Hierarchical testing using precomputed tests for modules

Murray, Brian Thomas, Ph.D.

The University of Michigan, 1994

HIERARCHICAL TESTING USING PRECOMPUTED TESTS FOR MODULES

by

Brian Thomas Murray

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

The return from your work must be the satisfaction which that work brings you and the world's need of that work

W. E. B. Du Bois

To my family.

ACKNOWLEDGEMENTS

I must express my heartfelt appreciation to my advisor John Hayes for his patient mentoring and guidance. By his instruction and example, I have learned a lot about computer science and how to do research. His skill in research, writing, and teaching are a continuing inspiration to me. Furthermore, I value his friendship as much as the skills I have learned. I also appreciate the friendship and interaction I have enjoyed with the many fellow graduate students who also worked with John during my tenure.

I am deeply grateful to my wife Alison for her patience and understanding during this project, which was always "almost finished." I also thank my daughters Lauren and Alyssa for patiently waiting for Daddy to finish "his book." My love for Alison and Lauren and Alyssa inspired me when the work was difficult. My parents deserve thanks for having encouraged me all my life and instilling in me the belief that no worthy goal is beyond my grasp.

I want to thank my supervisor John Hile for his support and advice throughout the years it has taken me to complete my research and my thesis. He has provided both material and management support, without which I could not continue. He has always been supportive of the concepts as well as the techniques and tools I have developed.

I must also thank R. G. "Ben" Bennetts and the folks who formed the test generation group at Cirrus Computers in the mid 1980's, including Dave Potts and Andrew Guyler among others. The lessons I learned working with them while developing methods to use Cirrus's Hitest as a hierarchical test generator ultimately led to the research reported here.

Finally, I want to thank General Motors for funding my research, originally through tuition refund and ultimately through the Technical Education Program. I especially want to thank Elaine Chapman Moore for the variety of administrative help she has given me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

LIST OF SYMBOLS

Symbol	
T_S	Test stimulus vector sequence for a module or circuit
T_R	Test response vector sequence from a fault-free module under test when T_S is applied to the module's inputs
T_{Ri}	Test response when fault f_i is active in the module under test
(T_R, T_{Ri})	Test response error or discrepancy represented as a vector sequence pair
$(T_S;T_R)$	Test package containing test data for a module or circuit
X, Z	Set of input and output ports
X_D, Z_D	Input and output data bus ports for a module or circuit
X_C, Z_C	Input and output control bus ports for a module or circuit
$V(X)$, $\{V(X)\}$, $S(X)$	Vector (signal value) at port X, all possible values at X, vector sequence constructed from $\{V(X)\}$
$P[M;(X_D;Z_D);\Omega]$	Propagation function on domain Ω from X_D to Z_D of M, represented by a set of input/output pairs $(\alpha_i;\beta_i)$, $1 \le i \le n$, where $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ is a partition of Ω, and β_i is the output of the function for α_i
#	Connection operation for parallel circuit junctions
$\circ$	Connection operation for series circuit junctions
$T[M;(X_D;Z_D)](S_C)$	Transmission function for module M, a propagation function on $\Omega = \{V(X)\}$ when the vector sequence S_C is applied to X_C
Trans(T)	Transparency index for transmission function T
B_i	Size of α_i in element $(\alpha_i;\beta_i)$ of a propagation function
V_n	Set of all possible values that may be applied to an n-bit bus

X	Unknown symbolic signal value
S	Stimulus symbolic signal value
R	Response symbolic signal value
C	Corrupted symbolic signal value
0_n	The n-bit all-0 vector
1_n	The n-bit all-1 vector
$\hat{S}$	Stimulus values not including 0_n or 1_n
$R_4 = \{X,S,R,C\}$	Basic symbolic signal value set
$R_7 = \{X,S,R,C,0_n,1_n,\hat{S}\}$	Basic symbolic signal value set for typical datapath operations
Ω	Domain of propagation functions, set of signals to be propagated

CHAPTER I
INTRODUCTION

Much of daily life is automated by computer-based digital systems whose failure could be catastrophic. Typical systems include real-time transaction-processing systems, airplane control systems, implanted medical instruments, and automotive control systems. For safety's sake, these systems, which are implemented by electronic integrated circuits (ICs), must be thoroughly tested before and during use. This thesis addresses the problem of test generation for complex digital systems, especially for application-specific integrated circuits (ASICs), and the relationship of testing to design.

1.1. Design and Testing

The reliability of all ICs is strongly dependent on how thoroughly they are tested. This is because ICs most often fail as result of fabrication defects rather than wearout, and thorough testing for these defects weeds out weak ICs. New fabrication technologies for ICs are constantly being developed and quickly brought to production to address requirements for higher circuit density. ICs manufactured using a new process often have many defects and it may take years for a process to mature to the point where the number of defects is negligible [2].

Problems of Testing. Testing is costly. It has become one of the most expensive aspects of manufacturing, a fact which has prompted many IC makers to seek testing methods that use cheaper testers, simpler test fixtures, and shorter test sequences. In addition to manufacturing cost, it typically takes about one third to one half of the design time to develop a method for testing a new IC [100]. Meanwhile, competition in the marketplace makes short development time essential. Computer-aided design (CAD) tools for very large scale integrated (VLSI) circuits have significantly reduced some aspects of design time, but corresponding gains have not been made in the area of test generation, despite the fact that testing has been thoroughly studied for more than thirty years [2]. Adequate testing solutions have remained elusive mainly because requirements change as technology and design styles evolve, and because many test problems are truly intractable.

The testability of ICs depends on how easily internal nodes of the circuits can be controlled and observed. The high density of new ICs—recent VLSI chips contain several million transistors—makes this difficult. Moreover, soaring clock and data rates are creating new ways in which ICs can fail. Timing errors and parametric faults are becoming more prevalent and often require new test techniques.

Hierarchical Design. A typical digital IC design scenario is shown in Figure 1.1. The first step is architecture design. In this step, specifications are translated into architecture components that the designers know how to build reliability. For example, decisions are made regarding such things as on-chip memory (RAM and ROM) size, what type of pipelining will be used, what instruction set will be used, whether microcoding will be used, etc. These design decisions can be analyzed by modeling the system at the register level and simulating it. The smallest elements in these models tend to be modules such as RAMs, ROMs, adders, and multiplexers.

After the architecture details are determined, the functionality of many of the modules may be specified in terms of Boolean equations and state diagrams. These in turn are translated into networks of logic gates and flip-flops, a process called logic design. Finally, the circuit can be designed in terms of individual transistors and other low-level circuit elements.

At each level in this top-down design process, design and simulation steps are carried out repeatedly until the specifications for that level are satisfied. Normally, designers try to minimize iteration between levels. This can be done by careful project planning and the use of predictive CAD tools such as timing analyzers and simulators, which allow many low-level circuit effects to be predicted at the gate level.

During architecture planning, designers may formulate a strategy for how the IC will be tested. The test strategy specifies, in abstract fashion, what fault types will be covered, how the IC will be partitioned for testing, how tests will be generated for each partition, how tests will be

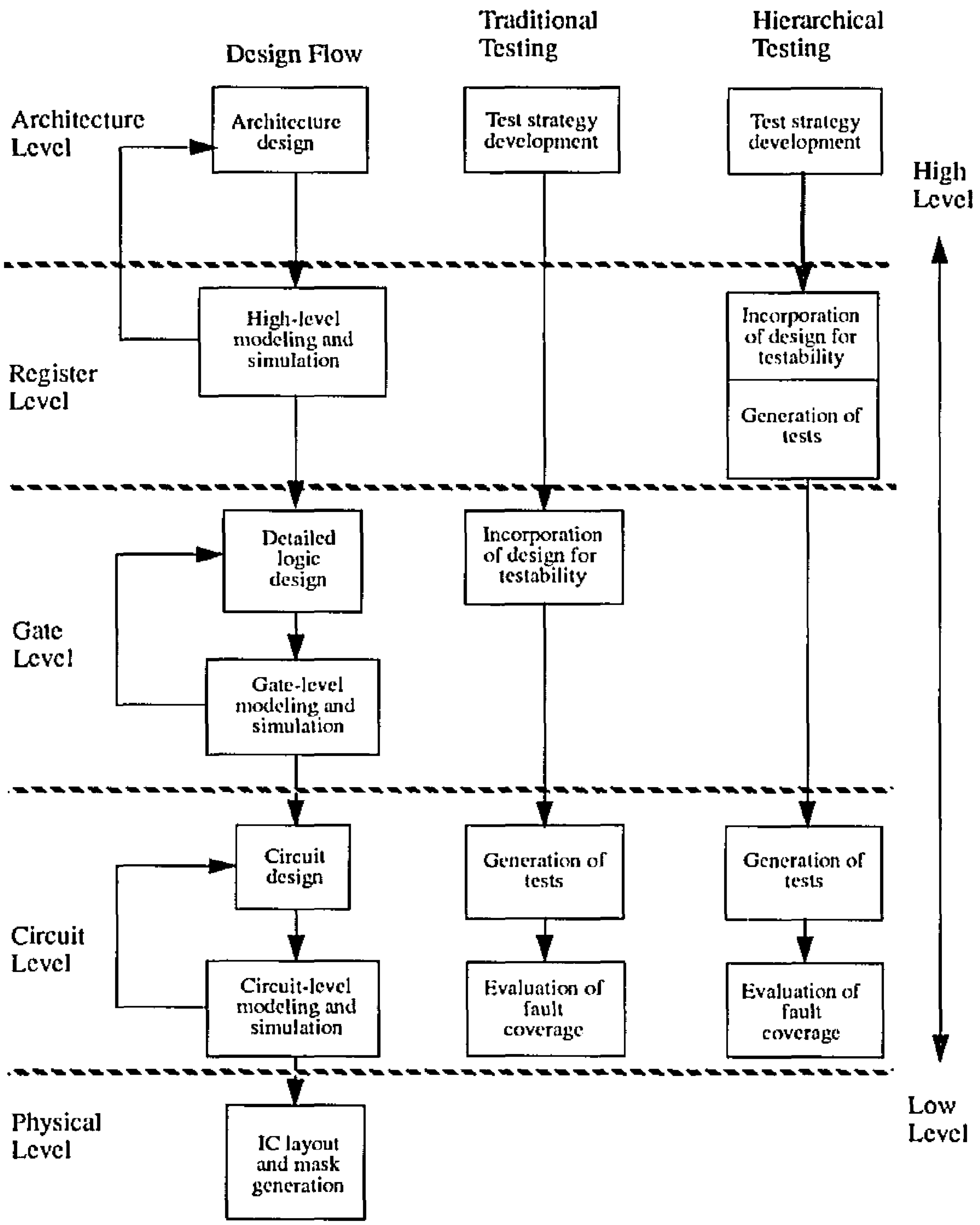

Design Flow
Traditional Testing
Hierarchical Testing
Architecture Level
Register Level
Gate Level
Circuit Level
Physical Level
High Level
Low Level
Architecture design
High-level modeling and simulation
Detailed logic design
Gate-level modeling and simulation
Circuit design
Circuit-level modeling and simulation
IC layout and mask generation
Test strategy development
Incorporation of design for testability
Generation of tests
Evaluation of fault coverage
Test strategy development
Incorporation of design for testability
Generation of tests
Generation of tests
Evaluation of fault coverage

Figure 1.1. Typical organization of design and test in the development of a new IC.

applied to each partition, etc. However, incorporation of any hardware to be used for testing (design for testability) is not usually done until the gate-level logic design stage is reached. In addition, test generation is only done after complete, detailed gate-level models have been developed since most CAD tools require these models as inputs. Current ASICs have hundreds of thousands of logic gates and soon will have millions. Gate-level test generation tools are already taxed to the limit and may soon become inadequate. In order to handle large circuits, these tools require highly constrained gate-level design styles, and therefore cannot effectively use the high-level structure of the circuit. Finally, test generation accounts for a large number of gate-level design iterations, and frequently prompts architecture changes to address problems that were not predicted during test strategy planning. This reflects the large gap between test planning and test generation, and the fact that predictive tools are almost non-existent.

Hierarchical Testing. Recently however, a number of high-level and hierarchical techniques for test generation have been proposed [9, 12, 18, 57, 58, 59, 62, 64, 75, 84, 87, 89, 90, 93, 94]. These techniques take advantage of the hierarchical structure found in most circuits and exploit circuit behavior that is only apparent at the register level. As shown in Figure 1.1, hierarchical testing techniques allow test strategy development, incorporation of design for testability, and test generation to take place at higher design levels. Since fewer functional objects are manipulated at the register level, design iterations are made more quickly. Test generation performance is also enhanced because there are fewer primitives to test, and because high-level circuit behavior, such as the ability to load test values into multi-bit registers, can be exploited more easily. The more important hierarchical techniques will be reviewed in Section 1.3.

Although hierarchical test generation techniques have been proposed since the mid-1970s [10], only recently have they become practical to use, as design styles have evolved to the point where hierarchical design is well-supported. New CAD tools are able to synthesize gate- and circuit-level simulation models from higher-level models [32, 95]. In many cases, designers compose circuits using libraries of large predefined modules such as ALUs and RAMs which have precomputed tests, thus obviating the need for test generation at lower levels of abstraction, a fact which we explore here. Overall, hierarchical techniques are well-suited to currently prevailing and future design styles. The test generation performance improvements obtained by the use of hierarchy offers the potential for dealing with ICs containing hundreds of millions of transistors.

The previously published hierarchical techniques have not been very successful—few have been incorporated in commercial systems. In many cases, they do not effectively represent

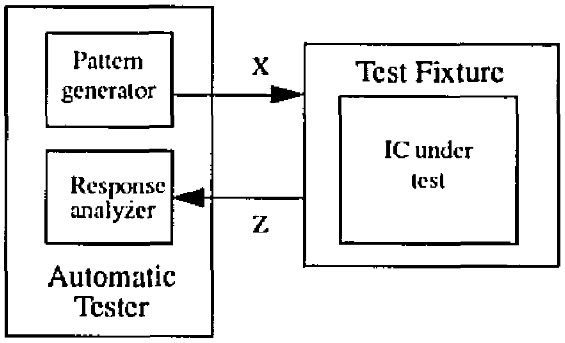


Figure 1.2. Automatic tester and the testing process.

the high-level structure of the circuit, and in other cases they rely on heuristics that have not proven to be widely applicable. Nevertheless, various hierarchical techniques are becoming important, such as the use of precomputed tests.

This thesis examines in detail the use of precomputed tests for modules in generating tests for new ICs incorporating these modules. In this chapter, we present a general introduction to testing theory and review the previous work in hierarchical testing with special emphasis on test generation using precomputed tests.

1.2. Basic Testing Theory

The testing process for integrated circuits is outlined in Figure 1.2. The pattern generator stimulates some of the inputs of the IC under test, and some time later the response analyzer reads the response sequence Z of the IC and checks for discrepancies between Z and an expected response Z_0 produced during test generation; a fault is detected when $Z \neq Z_0$. From this description, one can see that faults will be detected not only when the IC generates incorrect logic values, but also when values are generated at the wrong time. To produce and analyze the large amount of test data needed for modern high-speed ICs, automatic testers like the one depicted in Figure 1.2 have become large and expensive; some production testers cost millions of dollars.

Automatic testers are controlled by programs similar to those for general-purpose computers. The bulk of a typical test program is stimulus and response data, and it is the responsibility of the test generator to compute these data. Test generation is examined in detail in the next three subsections.

1.2.1 Circuit Modeling

Circuit models at all design levels are often classified as either structural or behavioral. Structural models emphasize the fact that circuits are composed of components (modules) which are interconnected by wires, or groups of wires, called buses. Behavioral or functional models, on the other hand, ignore circuit structure as much as possible, concentrating on only a few relevant structural details that allow us to describe the circuit's input/output behavior. Most models used in practice are mixed, that is, they contain aspects of both structure and function.

As noted above, high-level structural models contain large modules, which are themselves often described by behavioral models. In addition, the buses used to interconnect high-level modules are frequently treated as primitive entities which propagate abstract multi-bit signal values. Gate-level models are composed of a relatively small number of simple primitive modules (gates) that implement basic Boolean logic operations (AND, OR, NOT, EXCLUSIVE-OR, etc.). In many cases, circuit models are also hierarchical, that is, large modules defined at design level i can be recursively composed by interconnecting smaller modules defined at the next lower level $i - 1$. Thus, large modules may be described by multiple models, some structural and some behavioral.

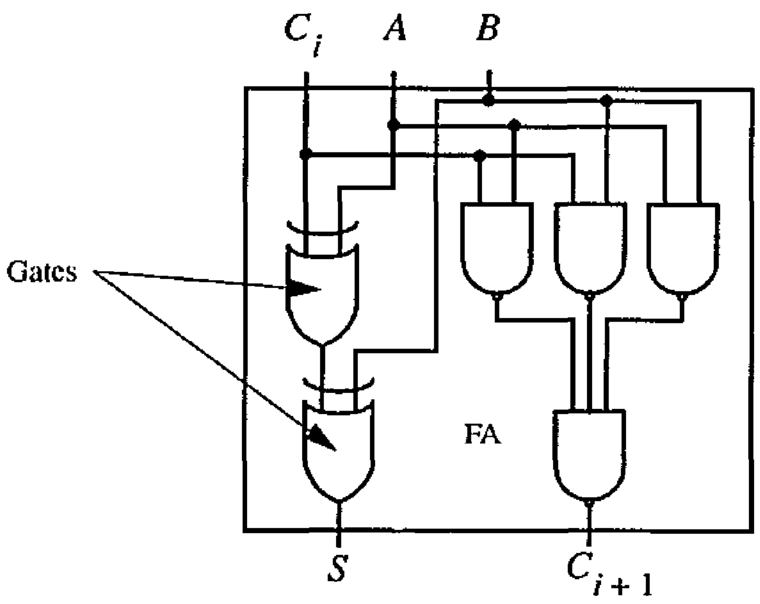
A simple example of hierarchical structure appears in the ripple-carry adder of Figure 1.3, which is composed of several full adder (FA) modules. Figure 1.3a shows a structural model of a FA module; this is also an example of gate-level design. A behavioral description of a FA is given by the following set of Boolean formulas defining the adder's sum (S) and output carry (C_{i+1}) signals in terms of input data (A and B) and input carry (C_i) signals

$$S = A \oplus B \oplus C_i$$

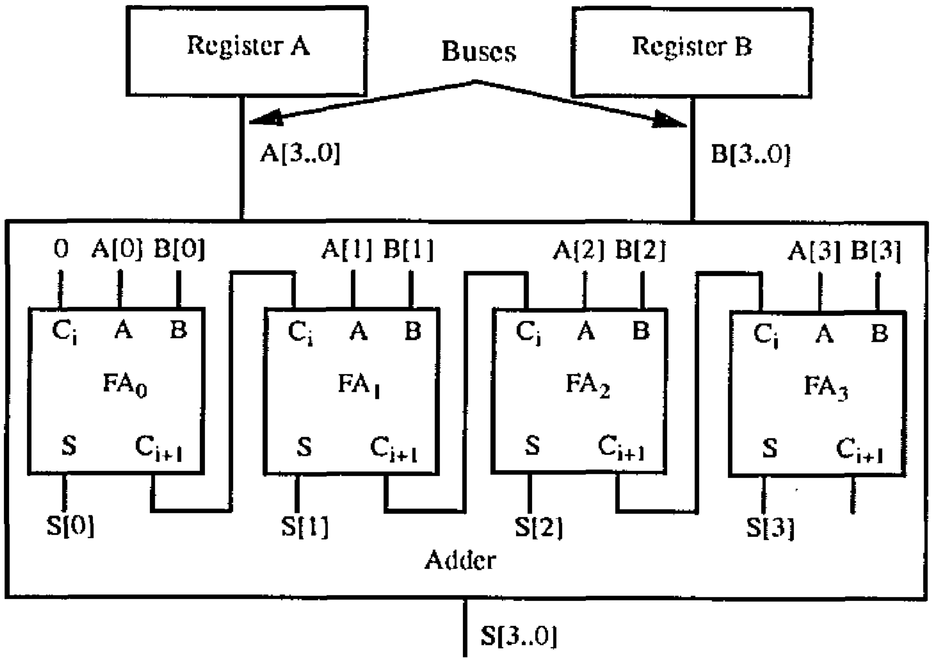
$$C_{i+1} = (A \wedge B) \vee (A \wedge C_i) \vee (B \wedge C_i)$$

Here, all variable values are binary (0 or 1), and $\wedge$, $\vee$, and $\oplus$ denote the Boolean operations AND, OR, and EXCLUSIVE-OR, respectively.

Figure 1.3b shows a group of FA modules connected to form a 4-bit adder, which is then combined with two registers to design a simple processor. This circuit provides an example of higher-level logic design at the register level. $A[3..0]$, $B[3..0]$, etc. denote buses defining higher-level versions of the binary signals in the full adder equations above. The operation of this adder can be succinctly represented by the high-level equation



(a)



(b)

Figure 1.3. Adder circuits viewed at two levels of abstraction: (a) gate(low), (b) register (high).

$$S[3..0] = A[3..0] + B[3..0]$$

Behavioral statements such as these are often combined into text descriptions of a circuit in a systematic form called a hardware description language (HDL). Note that there is no unique way of constructing behavioral models of this type. Recently however, designers have begun to standardize HDL formats. VHDL (the VHSIC Hardware Description Language [20]) has been mandated as a standard by the United States Government. Structural descriptions are modeled in HDLs as a list of wires and the corresponding modules they connect to; this is known as a netlist. Alternatively, they can be entered into a design system graphically, as a schematic diagram of symbols representing modules or functions, connected by lines representing wires and buses.

In the past, designers were often most comfortable working at the low, gate level, or with hierarchical models whose large (non-primitive) modules are described only by structural models. In this case, hierarchy is only a convenience for the entry and maintenance of the design; it is not used by CAD tools. The gate level is also the best supported in terms of design theory and CAD tools. Design data can be easily entered graphically, and low-level circuit models and layout can be reliably and efficiently generated from such gate-level models.

High-level simulation of behavioral descriptions, on the other hand, has often been considered a superfluous step which requires the development of additional simulation models that are not readily translated into lower-level models. However, analysis of many performance-enhancing design trade-offs is only possible by high-level simulation. Such design trade-offs are becoming increasingly important as IC densities grow. Recently, a number of new CAD tools have been developed which are able to automatically synthesize gate- and circuit-level descriptions from high-level behavioral descriptions [32]. These high-level synthesis tools show promise in automating digital design in a number of specialized areas, particularly signal-processing and digital control.

1.2.2 Fault Modeling

Faults in ICs primarily result from physical defects introduced during manufacturing. A variety of disturbances arise even in mature fabrication lines, and cause subtle failures that may not be detected for a long time without thorough testing. They can later show up as performance degradation, as errors in little-used functions, or as obvious catastrophic failures.

To generate tests, we use logical fault models that reflect changes in circuit function due to physical defects. A digital logic circuit is then said to fail if the function it implements differs from the function it was designed to implement. Fault models provide a consistent and technology-independent mechanism for how the logic function might fail, as well as a standard yardstick for measuring the quality of a set of tests. In developing a fault model, it is important to strike a balance between accuracy and complexity. The model must also match the characteristics of the design level(s) at which it is used.

The most common fault model is the low-level *single stuck line* (SSL) fault model. Physical failures are represented by maintaining a single line in the circuit at the constant value 0 or 1 regardless of how circuit operation stimulates the line. To test if a line is stuck at 0 (1), we must find a sequence of one or more primary input test patterns that will make the line 1 (0). This *exposes* the fault. The test must also arrange for an error caused by the fault to be propagated to a primary output where the error can be observed. Tests must usually be found for all $2N$ SSL faults in an N-line circuit. Typically a given test will detect or *cover* more than one fault. The fraction of faults covered by a set of tests is called the *fault coverage*. Experience has shown that ICs tested to provide high fault coverage (0.99 or better) for SSL faults generally have high reliability in the field. At present, the coverage of SSL faults is the only well-accepted measure of test quality, and all other measures must be calibrated against this model.

Few formal higher-level fault models exist, and those that do are often defined in imprecise terms. In Lin and Su's register-level fault model [64], for example, faults are classified according to their effect on some register-level components. These effects include such symptoms as register decoding errors, and data transfer errors. Other higher-level fault models are extensions of the SSL fault model. For example, Bhattacharya and Hayes [12] extended the SSL fault model to include all bits of a bus, leading to the concept of bus faults.

Since the SSL model only approximates the physical faults that occur in ICs, a considerable amount of work has been devoted to verifying it and to developing new, more accurate models. Problems with the SSL model, particularly for CMOS circuits, were identified as early as 1978 [98], so CMOS circuits are sometimes modeled at the (very low) switch level [47], where transistors are treated as idealized switches. Recently, designers have also become concerned about two alternative types of faults. The first type is the delay fault. Some defects cause the IC to generate correct logic values only after an excessive delay [63]. The second type is the bridging fault, where physically separate wires in a circuit are connected by a fault. By modeling the effect of manufacturing defects on layout, Ferguson and Shen were able to inductively show that the most likely CMOS fault is a bridge between adjacent wires [29]. Modeling delay and bridging faults can result in better quality tests, but test generation procedures for them are much more complex than for SSL faults. However, tests for these more accurate fault models can be generated and stored for register-level library modules, and later used as precomputed tests.

Finally, some circuits, such as RAMs and ROMs, have unique failure modes [2]. RAMs for instance, can fail so that certain patterns written to and read back from a particular address will expose a fault, otherwise the fault is not observed. These circuits are usually tested using completely different methods from those used for logic circuits [2]. Such tests can also be stored as precomputed tests.

1.2.3 Test Generation

In principle, it is possible to generate tests without the use of an explicit fault model. There are two approaches to such "black box" testing: random testing, where pseudo-randomly generated patterns are applied to the inputs of the circuit; and exhaustive testing, where all possible patterns are applied. Neither of these is practical for an entire VLSI circuit. Empirical evidence suggests that some faults are resistant to random testing [100], and most high-volume IC manufacturers believe that fault coverage of 99 percent is necessary. To see why exhaustive testing of entire circuits is not practical, consider an IC with 50 inputs (small compared to most modern VLSI circuits). Assuming that the circuit is combinational, the number of possible input vectors is $2^{50} = 1.126\times10^{15}$. A tester, operating at 50MHz would take more than 260 days to test this IC exhaustively. A sequential circuit would require an even larger number of input test vectors.

An alternative approach is to derive tests for faults in a structural circuit model algorithmically. Most research is devoted to algorithmic test generation. In Figure 1.4, we outline a generic procedure for generating tests for all faults in a circuit model. Step 5 of this procedure is performed by an algorithmic test generator. We next discuss the details of algorithmic test generation.

Combinational Circuits. The most studied approach to algorithmic test generation employs gate-level structural models; nearly all commercial test generators do so. We will discuss the two most widely used gate-level algorithms, versions of which can also be used at other abstraction levels.

```
test generation
{
    do {
        select an uncovered fault;
        generate a test for the fault;
        evaluate fault coverage thus far;
    } while (fault coverage is inadequate and time has not run out);
}
```

Figure 1.4. Generic test generation algorithm.

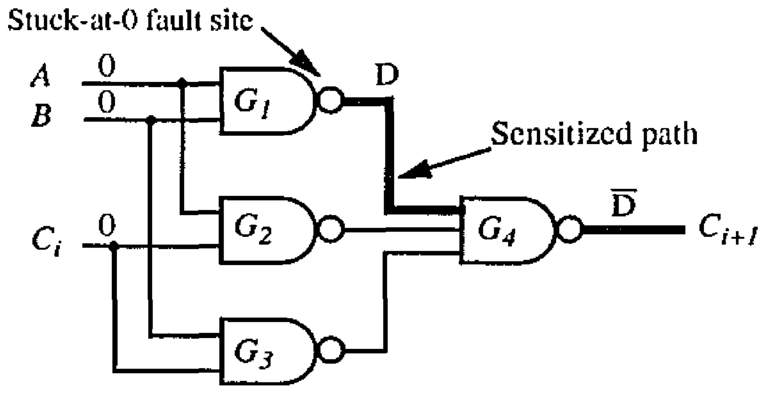


Figure 1.5. An example of D-propagation in a carry circuit.

Test generation algorithms have two basic steps: (1) expose the currently selected fault, and (2) propagate an error signal from the site of the fault to an observable output.

We first introduce some standard notation used for describing errors in test generation algorithms. If a line in a circuit is 0 (1) when it should be 1 (0), the error signal value on that line is represented by the symbol D ($\overline{D}$) for *discrepancy*. Consider the circuit in Figure 1.5. This circuit is the carry part of a full adder (Figure 1.3a). A stuck-at-0 fault at the output of gate G_1 can be exposed by attempting to make the output 1. A signal on this line will be detected as an error if it is 0 when it should be 1, that is, a D. The fault G_1 stuck-at-0 can be exposed, therefore, by assigning 0 to one or both of A and B.

Suppose that the output line of G_4 is observable. To observe the fault, we must propagate the D error signal from G_1 through G_4, which requires *sensitizing* G_4 to the error signal. We can do this only by assigning 1 to each of G_4's inputs from gates other than G_1. The output of G_4 will be $\overline{D}$ as shown, which still carries the error information. The error propagation step just described is called *D-propagation*. Now, the outputs of G_2 and G_3 have been assigned specific logic values, but not all of their inputs. If C_i is assigned the value 0, the circuit behavior will be completely specified. This process of determining complete and consistent specifications of circuit signal values is called *justification*.

