Dictionary Database Query

```
#!/usr/local/bin/perl

$outfile = "possible_gates";
if($#ARGV == 0){
 $clock = $ARGV[0];
} elsif($#ARGV >=1){
 $argset = $ARGV[0];
}
if($argset=~/^-c$/){
 $clock = $ARGV[1];
} if($argset=~/^-t$/){
 $test = $ARGV[1];
} if($argset=~/^-ct$/){
 $clock = $ARGV[1];
} if($argset=~/^-s$/){
 $sig = $ARGV[1];
} if($argset=~/^-cs$/){
 $clock = $ARGV[1];
 $sig = $ARGV[2];
} if($argset=~/^-cf$/){
 $clock = $ARGV[1];
} if($argset=~/^-tsf$/){
 $test = $ARGV[1];
 $sig = $ARGV[2];
 $outfile = $ARGV[3];
} if($argset=~/^-ts$/){
 $test = $ARGV[1];
 $sig = $ARGV[2];
} if($argset=~/^-sf$/){
 $sig = $ARGV[1];
 $outfile = $ARGV[2];
} if($argset=~/^-csf$/){
 $clock = $ARGV[1];
 $sig = $ARGV[2];
 $outfile = $ARGV[3];
}

open(OUT_FH, ">$outfile");
open(ERR_FH, ">>errors");
dbmopen(HIST1, 'failclock.db', 0666) || die "can't dbmopen failclock.db";
dbmopen(HIST2, 'failsig.db', 0666) || die "can't dbmopen failsig.db";
dbmopen(HIST3, 'failtest.db', 0666) || die "can't dbmopen failtest.db";
dbmopen(HIST4, 'failcs.db', 0666) || die "can't dbmopen failcs.db";
dbmopen(HIST5, 'failgate.db', 0666) || die "can't dbmopen failgate.db";
dbmopen(HIST6, 'failct.db', 0666) || die "can't dbmopen failct.db";

if (($argset=~/^-cs$/)||($argset=~/^-csf$/)){
if ($HIST4{"$clock:$sig"}){
 foreach $record (split(/ /,$HIST4{"$clock:$sig"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}
if (($clock==91)&&($sig!=~/^00000000x0000011111111$/)){
 foreach $record (split(/ /,$HIST4{$clock."b:".$sig})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}
if (($clock!=91)&&($sig=~/^00000000x0000011111111$/)){
 foreach $record (split(/ /,$HIST4{$clock.":".$sig."b"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}
if (($clock==91)&&($sig=~/^00000000x0000011111111$/)){
 foreach $record (split(/ /,$HIST4{$clock."b:".$sig."b"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}}

elsif (($argset=~/^-c$/)||($argset=~/^-cf$/)){
if ($HIST1{$clock}){
 foreach $record (split(/ /,$HIST1{$clock})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}
if ($clock==91){
 foreach $record (split(/ /,$HIST1{$clock."b"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}}

elsif (($argset=~/^-s$/)||($argset=~/^-sf$/)){
if ($HIST2{$sig}){
 foreach $record (split(/ /,$HIST2{$sig})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";
 }}
if ($sig=~/^00000000x0000011111111$/){
 foreach $record (split(/ /,$HIST2{$sig."b"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}}

elsif (($argset=~/^-ct$/)||($argset=~/^-ctf$/)){
if ($HIST6{$clock}){
 foreach $record (split(/ /,$HIST6{$clock})){
 print OUT_FH "$record\n";
 }}
if ($clock==91){
 foreach $record (split(/ /,$HIST3{$clock."b"})){
 print OUT_FH "$record\n";}}}

elsif (($argset=~/^-ts$/)||($argset=~/^-tsf$/)){
if ($HIST3{$test.":".$sig}){
 foreach $record (split(/ /,$HIST3{$test.":".$sig})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";
 }}
if ($sig=~/^00000000x0000011111111$/){
 foreach $record (split(/ /,$HIST3{$test.":".$sig."b"})){
 split(/:/,$HIST5{$record});
 print OUT_FH "$record\t->\t$_[0]\t$_[1]\t$_[2]\t$_[3]\n";}}}

dbmclose(HIST1);
dbmclose(HIST2);
dbmclose(HIST3);
dbmclose(HIST4);
dbmclose(HIST5);
dbmclose(HIST6);
close OUT_FH;
close ERR_FH;
```

```
/* see file: analyze_distance.com  for data preparation */


/* calcualte each measurement's difference from its mean */
data dis2;
  set distance;
  MDIS_BG  = DIS_BG - 4.4756831;
  MDIS_TR  = DIS_TR - 4.4870583;
  MDIS_EUD = DIS_EUD - 4.5400638;
  MDIS_SHT = DIS_SHT - 5.0866029;

  MST_BG  = MDIS_SHT - MDIS_BG;
  MST_TR  = MDIS_SHT - MDIS_TR;
  MST_EUD = MDIS_SHT - MDIS_EUD;
run;

proc means data=dis2 mean stderr t prt;
  var MST_EUD  MST_TR  MST_BG;
  title 'Paired Comparisons T Test';
run;


================================================================================

                  Paired Comparisons T Test

 Variable          Mean     Std Error            T  Prob>|T|
 ------------------------------------------------------------
 MST_EUD   6.2112933E-8     0.0136398  4.5537992E-6    1.0000
 MST_TR    2.6593807E-8     0.0135057  1.9690868E-6    1.0000
 MST_BG     5.428051E-8     0.0132487  4.0970309E-6    1.0000
 ------------------------------------------------------------
```