The most widely known test generation algorithm is the D-algorithm, first published in 1966 [33]. It provides a systematic implementation of the D-propagation and justification steps described above. In the case of D-propagation, several D's ($\overline{D}$'s) may be propagated simultaneously, since sometimes an error signal must be propagated along more than one path to reach an observable output. In the D-algorithm, D-propagation and justification operations make only local assignments of signal values. To justify a value on the output of gate G_i, the D-algorithm makes assignments to the inputs of G_i. If these are not primary inputs, assignments to them become objectives for subsequent justification steps.

Both D-propagation and justification involve decisions or choices. For example, to accomplish D-propagation in the circuit of Figure 1.5, C_i could have been assigned 1 instead of 0. If D-propagation or justification cannot be done at some point without invalidating signal values already assigned, the D-algorithm *backtracks*, that is, returns to an earlier decision point and makes an alternative decision. For example, an alternative path can be followed from a point on the D-propagation path where a signal line branches in several directions.

The D-algorithm has been successfully implemented for many years. Around 1980, it was shown to be inefficient for an important class of circuits called error-correction-and-translation circuits [40]; it may be inefficient for other useful circuits as well. Poor choices for D-propagation and justification in these circuits lead to an excessive number of backtracks and unacceptably long computation times. A major reason is the fact that backtracking might be initiated at any gate in the circuit.

The PODEM (Path Oriented DEcision Making) test generation algorithm avoids this problem by backtracking only at primary inputs [33]. In PODEM, internal values are not justified explicitly, as in the D-algorithm. To satisfy an internal objective such as a D or $\overline{D}$ on some internal line, a value is assigned to a primary input X_i and the circuit is simulated. If the simulation proves that the assignment does not satisfy the objective, the algorithm assigns another input value. If during simulation, two values conflict on a line, the algorithm backtracks by changing the value of the last assigned X_i. When both values have been tried unsuccessfully, the algorithm backtracks to the next-to-last assigned input. In this way, the algorithm can exhaustively explore all possible circuit states, but only implicitly. So while the effect of all circuit states is considered, not all possible assignments are made. In contrast to the D-algorithm, PODEM makes assignments only to primary inputs, not internal nodes of the circuit. Nevertheless, the assignment to be made should be related to the initial objective. In PODEM, a procedure called *backtrace* obtains this initial assignment. PODEM traces a path from the site of an internal objective to be satisfied to a primary input. Along this path it transfers its initial objective gate by gate until an assignment is made to a primary input.

A number of test generation techniques have been developed since PODEM [33, 52, 85], most of which are simply extensions to it. Their goal is to reduce the number of backtracks by identifying choices a test generation algorithm might make that cannot lead to a solution, without actually pursuing every decision. For example, the FAN algorithm [33] seeks to identify conflicts at fanout branches within a circuit, thereby avoiding backtracks at the primary inputs and the cost of simulating large parts of the circuit. Conflicting assignments at fanout branches cannot be satisfied by any assignment at primary inputs. Gate-level test generation speedups reported in [33] averaged about 3.5 over PODEM. In general, new techniques for gate-level test generation like FAN do not result in order-of-magnitude speedups over previous techniques.

Sequential Circuits. Algorithms such as the D-algorithm and PODEM cannot generate tests directly for sequential circuits because they assume that all assignments are instantaneously propagated. However, we can extend these algorithms to generate tests for some sequential circuits. Consider the standard (Huffman) model of a finite state machine in Figure 1.6a. We construct a pseudo-combinational iterative model of this circuit by:

1. Replacing each flip-flop, which has a fixed (clock determined) delay, by a pseudo flip-flop whose function is equivalent to the flip-flop's, but whose output is produced instantaneously, and
2. Connecting multiple copies of the circuit to form an acyclic circuit in the iterative form shown in Figure 1.6b

Each copy of the iterative circuit represents a different instant in time and is called a *timeframe*. Given a known initial state $q(t_0)$ at time t_0, the most common approach to sequential test gener-

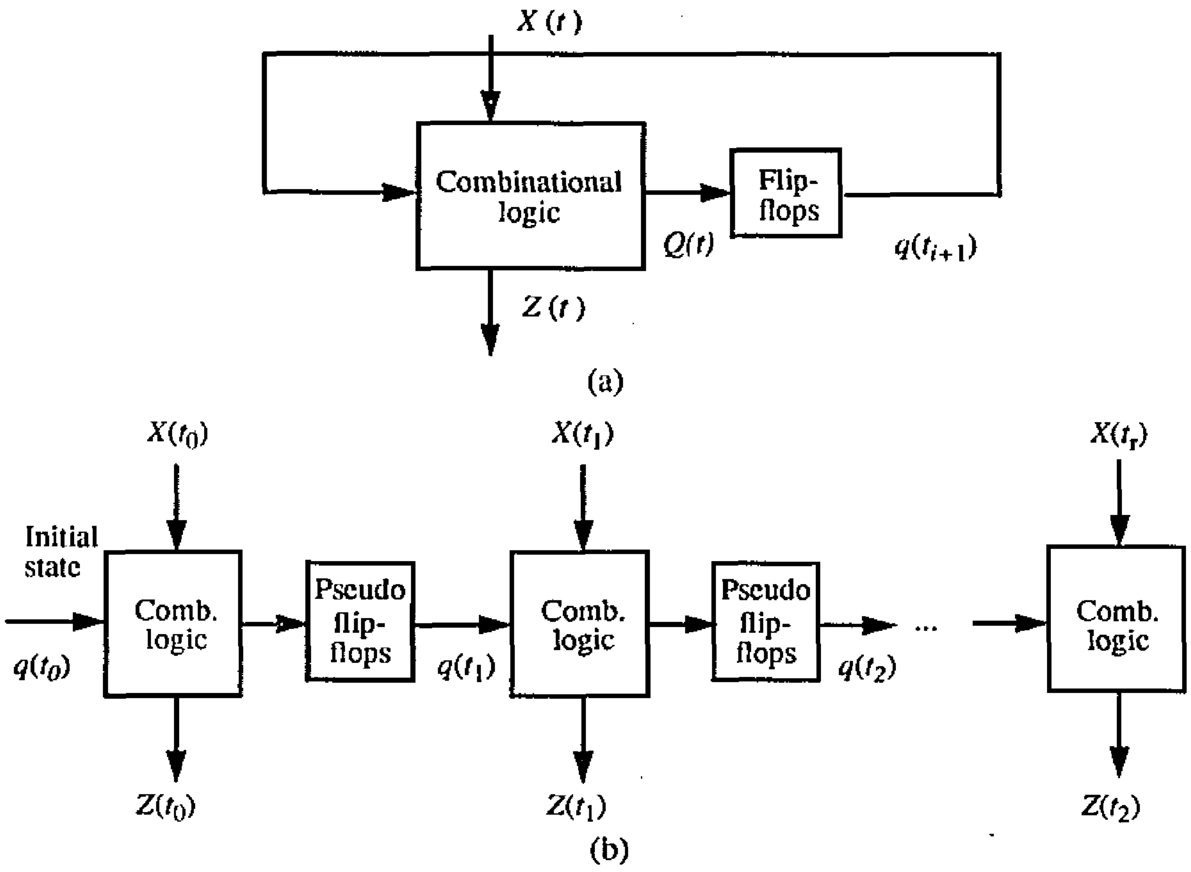


Figure 1.6. (a) Finite state machine and (b) an equivalent pseudo-combinational model.

ation is to construct an iterative model (Figure 1.6b) with r timeframes and execute a combinational test generator, ignoring the Z outputs in the first $r-1$ timeframes and the final state output $q(t_r)$. If the test generator cannot generate a test for the circuit in r timeframes, add a timeframe and begin test generation again [2].

If $q(t_0)$ is unknown, then the algorithm must justify objectives backwards in time, as well as propagating error signals forward in time. The *reverse-time* processing approach, exemplified by the extended backtrace (EBT) algorithm [67], avoids this complication by processing gates strictly backwards through the circuit and backwards in time, determining events which must come last, then next to last, etc. It starts at a primary output and follows a predetermined path to the site of a fault chosen for testing. All necessary assignments for one instance of time are determined before moving to the next earlier instance.

Another approach to sequential test generation is to randomly generate vectors from a given seed or set of seeds and fault-simulate them to determine their fitness for a test program according to a given cost metric. This approach is typified by the following algorithm [5]:

1. Create a set of trial test vectors and simulate using a fault simulator (discussed below).
2. Evaluate the trial vectors according to some cost function.
3. Select "optimal" trial vectors and add them to the test sequence.

In general, test generation for gate-level sequential circuits is not very well developed. This is an area where hierarchical approaches could make a major impact because most VLSI circuits are sequential. Gate-level techniques are mainly useful for small circuits of a few thousand gates, unless the circuits are modified to make them testable. We will discuss this further below.

Complexity Issues. Test generation is well known to be a difficult practical problem and a number of theoretical results support this conclusion. The fault detection problem, that is, the problem of computing a test to detect a given SSL fault, is NP-complete [50] for combinational circuits. The lower bound complexity of standard search-based test generation techniques is $\Omega(N^2)$ [41], and $O(N^3)$ in the average case [100], where N is the number of modules in the circuit. Finally, the typical approach (described above) of extending gate-level combinational algorithms to handle sequential circuits has worst-case complexity $O((2^N)^{2^{m+1}})$, where m is the number of state variables [16]. We see from this that if all else is equal, reducing the number of modules in the circuit by increasing the abstraction level should decrease test generation complexity.

Fault Simulation. Faults are simulated during test generation to determine the coverage of the current test (line 6 in Figure 1.4) and to reduce the number of tests needed. The process of analyzing coverage using a fault simulator is called *fault grading*. A fault simulator models faulty versions of a circuit as well as an unfaulted "good" version. If there are N faults to be considered, then the fault simulator simulates $N + 1$ different circuit responses to a given input in one simulation pass. The outputs from the simulation pass due to the N faulty circuits are compared with the one good circuit, and faults associated with incorrect circuit outputs are detected.

Usually, a test generated for a particular fault in a circuit will detect a number of other faults serendipitously. Fault simulation is not an essential step in test generation, and many test generators do not include it. However, fewer tests are generated when a fault simulator is used, which tends to speed up generation of a complete set of tests and results in more compact test sets.

On the other hand, fault simulation can be computationally expensive. A few hierarchical fault simulators have been developed [77], but these have not significantly reduced complexity in the general case and so are not widely employed. Fault simulation is most often included within test generation algorithms for combinational circuits [99]. By focusing on a limited range of circuits, these fault simulators achieve significant performance improvements over more general fault simulators, and thus are acceptable for inclusion in the test generation algorithm. This is a recurring theme in CAD: generality must be balanced with efficiency.

1.2.4 Design For Testability

A number of logic design techniques facilitate testing. For example, avoiding logical redundancy, providing for direct initialization of memory devices, and providing a mechanism for logically breaking global feedback loops, all make testing easier. Many companies compile long lists of such design for testability (DFT) rules. However, this ad hoc approach adds considerably to the designer's burden and may still not provide satisfactory testability.

A contrasting approach is systematic DFT. The basic idea of the most common systematic technique, scan design [25], is the separation of memory modules from combinational modules during testing. Memory modules, e.g. flip-flops, are chained together into a shift register or "scan chain" when a special test mode is activated. This partitions the circuit into a set of combinational subcircuits C_i, each of whose inputs and outputs are connected to the scan chain; see Figure 1.7. Each combinational subcircuit can be tested by shifting its input test data into the scan chain through the scan input X_S, letting the data ripple through the logic in parallel, loading the results in parallel into another part of the scan chain, and scanning the data out serially through Z_S. The scan chain itself can be tested by shifting a pattern of 1s to fill the chain, followed by the same number of 0s, and then the sequence 0101... Tests can be generated for the combinational subcircuits using any combinational test generator, since the subcircuit inputs and outputs may be treated as primary. Therefore, connecting all memory modules of a circuit into a scan chain obviates the need for sequential test generation.

There are many variations on the basic theme of scan design [7, 35, 92]. The most important design issue is whether to include all flip-flop elements in scan chains (full scan) or only some of them (partial scan). Partial scan is used to break feedback loops and to provide access to hard-to-test modules, but sequential test generation is still required to generate tests in this case. The use of scan design greatly simplifies test generation; however, scan elements add to the area of the chip

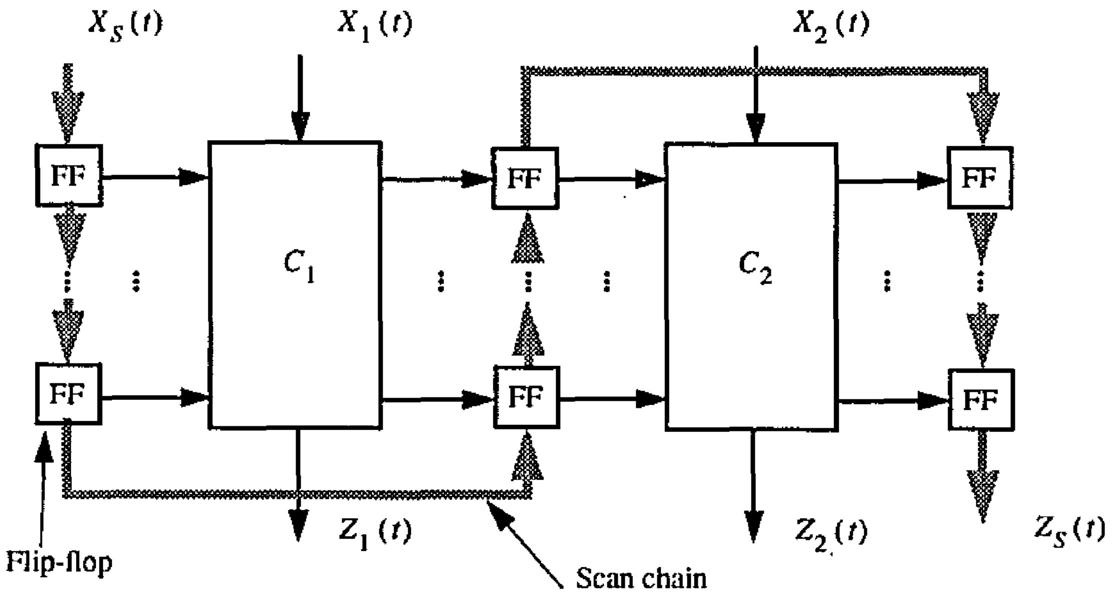


Figure 1.7. Scan testing scheme.

and cannot be used for dense memory modules such as RAMs. Very long scan chains can also be slow to load and unload during testing.

Hierarchical approaches to design for testability are just beginning to appear [9, 12, 19, 26, 51, 61, 73, 82]. Some of these identify registers for partial scan by analyzing behavioral descriptions [19], while others seek to modify behavioral descriptions to avoid optimizations that create unnecessary loops [61]. Bhattacharya studied modifications to individual register-level modules to improve their testability [12]. Finally, several hierarchical DFT techniques seek to ensure direct controllability and observability of every module [9, 26, 51, 73, 82]. Since abstract high-level representations of circuits are developed first and gate-level details are added later, hierarchical testability techniques can be introduced early in the design process. Moreover, since one module of a high-level model can potentially include many gates, a single testability feature in a high-level model can benefit a large number of gates.

1.3. Hierarchical Testing

We have given a brief review of classical (low-level) test generation theory and indicated its computation deficiencies. We have also indicated some places where we think hierarchical techniques can be used. Now we will summarize the more important hierarchical techniques that have been proposed, most of which simply address test generation. These range from well-defined extensions of gate-level algorithms to largely heuristic approaches influenced by research in artificial intelligence [13].

High-level Models. Several of the proposed hierarchical techniques model the circuit in terms of high-level functional blocks interconnected by single-bit lines. The fault models can allow for arbitrary faults in these blocks, and the test generation algorithms used may be simple extensions of the classical ones described above. Somenzi et al. [90] described such a technique based on the D-algorithm, Chandra and Patel [18] proposed a similar technique based on PODEM, and more recently, SOCRATES [84] has included higher-level primitives.

The fundamental advantage of higher-level modules is that there are fewer of them in the circuit to evaluate. However, the use of low-level (single-bit) interconnections between high-level modules negates this advantage as the space used to store propagation information is exponential in the number of bits, causing correspondingly long processing times. In addition, the algorithm has an exponential number of choices at each module, so there is a potential for excessive backtracking. This behavior was empirically observed by Chandra and Patel [18].

Bhattacharya and Hayes defined a test generation methodology that uses high-level interconnections (buses) and fault models in addition to high-level modules [12]. Faults in this model affect all bits of a bus. A bus is *totally stuck-at*-0 if all bits are stuck at logic level 0, and *totally stuck-at*-1 if all lines are stuck at logic level 1. An extended version of PODEM called VPODEM assigns vectors to buses to propagate these bus-level faults through the circuit model. The circuit and fault models, as well as the test generation algorithm, reduce to classical ones if components are restricted to single gates, and bus sizes are restricted to one, thus providing a truly hierarchical test generation method for large circuits.

The approach taken by VPODEM is especially suited to regular circuits like iterative logic arrays. In many such cases, it can be shown that a test generated for a total bus fault in the high-level model is guaranteed to detect all SSL faults on corresponding lines in a gate-level model of the circuit. Experiments conducted with medium scale ICs [12] suggest that tests generated for total bus faults in the high-level model detect more than 70 percent of the SSL faults in the corresponding gate-level model. The number of tests so generated is typically less by a factor of n, where n bits is the main bus size, than might be required to achieve the same fault coverage using gate-level models alone. The smaller number of tests combined with the reduced component count in the high-level model lead to a reduction in the total test generation effort, also by a factor of about n compared to standard techniques. Moreover, using VPODEM and a gate-level model of the same circuit, we can still obtain 100 percent SSL fault coverage by generating tests for SSL faults not detected by tests for total bus faults.

The circuit model used by Bhattacharya and Hayes [12] does not correspond directly to the circuit model as entered into a design system, so some work is required to construct the higher-level testing model. Moreover, the approach does not take full advantage of function and data abstractions in the original circuit, that is, its inherent high-level function.

Functional Approaches. Another class of test generators called "functional" test generators check for incorrect operation of high-level functions. For example, Thatte and Abraham proposed a high-level test generation scheme based on a graph model of the circuit under test [94]. Their method was primarily designed for microprocessors and programmable circuits of similar nature. Nodes of the graph are registers, and a directed edge I is inserted from node R_i to node R_j if the circuit can perform a register-transfer operation of the form $I{:}R_j \leftarrow R_i$, that is, function I maps values from register R_i into values in register R_j. Faults in this model typically represent erroneous data transfers; other fault types can be difficult or impossible to represent.

Other approaches use functional descriptions of the circuit based on HDLs [62, 64, 75]. Levendel and Menon model faults as D's injected into the variables of a HDL description [62]. Because faults are modeled at such a low level, the advantages of the higher-level circuit model are somewhat offset, as we have seen before. Lin and Su model a variety of faults, including incorrect instruction decoding, and incorrect register transfers. [64]. Recently, Rao et. al. [75] have proposed a similar approach for VHDL models. In their approach, as well as that of Thatte and Abraham [94], the faults are not described by well-accepted models, and the relationship of the faults to standard models is hard to quantify.

Artificial Intelligence-Based Methods. Finally, we will discuss three methods based on heuristic principles derived from artificial intelligence (AI). Saturn is a test generator with a strong focus on design hierarchy [89]. The circuit model that it uses has information about the structural hierarchy

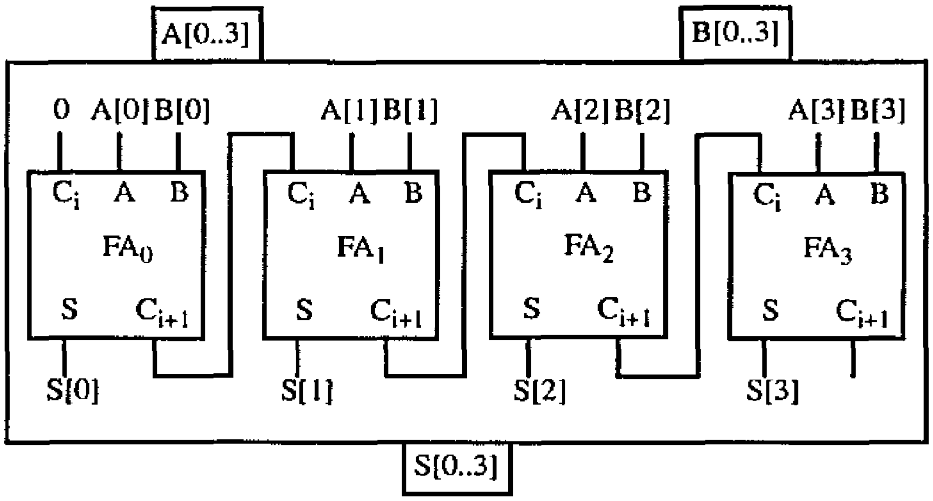


Figure 1.8. Four-bit adder as an example of hierarchical testing.

and also about the rules of circuit behavior at the various levels of abstraction. For example, the adder in Figure 1.3b is described structurally as composed of four full adder modules with appropriate connections. It can also be described in Saturn as a module performing the high-level functional operation $S[3..0] = A[3..0] + B[3..0]$. Similarly, the full adder modules are also described both behaviorally and in terms of their internal structure. Saturn will always attempt to propagate values through modules at the highest level of abstraction for which it has a model.

The test generation algorithm used by Saturn resembles the D-algorithm. It operates in bottom-up fashion by first generating tests for faults on gates in the circuit. When testing a gate, values are propagated to the boundaries of the gate's parent in the next highest level of the design hierarchy, then abstracted to the behavioral level of the parent.

As a demonstration of this bottom-up test generation philosophy, consider the four-bit adder of Figure 1.3b, which is repeated in Figure 1.8. To generate a test for this circuit, Saturn first generates a sequence of tests for all faults in one of the full adders, justifying internal signals only as far as the full adder inputs and propagating fault effects only as far as the full adder outputs. This test is then stored in a library for future use in testing full adders anywhere in the circuit.

Singh's work on Saturn was innovative and several test generators developed later used similar techniques, however, Saturn's performance on the few small examples cited was modest. An extension to Saturn called PF-TG (Program Fragment Test Generator) was developed by Shirley at MIT [88]. PF-TG generates tests by merging statements from precomputed tests, stored as fragments of test programs complete with loops and conditionals, into a test for the whole circuit using automatic program-writing techniques developed in AI research [8]. Its overall algorithm and circuit model are the same as Saturn's, but propagation and justification through modules is accomplished explicitly by knowledge stored in a library. The technique is completely heuristic, and dependent on the test programmer's skill in writing the test fragments for control and observation. However, the resulting test has the characteristics of a test program and can therefore take advantage of tester features unused by typical test generation algorithms.

Like Saturn and PF-TG, the circuit model for DB-TG [88] is described hierarchically and uses knowledge to constrain the search space. In contrast to PF-TG, however, this knowledge is not directly contributed by the user. Rather it is derived from earlier simulations of the circuit, using a symbolic simulator. During test generation, DB-TG relies on data it recorded earlier. A symbolic test value R is determined to have been propagated when it appears in unmodified form at a primary output. Finally, the symbolic values are replaced with sequences of precomputed test values which are required to test the module. As in the case of PF-TG, the approach is based almost entirely on heuristics, and DB-TG was tested only on one small circuit. Finally, as noted above, symbolic simulation for design verification is not part of the design cycle for most companies, nor does there appear to be a trend toward using it.

Finally, Krishnamurthy [57] used AI techniques to describe fault propagation and line justification methods for each module in a design hierarchy to improve the performance of the conventional D-algorithm. This approach has subsequently been used in a number of experimental hierarchical test generators, notably SOCRATES [84].

Precomputed Tests. A new, but growing class of test generators use hierarchical techniques as described above, but focus on the ability to justify precomputed tests for modules and propagate module test responses to outputs where they can be observed. This important capability is the main strategy in a number of experimental test generators [9, 58, 59, 68, 88, 93]. Currently evolving design styles, which rely heavily on CAD tools, are making this capability very desirable. Designers often reuse modules that have been stored in a library, and many of these modules do not have accurate gate-level models for test generation purposes. However, when they were designed, tests were generated for the modules and stored for future use. Testing using such precomputed tests

Module	Estimated number of gates	Estimated number of gate evaluations	Number of high-level modules	Number of module evaluations by *PathPlan*
Fltrdp	254	64,516	6	57
Vertdp	138	19,044	5	21
Rowdp	220	48,400	6	33
Alu	62	3,844	9	167
Progptr1	110	12,100	12	210
Progptr	156	24,336	6	68

Table 1.1 Performance of *PathPlan* relative to gate-level test generation.

therefore augments and complements well-established lower-level test generation schemes. In cases where precomputed tests may be easily applied to modules, it is advantageous to use them, especially when the modules are large. Not only is the test generation effort reduced, fault simulation for fault grading is minimized.

We developed an algorithm for test generation using precomputed tests and implemented it in a tool called *PathPlan* [68]. This program is one of the earliest to specifically use this approach. In *PathPlan*, precomputed tests are represented symbolically and propagated through modules in the circuit model. The test generation algorithm used is loosely based on the D-algorithm. We will discuss the design of *PathPlan* in detail in Chapter II.

Since the circuit modeling level is much higher than the usual gate level of the D-algorithm, the number of components is substantially reduced. Moreover, the number of backtracking choices available at each module is also greatly reduced. Some performance results for *PathPlan* are shown in Table 1.1 [68]. The modules in these circuits are implemented by module generators which produce layouts rather than netlists of gates. However, we can estimate the number of gates in each module and therefore in each circuit. As noted above, the lower bound complexity of test generation is $\Omega\left(N^2\right)$, where N is the number of modules. For instance, to generate a test for the gate-level version of Fltrdp (254 gates) using conventional techniques would require $64516c_1$ gate evaluations for some constant c_1 that depends on the test generator. With the same assumptions, *PathPlan* has the significantly lower bound of $36c_2$ on module evaluations for some constant c_2. Since the complexity of a module evaluation in *PathPlan* is similar to that of a gate-level algorithm, we will assume that c_1 and c_2 are within the same order of magnitude. The actual performance of *PathPlan* for Fltrdp is 57 module evaluations (see Table 1.1). Thus we see in this instance a potential speedup of perhaps three orders of magnitude over conventional test generation. Performance results for most other circuits in Table 1.1 are similar.

An approach similar to, and partly based on *PathPlan* has subsequently been implemented by Mitsubishi and used to test several circuits [11]. Despite this success, *PathPlan* is a preliminary system with a limited ability to propagate high-level signals. The propagation techniques used by *PathPlan* cannot handle general reconvergent fanout, or any irregularities in bus structure. In a circuit with a regular bus structure, all buses in the primary data path have a constant width n; no buses are truncated to smaller sizes.

Since the initial development of *PathPlan*, several other experiments in test generation using precomputed tests have been published. In [9], Beenker et al. of Philips describe their work on a test generation approach that relies heavily on special DFT techniques. The SPHINX tool box (later renamed Panther and marketed commercially) works in concert with the design system. Many modules implemented by the design system have precomputed tests, and all latches, registers and flip-flops employ full scan. Modules are grouped hierarchically to form "execution units" (EUs). Each EU is directly controllable and observable via buses and test registers and contains a small test controller module that controls the scan chains and routes tests to modules within the EU. The EUs themselves are grouped into "processors". Each processor also has a test controller that controls the propagation of tests to and from EUs and the test controllers within the EUs. Panther provides the tools to implement the scan chains and test controllers hierarchically. When the design is completed, every test stimulus vector can be applied directly to every module via scan chains and buses. Similar techniques were also proposed in [26, 51,73, 82]. Propagation of test information through other modules is minimized; Panther is not designed to propagate the precomputed tests through modules other than scan chains.

No new general principles of hierarchical test generation are developed in Panther. Test generation becomes a matter of scheduling tests to be applied. In [9], Beenker et al. describe how their approach is applied to an error-correction circuit with 225,000 transistors [103], thus further demonstrating the practicality of precomputed tests for commercial circuits. The performance of Panther in testing this chip is impressive. The test for the entire chip was generated in about 2 hours, which includes the time (80 minutes) to hierarchically generate tests for two programmable logic units which are then used as precomputed tests. It is claimed [9] that the DFT logic adds only about 8 percent to the chip area, however, the design style enforced by the CAD system naturally provides high controllability and observability and is not practical for all applications. Panther is tightly coupled to the design style, and the test generation tools assume that each individual module is directly accessible.

Su and Kime [93] have developed a tool called HPath for sensitizing multiple (multi-bit) paths in a hierarchical circuit. These paths deliver precomputed test data from primary inputs to module inputs and propagate test responses from module outputs to primary outputs. The HPath algorithm is based on heuristics, and contains two subfunctions: GPath, which is a symbolic version of PODEM designed to find sensitized paths in modules with gate-level models; and FPath, which, like *Pathplan*, finds paths through high-level circuits by using rules stored in libraries about propagation through modules. Hpath suffers from a similar inability to propagate error information through circuits with an irregular bus structure. Since HPath primarily uses heuristics, very few general principles are developed in [93] that might lead to the development of more advanced hierarchical test generators. In [93], Su and Kime report the average time required to sensitize paths to modules in several circuits, but this information is not compatible with other benchmark data.

Finally, Lee and Patel have reported on two versions of a test generator called ARTEST that they have developed for testing using precomputed tests [58, 59]. They assume that circuits are composed of two parts: a datapath containing large modules with precomputed tests, and a control unit which provides control signals to the datapath and is composed of gates and flip-flops. Each part has a separate test generation algorithm. The datapath is tested using hierarchical techniques. Lee and Patel assume that the exact description of the error signals associated with each module test is unknown. Therefore, when a stimulus vector v is propagated to the inputs of a module under test (MUT), the output is marked as "type faulty" if v matches a precomputed test vector for the MUT, or "type good" if v does not match any test vector. We will discuss this "typing" approach to error propagation further in the next chapter and in Chapter IV.

In the first version of ARTEST [58], the datapath is tested using a hierarchical algorithm similar to PODEM. Each test vector is justified individually; no attempt is made to propagate tests symbolically as in *PathPlan*. In the second version [59], a relaxation algorithm is used to justify internal signal objectives such as precomputed test stimuli. Symbolic expressions with undefined variables are propagated from the inputs of the circuit. This creates a system of equations which must be solved individually for each test vector to be applied to the MUT. Both versions of ARTEST use conventional gate-level techniques for testing the control unit. Faults propagated to the interface between the control unit and the datapath are propagated as high-level error signals (types) by the hierarchical test generator.

Lee and Patel have evaluated ARTEST's performance on small and medium sized examples (a few thousand gates). In [58], they compare the performance of the first version of ARTEST to a gate-level test generator HITEC [72]. Using both programs, they generate tests for a version of the Am2910 microprogram sequencer [3]. ARTEST uses 61.75 CPU seconds to generate a test compared with 2,297 seconds for HITEC—a speedup factor of 37.6. ARTEST performs well for several circuits that contain global feedback loops and reconvergent fanout. However, all circuits tested have a very regular bus structure. ARTEST cannot generate tests for circuits with an irregular bus structure, even if it is possible to successfully propagate the precomputed tests. More effective hierarchical error propagation techniques are essential to the development of testing methods for general circuits.

1.4. Summary and Thesis Overview

We have reviewed classical testing theory and practice, and have the following observations.

- Test generation for ICs continues to be an important and difficult problem. The changing nature of technology and design styles ensures that the problem will never be permanently solved. Present techniques are already inadequate in many cases and the problem is getting worse.

- Simulation and synthesis of digital systems increasingly take place at very high levels of abstraction. Hierarchical test generation tools provide an opportunity for addressing the testing problem at a point in the design process when changes and trade-offs are most easily made.
- Conventional test generation techniques are based on the SSL fault model. Other fault models such as bridging faults between transistors may be more accurate, but cannot be handled by conventional techniques. In addition, some types of circuits, such as RAMs and ROMs cannot be effectively tested using the SSL fault model.

Hierarchical testing techniques show great promise for improving test generation performance and for matching test generation tools to evolving computer-aided design styles. However, current techniques have many drawbacks, and so have not been widely implemented. Some of these techniques make use of the hierarchical modules of typical circuits, but retain a bit-level interconnection structure and related fault models [18, 90]. Thus, they do not realize the full potential of high-level error propagation because of the low-level interconnection structure. Other hierarchical testing techniques use higher-level models and bus-level interconnection structures, but use high-level fault models that are difficult to relate to more precise and well-accepted models [12, 64]. Still other techniques are based almost entirely on heuristics, so no generally applicable principles of test generation have been developed for them [88, 89]. Finally, few of the proposed techniques attempt a systematic approach to design for testability, despite the fact that hierarchical techniques can be implemented earlier in the design process. In general, the field of hierarchical test generation is still in its infancy.

Hierarchical test generation techniques that can use precomputed tests [9, 58, 59, 68, 88, 93] have two particular advantages: (1) tests can be generated using multiple fault models for the same circuit, including the SSL fault model, as well as more accurate and technology-specific fault models; (2) tests that exist for previously designed modules can be reused when it is inconvenient or impossible to regenerate tests for these modules in the circuit that contains them. Moreover, our initial test generator *PathPlan* demonstrates the possibility of significant performance advantages over conventional test generators. Nevertheless, all current hierarchical test generators that use precomputed tests are limited by their ability to generate tests for circuits with an irregular bus structure.

In this thesis, we present the theory and tools we have developed for generating tests for circuits using precomputed tests for modules. Our test generator *PathPlan*, which is widely cited, was one of the first automated CAD tools for this type of test generation. Since error propagation over irregular buses is a limitation of *PathPlan* and subsequent published tools, we have concentrated on research to develop improved error propagation techniques. To accomplish this task, we have formulated a theory of information propagation in bus-structured circuits. Some aspects of the theory are applied to the development of a new test generation tool *MATSim*, which analyzes error propagation. *MATSim* is incorporated in *PathPlan2*, a successor to *PathPlan*, which is not limited to circuits with a regular bus structure, but still generates tests with the same high performance achieved by *PathPlan*. Throughout the research reported here, we have sought to develop general theory and techniques that may be used by others, as well as implemented in our own tools.

In Chapter II, we discuss test generation using precomputed tests in more detail and contrast it with conventional techniques. We motivate the use of precomputed tests, and illustrate the problems of test propagation. Finally, we describe *PathPlan* as a demonstration of a test generator that uses precomputed tests, and identify the key extensions that are developed in the remainder of the thesis.

Chapter III presents a new, general theory of test data propagation, in which the information propagation characteristics of modules and circuits are succinctly represented algebraically. The theory formalizes intuitive notions of information propagation, and also allows us to identify some unexpected characteristics of propagation through modules. We use it to analyze circuits with irregular buses and prove several theorems relating the effect of module and circuit structure to information propagation.

Chapter IV addresses the problem test data propagation during hierarchical test generation. We discuss how test data can be propagated as symbolic expressions and matched to precomputed test stimulus sequences at the inputs to embedded modules. We show how to hierarchically analyze the propagation of errors produced by embedded modules being tested, using methods based on the propagation theory introduced in Chapter III. Finally, since many circuits cannot be adequately tested by precomputed tests, we propose a new design approach to modify them to improve testability. This also uses theory the developed in Chapter III, and augments traditional DFT approaches such as scan.

In Chapter V, we describe the design of our new test generation tools *MATSim* and *PathPlan2* that implement the propagation techniques covered in Chapter IV. They are intended to extend the capabilities of *PathPlan*, but are not based directly on it; they are completely new tools. *MATSim* provides the error propagation capability missing from previous work. It can be used manually or incorporated into automatic test generation tools such as *PathPlan2*. It thus provides a foundation for more advanced tools to automate test generation for future ICs. We present some experimental results showing that while *PathPlan2* is a more powerful and general program than *PathPlan*, its performance is at least as good.

Chapter VI summarizes the research described in this thesis and discusses future research in propagation theory and testing using precomputed tests for modules.

CHAPTER II
PRECOMPUTED TESTS

In this chapter, we introduce the method of testing using precomputed tests for modules and compare it with conventional techniques. We illustrate the method using some example circuits and introduce key concepts and terminology used throughout the thesis. Finally, we describe the design of *PathPlan*, our initial test generator that uses precomputed tests.

2.1. Precomputed Versus Conventional Methods

The defining assumption of precomputed test methods is that a library of tests exists for every module in the circuit under consideration at some level of abstraction. In most cases, these modules are much larger than logic gates. Each module's library test covers some set of faults in the module, such as the set of all SSL faults. The goal of precomputed test methods is to combine the tests for all the individual modules into a test for a multi-module circuit. This contrasts with conventional methods, where the goal is to cover a set of faults. We are interested in precomputed test methods that use a structural circuit model, such as a netlist, and accomplish their goal by justifying a ***module test stimulus sequence*** T_S at the inputs of each module under test (MUT) in a given circuit, and propagating the corresponding ***module test response sequence*** T_R from the outputs of the MUT to the primary outputs of the circuit, as illustrated in Figure 2.1. T_R may be propagated directly to a primary output or to some other observation point, such as a scan path. In Figure 2.1, T_S is justified to a primary circuit input where it is denoted CT_S for circuit test stimulus sequence. The response T_R is propagated along two paths to primary circuit outputs. The responses at the outputs are denoted CT_{R1} and CT_{R2} for circuit test response sequence. Note that

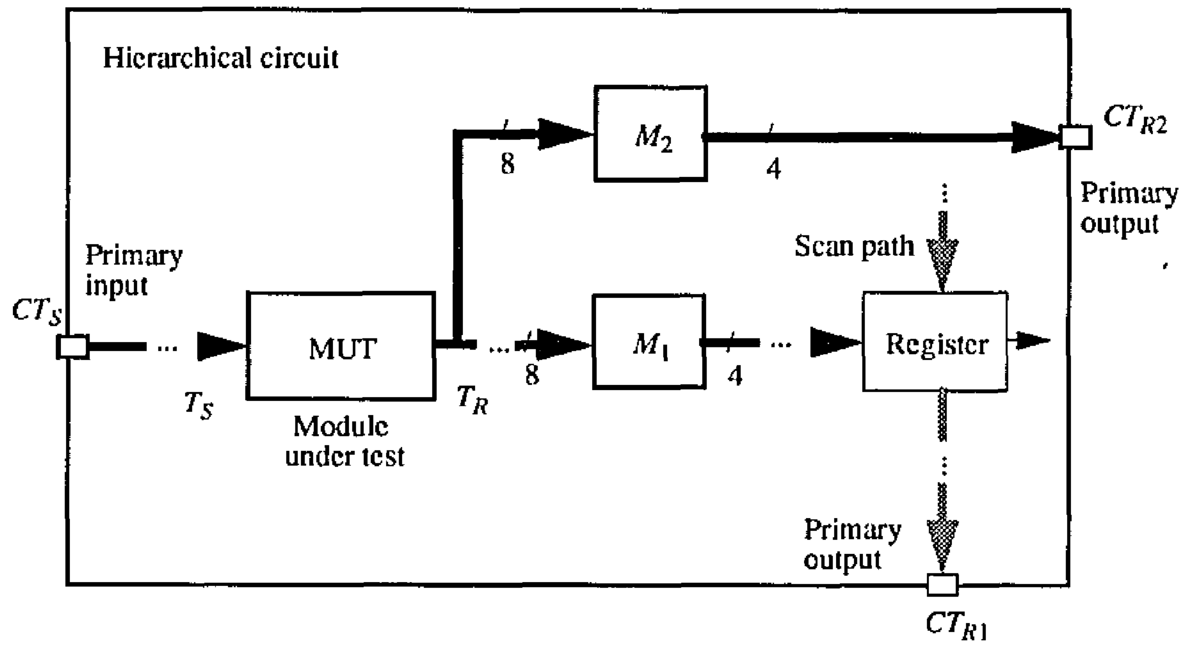


Figure 2.1. Propagating and justifying module tests in a circuit.

T_R is propagated through modules M_1 and M_2, both of which have output buses smaller than their input buses. This is an example of an irregular bus structure. The test generator must select each module in turn to be the MUT and justify T_S and propagate T_R until all modules are tested, or until it is proven that they cannot be.

Precomputed test methods and conventional methods differ in the nature of their objectives. Recall that when testing for SSL faults, for instance line l of the MUT stuck-at-v, the initial objective is set l to $\bar{v}$. In precomputed test methods, the initial objective for the test generator is to apply the set T_S of test stimuli to all inputs of the MUT. During test generation, values propagated to the inputs of the MUT are compared to T_S and if there is a match, T_R is propagated from the output of the MUT. Clearly, applying T_S is a more complex objective than setting $l = \bar{v}$ since several MUT input ports are usually simultaneously specified as a group or vector. Moreover, although faults in the MUT may be covered by many different test vectors, in precomputed test methods vectors that do not match T_S are ignored. This leads to the main drawback of precomputed test methods: various constraints imposed by the circuit containing the MUT may prevent either T_S or T_R, or both from being propagated, despite the fact that all non-redundant faults in the MUT could be covered by different tests.

Nevertheless, a basic assumption in the use of precomputed tests is that tests for faults within modules must be reused and cannot be regenerated in the context of a larger circuit. Our goal is to ensure that precomputed test data are correctly and efficiently propagated if they can be. When they cannot be successfully propagated, then fault coverage will be reduced, unless the circuit is modified to improve testability.

Another characteristic feature of precomputed test methods is that faults within the MUT produce arbitrary multi-bit errors at the outputs of the MUT. Due to fanout within the MUT, a single fault can produce errors at more than one module output bit, and each vector in T_S can sensitize more than one fault, causing multiple single-bit errors. Therefore, when the MUT contains a fault, an individual test response vector v_i in T_R can assume an arbitrary error value $v_i^c \neq v_i$. For a given MUT, fault model, and test stimulus sequence T_S, there may be many possible error values. When generating a test, we must ensure that every v_i^c produces a different value than v_i at a primary output, that is, propagate the error. In contrast, to control complexity most conventional test generators consider only a small number of the possible error values. For example, VPODEM [12] uses only two fixed error values: the all-zero vector (due to a bus totally-stuck-at-0) and the all-one vector (due to a bus totally-stuck-at-1). These errors are produced as a result of assumed faults on lines and buses according to the VPODEM fault model. Such simplifying fault model assumptions are unacceptable for testing using precomputed tests.

Finally, most precomputed test methods use hierarchy and signal abstraction because the circuits for which these techniques are appropriate contain several large modules, often interconnected by a well-defined bus structure. In addition, the multi-bit objectives and multi-bit error signals are more efficiently propagated over buses. Efficiency in propagation is important because precomputed test sequences for large modules can be quite long.

Hierarchical representations of signal values allow precomputed test methods to take advantage of the bus structure in a circuit to improve efficiency. If we model module connections using single-bit wires, each carrying signals independently of adjacent wires, then signal propagation must take place at this same low level. The performance advantage frequently obtained by avoiding testing within modules is at least partially offset by the complexity of simultaneously, but independently propagating many individual intermodule signals. On the other hand, circuits whose modules are interconnected by large regular buses, which can be modeled as monolithic signal-carrying objects, can be tested with good performance, as demonstrated by *PathPlan*. Unfortunately, few circuits are composed solely of regular buses. As noted, buses are often truncated to smaller sizes, thus losing some of their ability to propagate information. Information can also be lost as signals are propagated through some types of modules. Such cases present nontrivial, but often surmountable barriers to high-level signal propagation.

We have defined the precomputed test method and contrasted it with conventional techniques. Next we motivate the method by means of some examples, and discuss the class of circuits that it can effectively test.

2.2. Using Precomputed Test Methods

Although they often improve test generation, the main reason for using precomputed tests is that circuits often contain modules that cannot be tested by classical techniques. Precomputed test methods are most appropriate for circuits that contain modules from two groups: (1) library modules, and (2) modules not tested using the SSL fault model.

The first group consists of modules designed and tested independently, and stored in a library. Many modern circuits are composed of modules that are built by separate design teams. For instance, a microprocessor may be partitioned into CPU, memory management unit, cache controller, I/O circuits, etc., each designed separately. Design teams responsible for a module often also have responsibility for generating tests for it. In many cases, the modules are so large that it is impractical to regenerate tests for them when they are interconnected at the next level in the design hierarchy; the original tests are reused. In other cases, the module library is sold to other designers as part of a design service. Customers of the design service include the modules in their circuits and use functional models of the modules to verify their design. However, the structure of a library module is proprietary and is often not provided in sufficient detail for test generation. Nevertheless, precomputed test data for the modules are not proprietary and can be provided with the functional models.

The principal modules in the second group, modules not tested using the SSL fault model, are embedded RAMs and ROMs. These modules cannot be tested by classical techniques and are usually treated as a special case when testing an IC containing them. Also in this group are modules tested using more accurate, technology-specific fault models, e.g. bridging faults between transistors. Such fault models are useful in developing better quality tests, but are so detailed that test generation is impractical for typical VLSI circuits. These detailed fault models can be used to generate tests for small submodules, and precomputed test techniques can be used to compose the module tests into a test for the entire circuit.

An example of a circuit that is relatively easy to test using precomputed test methods is shown in Figure 2.2. This circuit was generated by AutoCircuit, a high-level synthesis tool [32] being developed at General Motors Research and Development Center and based on the System Architect's Workbench from Carnegie Mellon University [95]. The circuit Encode is part of a special-purpose communications chip. The modules in this circuit are synthesized by module generators. They are similar to library modules and gate-level netlists suitable for test generation are not available for all modules. Nearly all buses in this datapath have the same width (42 bits). The bus structure is very simple, while the total number of transistors is fairly large—67,000 for the entire chip, including four multi-port RAMs.

Each module in the datapath schematic of Encode is uniquely named PROCR_*i*, MUXR_*j*, or REGR_*k*, where MUXR_*j* is a multiplexer and REGR_*k* is a register. Most other functional modules are named PROCR_*j*. Below the module name is the width of the primary data input bus and the module type. The CONCAT module adds lines to a bus. Now consider the problem of testing MUXR_3 in Figure 2.2. The CONCAT modules (PROCR_3—PROCR_6) shift a constant value 0 or 1 into the low order bit of the inputs to MUXR_3 and MUXR_4. The value of this bit can be controlled by inputs t_1 through t_4. Therefore, the test vector set T_S for MUXR_3 can easily be justified through path (MUXR_1, PROCR_1, REGR_1, REGR_3, PROCR_3) and path (MUXR_1, MUXR_2, REGR_2, REGR_4, PROCR_4). The test response T_R can easily be propagated through (REGR_5, MUXR_5, MUXR_7, REGR_5). These two paths are transparent to the propagation of signals.

An example of a more difficult type of circuit that we would like to handle using precomputed test methods appears in Figure 2.3. This circuit is called Divfilt, and implements a digital filter containing about 2,900 transistors. Typically, several such circuits are combined to form an IC,

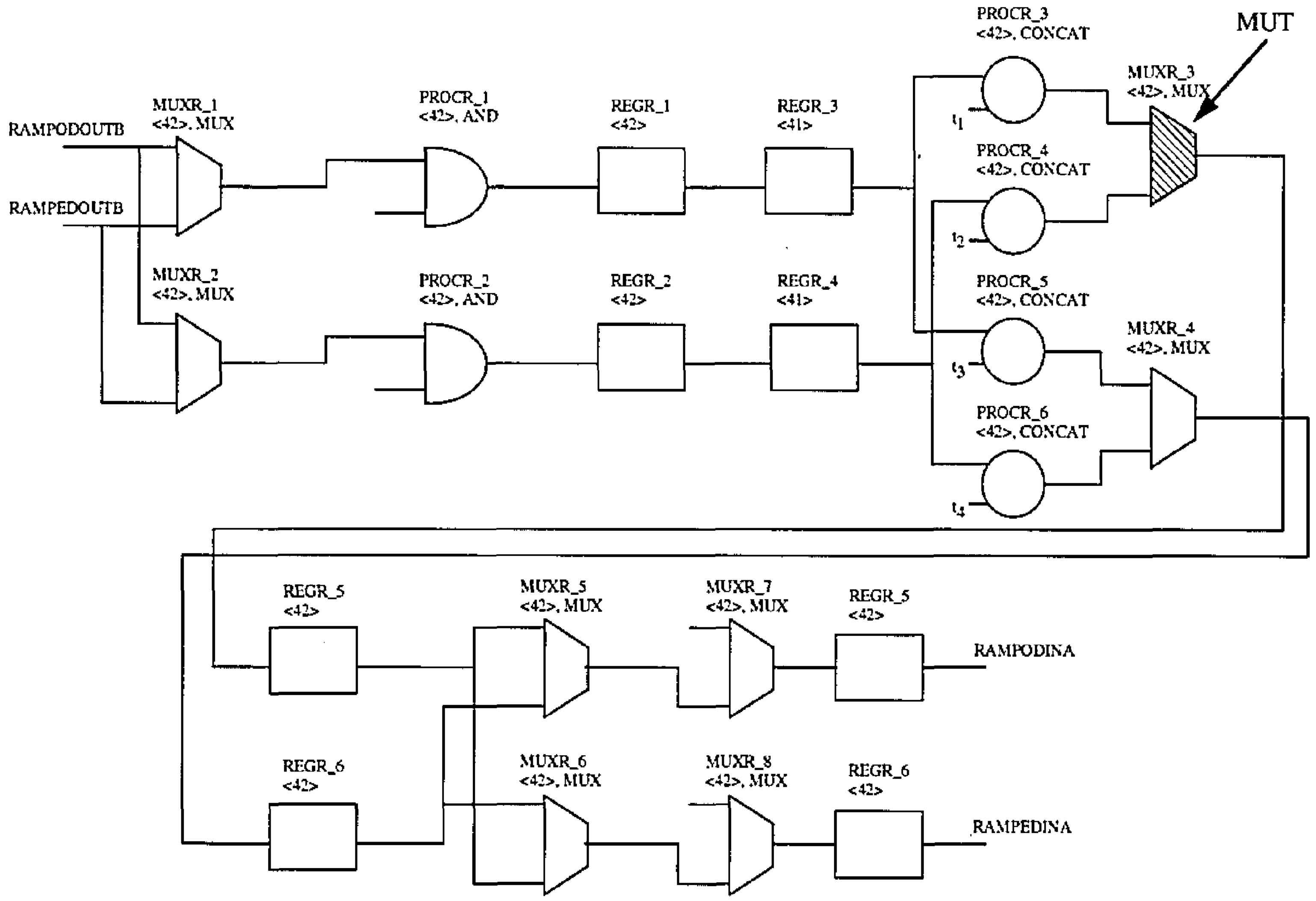


Figure 2.2. Example of a circuit Encode that is easy to test using precomputed tests.

or are integrated on-chip with a microprocessor. Like Encode, Divfilt was synthesized by AutoCircuit and contains primarily library modules. The FREAD modules in this circuit have the undesirable feature of truncating bus width. Note the large number (12) of FREAD modules out of a total of 51 modules. There are also several cases of reconvergent fanout, e.g. from the output of PROCR_25 to the inputs of PROCR_1. Finally, note that Divfilt is highly sequential; there are ten register modules and several feedback paths throughout the circuit. The complicated bus structure is the result of the optimizations used by AutoCircuit to reduce the number of modules. Many modules are reused in several different operation.

Despite the complexity of its bus structure, Divfilt does make extensive use of multi-bit buses. High-level propagation techniques can therefore be used for testing some modules. In addition, Divfilt's complexity is mitigated by the fact that relatively few types of library modules are used. AutoCircuit typically designs on-the-fly large sequential modules such as counters, from lower-level modules such as adders and registers, because the latter can be shared with other functions. Large modules with behavioral models are sometimes helpful in identifying circuit behavior useful for testing. For instance, the fact that a collection of modules are combined to form a stack circuit can sometimes aid test strategy development. However, it is easier to propagate errors in T_R through a small set of well-characterized primitive functions, as we will later show. The modules that AutoCircuit uses most are adders/subtracters, multiplexers, FREADs, CONCATs, ANDs, ORs, NOTs, RAMs, and registers.

Now consider the problem of testing module MUXR_1 in Divfilt. Its test response T_R will have to be propagated through at least 12 modules to reach a primary output (SEND_2), along a propagation path with significant loss of information (bits). This circuit is extremely difficult to test, either by conventional or precomputed methods. Many faults in modules cannot be detected because buses that carry the errors they cause are truncated, making it impossible to differentiate an error from correct circuit behavior. The IC defects that cause the faults can remain latent in the circuit and cause failures later on. Clearly, some circuit modifications will be needed to test Divfilt.

Microcontroller products offered by such companies as Motorola and Texas Instruments provide further examples of circuits with library modules. Precomputed testing methods are frequently used to test these circuits. Microcontrollers from Motorola contain CPUs (microproces-

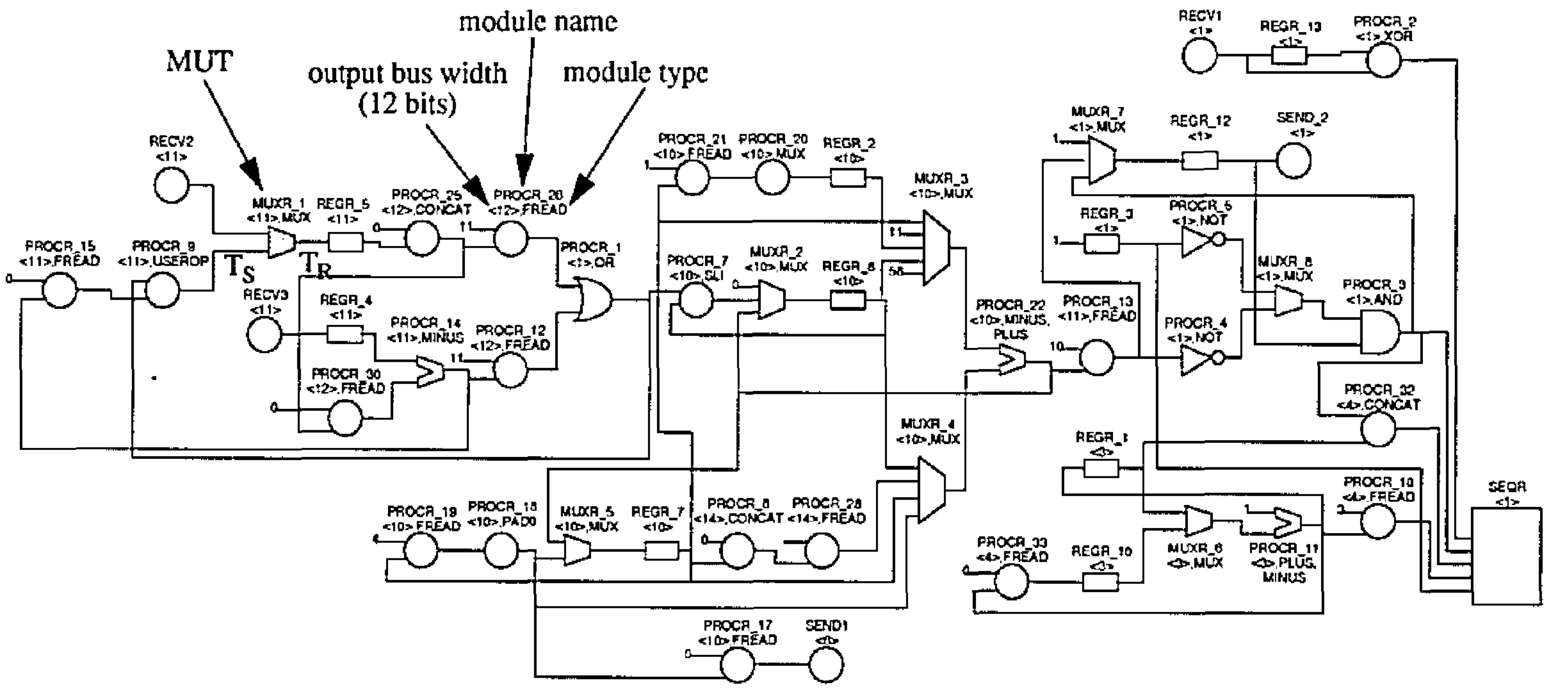


Figure 2.3. Example of a circuit with library modules Divfilt which is difficult to test using precomputed tests.

sors) such as the 68332 and 68HC11, as well as dozens of other modules such as RAMs, ROMs, I/O ports, and bus controllers. These modules are designed in several countries. They are stored in a library and connected together in different ways to construct different microcontrollers. Each module has associated precomputed tests and simulation models, and it is usually impractical for completely new tests to be written for the modules when they are composed into a microcontroller.

In general, since on-chip data and address buses provide good controllability and observability, microcontrollers are usually somewhat easier to test using precomputed tests than Divfilt. However, because a number of special purpose circuits are also included on-chip with the CPU, microcontrollers do have some irregular buses. Therefore, in this thesis, we will focus on the propagation of test information in circuits like Divfilt. Effective techniques for testing such circuits are also useful for dealing with circuits with more regular buses such as Encode or typical microcontrollers.

Now that we have defined the precomputed test method and illustrated the types of circuits that must be tested using precomputed tests, we turn our attention to developing our key concepts and terminology.

2.3. Propagating Precomputed Tests

In this section, we formalize circuit terminology we have used informally until now, and introduce the concept of module and circuit transparency—the ability to propagate T_R. We refer to all components in the circuit under test as *modules*. Signals enter and leave modules through input and output *ports*. All n-bit module interconnections are called *buses*, even when $n = 1$. If X is a bus, then $|X| = n$ denotes the *width* of the bus. Individual subsets of bus lines are described using standard array notation; for example, if $|X| = 8$, then $X = X[7..0]$. The most significant bit of an n-bit bus is $X[n\text{-}1]$. This is commonly referred to as Big Endian notation [21]. The signal value associated with a bus or port X is denoted $V(X)$ and the set of all possible values that can be assigned to X is denoted by $\{V(X)\}$. Specific values on multi-bit buses are represented as binary (marked with a subscript 2) or decimal numbers. For example, the bit pattern $V(X[2..0]) = 111$ is denoted 111_2 in binary and 7 in decimal. Subscripts are omitted when discussing single-bit values, since there is no ambiguity in interpretation. In most cases, we represent values as decimal

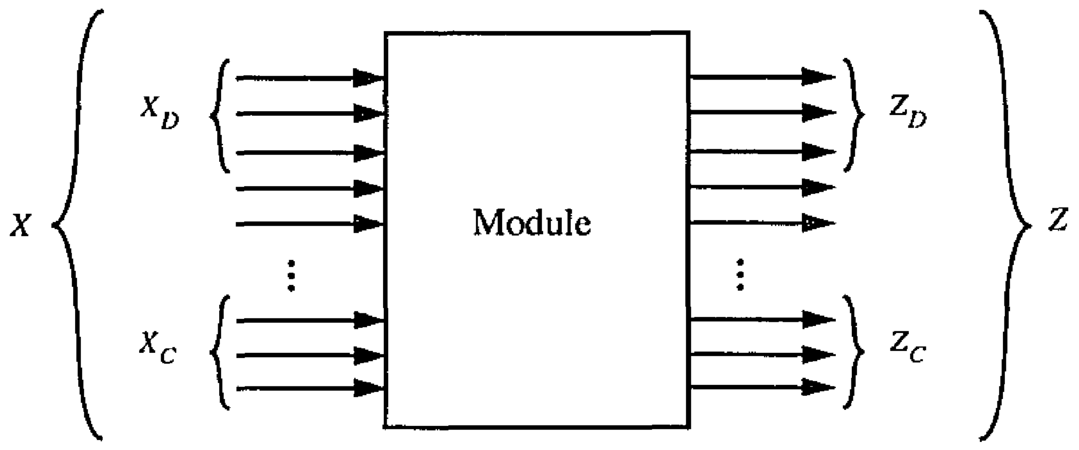


Figure 2.4. Notation for module input/output signals.

numbers. Circuits are directed graphs whose nodes are modules and whose edges are buses. Buses start at module output ports and end on module input ports. When considering a module in isolation, we may unambiguously refer to its ports and the buses attached to them by the same name.

Let M (Figure 2.4) be a module with input ports X and output ports Z. T_R is propagated from a subset of the input ports $X_D \subseteq X$ to a subset of the output ports $Z_D \subseteq Z$. Propagation is often controlled by a third set of ports $X_C \subseteq X - X_D$. A test stimulus set T_S is justified at Z_D by assignments to X_D and X_C. The buses X_D, X_C, Z_D, and Z_C are called respectively, the *input data bus*, the *input control bus*, the *output data bus*, and the *output control bus*, and $(X_D;Z_D)$ is a data-bus pair. We assume in our analysis of propagation that X_D, X_C, and Z_D may be freely chosen. Library modules, such as adders and multiplexers, often have standard X_D's, X_C's, and Z_D's based on functional considerations. However, when the modules are used in a circuit, the data and control buses suitable for testing purposes may differ from these standard configurations. Some typical datapath modules are shown in Figure 2.5. Each module is shown with a particular assignment of inputs to X_C and X_D, and outputs to Z_D. Figure 2.5d shows the usual assignment of buses for a multiplexer, while Figure 2.5e shows a less common but still useful assignment. Figure 2.5h shows a decoder with only two decoded values being used. This module is a version of a classic decoder module which has one output bit for each possible input value. Frequently, only a few input values need to be decoded, and VLSI circuits often have large buses which prohibit the use of complete decoders, so module generators are used to create small versions such as the module

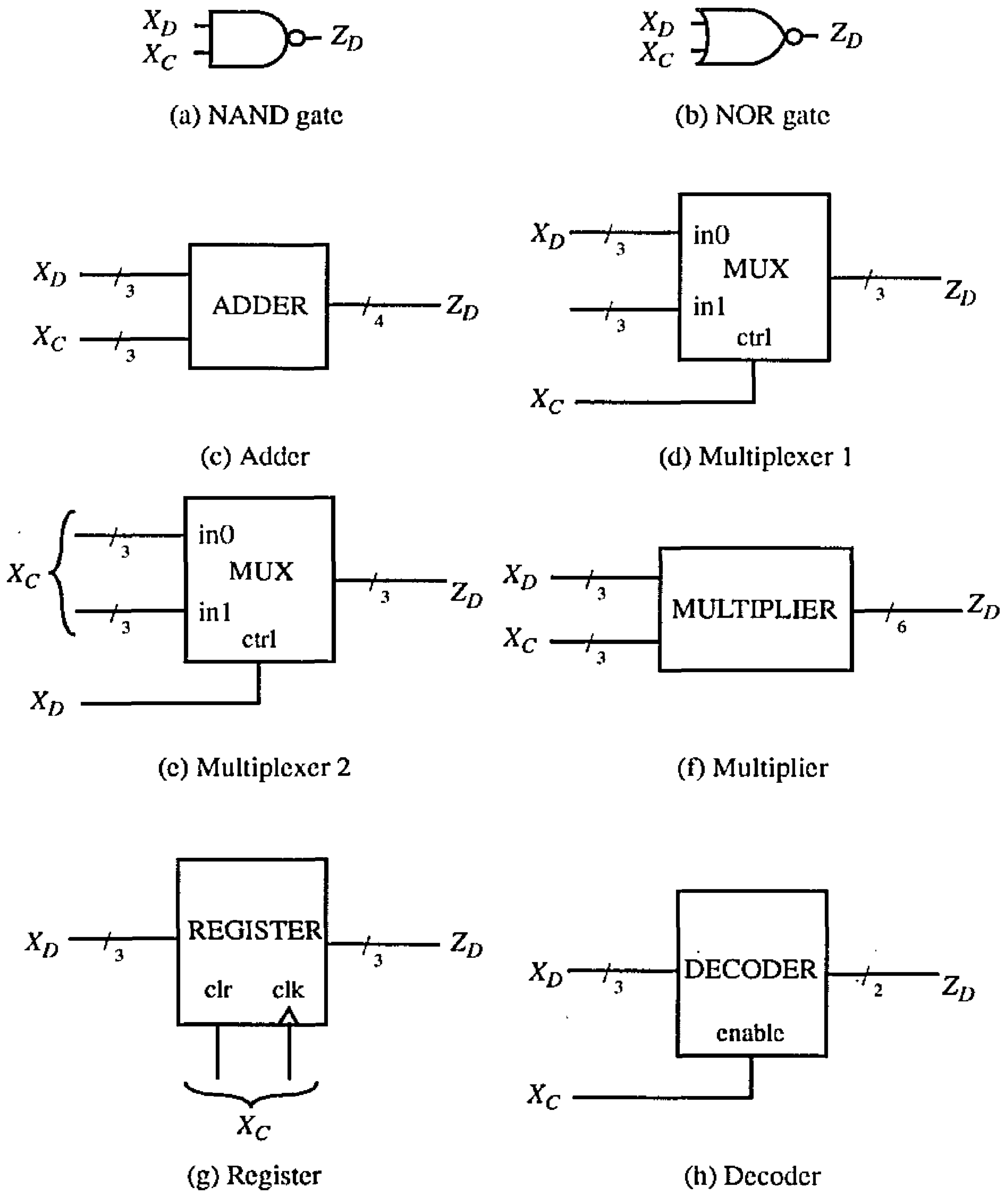


Figure 2.5. Typical modules and their buses

in Figure 2.5h. Here we assume that only inputs 0 and 4 are decoded, so that when the decoder is enabled and $V(X_D) = 0$, then $V(Z_D) = 1$, and if $V(X_D) = 4$, then $V(Z_D) = 2$. All other values at X_D map to 0 at Z_D.

The *propagation problem* (also called the observability problem) for a module M may be stated as: given a data input value $V(X_D)$, determine a control input value $V(X_C)$ so that the data output $V(Z_D)$ is distinguished from the output $V(Z_D)'$ due to any $V(X_D)' \neq V(X_D)$. M is said to be *transparent* with respect to $V(X_C)$ if such a control value exists. That is, M is transparent if any change in value at X_D is reflected in a change in value at Z_D. For example, Multiplexer 1 (Figure 2.5d) is transparent when $V(X_C) = 0$. The concept of transparency is also defined similarly by Marhöfer [66], and the path from X_D to Z_D through a transparent module is called an F-path by Freeman [30]. M is *partially transparent* if for some $V(X_C)$, $V(Z_D)'$ is distinguished from $V(Z_D)$ due to at least one $V(X_D)' \neq V(X_D)$. In this case, some changes at X_D can be distinguished at Z_D, but not all. For example, the decoder in Figure 2.5h is only partially transparent when $V(X_C) = 1$, since not all of the inputs are decoded—most map to 0.

Some propagation paths with specific transmission properties have been defined. Two of the more important ones are *I-mode* paths and *T-mode* paths [1], both of which are fully transparent. $V(X_D)$ is propagated from X_D to Z_D without modification along propagation paths with I-modes. $V(X_D)$ is propagated from X_D to Z_D either unchanged or inverted—a simple transformation, along paths with T-modes. An example of an I-mode is demonstrated by Multiplexer 1 (Figure 2.5d) when $V(X_C) = 0$, and an example of a T-mode is demonstrated by the NAND gate in Figure 2.5a when $V(X_C) = 1$. Other useful functional path characteristics or modes may easily be envisioned.

The *justification problem* (also called the controllability problem) for a module M is to determine an assignment to X_D and X_C such that $V(Z_D) = T_S$. Frequently, as in the case of observability, X_C is used to control information propagation from X_D to Z_D. In these cases, each $V(Z_D)$ corresponds to a unique $V(X_D)$. For example, when $V(X_C) = 0$ in Multiplexer 1 (Figure 2.5d), then any desired value v at Z_D can be obtained by applying v to X_D. Both I-modes and T-modes can be used in this way for justifying as well as propagating, a fact that is exploited by *PathPlan*.

2.4. Representing Information

In this section we discuss the representation of precomputed test data. These data are propagated as signal values in the circuit model. Two types of information need to be propagated: (1) correct or fault-free signal values, including T_S, T_R, and various control signals, and (2) error signal values that represent the effect of faults.

Vector Sequences. As discussed above, our precomputed test methods use hierarchy and abstraction in the propagation of signals as well as in the description of circuit structure and behavior. In order to represent signal abstraction, we consider all signal values to be vector sequences [45,12]. A basic *vector sequence* is an $n \times m$ matrix of logic values representing a sequence of Boolean vectors. Each of the n rows of a vector sequence A represents the signal values for one bit of a bus over time. Column t of A represents the set of logic values on the bits of an m-bit bus at time instance t. These values may be freely interpreted as any valid encoding over the bits, such as integers modulo m, Gray codes, etc. The time units are typically clock cycles, but may be interpreted as gate delays, groups of clock cycles, or the like.

In general, we consider T_S and T_R to be vector sequences. As an example, the following stimulus/response vector sequence pair represents a sequence of three vectors on the input and output ports of a 3-bit adder such as that of Figure 2.5c.

$$\left(\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} ; \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \right) \tag{2.1}$$

Vector sequence matrices can be refined into submatrices in hierarchical fashion. For example, a natural partition of the input stimulus vector sequence part of (2.1) for the adder is

$$A_1 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{2.2}$$

which also illustrates how vector sequences can be represented by symbols. If we assign the output part of the pair (2.1) to the symbol A_3, then the pair (2.1) can be written as

$$\left(\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} ; A_3 \right) \tag{2.3}$$

We sometimes use a shorthand notation that allows vector sequences to be written on one line. Horizontal sequences, implying a range of time instances, are enclosed in square brackets. Vertical vectors, implying a range of values in space, are enclosed in parentheses. For example, if

$$A = \begin{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & \begin{matrix} A_{13} \\ A_{23} \end{matrix} \\ \begin{matrix} A_{31} & A_{32} \end{matrix} & A_{33} \end{bmatrix} \tag{2.4}$$

and

$$A' = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{2.5}$$

then we write $A = [\,(A', [A_{31}, A_{32}]), (A_{13}, A_{23}, A_{33})\,]$. If we interpret the individual vectors of a vector sequence as integers, then using our shorthand notation, the vector sequence A_1 in (2.2) can be rewritten as $A_1 = [3, 1, 0]$. In these examples, the basic elements of vector sequences are constants, but they can also be variables, as we show below.

Certain vector sequences are used so frequently by themselves and in the construction of other sequences, that they have been given special names and symbols [45]. These include the n-bit all-0 vector sequence 0_n and the all-1 sequence 1_n which are used later, as well as C_n, the $n \times 2^n$ counting sequence (the output of an n-bit counter), and D_n, the $n \times n$ diagonal sequence.

Errors. Let Z be a bus in a circuit under test C, and let $V(Z) = v$ when C is working correctly.

Let $V(Z) = v^e$ when C has a fault. The pair (v, v^e) is called a *discrepancy*, and represents an error signal on Z. In conventional (gate-level) test generation, error signals are usually represented by the symbols D = (1,0) and $\overline{\text{D}}$ = (0,1). Error signals are frequently combined with fault-free signal values in a single set. In the D-algorithm and PODEM for instance, the signal values are elements of a five-valued algebra $D_5 = \{0,1,X,D,\overline{D}\}$, where 0 represents the pair (0,0) and 1 represents (1,1). The value X implies uncertainty, which can be represented by the unordered set {0,1}, therefore X represents ({0,1}, {0,1}).

We can apply logic functions such as AND, OR, and NOT to D_5 by appropriately combining operations on the basic elements 0 and 1. We use the general method for extending basic algebras presented in [46]. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of constant values, for instance {0,1}, and let $\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\}$ be an associated set of operations, such as {AND, OR, NOT} so that together A and Φ constitute an algebra denoted (A, Φ). A is referred to the *basis set*. There are two general methods for extending A to a new set A' that allow the operation set Φ' associated with A' to be easily constructed from the operations in Φ.

The first method creates A' from ordered n-tuples of the form $a_i' = (a_{i1}, a_{i2}, \ldots, a_{in})$, where each a_{ij} is an element of A. The interpretation of a_i' in this case is that the signal may assume value a_{ij} under condition j. A set A' constructed in this fashion is called a *P-set*. The discrepancies D = (1,0) and $\overline{\text{D}}$ = (0,1), and the fault-free values (0,0) and (1,1) in D_5 are P-set pairs. The other method is to construct A' from a set of subsets of A, so that each $a_i' \in A'$ is itself an unordered set of the form $a_i' = \{a_{i1}, a_{i2}, \ldots, a_{ik}\}$, where each a_{ij} is an element of A. The interpretation of a_i' in this case is that the signal may assume any value in the set. A set A' constructed in this fashion is called a *U-set* and is typically associated with uncertainty about the current value. The elements of X = ({0,1}, {0,1}) in D_5 are examples of U-sets. Clearly, P-sets and U-sets can be combined as they are in $D_5 = \{0,1,X,D,\overline{D}\} = \{(0,0), (1,1), (\{0,1\},\{0,1\}), (1,0), (0,1)\}$.

Next we show how to obtain the operations in Φ' by combining operations in Φ. Consider an operator $\phi(a_1, a_2, \ldots, a_m)$ defined on A. ϕ can be extended for any P-set as follows:

$$\phi'(a_1', a_2', \ldots, a_m') = (\phi(a_{11}, a_{21}, \ldots, a_{m1}), \phi(a_{12}, a_{22}, \ldots, a_{m2}), \ldots \phi(a_{1n}, a_{2n}, \ldots, a_{mn})) \tag{2.6}$$

where $a_i' = (a_{i1}, a_{i2}, \ldots, a_{in}) \in A'$ and $a_{ij} \in A$ for $1 \le i \le m$ and $1 \le j \le n$. Similarly, ϕ can be

AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

Table 2.1 The AND operator defined for D_5.

extended for any U-set as follows:

$$\begin{aligned}\phi'(a_1', a_2', \ldots, a_m') = \{&\phi(a_{11}, a_{21}, \ldots, a_{m1}), \phi(a_{12}, a_{21}, \ldots, a_{m1}),\\ &\ldots\\ &\phi(a_{1n}, a_{21}, \ldots, a_{m1}), \phi(a_{11}, a_{22}, \ldots, a_{m1}), \phi(a_{12}, a_{22}, \ldots, a_{m1}),\\ &\ldots\\ &\phi\left(a_{1n_1}, a_{2n_2}, \ldots, a_{mn_m}\right)\}\end{aligned} \tag{2.7}$$

where $a_i' = \{a_{i1}, a_{i2}, \ldots, a_{in}\} \in A'$ and $a_{ij} \in A$ and $1 \le i \le m$ and $1 \le j \le n_i$. In other words, the new set is constructed by combining members from each of the sets $a_1', a_2', \ldots, a_m'$ in all possible ways. Equations (2.6) and (2.7) define an extension rule which is satisfied by many useful multiple-valued logics [46].

We can now use (2.6) and (2.7) to construct logic functions for D_5. For example, consider the AND operation and D_5.

$$\begin{aligned}\text{AND}(\text{D}, 1) &= \text{AND}((1,0), (1,1))\\ &= (\text{AND}(1,1), \text{AND}(0,1)) = (1,0) = \text{D}\end{aligned}$$

while

$$\begin{aligned}\text{AND}(0, \text{X}) &= \text{AND}((\{0\}, \{0\}), (\{0,1\}, \{0,1\}))\\ &= (\text{AND}(\{0\}, \{0,1\}), \text{AND}(\{0\}, \{0,1\}))\\ &= (\{\text{AND}(0,0), \text{AND}(0,1)\}, \{\text{AND}(0,0), \text{AND}(0,1)\})\\ &= (\{0,0\}, \{0,0\}) = (\{0\}, \{0\}) = 0.\end{aligned}$$

The complete function table for an AND gate is shown in Table 2.1. Tables for the other logic functions are similar.

This algebra (D_5, {AND, OR, NOT}) is used by test generation algorithms to compute values at the inputs and outputs of gates in the circuit. In PODEM, for example, after a proposed assignment v of logic values to the primary inputs is established, an implication operation is performed to determine an equilibrium logic state which is consistent with v. Each gate in the circuit is updated according to the function tables such as Table 2.1 to determine what its output will be in response to a change on the gate's inputs. It is possible to use tables such as Table 2.1 to unambiguously analyze the results of error propagation along reconvergent paths in a circuit. If both inputs to an AND gate are D, then the output is D. On the other hand, if one input is D and the other $\overline{\text{D}}$ then the output is 0, that is, the error signal is blocked.

For precomputed testing, we want to propagate error signals on buses, not individual bits of buses. Test response errors at the output of the MUT have the form (T_R, T_{Ri}), where T_{Ri} is the response of the MUT to T_S when fault f_i is active. For example, if fault f_i is a stuck-at-1 fault on the least significant bit of the output to a 3-bit adder, then the discrepancy associated with fault f_i for the test response given in (2.1) is

$$\left(\begin{bmatrix}0&0&0\\1&0&0\\1&1&0\\0&1&0\end{bmatrix},\begin{bmatrix}0&0&0\\1&0&0\\1&1&0\\1&1&1\end{bmatrix}\right) \tag{2.8}$$

We can represent individual vector sequence discrepancies like (2.8) as P-set pairs, similar to those in D_5. However the number of elements in a set of such pairs is not fixed, it depends on the size of the buses in a circuit and on the length of the sequences. We cannot define a fixed set such as D_5, because bus size varies between circuits, and so we cannot define corresponding functions to propagate the values in a circuit model.

However, by abstracting signals, we can create a small fixed set of symbolic error signals. Lee and Patel pursued this approach for the test generator ARTEST [58]. They constructed a set of symbolic error signals as a P-set similar to D_5 using the ad hoc basis set {X,V,U,V',U'} with the following interpretation

X = an unassigned value

V = an assigned, known value

Error signal	Symbol	Interpretation
(X,X)	X	Unassigned
(X,V)	CF	Constant faulty
(X,U)	VF	Variable faulty
(V,X)	CG	Constant good
(V,V)	C	Constant good and faulty
(V,V')	CGCFE	Constant good and constant faulty effect
(V,U)	CGVF	Constant good and variable faulty
(V,U')	CGVFE	Constant good and variable faulty effect

Table 2.2 The set of symbolic error signals used in ARTEST [58].

U = an assigned, but unknown value

V' = an assigned known value different from the correct value

U' = an assigned but unknown value, different from the correct value

The symbolic error signal P-set constructed from this basis set is shown in Table 2.2 together with the symbols and their interpretation provided by Lee and Patel [58]. These interpretations are meant to clarify the meaning of the signal but are frequently ambiguous. Some signals such as X are similar to values in other signal value sets such as D_5. Others, such as CGVFE, are unique to ARTEST. The signal CGVFE is used to represent all errors of the form (T_R, T_{Ri}) simultaneously. The signal C is used to represent all fault-free control signals and T_S. If CGVFE is propagated to a primary output, then all test response errors are propagated.

Since the elements of the signal value set described above are all abstract symbolic values, they cannot be used alone to determine the state of a circuit during test generation. Therefore, they are combined in ARTEST with an explicit fault-free signal value and the resulting signal is a pair of the form (*v*, *t*), where *v* is a vector and *t* is an element of the symbolic signal value set given in Table 2.2 and is referred to as the *type*.

The type set used in ARTEST has two drawbacks. First, the set is constructed in ad hoc fashion so that the method described above for extending basic operations cannot be systematically applied. Types are propagated in ARTEST using a set of rules for each module. Second, CGVFE can only be propagated along fully transparent paths. Since all test response errors are combined in CGVFE, if any particular error (T_R, T_{Ri}) cannot be propagated along a propagation path, then ARTEST must pessimistically assume that none can. Therefore, as previously discussed, ARTEST cannot generate tests for circuits with an irregular bus structure, such as Divfilt (Figure 2.3).

In Chapter IV, we will present a hierarchical approach to test response error propagation. Our highest-level signals are similar to those described above for ARTEST and propagate test response signals along fully transparent paths. Less abstract error signals are used to propagate test response errors along partially transparent paths. The signal value sets and their associated functions are rigorously defined using P/U-sets and the operation extension methods described above.

Test Packages. We refer to the information unit containing all test, propagation, and control information for a module or circuit as a *test package*. Test packages are the elements of the module library used for testing by the precomputed test method. Like packages in the hardware description language VHDL [20], they hide and abstract information. The simplest form is the stimulus/response pair $(T_S;T_R)$, where T_S and T_R are, in general, vector sequences. A test package exhibits the same hierarchical structure as the underlying vector sequences. In particular, $(T_S;T_R)$ may be partitioned into control and data parts as implied by the partitioning of buses depicted in Figure 2.4. Such a test package can be denoted by $(V(X_D), V(X_C);V(Z_D), V(Z_C))$. Each of the buses X_D, X_C, Z_D, Z_C may be further refined (in space) into the natural buses of the circuit, as in the example modules in Figure 2.5. This implies a natural correspondence between entries in a test package and the ports of a module.

Since T_S and T_R can be represented by vector sequences, refinements of the test package $(T_S;T_R)$ in time can be concisely represented. For example, the multiplexer of Figure 2.5b can be tested as follows: select the data input in0 for k clock cycles by applying a sequence of k 0's to ctrl, and apply a sequence of k patterns in a vector sequence A_1 to in0. The process is repeated in k-cycle sequence for the other data input in1 with S(ctrl) = 1_k. The resulting test package for this case is

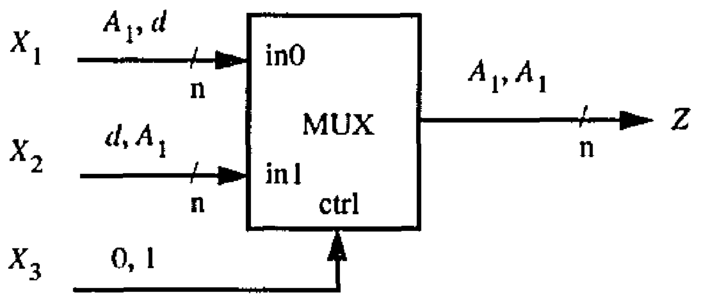


Figure 2.6. Bus assignment example

$$TP_1 = (T_S;T_R) = \left(\begin{bmatrix} A_1 & d \\ d & A_1 \\ 0 & 1 \end{bmatrix} ; \begin{bmatrix} A_1 & A_1 \end{bmatrix} \right)$$

The value d denotes "don't care", and implies that any n-bit vector sequence can be applied as part of the test. This test package can also be written as $([(A_1, d), (d, A_1)], [0, 1]; [A_1, A_1])$. Figure 2.6 shows the various components of the test package applied to the input/output ports of the multiplexer.

We commonly separate test packages into two main types: those containing test data for module faults called *fault test packages* (FTPs), and those containing propagation information for modules called *propagation test packages* (PTPs). These may be loosely compared to the fault D-cubes and propagation D-cubes of classical testing theory [16]. The test package for the multiplexer described above is an example of a FTP. PTPs define functions mapping specific inputs to outputs for the purposes of propagating test information. In a test package of the form $(S(X_D), S(X_C); S(Z_D), S(Z_C))$, $S(X_D)$ or $S(Z_D)$ may be vector sequences associated with several ports. If a variable α appears within $S(X_D)$, then α should also appear in the corresponding position within $S(Z_D)$. As an example, the PTP for propagating sequences through the data port X_1 of the multiplexer in Figure 2.6 is denoted by $TP_2 = ((\alpha, d), 0; \alpha)$. In TP_2 the variable α appears in positions corresponding to X_1 and Z, and the constant value 0 corresponds to the control input X_3. During test generation, all values in the PTP must match the values on their corresponding buses. Therefore, TP_2 implies that when $V(X_3)$ is 0, the same vector sequence must be assigned to both X_1 and Z, since both must match the same variable. The corresponding package for propagation through X_2 is $((d, \alpha), 1; \alpha)$. We will demonstrate how PTPs are used in a test generation algorithm in the next section.

2.5. *PathPlan*

PathPlan (for path planning) is our initial version of a hierarchical test generator using precomputed tests. It was developed at General Motors Research Laboratories in 1987 and was one of the earliest reported automatic test generators designed specifically to generate tests for circuits using precomputed tests for modules [68]. It has not been used to test commercial circuits, however, a similar program partly based on *PathPlan* has been used for production testing [11]. Our work developing *PathPlan*, as well as the work to develop necessary extensions to it, form the basis for this thesis.

PathPlan propagates symbolic references to vector sequences representing T_S and T_R through circuit models using an algorithm similar to the D-algorithm. The module test stimulus T_S is justified module by module from the MUT to primary inputs (PIs) and the module test response T_R is propagated module by module to primary outputs (POs). T_S and T_R are stored as FTPs for each module in a circuit to be tested. PTPs are used to transfer information through modules. To simplify the processing of test response errors, *PathPlan* restricts propagation to cases that require only simple transformations of the symbolic signals being propagated. If A is a vector sequence representing T_R, then A or its logical inverse $\bar{A}$ must be propagated along a path from the output of the MUT to a primary output in order for error propagation to be considered successful by *PathPlan*. As noted earlier, this mode of propagation is referred to as T-mode propagation. A T-mode path is transparent, therefore any test response error (T_R, T_{Ri}) at the output of the MUT will be propagated along such a path. *PathPlan* does not explicitly represent error signals, since all errors are implicitly propagated along T-mode paths.

In *PathPlan*, vector sequence signals are assigned to buses and propagated along data paths using a procedure called instantiation. The same instantiation procedure is used for both justification and propagation. We will describe how it works by means of some simple examples. Instantiation is fairly trivial when assigning the various stimulus and response components of a FTP to a MUT in a given circuit. Consider an instance of the multiplexer of Figure 2.6 in a circuit, and assume that its test is characterized by the test package TP_1 = ([(A_1, d), (d, A_1)], [0,1]; [A_1,

A_1]) derived above. The signals assigned to all the ports in this instance of the multiplexer are initially d. When the instance becomes the MUT, then the simplest form of instantiation is used to assign values to the multiplexer's ports as follows: A_1 is assigned to port X_1, d is assigned to port X_2, 0 is assigned to port X_3, and A_1 is assigned to port Z.

Instantiation is also used for the more complex process of signal propagation in the following way. Again referring to Figure 2.6, suppose that a vector sequence A_R representing T_R has already been assigned to port X_1 of the multiplexer instance, but that all other ports are still initialized to d. A_R can be propagated through the multiplexer using the PTP $TP_2 = ((\alpha, d, 0);\alpha)$. First, since the variable α in position 1 of TP_2 corresponds to port X_1, instantiation assigns A_R to α. Next it substitutes A_R for variables named α everywhere in TP_2 producing $TP_3 = ((A_R, d, 0);A_R)$. Finally, it intersects each value in TP_3 with the value already assigned to its corresponding port, where intersection of two vector sequences is defined by the intersection of their corresponding bits in the usual way [79]:

$$0 \cap 0 = 0 \cap d = d \cap 0 = 0$$
$$1 \cap 1 = 1 \cap d = d \cap 1 = 1$$
$$d \cap d = d$$
$$1 \cap 0 = 0 \cap 1 = \varnothing$$

Here the empty set symbol $\varnothing$ denotes conflict. In most cases, conflicts can be analyzed symbolically, since different vector sequences are assigned different symbols in *PathPlan*. For example, if A_1 and A_2 refer to different vector sequences, then A_1 cannot be intersected with A_2 without conflict. Instantiation can be used in a symmetric way to transfer a vector sequence A_S representing T_S along a path from an input to the MUT to a primary input.

A key element of instantiation when used for propagation is the assignment of vector sequences to variables in the input (output) part of a PTP and the subsequent substitution of the same vector sequence for variables in the output (input) part of the PTP. In the propagation example above, the value A_R on port X_1 was propagated to port Z when the same variable α appeared in the positions of the PTP corresponding to those ports. The process of substituting the same value for all variables of the same name in a test package to be instantiated is called *unification*. The unification procedure is widely applied in computer science [53], most notably in manipula-

```
PathPlan
{
    while (there are modules left to test) {
        select a module to test;
        while (there are FTPs for the MUT) {
            initialize circuit;
            select a FTP;
            instantiate( ports of the MUT, FTP );
            while (there are test responses to propagate) {
                choose $T_R$;
                if (there are modules in the test frontier) {
                    current module = MUT;
                    record choice;
                    propagate;
                }
                if (propagation is successful) {
                    justify;
                    if (justification is successful) simulate implications;
                    if (no conflicts) record success;
                }
            }
        }
    }
}
```

Figure 2.7. *PathPlan* algorithm.

tions of expressions by compilers, in automated theorem proving, and in logic programming.

The instantiation process may be summarized as follows:

1. For each position in the test package *TP* containing a variable, assign the value on the corresponding bus in the circuit to that variable.
2. Unify the variables if possible
3. Intersect the values in the test package with the corresponding values in the circuit, and assign the results to the buses in the circuit if there is no conflict

If a conflict is encountered in steps 2 or 3, other test packages are tried, if available.

Figure 2.7 shows the main body of the test generation program *PathPlan*. It uses two primary subprocedures, *propagate* and *justify* defined in Figures 2.8 and 2.9, respectively. There are often several FTPs to be instantiated for a given MUT; individual tests can be hierarchically decomposed into smaller FTPs to be applied separately as discussed above, and alternative FTPs can be used when the application of an initial FTP is impossible. The *test frontier* contains a list of

```
propagate
{
    while (TRUE) {
        if (destination module exists and more PTPs)
            select a PTP;
        else if (test frontier not empty)
            select a destination module;
        else return( FAILURE );
        while (there are PTPs for destination module but no
                successful instantiation) {
            instantiate( ports of destination module, PTP );
            if (instantiation successful) {
                record choice;
                current module = destination module;
                if (current module is connected to PO)
                    return( SUCCESS);
            } else {
                select new PTP;
                if (no PTP) backtrack to last recorded choice;
            }
        }
    }
}
```

Figure 2.8. Propagation procedure of *PathPlan*.

the modules chosen for forward propagation. These are the modules whose outputs are unassigned, and at least one of whose inputs has been assigned T_R. Modules can be chosen for justification if their outputs have been assigned but not all their inputs have been assigned yet.

PathPlan has been implemented and used to test several practical circuits. It consists of about 7,000 lines of C and accepts circuit descriptions written in an HDL similar to the commercial test generator and simulator Hitest [76]. In order to demonstrate the use of *PathPlan*, we describe its application to the Gaussian filter chip which is an IC designed at General Motors R&D Center for use in image processing applications [65]. It is a small but nontrivial CMOS design with approximately 40,000 transistors. All modules are synthesized using module generators as in Divfilt and Encode. Two on-chip RAMs support line buffering operations. Figure 2.10 shows a basic block of the circuit called Fltrdp. Three of these blocks are used in the Gaussian filter and form a major portion of that circuit.

Fltrdp is composed of five types of modules: ADDERX, ADDER, FO, MUX, and LATCH. Buses 5, 12, and 13 are 1-bit buses; all others are 8 bits in width. ADDERX is an adder

```
justify
{
    while (there are buses to be justified) {
        select a bus X to be justified;
        current module = module driving X;
        select a PTP for current module;
        while (there are more PTPs or until successful instantiation) {
            instantiate( ports of current module, PTP );
            if (successful) record choice;
            else {
                select PTP;
                if (no PTP) backtrack to last recorded choice;
                if (there are no more justification choices)
                    propagate;
                if (propagation fails) return( FAILURE );
            }
        }
    }
    return( SUCCESS );
}
```

Figure 2.9. Justification procedure of *PathPlan*.

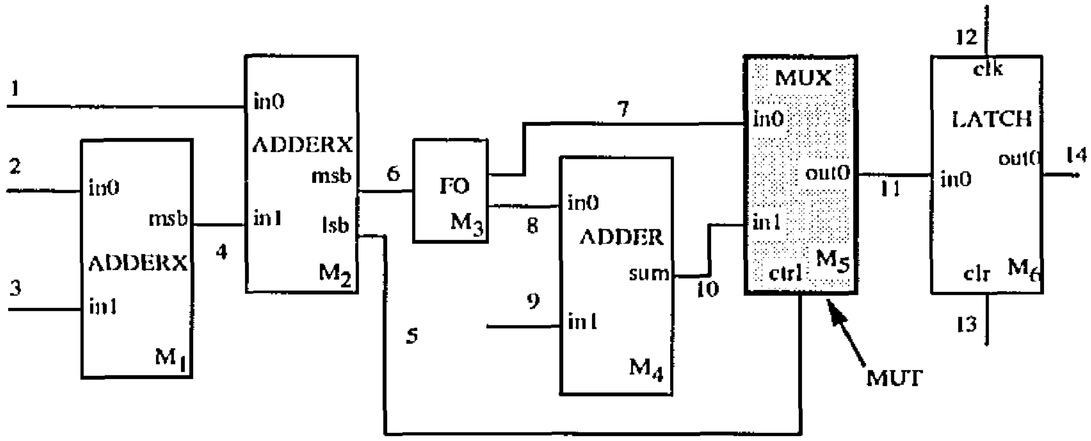


Figure 2.10. Basic block Fltrdp of the Gaussian filter chip.

whose carry-out bus has been combined with the most significant 8 bits of the sum, forming the output bus msb. The least significant bit of the sum is called lsb. ADDER is self-explanatory, LATCH is an edge-triggered latch module, and MUX is a multiplexer. *PathPlan* models fanout explicitly as a module; FO is a fanout module. The carry-in value to all adders is 0, and any outputs not used are not shown. In this example, ADDERX is designated as a module type different from ADDER to mitigate *PathPlan*'s limited ability to handle bus irregularities.

We will show two passes of PathPlan used to derive a test for the multiplexer MUX in

Fltrdp. As before, the FTP for testing MUX will be

$$(T_S;T_R) = ([(A_1,d),(d,A_1)],[0,1];[A_1,A_1])$$

which matches ports in0, in1, ctrl, and out0, respectively. The circuit will be initialized so that every bus has value d. After instantiating $((A_1,d),0;A_1)$, the list of the values on all 14 buses of Fltrdp is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
d	d	d	d	0	d	A_1	d	d	d	A_1	d	d	d

The MUT is not connected to a primary output so we include the destination module M_6 in the test frontier and propagate. The test package for propagating values in parallel through the latch is $((C,\alpha,0);\alpha)$, where C is a clocking sequence ([0101...]). After instantiation, the values on the buses become

1	2	3	4	5	6	7	8	9	10	11	12	13	14
d	d	d	d	0	d	A_1	d	d	d	A_1	C	0	A_1

Since bus 14 is connected to a primary output, propagation has succeeded. As all inputs to the latch except in0 are primary inputs, the next nontrivial justifications to be made are on buses 7 and 5. The test package for propagating values backward through ADDERX is $((\alpha,\alpha);(0,\alpha))$. This test package represents a strategy for backward propagation (justification) in which a value A is added to itself, thus propagating A to the most significant bits of the sum (msb). In this case, $X_C = X_D$. The PTP for the FO module M_3 is $(\alpha;(\alpha,\alpha))$. We now select bus 7 for justification. After instantiation, the values on the buses are:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
d	d	d	d	0	A_1	A_1	A_1	d	d	A_1	C	0	A_1

Finally, the outputs on modules M_1 and M_2 are justified. After these justifications, the values on buses are:

1	2	3	4	5	6	7	8	9	10	11	12	13	.14
A_1	A_1	A_1	A_1	0	A_1	A_1	A_1	d	d	A_1	C	0	A_1

The first component of T_S has now been propagated to primary inputs.

We next try to instantiate $((d, A_1), 1;A_1)$ in the same manner. The sequence of steps is shown in the following list of bus values.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
d	d	d	d	1	d	d	d	d	A_1	A_1	d	d	d
d	d	d	d	1	d	d	d	d	A_1	A_1	C	0	A_1
d	d	d	d	1	d	d	0	A_1	A_1	A_1	C	0	A_1
d	d	d	d	1	0	0	0	A_1	A_1	A_1	C	0	A_1
0	d	d	1	1	0	0	0	A_1	A_1	A_1	C	0	A_1
0	0	1	0	1	0	0	0	A_1	A_1	A_1	C	0	A_1

As discussed in Chapter I, we measure performance in terms of module evaluations, and in *PathPlan* we measure module evaluations by counting instantiations. For the circuit in Figure 2.10, *PathPlan* requires 57 module evaluations to fully test modules M_5 and M_6, and partially test modules M_1 and M_2 and M_4. Module M_1 is only partially tested because the lsb output is unused. Neither module M_2 nor module M_4 can be fully tested at this level of abstraction by *PathPlan* using T-modes due to reconvergent fanout. In the case of module M_2, T_R is propagated along buses 5 and 6 and reconverges at module M_5. There is no T-mode for module M_5 that propagates vector sequence symbols simultaneously on both control and data inputs. Note that if errors in T_R appear at both the lsb and msb outputs of module M_2, or only at the msb output, then they are always propagated through M_5. However, an error that appears at lsb alone can only be propagated if bus 9 is not 0, that is, when the in0 and in1 input signals to module M_5 are different. To exploit this fact, we can partition the FTP for M_2 into two different parts; those that produce errors only at lsb are propagated together, and the rest are propagated separately. However, we must still use an ad hoc PTP (one not using the T-mode propagation) for module M_5 to propagate T_R. This

Module	Estimated number of gates	Estimated number of gate evaluations	Number of high-level modules	Number of module evaluations by *PathPlan*
Fltrdp	254	64,516	6	57
Vertdp	138	19,044	5	21
Rowdp	220	48,400	6	33
Alu	62	3,844	9	167
Progptr1	110	12,100	12	210
Progptr	156	24,336	6	68

Table 2.3 Performance of *PathPlan* relative to gate-level test generation.

approach is not systematic; it does not address the general problem of reconvergent fanout. Reconvergent fanout is also a problem when using symbolic vector sequences to test module M_4 in *PathPlan*. With T-mode propagation, we cannot simultaneously control bus 8 to apply T_S to module M_4, and bus 5 to propagate T_R through module M_5.

Some results of applying *PathPlan* to other practical circuits are shown in Table 2.3, which repeats Table 1.1. Fltrdp has been described above. The modules named Vertdp and Rowdp are datapath circuits similar to Fltrdp and used in the template-matching IC described in [65]. Vertdp has two registers, an inverting buffer, an adder, and a multiplexer. Rowdp has one register, an inverting buffer, two adders, and two multiplexers. Both can be tested completely by a test generated by *Pathplan*. Alu is a high-level model of the 74181 ALU/function generator described in [12]. Most of the modules in the high-level model employed by Alu are word gates. Finally, Progptr1 and Progptr2 are circuits used in a control unit. They consist primarily of multiplexers, but the bus size of the multiplexers in Progptr1 is 4, while the bus size of the multiplexers in Progptr2 is 8. The fact that the modules are smaller, and that there are more of them accounts for the lower performance of *PathPlan* in testing Progptr1.

PathPlan is an early version of a test generator designed specifically for testing using precomputed tests, therefore it has some restrictions that have been addressed by subsequent research. In particular,

1. It can only handle acyclic combinations of primitive modules
2. It uses only T-mode propagation

Restriction 1 can easily be addressed in principle by extending *PathPlan* to include the same modifications used in conventional gate-level test generators to handle sequential circuits [68]. Restriction 2 simplifies the algorithm but shrinks the solution space by leaving out many possible solutions. For example, suppose that module M_4 of Fltrdp (Figure 2.10) is the MUT. As discussed above, T_R cannot be propagated through module M_5 due to *PathPlan*'s reliance on T-modes. This restriction can be addressed by extending *PathPlan* to process more complex symbolic expressions, which we do in Chapter IV. Exclusive T-mode propagation also limits the kinds of transparent propagation paths that can be used to propagate errors. For example, an incrementer is a transparent module, but does not have a T-mode. ARTEST addresses this issue by using types as described earlier.

The main limitation of *PathPlan* is also shared by all other previously reported hierarchical test generators using precomputed tests for modules. They cannot analyze error propagation through circuits with truncated buses and arbitrary reconvergent fanout. It is this limitation that we address in the remainder of the thesis.

Despite its limitations, *PathPlan* does have some advantages which make it useful for testing circuits with large complex modules and a regular bus structure. These advantages stem from *PathPlan*'s simplicity. The main test generation algorithm and related procedures are implemented in about 1000 lines of C. The rest of the code supports libraries used by *PathPlan* as well as compilers for the hardware description language and the test package library. All the functional behavior for each module is contained in PTPs. *PathPlan* can handle both combinational and sequential primitive modules of arbitrary size and complexity. It only requires PTPs for the modules sufficient for propagating vector sequences through them during testing using T-modes. Performance measured by module evaluations depends on the number of PTPs stored for each module since these must be searched to find one that can be instantiated. A few simple propagation modes are often sufficient.

One additional aspect of test generation using precomputed tests often leads to problems; the propagation of signals through the MUT in timeframes other than when the FTP is instantiated.

This issue is handled in DB-TG [88] by optimistically assuming that the MUT only affects signals in one timeframe. *PathPlan* assumes that propagation is blocked. This problem is rarely mentioned in the literature. In some instances, the errors produced by the MUT can be propagated through the MUT even when it is faulty. The effect of each fault in the MUT on the error signal must be analyzed to determine whether the error signal is propagated. We will address this issue in Chapter IV.

Finally, we consider the affect of design-for-test techniques on test generation using precomputed tests. As discussed in Chapter I, many of the classical methods used for improving circuit testability are equally useful for precomputed test methods. In particular, full and partial scan design can be used to provide controllability and observability thus making circuits easier to test. In addition, direct access to modules can be provided by routing internal buses through special multiplexers. Variations on these techniques are used in [9, 26, 51, 82, 73]. An alternative approach to improving circuit observability for propagation is to modify non-transparent modules to increase their transparency; we will discuss this approach in Chapter IV.

2.6. Summary

In this chapter, we have introduced the precomputed test method and provided examples of the types of circuits that it can test. This method is appropriate for circuits with modules that cannot be tested using classical techniques. Each module must have an applicable precomputed test set. Test stimulus vectors T_S are propagated through structural models of a circuit under consideration to a module under test, and the corresponding response T_R of the module is propagated to a primary output or other observation point. The circuits tested using precomputed test methods have large modules connected by multi-bit buses. Frequently, these buses contain irregularities such as reconvergent fanout and truncations.

We also identified the key aspects of precomputed testing as well as the main components of our method and contrasted them with similar concepts in classical methods. Since the circuits for which precomputed test methods are appropriate contain large modules with long tests, as well as multi-bit buses, for efficiency, test information should be propagated at a high level of abstraction whenever possible. In our method, T_S and T_R have a hierarchical representation called a vector sequence. At the highest level of abstraction, vector sequences are propagated as symbols. In conventional methods, single-bit signals are used for propagating fault-free and faulty (error) data. Error signals are propagated more abstractly in most precomputed test methods. High-level symbolic error signals cannot be easily propagated through irregular buses. This is a main limitation of previously published hierarchical techniques for generating tests using precomputed tests for modules.

Finally, we described our initial test generator *PathPlan* in detail. *PathPlan* represents precomputed test stimulus and response sequences as vector sequences stored as test packages for each module. It generates tests by propagating symbolic references to vector sequences through a circuit model and performs only simple transformations on the signals. *PathPlan* can be used to test many useful circuits, however, as with other techniques using precomputed tests, it is ineffective at propagating error information through circuits with an irregular bus structure. The representation and propagation of error information through complex bus-structured circuits is an important research issue that is a central focus of this thesis. Another restriction of *PathPlan* is its inability to propagate arbitrary expressions of vector sequences. Solutions to these problems are examined in Chapter IV. First however, we develop our general theory of propagation.

CHAPTER III
THEORY OF PROPAGATION

This chapter presents a theory of propagation, a formal method for characterizing the information transmission properties of logic modules and circuits. Its goal is to automate analysis of error propagation in complex bus-structured circuits during test generation. The resulting theory is very general and has broad applicability. We have applied it to error propagation for test generation and also to design for testability.

3.1. Propagation Algebra

In this section, we formalize the propagation characteristics of modules and circuits and define the basic elements of propagation theory.

3.1.1 Propagation Functions

Let M be a module with input X and output Z, and let F be the function of M mapping values at X to values at Z. Often we are only interested in propagating values from a subset of the inputs to a subset of the outputs. If we select values only from the port $Z_D \subseteq Z$, then we shrink the codomain of F. We call the resulting function a *module subfunction* and denote it by $F[X;Z_D]$. Consider the 3-bit, 2-input multiplexer shown in Figure 3.1. Let Z_D be the two least significant bits of Z, that is, Z_D = out0[1..0]. The input is X = (ctrl, in0, in1). Signal values on Z_D can only range between 0 and 3. For example, F(1,2,4) = 4, while F[(ctrl,in0,in1);out0[1..0]](1,2,4) = 0, and F(1,2,5) = 5, while F[(ctrl,in0,in1);out0[1..0]](1,2,5) = 1.

Next, we consider mappings from a restricted set of input ports (X_C, X_D). Since the values assigned to other input ports are unspecified, the result is a set of subfunctions, one for each

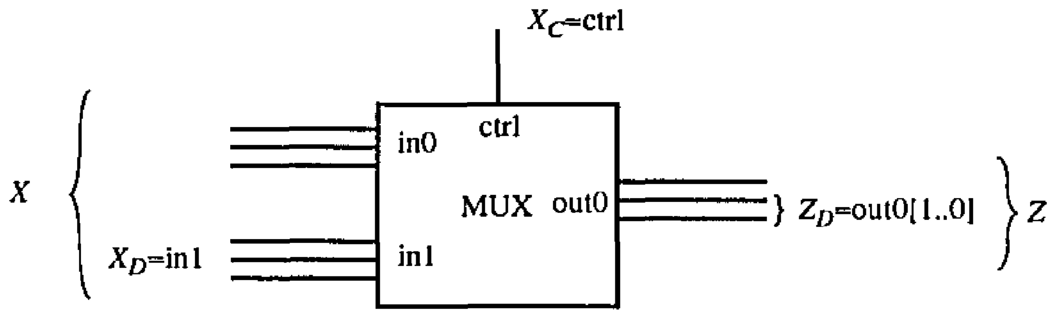


Figure 3.1. Three-bit, 2-input multiplexer with input data bus X_D = in1, control bus input X_C = ctrl, and output data bus Z_D = out0[1..0].

combination of values on $X - X_C - X_D$. We denote the subfunction mappings from values at X_C and X_D to values at Z_D as $F[(X_C, X_D); Z_D](x_1, x_2)$, where x_1 is a value assigned to X_C and x_2 is a value assigned to X_D. Consider again the multiplexer shown in Figure 3.1.

F[(ctrl, in1);out0[1..0]]$(0,x_2) = \{0,1,2,3\}$ for all x_2

F[(ctrl, in1);out0[1..0]]$(0,x_2) = x_2,\ x_2 \in \{0, 1, 2, 3\}$

F[(ctrl, in1);out0[1..0]]$(0,x_2) = x_2 - 4,\ x_2 \in \{4, 5, 6, 7\}$

If it is clear that we are referring to subfunctions, and if X_C, X_D, and Z_D, are known, then we will simply write $F[(X_C, X_D); Z_D]$ as F.

Finally, we consider the set of information or signal values to be propagated from X_D to Z_D. The full set of values that can be applied to a module at X_D is $\{V(X_D)\}$; however, we are often only interested in a subset of these, $\Omega \subseteq \{V(X_D)\}$. For instance, Ω can be the set of possible responses or outcomes from a test. The module subfunction defines an equivalence relation R on Ω. For $x_1, x_2 \in \Omega$, and some constant value c assigned to X_C, let $x_1 R x_2$ if and only if

$$F[(X_C, X_D); Z_D](c, x_1) = F[(X_C, X_D); Z_D](c, x_2)$$

R is an equivalence relation since set equality is reflexive, transitive, and symmetric. If x_1 and x_2 are equivalent, then they cannot be distinguished at Z_D. Indistinguishability of signal values, defined as equivalence, plays a central role in our analysis of information propagation.

A partition π on a set S is a collection of disjoint subsets of S called *blocks*, whose union is S. Two elements $s_1, s_2 \in S$ are equivalent, denoted $s_1 \equiv s_2(\pi)$, if and only if they are in the same block. If R is an equivalence relation on S, then the set of equivalence classes of R defines a partition π on S and vice versa. In particular, the equivalence relation on Ω defined by

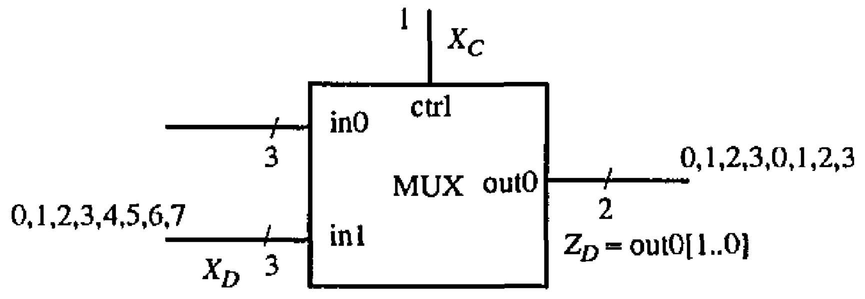


Figure 3.2. Set of values applied to X_D of a 3-bit, 2-input multiplexer and corresponding outputs at Z_D for $V(X_C) = 1$.

$F[(X_C, X_D); Z_D]$ induces a partition on Ω. As an example, for the 3-bit, 2-input multiplexer in Figure 3.1 with $\Omega = \{V(X_D)\} = \{0, 1, 2, 3, 4, 5, 6, 7\}$, we have the following:

F[(ctrl,in1);out0[1..0]](1,0) = F[(ctrl,in1);out0[1..0]](1,4) = 0

F[(ctrl,in1);out0[1..0]](1,1) = F[(ctrl,in1);out0[1..0]](1,5) = 1

F[(ctrl,in1);out0[1..0]](1,2) = F[(ctrl,in1);out0[1..0]](1,6) = 2

F[(ctrl,in1);out0[1..0]](1,3) = F[(ctrl,in1);out0[1..0]](1,7) = 3

This is depicted in Figure 3.2. The subfunction F[(ctrl, in1);out0[1..0]](1,x) forms a partition $\{\{0,4\},\{1,5\},\{2,6\},\{3,7\}\}$ on Ω.

In our analysis, it is frequently necessary to a treat a set of related subfunctions as a single unit. Let $\vec{F} = (F_1, F_2, \ldots, F_k)$ be an ordered k-tuple of subfunctions, each with the same input data bus X_D and domain $\Omega \subseteq \{V(X_D)\}$. $\vec{F}$ is called a *subfunction vector.* We can compose subfunction vectors in various ways. For instance, each subfunction F_i may be defined as $F[(X_C, X_D); Z_{Di}](c, x)$, where $V(X_C)$ is a constant value c, and each $Z_{Di} \subseteq Z$ is an output port disjoint from the output ports in subfunctions F_j, $j \neq i$. This is depicted in Figure 3.3a. In this case, each subfunction vector represents the mapping of values from a single input data port X_D to values on multiple output data ports Z_{Di}. Consider again the 3-bit, 2-input multiplexer in Figure 3.1. Let X_C = ctrl and X_D = in1 as before. Let Z_{D1} = out0[1..0] and Z_{D2} = out0[2], so that F_1 = F[(ctrl,in1);out0[1..0]] and F_2 = F[(ctrl,in1);out0[2]]. Then,

$$(F_1(1,x), F_2(1,x)) = \begin{cases} (1,x), 0 \le x \le 3 \\ (1, x-4), 4 \le x \le 7 \end{cases}$$

Alternatively, each subfunction F_i can be defined as $F[(X_C, X_D); Z_D](x_i, x)$, where

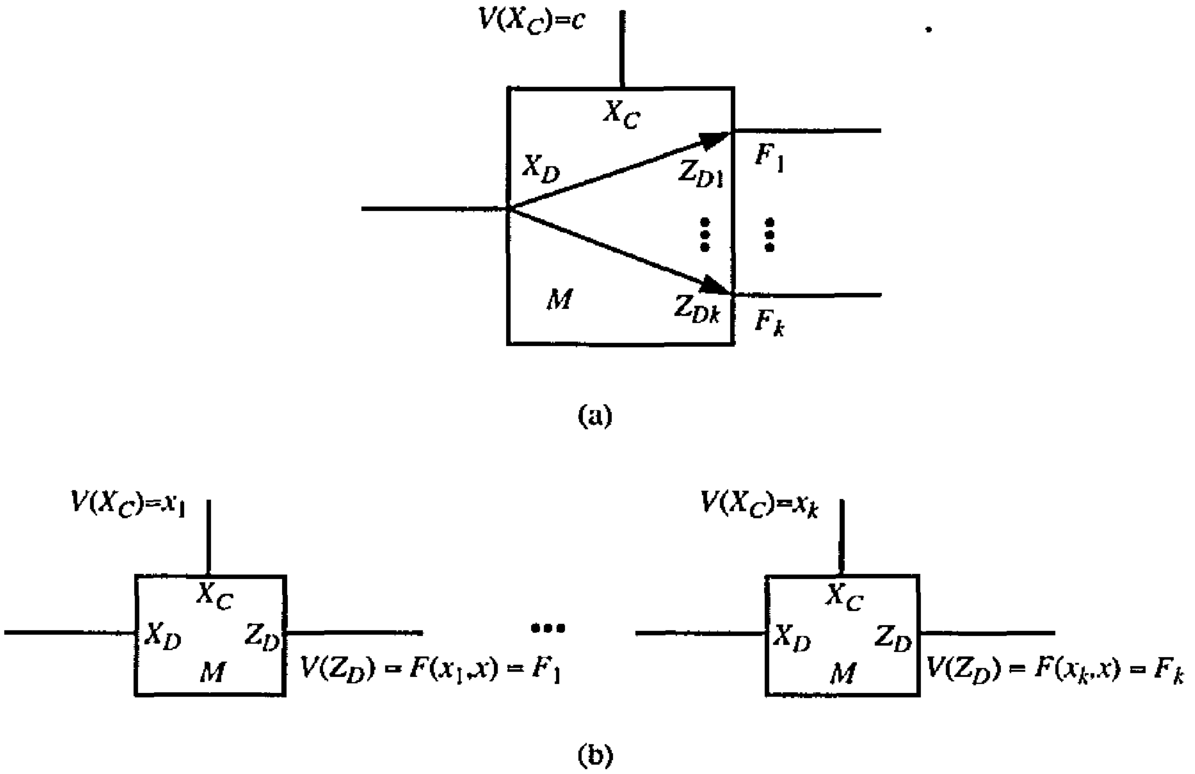


Figure 3.3. Two compositions of a subfunction vector $(F_1,F_2,...,F_k)$: (a) space and (b) time.

each x_i is a different value assigned to X_C. This subfunction vector represents the mapping of values from a single input data bus X_D to values at a single output data bus Z_D due to a sequence of control values assigned to X_C for each mapping. Each F_i represents the mapping due to a different value in this sequence, as depicted in Figure 3.3b. For the 3-bit, 2-input multiplexer in Figure 3.1,

$$(F(0,0), F(1,0)) = (\{0,1,2,3\}, 0)$$

$$(F(0,1), F(1,1)) = (\{0,1,2,3\}, 1)$$

Here the control sequence applied to X_C is [0,1]. These are the two common interpretations of a subfunction vector used in our analysis.

A subfunction vector also defines a partition on Ω related to the partitions π_i defined by its constituent subfunctions. Let x_1 and x_2 be two elements of Ω and let R be the relation on Ω such that x_1Rx_2 if and only if x_1 and x_2 are both contained in the same block in every partition π_i. This is an equivalence relation because, once again, set inclusion is reflexive, transitive, and symmetric. The structure of the partition defined by subfunction vectors is a key element in determining the propagation characteristics of a module or circuit with respect to the set of information

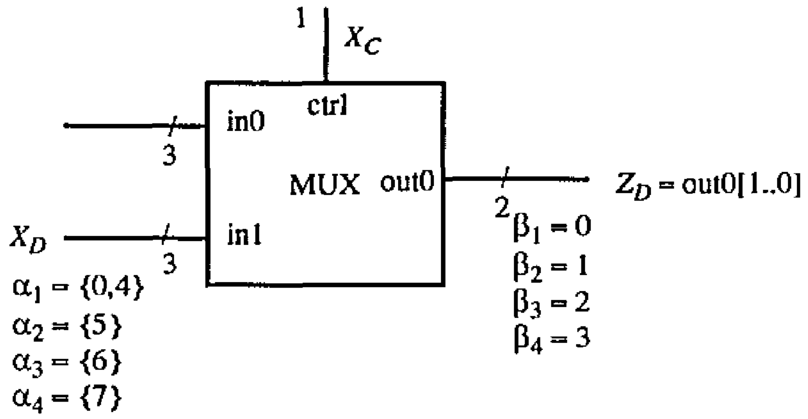


Figure 3.4. Propagation function P[MUX;(ctrl,in1);{0,4,5,6,7}] for $V(X_C) = 1$.

values in Ω.

Definition 3.1: Let M be a module with input control bus X_C and input data bus X_D, and let $\vec{F}$ be a subfunction vector derived from the function of M. Let α_i denote the ith block of the n-block partition π_Ω on Ω defined by $\vec{F}$, and let $\beta_i = \vec{F}(\alpha_i)$. Then the set $P[M;(X_C, X_D);\Omega] = \{(\alpha_1;\beta_1), (\alpha_2;\beta_2), \ldots, (\alpha_n;\beta_n)\}$ is a *propagation function* on Ω. $P[M;(X_C, X_D);\Omega]$ is said to be *based on* $\vec{F}$, π_Ω is said to be *embedded* in $P[M;(X_C, X_D);\Omega]$, and each pair $(\alpha_i;\beta_i)$ is referred to as a *block* of the propagation function.

Consider the 3-bit, 2-input multiplexer of Figure 3.1 shown again in Figure 3.4. Let $\Omega = \{0, 4, 5, 6, 7\}$ and $F = F$[(ctrl, in1), out0[1..0]](1,x), then $F(1,0) = F(1,4) = 0$, $F(1,5) = 1$, $F(1,6) = 2$, and $F(1,7) = 3$. Therefore, the propagation function on Ω based on F is

$$P\text{[MUX;(ctrl, in1);\{0,4,5,6,7\}]} = \{(0,4;0), (5;1), (6;2), (7;3)\} \quad (3.1)$$

which is depicted in Figure 3.4. The partition π_Ω embedded in P is $\{(0,4), (5), (6), (7)\}$.

As another example, consider the adder module in Figure 3.5. Here, X_D is the addend, X_C is the augend, and Z_D is the three most significant bits of the sum. Let $\Omega = \{0, 1, 2, 3\}$ and $\vec{F} = (F(0,x), F(1,x))$. Then, $\vec{F}(0) = (0,0)$, $\vec{F}(1) = (0,1)$, $\vec{F}(2) = (1,1)$, and $\vec{F}(3) = (1,2)$. Therefore, the propagation function on Ω based on $\vec{F}$ is

$$P\text{[ADDER; (in1, in2); \{0,1,2,3\}]} = \{(0;(0,0)), (1;(0,1)), (2;(1,1)), (3;(1,2))\} \quad (3.2)$$

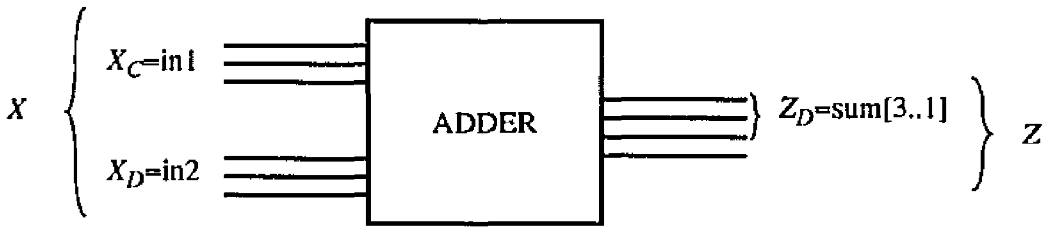


Figure 3.5. Three-bit adder module with input data bus X_D as addend, input control bus X_C as augend, and output data bus Z_D = sum[3..1].

and the embedded partition is $\pi_\Omega = \{(0), (1), (2), (3)\}$.

The mapping of inputs to outputs represented by a single block of a propagation function is similar to the mapping described by a singular cube, which is a representation of an incomplete Boolean function introduced by Roth [78], and widely used in describing logic synthesis algorithms. It is written $u|v$, where u is a set of values u_i for input variables (ports), and v is a set of values v_i for output variables (ports). Here each u_i or v_j must be either 0, 1, or d (don't care). For instance, the block $(0, 4;0)$ in equation (3.1) is equivalent to the cube $d00_2|00_2$.

Since propagation functions use partition theory, several definitions and operations derived directly from partition theory can be applied to them. For instance, the partition on S consisting of all singleton (one-element) blocks is the *zero partition*, and the partition consisting of a single block containing all elements of S is the *unit partition*. By analogy, the propagation function where the embedded partition π_Ω is the zero partition on Ω, and where $\alpha_i = \beta_i$ for all i, is the *zero propagation function*. The propagation function $P = \{(\alpha;\beta)\}$ where the embedded partition π_Ω is the unit partition on Ω and $\beta = \Omega$, is called the *unit propagation function*. Consider again the 3-bit, 2-input multiplexer of Figure 3.1. If $V(X_C) = 1$, then

$$P[MUX;(ctrl, in1);\{0,1,2,3\}] = \{(0;0), (1;1), (2;2), (3;3)\}$$

is the zero propagation function on Ω. If $V(X_C) = 0$, then

$$P[MUX;(ctrl, in1);\{0,1,2,3\}] = \{(0,1,2,3;\{0,1,2,3\})\}$$

is the unit propagation function on Ω.

If two values x_1 and x_2 are both in the same α_i, then they produce the same output $V(Z_D)$ and so they cannot be distinguished. In this case, some information is lost in propagating values from X_D to Z_D. Since every element of Ω is contained in the single α_i of the unit propagation function, the associated module propagates no information from X_D to Z_D. On the other hand, since each α_i of the zero propagation function contains exactly one value, the associated module propagates all information.

We now define some basic operations for combining propagation functions. These are used in the application of propagation functions to transparency analysis and error propagation, which are discussed later. We begin with the intersection operation for propagation functions which is derived from partition intersection (see also Appendix A).

Definition 3.2: Let P_1 and P_2 be propagation functions on the set Ω. The *intersection of* P_1 *and* P_2, denoted $P_1 \cap P_2$, is the propagation function on Ω such that

1. If π_{Ω_1} is the partition embedded in P_1, π_{Ω_2} is the partition embedded in P_2, and π_{Ω_3} is the partition embedded in $P_1 \cap P_2$, then $\pi_{\Omega_3} = \pi_{\Omega_2} \cap \pi_{\Omega_2}$, the partition intersection of π_{Ω_1} and π_{Ω_2};
2. If $x_1 \in \alpha_i$ in P_1 and $x_1 \in \alpha_j$ in P_2, then there is a block $(\alpha_k;\beta_k)$ in $P_1 \cap P_2$ such that $x_1 \in \alpha_k$ and $\beta_k = (\beta_i, \beta_j)$.

A method of computing intersection is implicit in the definition. To illustrate intersection, consider the adder module in Figure 3.5. Let $\Omega = \{0, 1, 2, 3\}$, let P_1 be the propagation function with $V(X_C) = 0$, and let P_2 be the propagation function with $V(X_C) = 1$. Then

$$P_1 = \{(0, 1;0), (2, 3;1)\}$$

$$P_2 = \{(0;0), (1, 2;1), (3;2)\}$$

$$P_1 \cap P_2 = \{(0;(0, 0)), (1;(0, 1)), (2;(1, 1)), (3;(1, 2))\}$$

$P_1 \cap P_2$ is the same as the propagation function P in equation (3.1). P_1 and P_2 are based on subfunction vectors $\vec{F}_1$ and $\vec{F}_2$ respectively, and P is based on subfunction vector $\vec{F}$. Note that $\vec{F}$ is the concatenation of $\vec{F}_1$ and $\vec{F}_2$, that is, $\vec{F} = (\vec{F}_1, \vec{F}_2)$. This situation is formalized in the following theorem

Theorem 3.1: *Let* P_1 *and* P_2 *be the propagation functions based on subfunction vectors* $\vec{F}_1$ *and* $\vec{F}_2$, *respectively. Then* $P_1 \cap P_2$ *is based on* $(\vec{F}_1, \vec{F}_2)$.

Theorem 3.1 follows directly from part 2 of Definition 3.2. A number of key aspects of propagation are based on propagation function intersection, as we will show later.

We can also define a union operation for propagation functions based on partition union.

Definition 3.3: Let P_1 and P_2 be propagation functions on the set Ω. The *union of* P_1 *and* P_2, denoted $P_1 \cup P_2$, is the propagation function on Ω such that

1. If π_{Ω_1} is the partition embedded in P_1, π_{Ω_2} is the partition embedded in P_2, and π_{Ω_3} is the partition embedded in $P_1 \cup P_2$, then $\pi_{\Omega_3} = \pi_{\Omega_2} \cup \pi_{\Omega_2}$, the partition union of π_{Ω_1} and π_{Ω_2};
2. If $(\alpha_i;\beta_i)$ is a block of $P_1 \cup P_2$, $(\alpha_j;\beta_j)$ is a block of either P_1 or P_2, and if $\alpha_i \cap \alpha_j \neq \varnothing$, then for all k, the kth element of β_j is contained in the kth element of β_i.

The union operation can be computed by generating the partition union using only the α's of P_1 and P_2, then forming the elementwise union of the β's. As an example of this operation, let $P_1 = \{(0,1;0),(2,3;1)\}$ and $P_2 = \{(0;0),(1,2;1),(3;2)\}$, as in the intersection example above. Then $P_1 \cup P_2 = \{(0,1,2,3;\{0,1,2\})\}$

Many of the properties of a propagation function P on Ω depend only on the embedded partition π_Ω. This partition determines the information propagated by the module subfunction on which P is based. Let P_1 and P_2 be two propagation functions based on subfunction vectors $\vec{F}_1$ and $\vec{F}_2$, respectively. If the embedded partition of P_1 and P_2 is the same, then $\vec{F}_1$ and $\vec{F}_2$ propagate the same information.

Definition 3.4: Let P_1 and P_2 be module functions on a set Ω. Let π_{Ω_1} be the partition embedded in P_1, and π_{Ω_2} be the partition embedded in P_2. Then P_1 and P_2 are *congruent*, denoted $P_1 \equiv P_2$, if and only if $\pi_{\Omega_1} = \pi_{\Omega_2}$. Let $(\alpha_i;\beta_i) \in P_1$ and $(\alpha_j;\beta_j) \in P_1$. Then as a special case, $P_1 = P_2$ if and only if $P_1 \equiv P_2$ and $\alpha_i = \alpha_j$ implies $\beta_i = \beta_j$.

For example, if $P_1 = \{(0;0),\{1;1\}\}$ and $P_2 = \{(0;1),(1;2)\}$, then $P_1 \equiv P_2$. The congruence relation is important in the application of propagation functions. It is commonly the case that two propagation functions are congruent, but not equal. However, propagation functions need not be equal to propagate the same information—they need only be congruent.

The congruence relation is clearly reflexive, transitive, and symmetric, Therefore, the set of all propagation functions on a set Ω is partitioned into equivalence classes by congruence. Two propagation functions are in the same class if and only if they are congruent. Each class is defined by the embedded partition common to all elements within the class.

As mentioned, the zero propagation function with one element in each α_i transmits all information, while the unit propagation function, with only one block containing all elements of Ω transmits none. By extension, propagation functions that have more blocks with fewer elements transmit more information than propagation functions with fewer blocks and more elements per block. This property is expressed algebraically in the next definition.

Definition 3.5: Let P_1 and P_2 be propagation functions on a set Ω. Let $\pi_{\Omega 1}$ be the partition embedded in P_1 and $\pi_{\Omega 2}$ be the partition embedded in P_2. Then P_1 *is less than or congruent to* P_2, denoted $P_1 \leq P_2$, if and only if $\pi_{\Omega 1} \leq \pi_{\Omega 2}$. If P_1 and P_2 are not congruent, then we can say that P_1 *is strictly less than* P_2, denoted $P_1 < P_2$. If $P_1 \leq P_2$ or $P_2 \leq P_1$, then P_1 and P_2 are said to be *comparable*, otherwise they are incomparable.

For example, let $P_1 = \{(0,1,2,3;0), (4,5,6,7;1)\}$ and $P_2 = \{(0,1;0), (2,3;1), (4,5;2), (6,7;3)\}$, and let $\pi_{\Omega 1}$ and $\pi_{\Omega 2}$ be the embedded partitions of P_1 and P_2, respectively. Then $\pi_{\Omega 1} \leq \pi_{\Omega 2}$, since all of the blocks of $\pi_{\Omega 1}$ are contained in blocks of $\pi_{\Omega 2}$ (see Appendix A). Therefore $P_2 \leq P_1$, and more information is propagated by P_2 than P_1.

Clearly, if $P_1 \leq P_2$ and $P_2 \leq P_1$, then $P_1 \equiv P_2$. The set P_Ω of all propagation functions on a set Ω, together with the ordering relation $\leq$ is a partially ordered set, since the set Π_Ω of all partitions on Ω is a partially ordered set. In fact, the set Π_Ω, together with the partition intersection $\cap$ and union $\cup$ operations form a lattice denoted $(\Pi_\Omega, \cap, \cup)$. Lattices are algebras characterized by the fact that the two operations satisfy the idempotence, commutative, associative, and absorption laws (see Appendix A). The algebra formed by P_Ω and the propagation function intersection and union operations $(P_\Omega, \cap, \cup)$ is homomorphic to $(\Pi_\Omega, \cap, \cup)$, that is, the operations behave the same way in each algebra (see Appendix A). For example, let x_1 and x_2 be elements of Π_Ω, then the commutativity property for partition intersection can be written $x_1 \cap x_2 = x_2 \cap x_1$. On the other hand, if x_1 and x_2 are elements of P_Ω, then the commutativity property for propagation intersection can be written $x_1 \cap x_2 \equiv x_2 \cap x_1$. In general, if we replace equality by the congruence relation, then $(P_\Omega, \cap, \cup)$ can be considered a lattice—all lattice properties and theorems apply. Therefore, we will refer to the algebra $(P_\Omega, \cap, \cup)$ as the *propagation function lattice.* A lattice with only two elements is a Boolean algebra, therefore, if $\Omega = \{0, 1\}$, any propagation function on Ω is a Boolean algebra and subject to laws similar to those for standard logic gates.

The classification of $(P_\Omega, \cap, \cup)$ as a lattice makes a wealth of theorems and properties applicable [14,42]. We will show that lattice intersection is extremely useful in analyzing propagation through circuits. Other lattice concepts are also useful.

3.1.2 Module Connections and Propagation Algebras

We have defined propagation functions to represent specific input-output mappings for individual modules and showed that together with the intersection and union operations they form a lattice, an algebra with several useful properties. We now show how to combine propagation functions for individual modules into a propagation function for a multi-module circuit using appropriate connection operations. These operations form a separate, derived algebra which we use to analyze the information transmission properties of multi-module circuits.

Most circuits composed of high-level modules can be modeled as directed graphs in which modules are edges and connections are vertices, since most modules are unidirectional. Even circuits with tristate buses can often be modeled as directed graphs for particular operation cycles [12]. Since all connections in a directed graph can be treated as series or parallel connections (see Appendix A), we can model the behavior of the circuits of interest using operations based on just these two fundamental types of connections.

Two modules M_1 and M_2 are connected in *series* if a bus joins the output data bus of module M_1 to the input data bus of M_2. In order to define a parallel connection, we must first define two types of junctions: fanout connections and merge connections. If a bus L is connected to the input data buses (X_D's) of $n > 1$ modules, then the junction of L and these data buses is a *fanout junction.* Information propagated on L is copied to the inputs of the n modules. Now, let $L_1, \ldots, L_n$ be a set of buses that connect the output data buses (Z_D's) of $n > 1$ modules to primary outputs or to the input data bus of a single module. $L_1, \ldots, L_n$ are concatenated to form a single bus L. The point where $L_1, \ldots, L_n$ becomes L is referred to as a *merge junction.* Information that is propagated independently on each L_i is combined with the information on the other buses to form

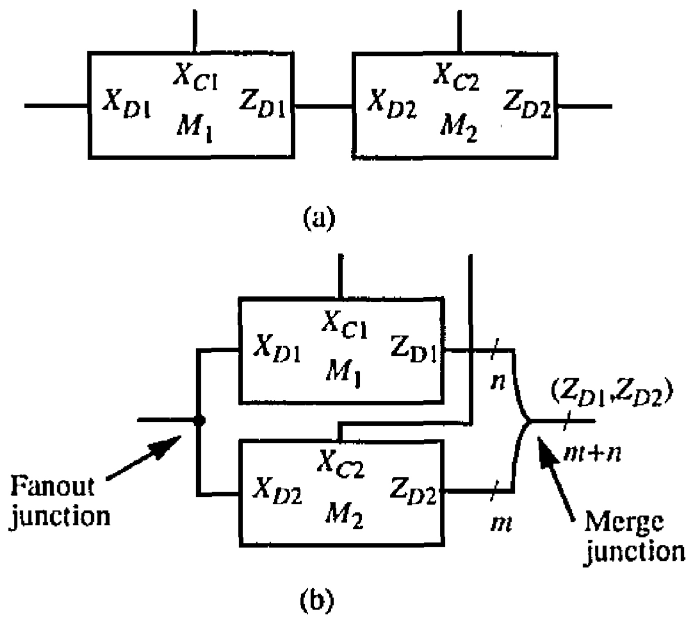


Figure 3.6. Series (a) and parallel (b) connections of modules.

a single information unit. Two modules are connected in *parallel* if their input data buses meet at a fanout junction and their outputs meet at a merge junction. Examples of series and parallel connections are shown in Figures 3.6a and 3.6b, respectively.

In classical circuit theory, where electrical components such as resistors, inductors, and capacitors form the edges of a graph G, and component connections form vertices of G, series-parallel connections greatly simplify analysis. Let C_1 and C_2 be two circuit components with admittances Y_1 and Y_2, respectively. If C_1 and C_2 are connected in parallel, then the admittance of the combination is given by $Y = Y_1 + Y_2$. If C_1 and C_2 are connected in series, then their combined admittance is given by

$$\frac{1}{Y} = \frac{1}{Y_1} + \frac{1}{Y_2} = \frac{Y_1 + Y_2}{Y_1 Y_2} . \tag{3.3}$$

If we define a "reduced sum" operation * as $A*B = \frac{AB}{A+B}$, then we can express equation (3.3) as $Y = Y_1 * Y_2$ [28]. We now have a parallel connection operation + and a series connection operation * for admittances. Propagation functions represent information transmission properties of modules and circuits that are analogous to admittances, which represent electrical current transmission capability. Therefore, we will define parallel and series connection operations for propagation functions that are analogous to + and *.

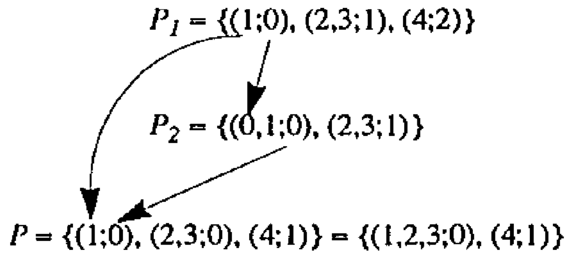
The series and parallel connections for modular bus-structured circuits shown in Figure 3.6 are also similar in concept to the connections in switching circuits studied first by Shannon [86]. However, in Shannon's model, the outputs of switches (modules) in parallel are wired together and information is transmitted serially after the junction. In our model, the outputs remain separate and information is propagated along multi-bit buses in parallel. For analysis purposes, we merge information on the parallel buses into a single vector. Shannon's model leads to a definition of series and parallel connection operations that form a Boolean algebra. The module connections described above lead to a different algebra, which we describe next.

If two modules M_1 and M_2, with propagation functions P_1 and P_2 respectively, are connected in parallel, then the combination can be considered to be one module, with one input data bus and two output data buses. We saw above that the propagation function for the combined module is given by the intersection of P_1 and P_2. Thus we have a partial correspondence between connection operations and the propagation function lattice. We denote the parallel connection operation as $P_1 \# P_2$.

Now let M_1 and M_2 be two modules connected in series to form a module with one input data bus and one output data bus. If F_1 and F_2 are module subfunctions for M_1 and M_2, respectively, then the resulting module subfunction for the series combination is the composition $F_2(F_1)$ of F_1 and F_2, that is, the outputs of F_1 form the inputs of F_2. The function $F_2(F_1)$ is denoted $F_1 \circ F_2$. The composition of two subfunction vectors $\vec{F}_1 = (F_{11}, F_{12}, \ldots, F_{1k})$ and $\vec{F}_2 = (F_{21}, F_{22}, \ldots, F_{2k})$ is the pairwise composition of the individual elements.

$$\vec{F}_1 \circ \vec{F}_2 = (F_{11} \circ F_{21}, F_{12} \circ F_{22}, \ldots, F_{1k} \circ F_{2k})$$

Now let $P_1 = \{(1;0), (2,3;1), (4;2)\}$ and $P_2 = \{(0,1;0), (2,3;1)\}$ be propagation functions based on $\vec{F}_1$ and $\vec{F}_2$, respectively, and let P be the propagation function based on $\vec{F}_1 \circ \vec{F}_2$. When P_1 and P_2 are combined to form the series composition P, the output part β_{1i} of each block in P_1 must be contained in the input part α_{2j} of some block in P_2, since the outputs of $\vec{F}_1$ become the inputs of $\vec{F}_2$. For instance, (1;0) is a block in P_1, and (0,1;0) is a block in P_2. The 0 in the output part of (1;0) corresponds to the 0 in the input part of (0,1;0). If $\beta_{1i} \in \alpha_{2j}$, then a block is added to P whose input part is α_{1i} (1 in this example), and whose output part is β_{2j} (0 in this example). This is depicted in Figure 3.7.



$P_1 = \{(1;0), (2,3;1), (4;2)\}$
$P_2 = \{(0,1;0), (2,3;1)\}$
$P = \{(1;0), (2,3;0), (4;1)\} = \{(1,2,3;0), (4;1)\}$

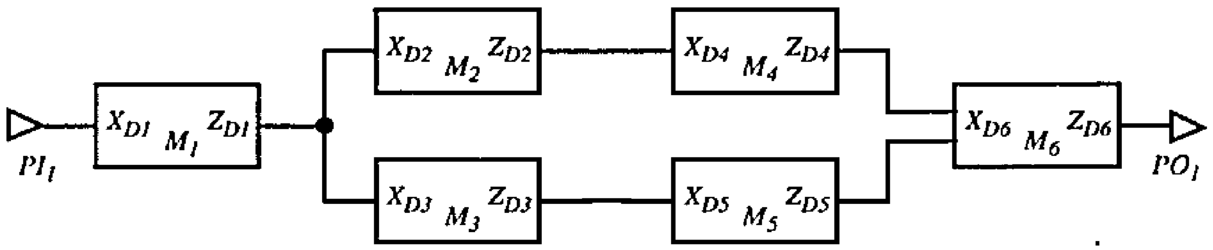
Figure 3.7. The series composition of two propagation functions P_1 and P_2.

In the general case, the β_i's of a propagation function are not singletons, they are vectors of sets. As discussed above, the composition of two vectors is the vector formed by the pairwise composition of the elements. Let $\beta_i[j]$ denote the jth element of the vector β_i. The following definition covers the general case of series composition.

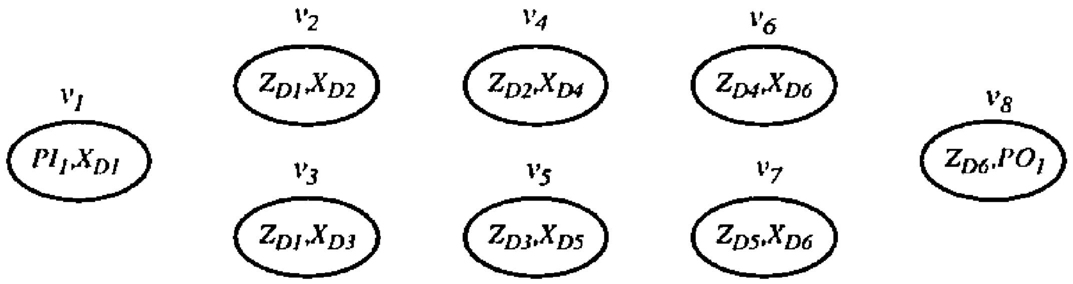
Definition 3.6: Let P_1 and P_2 be propagation functions on the sets Ω_1 and Ω_2 respectively. The *series connection* of P_1 and P_2, denoted $P_1{}^\circ P_2$, is the module function on Ω_1 such that if $(\alpha_{1i};\beta_{1i}) \in P_1$, $(\alpha_{2j};\beta_{2j}) \in P_2$, and $\beta_{1i}[q] \cap \alpha_{2j} \neq \varnothing$, then there is a block $(\alpha_k;\beta_k) \in P_1{}^\circ P_2$ where $\alpha_{1i} \subseteq \alpha_k$ and $\beta_{2j}[q] \subseteq \beta_k[q]$. If z is an element of some $\beta_{1i}[q]$, then z must be an element of some α_{2j}, otherwise $P_1{}^\circ P_2$ is *inconsistent*.

The series connection operation can be computed by a straightforward application of the definition. As an example, let $P_1 = \{(0,1;(\{0,1\},\{0,1\})), (2;(\{0,1\},\{1,2\})), (3;(\{1,2\},\{1,2\}))\}$ and $P_2 = \{(0;(\{0,1\},\{0,1\})), (1;(\{0,1\},\{1,2\})), (2,3;(\{1,2\},\{1,2\}))\}$, then $P_1{}^\circ P_2 = \{(0,1;(\{0,1\},\{0,12\})), (2;(\{0,1\},\{1,2\})), (3;(\{0,1,2\},\{1,2\}))\}$.

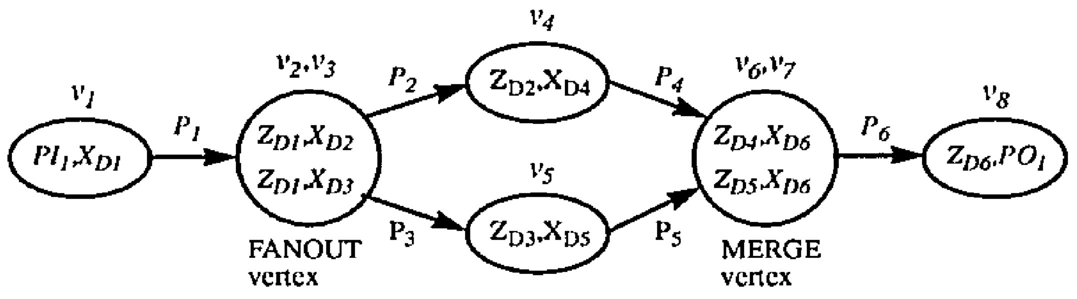
The series connection operation for propagation functions is obviously quite different from the union operation. The latter depends only on interactions between the input parts of propagation function blocks (the α's). The union operation is not even defined for P_1 and P_2 above, since these two propagation functions are defined on different sets of module inputs. Therefore, the algebra formed by the set Ψ of all propagation functions and the series and parallel connection operations just discussed, is not the same as the propagation function lattice described in Section 3.1.1. We refer to this new algebra, $(\Psi, \#, {}^\circ)$ as the *propagation algebra*, and we will examine its properties below. However, first we discuss a graph representation of circuits that illustrates the



(a) Datapath circuit C_1



(b) Connection vertices for C_1

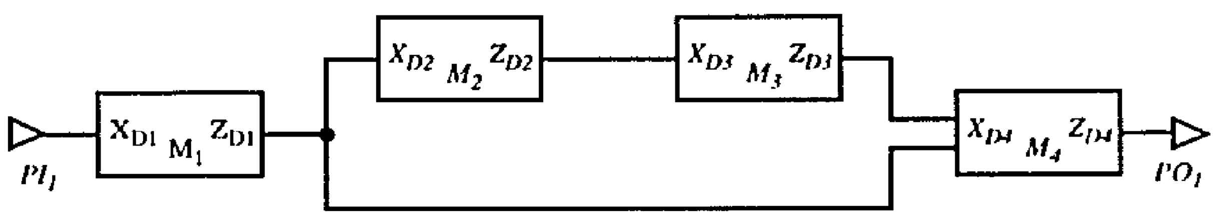


(c) Propagation diagram for C_1

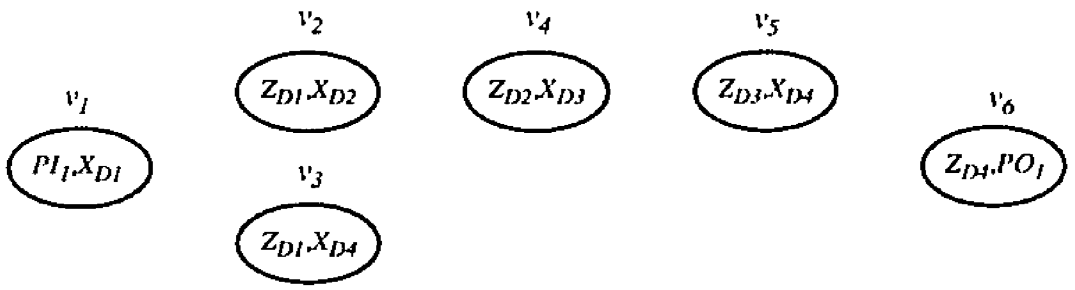
Figure 3.8. Example of a propagation diagram for a circuit with series and parallel connections.

series-parallel structure of modular, high-level logic circuits. This graph defines an expression in the propagation algebra.

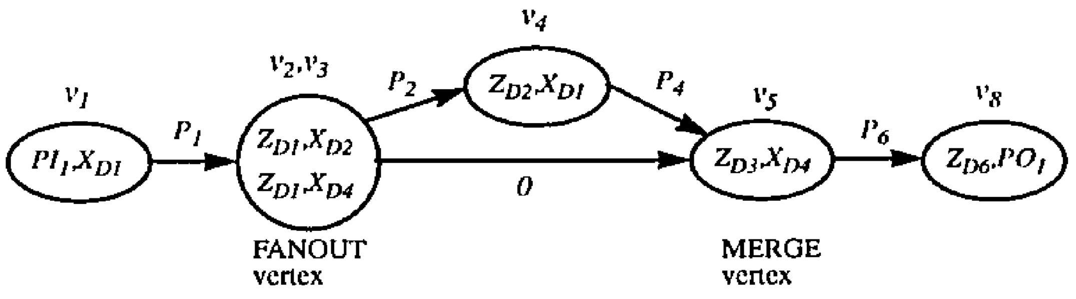
A *propagation diagram* is a directed graph whose edges represent propagation functions and whose vertices are module interconnections. Examples of propagation diagrams and the process used to construct them are shown in Figures 3.8 and 3.9. Circuit C_1 in Figure 3.8a is an example of a datapath circuit with series and parallel connections. Data bus inputs and outputs have been selected for each of the modules in this circuit; control buses are not shown. Data propagated to the output of module M_1 fans out along two parallel paths and reconverges at module M_6.



(a) Datapath circuit C_2



(b) Connection vertices for C_2



(c) Propagation diagram for C_2

Figure 3.9. Example of a propagation diagram for a datapath circuit C_2 with a bus connected in parallel with two modules.

The first step in the development of a propagation diagram is to create a vertex for every module-to-module connection. Vertices are labeled by pairs of the form (Z_{Di}, X_{Dj}), (PI_k, X_{Dj}), or (Z_{Di}, PO_k), where PI_k is a primary input and PO_k is a primary output. The complete set of vertices produced by this step is shown in Figure 3.8b. We do not allow input data buses or output data buses to be included in more than one vertex, therefore, vertex v_2 is combined with vertex v_3, and v_6 is combined with v_7. Vertices with the same Z_{Di} are combined into a FANOUT vertex, for example, (v_2, v_3). Vertices with the same X_{Di} are combined into a MERGE vertex, for example, (v_6, v_7). Finally, an edge from a vertex v_i to a vertex v_j represents a propagation function for the module whose input data bus is contained in v_i and whose output data bus is contained in v_j. The final propagation diagram is shown in Figure 3.8c.

Circuit C_2 shown in Figure 3.9a is an example of a module with a bus connected in parallel with two modules in series. This circuit results in the set of vertices shown in Figure 3.9b. In this case, vertices v_2, v_3, and v_5 are candidates for being combined into one vertex. However, a vertex may not be both a fanout vertex and a merge vertex—this would create a cycle. If a combined vertex contains pairs with the same Z_{Di} and X_{Dj}, we separate the pairs into two vertices so that pairs with the same Z_{Di} are in a fanout vertex, and the rest are in a merge vertex. Therefore, we combine vertices v_2 and v_3 into a fanout vertex and v_5 becomes a merge vertex. The two new vertices are connected by an edge labeled 0 for the zero propagation function. This edge represents a bus whose only function is to transmit information with no modification.

Expressions in the propagation algebra (*propagation expressions*) represent the transmission of information in a set Ω from a primary input port or output data bus of a module M_1 to a primary output or output data bus of another module M_2. A propagation expression can be constructed by traversing the propagation diagram from one vertex to another. The expression representing any path $P_1, P_2, \ldots, P_n$ in the diagram is the series connection of the propagation functions on the path, $P_1 \circ P_2 \circ \ldots \circ P_n$. The expression on the outgoing edge of a merge vertex is the parallel connection of the expressions on the incoming edges. For example, the expression corresponding to the propagation of information from the input port of C_1 to the output port is

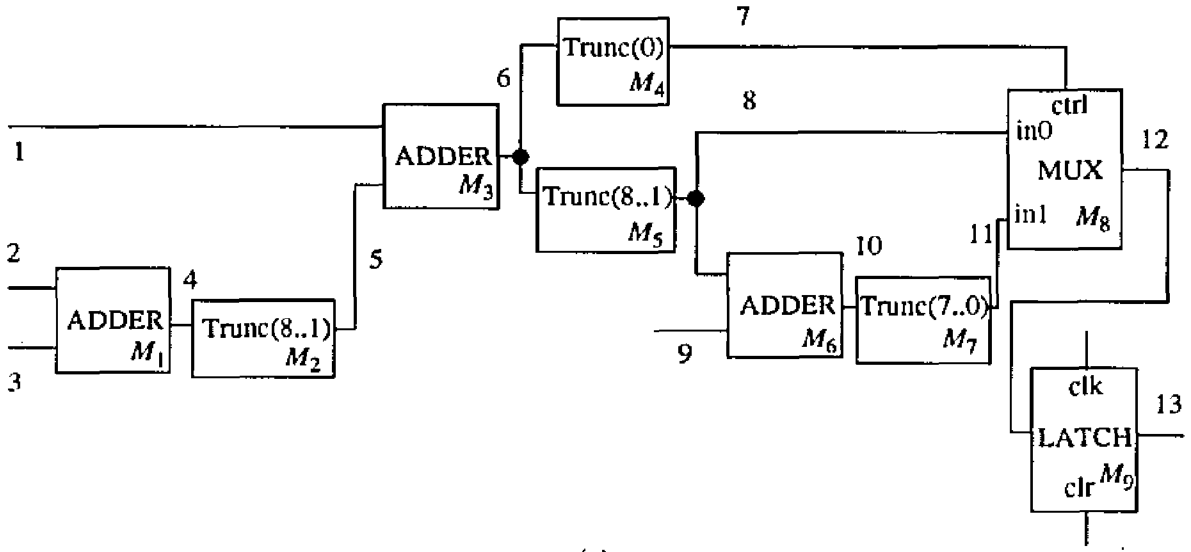
$$(P_1 \circ P_2 \circ P_4 \# P_1 \circ P_3 \circ P_5) \circ P_6 \tag{3.4}$$

and the propagation expression for C_2 is

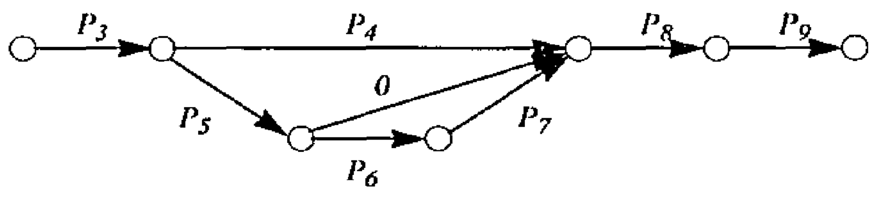
$$(P_1 \circ P_2 \circ P_3 \# P_1 \circ 0) \circ P_4 . \tag{3.5}$$

In order to reduce the number of parentheses in propagation expressions and improve readability, we have assigned a higher precedence to the series connection operation $\circ$.

To illustrate propagation diagrams and propagation expressions in terms of a familiar circuit, consider the Fltrdp circuit first discussed in Section 2.5. Fltrdp has been redrawn in Figure 3.10a with bus truncations explicitly represented by truncate modules. As before, all modules and buses are numbered. The output buses of the adders, modules M_1, M_3, and M_6, are 9 bits wide. Bus 7 is 1 bit wide. The rest of the buses are 8 bits wide. The propagation diagram shown in



(a)



(b)

Figure 3.10. Fltrdp datapath circuit, (a) schematic and (b) propagation diagram.

Figure 3.10b represents a propagation path from input bus 1 to the output bus 13. Here the connection labels in the vertices have been omitted for readability. The corresponding propagation expression is

$$P_3 \circ (P_4 \# (P_5 \circ (0 \# (P_6 \circ P_7)))) \circ P_8 \circ P_9 .$$

The propagation algebra has several useful properties, some of which are shown in Table 3.1. Also shown are several cases where properties do not hold (the numbers of these cases are in parenthesis). In this table, we assume that P_i is a propagation function on Ω_i for module M_i. We also assume that all series connection operations are consistent with real circuit connections. For instance, $P_1 \circ P_2$ is only defined if the output bus of M_1 is the same size as the input bus of M_2. Finally, we assume that if P_1 and P_2 are combined in parallel, then $\Omega_1 = \Omega_2$, that is, they have the same domain. We use the stronger equality relation in place of congruence whenever it is applicable. Note that the commutative, idempotent, and absorptive laws do not hold for the series connection operation $\circ$, thus the propagation algebra is not a lattice. However, since the computa-

Property	No.	Statement of property and its status
Identity	1a 1b	$0°P_1 = P_1$ and $P_1°0 = P_1$ for every element P_1. $1\#P_1 = P_1$ and $P_1\#1 = P_1$ for every element P_1.
Distributivity	2a (2b)	$P_1°(P_2\#P_3) = (P_1°P_2)\#(P_1°P_3)$ $P_1\#(P_2°P_3) \not\equiv (P_1\#P_2)°(P_1\#P_3)$
Commutativity	3a (3b)	$P_1\#P_2 \equiv P_2\#P_1$ $P_1°P_2 \not\equiv P_2°P_1$
Idempotence	4a 4b	$P_1\#P_1 \equiv P_1$ $P_1°P_1 \equiv P_1$ if and only if $P_1 \equiv 0$ or $P_1 \equiv 1$.
Absorption	5a (5b)	$P_1\#(P_1°P_2) \equiv P_1$ $P_1°(P_1\#P_2) \not\equiv P_1$
Associativity	6a 6b	$P_1\#(P_2\#P_3) \equiv (P_1\#P_2)\#P_3$ $P_1°(P_2°P_3) = (P_1°P_2)°P_3$
Miscellaneous	7 8 9	$P_1\#P_2 \leq P_1$ and $P_1\#P_2 \leq P_2$ $0\#P_1 \equiv 0$ $P_1 \leq P_1°P_2$

Table 3.1 Common algebra laws that do and do not hold for propagation algebra. The numbers of the laws that do not hold are in parenthesis.

tion of the parallel connection operation # is the same as the intersection operation in the propagation set lattice described above, the propagation algebra is a semi-lattice [42].

Apart from the four major properties of commutativity, idempotence, absorption, and associativity, Table 3.1 lists some other typical properties of algebras encountered in digital systems. For instance, the zero propagation function acts as an identity for the series connection operation, since connecting a bus to the input or output of a module M does not affect the propagation characteristics of M. Another important property is the distributive property 2a. This property allows us to rewrite expression (3.4), for instance, as $P_1°(P_2°P_4\#P_3°P_5)°P_6$. Distributivity implies that the propagation characteristics of a module are transferred along all paths leading away from its output. Rewriting propagation expressions can aid in analysis, as we show below. Property 7 indicates that two modules connected in parallel propagate at least as much information as each does individually. Property 8 indicates that the propagation characteristics of a bus in parallel with any module dominate the propagation characteristics of the combination. Finally, property 9 indicates that combining two modules in series can only reduce the amount of information that is propagated.

We can simplify propagation expressions, that is, rewrite them with fewer propagation function symbols, by using the properties in Table 3.1. The following series of steps simplify expression (3.5):

$(P_1 \circ P_2 \circ P_3 \# P_1 \circ 0) \circ P_4 = P_1 \circ (P_2 \circ P_3 \# 0) \circ P_4$ by property 2a

$P_1 \circ (P_2 \circ P_3 \# 0) \circ P_4 \equiv P_1 \circ 0 \circ P_4$ by property 7

$P_1 \circ 0 \circ P_4 \cong P_1 \circ P_4$ by property 1a

This set of simplifications illustrates the dominance of a transparent path in a parallel connection.

We have presented a very general theory for studying the propagation of information in bus-structured circuits. The two main concepts are:

1. Subfunctions and subfunction vectors define a partition on the set of information symbols to be propagated through a module or circuit. Hence, many aspects of information propagation can be studied by analogy to partition theory. We have formalized this insight in the form of propagation functions.

2. Bus-structured circuits can often be modeled for analysis as directed graphs forming series and parallel junctions at vertices. Each junction corresponds to an operation that combines propagation functions. The set of propagation functions and series and parallel connection operations form a propagation algebra, and a graph representing a circuit corresponds to a propagation algebra expression.

The propagation algebra satisfies several common algebraic laws and properties that reflect intuition about information propagation. These laws and properties can be applied to propagation expressions to manipulate them mechanically, thus raising the possibility of automating analysis.

To apply propagation theory in precomputed testing, we specialize propagation functions by restricting the set of input symbols Ω. We have studied two applications of this approach: transparency analysis and error propagation analysis. We discuss transparency analysis next and error propagation analysis in Chapter IV.

3.2. Transparency Analysis

A module or circuit is *transparent* if any error can be propagated through it. As discussed in Chapter II, transparency is a property of modules and circuits that greatly simplifies error propagation. If there is a known transparent path from a bus X in a circuit to a primary output, then any test response error (T_R, T_{Ri}) propagated to X is implicitly propagated to the primary outputs. We define a special type of propagation function to analyze this property. In this function, the sequences applied to X_C to control propagation are explicitly represented as a parameter. Recall that our convention is to use square brackets [...] for ordered sequences in time.

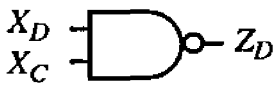
Definition 3.7: Let M be a module or circuit with input data bus X_D, input control bus X_C, and output data bus Z_D. Let $P_i = P\left[M; (X_C, X_D); \{V(X_D)\}\right]$ when $V(X_C)$ has the fixed value v_i, and let S_C be the vector sequence $[v_1, v_2, ..., v_k]$ for any k timesteps, $k \geq 1$. Then

$$T[M; (X_C, X_D)]\ (S_C) = P_1 \cap P_2 \cap \ldots \cap P_k$$

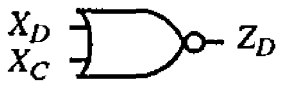
is a *transmission function* for module M.

Transmission functions describe the ability of a module to propagate information sequences (not just a specific set of values) from X_D to Z_D, as well as how this propagation is controlled by sequences applied to X_C. Consider the set of typical datapath modules in Figure 3.11 (which repeats Figure 2.5). Transmission functions for these modules with typical values for S_C and $k = 1$ or 2 appear in Table 3.2. We assume that the register (Figure 3.11g) has already been reset to 0. We assume that the decoder (Figure 3.11h) decodes only inputs 0 and 4, that is, the least significant bit of the output is set when the input is 0 (output value 1), and the most significant bit of the output is set when the input is 4 (output value 2). If the input is not 0 or 4, then the output is 0.

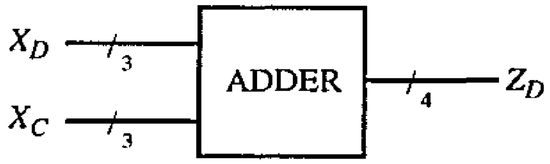
As previously discussed, if P_1 and P_2 are propagation functions on a set Ω for modules M_1 and M_2 respectively, then $P_1 < P_2$ implies that P_1 propagates more of the information contained in Ω than P_2. For a different domain Ω', it may be that $P_2 < P_1$. However, if $\Omega = \{V(X_D)\}$, as in the case of transmission functions, then $P_1 < P_2$ implies that M_1 propagates more information than M_2. The unit propagation function transmits no information, and any



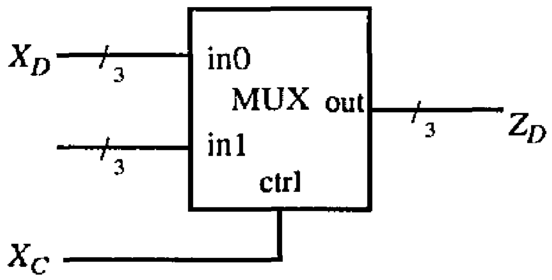
(a) NAND gate



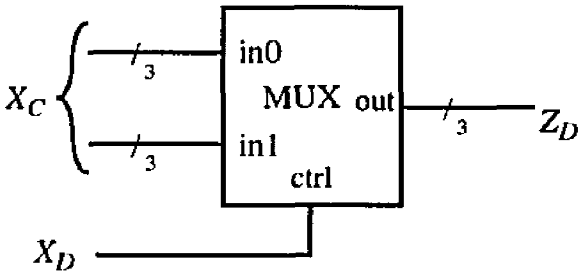
(b) NOR gate



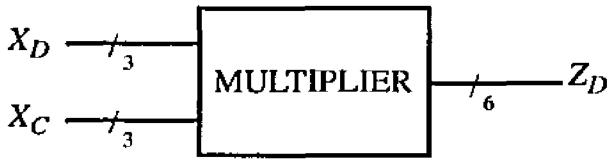
(c) Adder



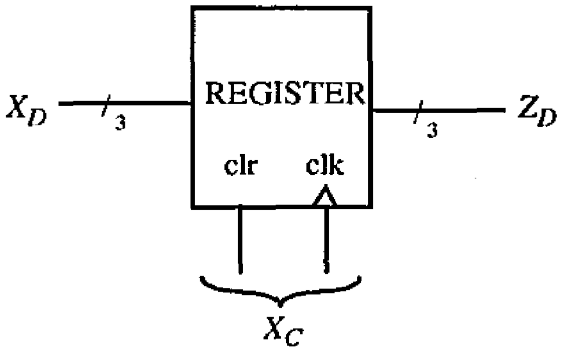
(d) Multiplexer 1



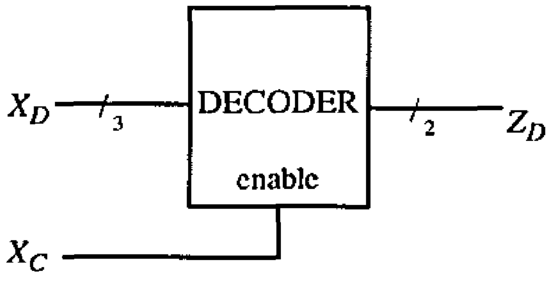
(e) Multiplexer 2



(f) Multiplier



(g) Register



(h) Decoder

Figure 3.11. Typical modules and their buses

Module	Transmission functions
(a) NAND gate	$T([0]) = \{(0,1;1)\}$ $T([1]) = \{(0;1), (1;0)\}$
(b) NOR gate	$T([0]) = \{(0;1), (1;0)\}$ $T([1]) = \{(0,1;0)\}$
(c) Adder	$T([0]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$ $T([1]) = \{(0;1), (1;2), (2;3), (3;4), (4;5), (5;6), (6;7), (7;8)\}$ $T([2]) = \ldots$
(d) Multiplexer 1	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$
(e) Multiplexer 2	$T([0]) = \{(0,1;0)\}$ $T([1]) = \{(0;0), (1;1)\}$ $T([2]) = \ldots$
(f) Multiplier	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$ $T([2]) = \ldots$
(g) Register	$T([0,1]) = \{(0;(0,0)), (1;(0,1)), (2;(0,2)), (3;(0,3)), (4;(0,4)),$ $(5;(0,5)), (6;(0,6)), (7;(0,7))\}$
(h) Decoder	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;1), (1,2,3,5,6,7;0), (4;2)\}$

Table 3.2 Transmission functions for the modules in Figure 3.11

transmission function $T(S_C) < 1$ propagates at least some information. This leads to the following definition.

Definition 3.8: Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D, and let $T(S_C)$ be a transmission function for M. If $T(S_C) < 1$, then $T(S_C)$ and the associated module M are said to be *sensitized* by S_C. Alternatively, $T(S_C)$ and M are said to be *partially transparent*, since at least some information is propagated by M.

Sensitization is a property of a module, and not dependent on a particular set of inputs applied to the module's X_D. It corresponds exactly to the conventional definition when applied to common logic gates. For instance, let M be a two-input NAND gate. Let X_D be one input port and X_C be the other; the output port is Z_D. In this case, $T([0]) = \{0,1;1)\}$ and $T([1]) = \{(0;1), (1;0)\}$.

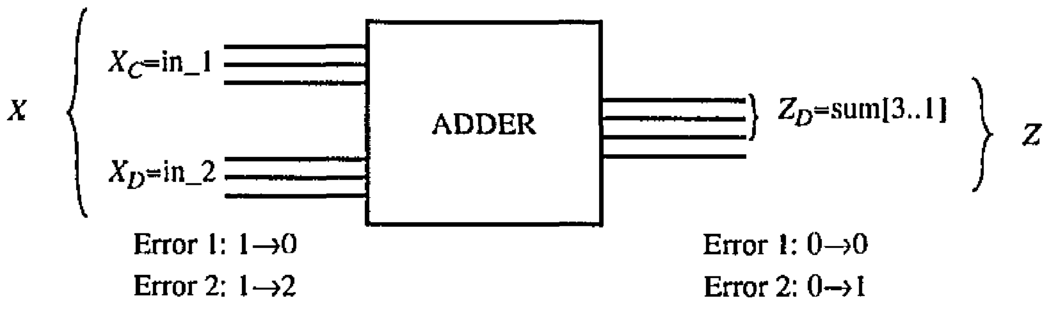


Figure 3.12. Three-bit adder module with input data bus X_D as addend, input control bus X_C as augend, and output data bus Z_D = sum[3..1].

Since $T([0]) \equiv 1$, the gate is not sensitized for the control input 0, as expected. Similarly, $T([1]) < 1$, so the gate is sensitized for the control input value 1.

Less obvious is the application of this definition of sensitization to larger, bus-structured modules. Consider the adder in Figure 3.12. As in previous examples, X_D is the addend, X_C is the augend, and Z_D comprises the three most significant bits of the sum. For this module, $T([1]) = \{(0;0), (1,2;1), (3,4;2), (5,6;3), (7;4)\}$. The adder is sensitized since $T([1]) < 1$; in fact, it is sensitized for all control inputs. However, some errors cannot be propagated from X_D to Z_D. For instance, if a fault causes a correct $V(X_D) = 1$ to be changed to 0 (error 1 in Figure 3.12), then the resulting error cannot be propagated through the adder. On the other hand, if an error changes $V(X_D) = 1$ to 2, then the error can be propagated. The special case when all errors are propagated is covered by the following theorem.

Theorem 3.2: *Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. M is transparent from X_D to Z_D for a particular control sequence V_C if and only if $T(V_C) \equiv 0$.*

The proof of Theorem 3.2 follows from the preceding discussion. Transmission functions provide a succinct definition of transparency that we can use in formal analysis.

3.2.1 Propagation Characteristics of Modules and Circuits

The structure of the buses connected to a module M greatly influences the transparency of M. For example, if $|X_D| > |Z_D|$ then M cannot be transparent from X_D to Z_D when the length of the control sequence $|S_C|$ for $T(S_C)$ is 1. In this case, M's domain is larger than its codomain. We will discuss the case where $|S_C| > 1$ in Section 3.4.

Most modules have only a few "natural" data bus pairs $(X_D;Z_D)$ with associated transparent propagation modes. For instance, in a multiplexer, each control bus value selects a different data input to propagate to the output. Using transmission functions, we can examine the behavior of less "natural" assignments of input ports to X_C and X_D. This is useful since library modules are often connected in unexpected ways when used in a circuit.

Theorem 3.3: *Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. Let $S_C = [v_1, v_2, ..., v_k]$ be a k-step sensitizing control sequence, that is, a sequence of k values assigned to X_C.*

(a) Let X_C' be an alternative input control bus such that $X_C \subset X_C'$ and let $S_C' = [v_1', v_2', ..., v_k']$ be a sequence of k values assigned to X_C'. If the value of each bit of v_i is the same as the corresponding bit of v_i' for $1 \le i \le k$, then

$$T[M;(X_C, X_D)](S_C) \equiv T[M;(X_C', X_D)](S_C')$$

(b) Let X_D' be an alternative input data bus. If $X_D \subset X_D'$, then

$$T[M;(X_C, X_D')](S_C) \not\leq T[M;(X_C, X_D')](S_C)$$

Proof: For part (a), let T be based on subfunction vector $\vec{F} = (F_1, ..., F_k)$. Recall that each F_i is itself a set of mappings, one mapping for each unspecified assignment to $X - (X_C - X_D)$. By making X_C larger, we specify more of the domain. This reduces the number of mappings, but does not change them; the embedded partition of T remains the same. For part (b), note that increasing the size of X_D increases the number of elements in each α_i without increasing the number of blocks. Therefore, blocks of the embedded partition of $T[M;(X_C, X_D')]$ cannot be contained in blocks of the embedded partition of $T[M;(X_C, X_D)]$. □

Theorem 3.3*a* implies that, for a given data bus pair $(X_D;Z_D)$, if the corresponding X_C is unknown, then we may safely choose $X_C = X - X_D$. If a control bus exists within $X - X_D$ for which a sensitizing assignment can be found, then there exists a sensitizing assignment for the entire set of ports $X - X_D$. Theorem 3.3(b) implies that there exists a transparent data bus pair $(X_D;Z_D)$ of maximum size. Intuitively, if making X_D larger reduces the transparency of a module, then there must be a more transparent X_D' embedded within every sensitized X_D.

As an illustration of Theorem 3.3, consider Multiplexer 1 in Figure 3.11d. Let $X_C{}' =$ (in0,ctrl) and $Z_D{}' =$ out[1..0], the two least significant bits of the output. As long as V(ctrl) = 1, the multiplexer is sensitized (but not necessarily transparent), regardless of V(in0). On the other hand, if X_C and Z_D are as shown in Figure 3.11d, and $X_D{}' =$ (in0,in1), then the transmission function will be less transparent than for $X_D =$ in1, since we are attempting to funnel more possible values to the same output.

Definition 3.9: Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. If $X_D{}' \subseteq X_D$ and $Z_D{}' \subseteq Z_D$, then $(X_D{}';Z_D{}')$ is a *subpath* of the data bus pair $(X_D;Z_D)$

The following corollary to Theorem 3.3, which is also noted by Marhöfer [66], illustrates the importance of subpaths.

Corollary 3.1: *Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. Then $T(S_C) < 1$ if and only if there is a subpath $(X_D{}';Z_D{}')$ such that the corresponding transmission function $T[M;(X_D{}', X_C)]\ (S_C) \equiv 0$.*

Proof: If $T(S_C) < 1$, then there are at least two non-empty blocks $(\alpha_i;\beta_i)$ and $(\alpha_j;\beta_j)$. Let $x_1 \in \alpha_i$ and $x_2 \in \alpha_j$, then x_1 and x_2 differ in at least one bit position. Let one of these bits be $X_D{}'$, let $Z_D{}' = Z_D$, and let the corresponding transmission function be $T'(S_C)$. Clearly, $T'(S_C)$ can contain only two blocks, and the embedded partition must be $\{\{0\}, \{1\}\}$. Therefore, $T'(S_C) \equiv 0$. Now assume that there is a transparent subpath $(X_D{}';Z_D{}')$. By Theorem 3.3b we know that increasing the size of $X_D{}'$ can make the corresponding transmission function less transparent. However, each unique $V(X_D{}')$ must still reside in a different block, therefore $T(S_C)$ can never be the unit propagation function. □

Continuing the example above for Multiplexer 1 in Figure 3.11d, let $Z_D{}' =$ out[1..0]. Then, T([1]) = {(0,4;0), (1,5;1), (2,6;2), (3,7;3)}, which demonstrates that the module is sensitized, but not transparent. If $X_D{}' =$ in1[1..0], then $(X_D{}';Z_D{}')$ is a subpath of $(X_D;Z_D)$ and the corresponding transparency function is $T([1]) = \{(0;0), (1;1), (2;2), (3;3)\} \equiv 0$.

We now turn our attention to the algebraic analysis of circuits for transparency. The goal of this analysis is to determine whether a propagation expression composed of transmission functions is congruent to the zero propagation function and therefore transparent, or else cannot be congruent to zero and therefore only either partially transparent or non-transparent (congruent to the unit transmission function). We frequently leave out the control sequence parameters in these expressions and assume that a controlling sequence exists (or does not exist) with the required transparency. Transmission functions can be treated as variables. For example, in a typical propagation expression $T_1 \circ (T_2 \# T_3)$, each transmission function "variable" T_i has a number of valid assignments, corresponding to various control sequences applied to the module associated with T_i.

Many key properties of the propagation algebra appear in Table 3.1. However, most of those properties apply only to parallel connections. Here we consider the transparency of series connections. A partially transparent module in a path consisting of series-connected modules usually reduces the transparency of the entire path. The following theorem makes this general statement precise.

Theorem 3.4: *Let T_1 and T_2 be transmission functions for modules M_1 and M_2 respectively.*

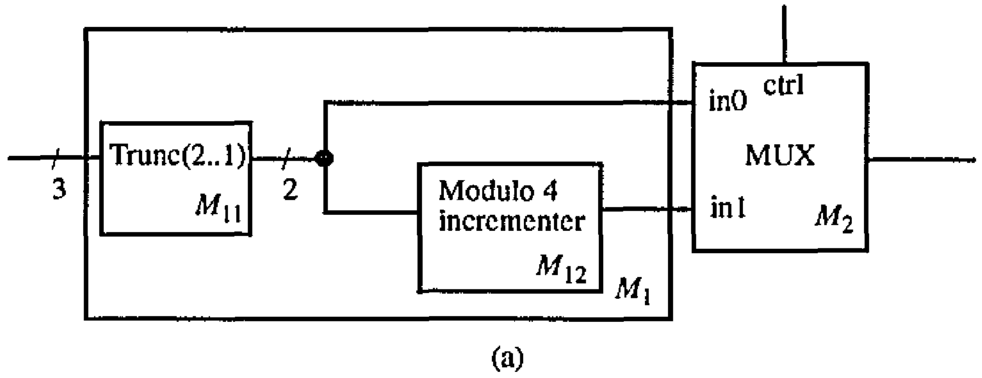
(a) $T_1 \circ T_2 \equiv 0$ *implies* $T_1 \equiv 0$.

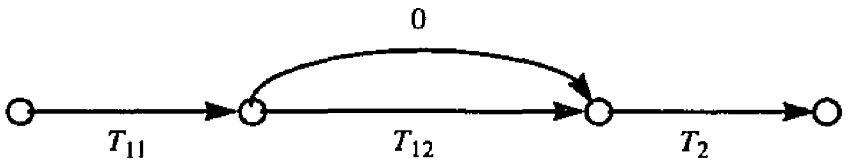
(b) $T_1 \circ T_2 \equiv T_1$ *if and only if there is no block* $(\alpha_{2i};\beta_{2i})$ *in* T_2, *and no two blocks* $(\alpha_{1j};\beta_{1j})$ *and* $(\alpha_{1k};\beta_{1k})$ *in* T_1 *such that* α_{2i} *contains* $\beta_{1j}[q]$ *(the qth element of* β_{1j}*) and* $\beta_{1k}[q]$ *for any q.*

Proof: Part (a) follows directly from property 9 of Table 3.1, that is, $T_1 \circ T_2 \geq T_1$. For part (b), by definition, $T_1 \circ T_2 \equiv T_1$ if and only if no blocks of T_1 are combined by the connection operation. This is true if and only if α_{2i} contains at most one $\beta_{1j}[q]$. □

Theorem 3.4*a* states that the first module in a series must be transparent for the entire series connection to be transparent. Property 4b in Table 3.1 is a special case of this theorem. Theorem 3.4b states that M_2 need not be transparent, but its corresponding transmission function must distinguish those values that can be generated by M_1. The module M_1 may add spurious or redundant information that is rejected by M_2.

Theorem 3.4 is useful for analyzing cases of reconvergent fanout. Consider the circuit in Figure 3.13a, which truncates the least significant bit of the input bus and propagates either the





(b)

Figure 3.13. (a) Circuit diagram and (b) propagation diagram for circuit with transparent reconvergent fanout.

resultant two-bit value v or v+1 (*mod* 4) to the output, depending on the control value applied to the multiplexer. The propagation diagram for this circuit appears in Figure 3.13b. As shown in Figure 3.13a, we can treat the circuit as the series connection of M_1 and M_2, where M_1 is hierarchically composed of modules M_{11} and M_{12}. Let T_i be the transmission function for module M_i. The following set of equations illustrate Theorem 3.4b.

$$T_{11} = \{(0,1;0),\ (2,3;1),\ (4,5;2),\ (6,7;3)\}$$

$$T_{12} = \{(0;1),\ (1;2),\ (2;3),\ (3;0)\}$$

$$T_{11}{}^\circ T_{12} = \{(0,1;1),\ (2,3;2),\ (4,5;3),\ (6,7;0)\}$$

$$T_1 = T_{11}\#T_{11}{}^\circ T_{12} = \{(0,1;1),\ (2,3;6),\ (4,5;11),\ (6,7;12)\}$$

$$T_2([0]) = \{(0,1,2,3;0),\ (4,5,6,7;1),\ (8,9,10,11;2),\ (12,13,14,15;3)\}$$

$$T_1{}^\circ T_2 = \{(0,1;0),\ (2,3;1),\ (4,5;2),\ (6,7;3)\} \equiv T_1$$

In this case, T_2 is not transparent, but $T_1{}^\circ T_2 \equiv T_1$.

In order to apply Theorem 3.4b in the analysis of modules in series, we must know the structure of all of the transmission functions in the series chain, which complicates the analysis.

Consider the Fltrdp datapath circuit of Figure 3.10. We can treat the network from the primary input bus 1 to the inputs of the multiplexer module 8 as a single module M. However, since the input to M (bus 1) is 8 bits wide, and the input to module 8 (all inputs form X_D) in this configuration is 17 bits wide, we must apply Theorem 3.4b to analyze transparency. The transmission function for module 8 has $2^{17} = 131,072$ elements to distribute among the blocks. It is therefore difficult to analyze this case by inspection.

Nevertheless, we can define a condition that enables us to make a strong statement about the transparency of the second module in a series connection. Let M be module with output data bus Z_D and let T be a transmission function for M. Let $|T|$ be the number of blocks in T. If $|Z_D| = \log_2|T|$, then T has the maximum number of blocks for a combinational function (the control sequence has length 0 or 1) and is said to be *bus-size limited.* The codomain of the subfunction upon which T is based contains all of the possible values that can be produced on Z_D. In this case, no spurious information can be added by M. The following corollary, which modifies Theorem 3.4b for the case in which M_1 is bus-size limited, summarizes the preceding discussion.

Corollary 3.2: *Let T_1 and T_2 be transmission functions for modules M_1 and M_2 respectively, and let Z_{D1} be the output data bus of M_1. If $|Z_{D1}| = \log_2|T_1|$, then $T_1 \circ T_2 \cong T_1$ if and only if $T_2 \cong 0$.*

Consider $T_{11} \circ T_{12}$ in the example above. T_{11} meets the requirement that $|Z_{D1}| = \log_2|T_{11}|$. Since $T_{12} \cong 0$, it is true that $T_{11} \circ T_{12} \cong T_{11}$. On the other hand, let $T_3 = \{(0,1;0), (2,3;2)\}$ (not congruent to zero). Then $T_{11} \circ T_3 = \{(0,1,2,3;0), (4,5,6,7;2)\} \ncong T_{11}$.

A primary goal in developing a theory of propagation is to uncover methods for analyzing propagation in circuits with irregular buses. While error propagation may be blocked by non-transparency along any single path in a circuit, the combination of paths in parallel can allow all test response errors (T_R, T_{Ri}) to be propagated. Some errors can be propagated along one path, and the rest along other paths. This combination of paths in parallel creates a distributed propagation path. We can determine whether such a path is transparent by intersecting the transmission functions of its constituent single paths. If the result is zero, then the distributed path is transparent.

Let M_1 and M_2 be two modules connected in parallel, and let T_1 and T_2 be transmission functions corresponding to M_1 and M_2 respectively. If $T_1 > 0$, $T_2 > 0$, and $T_1 \# T_2 \cong 0$, then the connection is a distributed transparent path, and T_1 and T_2 are *complements*. Unfortunately, propagation functions do not have unique complements—a key algebraic property. For example, consider

$$T_1 = \{ (0, 1;0), (2, 3;1), (4, 5;2), (6, 7;3) \} \tag{3.6}$$

$$T_2 = \{ (0, 2, 4, 6;0), (1, 3, 5, 7;1) \} \tag{3.7}$$

$$T_3 = \{ (0, 2;0), (1, 3;1), (4, 6;2), (5, 7;3) \} \tag{3.8}$$

Since $T_1 \# T_2 \cong 0$ and $T_1 \# T_3 \cong 0$, both T_2 and T_3 are complements of T_1. The lack of a unique complement sometimes limits our ability simplify propagation expressions algebraically.

Modules with different types of transmission functions may be combined in parallel to produce a circuit with distributed transparency. In general, we must compute the intersection of the corresponding transmission functions in order to determine transparency. However, in some cases we can simplify propagation expressions algebraically, without explicitly computing the intersection. To show this, we make use of special pairs of complements which we now define.

Definition 3.10: Let $T \# T_{max} \cong 0$ such that there is no $T_a > T$ and no $T_b > T_{max}$, where $T \# T_b \cong 0$ or $T_a \# T_{max} \cong 0$. Then T_{max} is said to be a *maximal complement* of T_1, and vice versa.

Among comparable transmission functions, maximal complements are unique up to congruence. Maximal complements have useful properties similar to unique complements in some cases. T_1 (3.6) and T_2 (3.7) given in the example above are maximal complements, while T_1 and T_3 (3.8) are not. The following theorem shows how maximal complements can determine a specific transparency requirement for parallel propagation paths.

Theorem 3.5: *Let M_1, M_2, M_3, and M_4 be modules connected as in Figure 3.14 with transmission functions T_1, T_2, T_3, and T_4, respectively, and let Z_{D1} and Z_{D3} be the output data buses of M_1 and M_3. If M_1 and M_3 are bus-size limited, that is, $|Z_{D1}| = \log|T_1|$ and $|Z_{D3}| = \log|T_3|$, and if T_1 and T_3 are maximal complements, then*

$$T_1 \circ T_2 \# T_3 \circ T_4 \cong 0 \textit{ if and only if } T_2 \cong 0 \textit{ and } T_4 \cong 0.$$

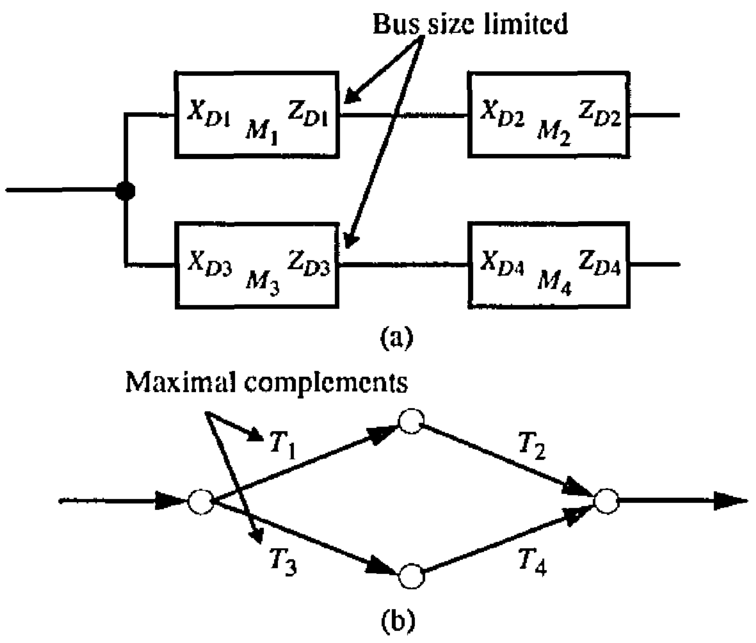


Figure 3.14. (a) Module connections for Theorem 3.5, and (b) corresponding propagation diagram.

Proof: Clearly, if $T_2 \equiv 0$ and $T_4 \equiv 0$, then $T_1 \circ T_2 \# T_3 \circ T_4 \equiv 0$, since $T_1 \circ T_2 \equiv T_1$ and $T_3 \circ T_4 \equiv T_3$. Now let $T_1 \circ T_2 \# T_3 \circ T_4 \equiv 0$ and suppose that $T_2 > 0$. By Property 9 (Table 3.1), $T_1 \circ T_2 \geq T_1$, but according to Corollary 3.2, $T_1 \circ T_2 \not\equiv T_1$. This means that $T_1 \circ T_2 > T_1$ and $T_1 \circ T_2 \# T_3 \circ T_4 \not\equiv 0$ by definition of maximum complement; we have a contradiction. Therefore $T_2 \equiv 0$. Clearly, $T_4 \equiv 0$ by the same reasoning. □

According to Theorem 3.5, if M_1 and M_2 are two bus-size limited modules connected in parallel whose transmission functions are maximal complements, then the only way that the resultant distributed propagation path can be transparent is if all modules connected in series with M_1 and all modules connected in series with M_2 are transparent. This implies that each of M_1 and M_2 propagates only the information that the other does not.

Theorem 3.5 is often useful in analyzing circuits. If the set of modules connected to a fanout point X are bus-size limited with transmission functions that are maximal complements, then non-transparency in any branch of the distributed propagation path beginning at X makes the entire circuit non-transparent. Conversely, Theorem 3.5 defines a transparency requirement for the modules on parallel propagation paths that can be determined algebraically. We will present specific examples using Theorem 3.5 in circuit analysis below. First, we consider the types of modules whose transmission functions have the required properties.

Theorem 3.6: *Let M_1 and M_2 be two truncate modules with the same input data bus X_D such that M_1 propagates only bits of X_D not propagated by M_2 and vice versa. Let T_1 and T_2 be the propagation functions of M_1 and M_2, respectively. Then T_1 and T_2 are maximal complements.*

Proof: Clearly, $T_1 \# T_2 \equiv 0$. We need to show that there is no $T_a > T_1$ and no $T_b > T_2$ such that $T_1 \# T_b \equiv 0$ or $T_a \# T_2 \equiv 0$. Let $(\alpha_{1i};\beta_{1i})$ be a block in T_1. M_2 propagates bits that M_1 does not (and vice versa), therefore the elements in α_{1i} must all be in separate blocks of T_2 since they are all mapped to different outputs (β 's). Let the input data bus of M_1 and M_2 be N bits wide and the output data buses of M_1 and M_2 be N_1 and N_2 bits wide, respectively; then $N = N_1 + N_2$. The 2^N possible inputs are distributed evenly among the 2^{N_1} blocks in T_1 and 2^{N_2} blocks in T_2, so there are 2^{N_2} elements in each block of T_1 and 2^{N_1} elements in each block of T_2. Thus each block in T_1 contains exactly one element from each block in T_2 and vice versa.

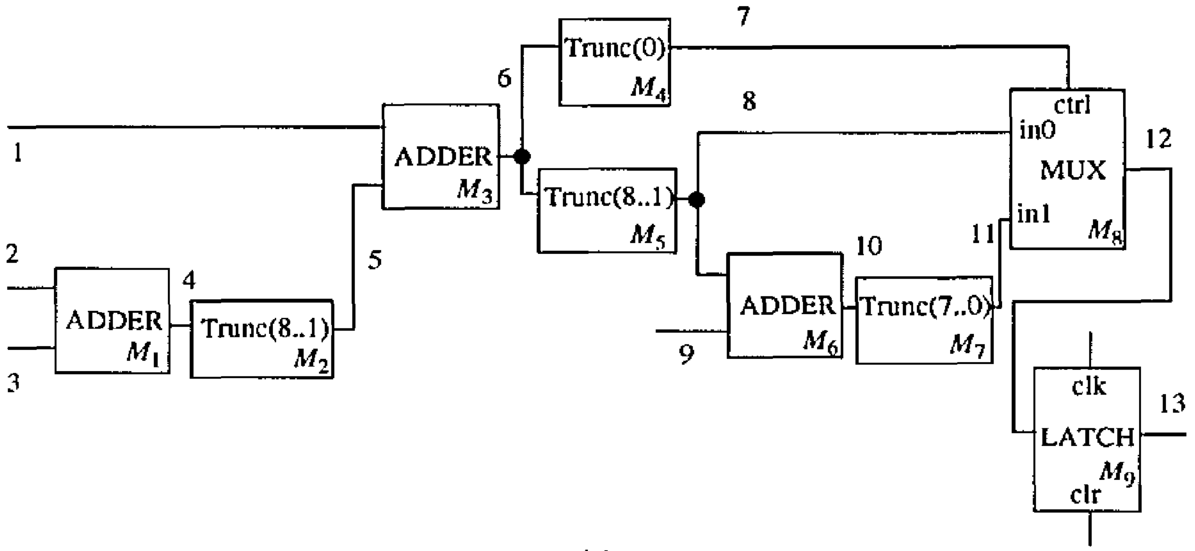
Now, suppose that there is a transmission function $T_b > T_2$ such that $T_1 \# T_b \equiv 0$. This implies that there is a block $(\alpha_{bi};\beta_{bi})$ in T_b, and at least two blocks $(\alpha_{2j};\beta_{2j})$ and $(\alpha_{2k};\beta_{2k})$ in T_2, such that α_{bi} contains all the elements from α_{2j} and α_{2k}. Now for some block $(\alpha_{1q};\beta_{1q})$ in T_1, α_{2j} and α_{2k} each contain exactly one element of α_{1q}. Therefore, α_{bi} contains two elements of α_{1q}. These two elements will also be in the same block of $T_1 \# T_b$, so $T_1 \# T_b > 0$, a contradiction. Therefore, there is no $T_b > T_2$ such that $T_1 \# T_b \equiv 0$. By the same reasoning, there can be no $T_a > T_1$ such that $T_a \# T_2 \equiv 0$. □

Sets of two or more truncate modules connected to the same input (fanout point), whose transmission functions are maximal complements, are common in circuits. Buses are frequently divided into two or more parts, each used in a separate calculation.

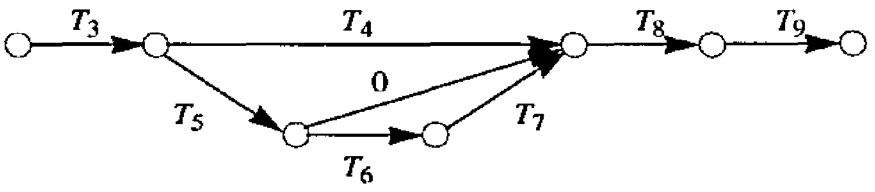
3.2.2 Examples of Transparency Analysis

We now present some additional circuit examples to illustrate our analysis technique.

Fltrdp. The Fltrdp circuit and corresponding propagation diagram are shown again in Figure 3.15. In this case, edges of the propagation diagram are labeled with transmission functions. As usual, transmission function T_i is associated with module M_i. The corresponding expression for the propagation path from bus 1 to bus 13 is given by



(a)



(b)

Figure 3.15. Fltrdp datapath circuit: (a) schematic and (b) propagation diagram.

$T_3 \circ (T_4 \# (T_5 \circ (0 \# (T_6 \circ T_7)))) \circ T_8 \circ T_9$

$\cong T_3 \circ (T_4 \# (T_5 \circ 0)) \circ T_8 \circ T_9$ by property 8

$\cong T_3 \circ ((T_4 \circ 0) \# (T_5 \circ 0)) \circ T_8 \circ T_9$ by property 1a (identity)

$\cong T_3 \circ 0 \circ T_8 \circ T_9$ by Theorem 3.5

Adders like module M_3 are transparent for any X_C, so $T_3 \cong 0$. Likewise, module M_9 (the latch) is transparent when clocked. However, because of the reconvergent fanout at module M_8, we cannot determine whether the entire circuit is transparent without analyzing the individual transmission functions that make up the propagation expression. Transparency cannot be determined in this case by rewriting the expression.

Iterative logic array. A ripple-carry adder, such as the four-bit adder shown in Figure 3.16, is an example of a one-dimensional ILA. Some portion of the output of each module M_i (the carry-out C_{i+1} in this case) is used to connect it to another submodule M_{i+1} in the array. Each submodule also has a carry-in input port C_i. ILAs can be recursively described; an n-bit version I_n is con-

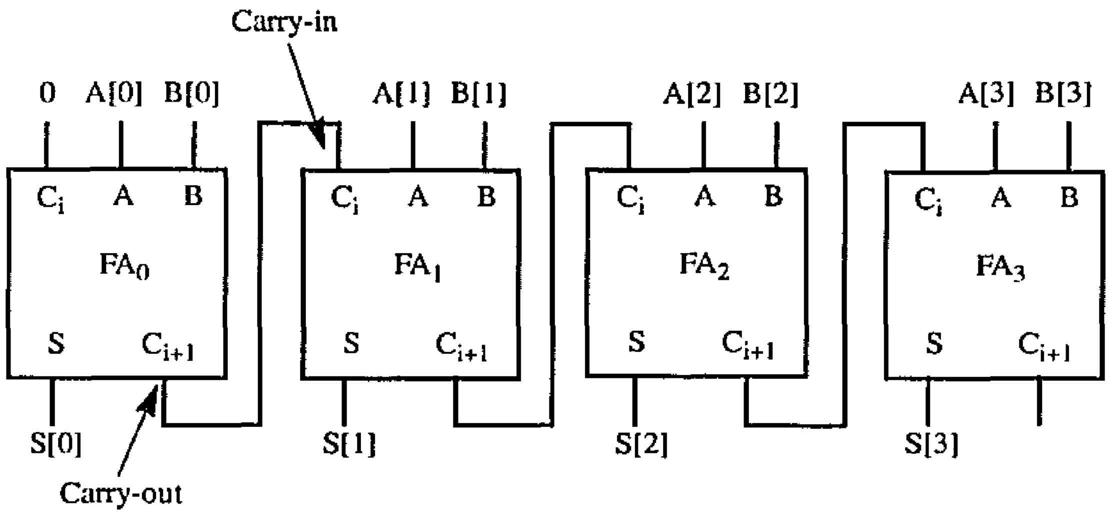


Figure 3.16. Four-bit ripple-carry adder as an example of a one-dimensional iterative logic array.

structed from an $(n-1)$-bit version, I_{n-1} and an additional submodule, M_n. By modeling the ILA as a series-parallel circuit, we can use Theorems 3.5 and 3.6 to prove the following theorem.

Theorem 3.7: *An n-bit one-dimensional ILA M_n $(n > 1)$ is transparent if and only if all of the individual cells which compose it are transparent.*

Proof: We begin with a recursive model of the ILA illustrated by the ripple-carry adder in Figure 3.17a. The n-bit adder is composed of an $(n-1)$-bit adder A_{n-1} connected to a full adder module FA_n. Each module in this diagram has input data and control buses and output data and control buses. The output control bus $Z_{C1} = C_n$ connects the two modules. We also assume that the carry-in line to the $(n-1)$-bit adder is contained in X_{D1}. The adder is redrawn for analysis in Figure 3.17b and illustrates some typical transparency modeling techniques. For instance, all individual module data buses connected to primary inputs are assumed to be concatenated into one primary input data bus X_D. Similarly, all individual module output data buses connected to a primary output are concatenated into one primary output data bus Z_D.

Bus X_D fans out to truncate modules, the outputs of which are the individual module data bus inputs X_{Di}. This modeling technique allows us to represent the distribution of data from an single input bus outside the ILA to the individual inputs of constituent modules inside the ILA. Similarly, the outputs of individual modules here are treated as one bus outside the ILA. In this diagram, we also assume that the outputs of the $(n-1)$-bit adder and the full adder contain both

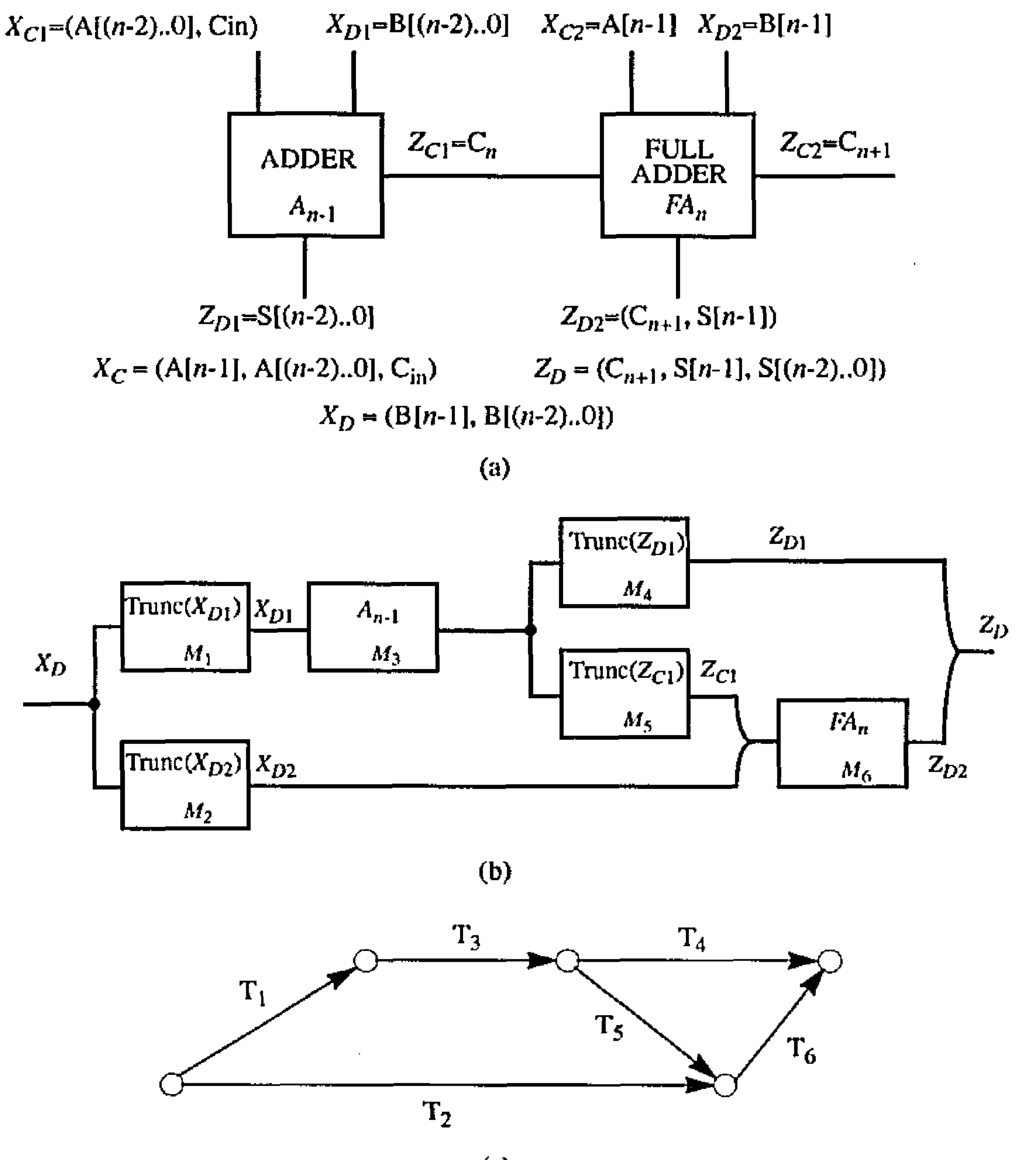


Figure 3.17. (a-b) Recursive description, and (c) propagation diagram of an n-bit, ripple-carry adder.

data and control. Finally, note that the output of the $(n-1)$-bit adder, which initially contains both Z_{D1} and Z_{C1}, fans out to two truncate modules. One of these truncate modules, Trunc(Z_{D1}), selects the bits of Z_{D1} associated with data, and the other, Trunc(Z_{C1}), selects the bits associated with control. Figure 3.17c shows a propagation diagram for Figure 3.17b, in which transmission function T_i is associated with the module M_i. The expression that represents the transparency of the circuit is

$$
\begin{aligned}
&(T_1 \circ T_3 \circ T_4) \# (((T_1 \circ T_3 \circ T_5) \# T_2) \circ T_6) \\
&\cong (T_1 \circ T_3 \circ T_4) \# ((T_1 \circ T_3 \circ T_5\ T_6) \# (T_2 \circ T_6)) && \text{by distributivity} \\
&\cong T_1 \circ (T_3 \circ (T_4 \# (T_5 \circ T_6))) \# T_2 \circ T_6 && \text{by distributivity}
\end{aligned}
$$

The pair of truncate modules M_1 and M_2 meets the requirement of Theorem 3.6, as does the pair of modules M_4 and M_5. Therefore, T_1 and T_2 are maximal complements and by Theorem 3.5, $T_3 \circ (T_4 \# (T_5 \circ T_6))$ must be congruent to zero. By the same reasoning, T_6 must also be congruent to zero in order for $T_1 \circ (T_3 \circ (T_4 \# (T_5 \circ T_6))) \# T_2 \circ T_6$ to be congruent to zero. If $T_3 \circ (T_4 \# (T_5 \circ T_6)) \cong 0$ then $T_3 \cong 0$ by Theorem 3.4a. □

Theorem 3.7 supports the intuitive notion that it is possible to determine the transparency of an iterative module by examining a small version of that module, at least in the case of full transparency.

Divfilt. Next consider the Divfilt datapath circuit discussed in Chapter II and displayed again in Figure 3.18. We will analyze the transparency of the boxed group of modules in the upper left hand corner of the figure. The propagation diagram for this group of modules is shown in Figure 3.19. The propagation expression representing the transparency from primary input port RECV2 to the output of PROCR_1 is

$$T_1 \circ T_2 \circ T_3 \circ (T_7 \# (T_4 \circ T_5 \circ T_6)) \circ T_8 \qquad (3.9)$$

We want to find a valid assignment of transmission functions to the T_i's in this expression that make it congruent to zero. We assume that control buses for each module can be independently assigned. As discussed above, each T_i is a variable—it assumes a different form for each control sequence applied to M_i. We are only interested in propagation modes where $T_i < 1$ if such modes exist. In some cases, there is only one transmission for a module (e.g., $X_C = \varnothing$). In other cases, M_i will only have one control assignment so that $T_i < 1$. In the last two cases $T_i \cong 0$ since there is only one transmission function of interest.

Multiplexer MUXR_1 has only one sensitized mode of operation, and this mode is transparent. Therefore, we set $T_1 \cong 0$. Registers such as REGR_5 are likewise transparent for only one control sequence. The transmission functions for CONCAT modules and FREADs (truncate modules) are constants, in the case of FREADs, $T_i < 1$. The OR gate PROCR_1 has no natural control

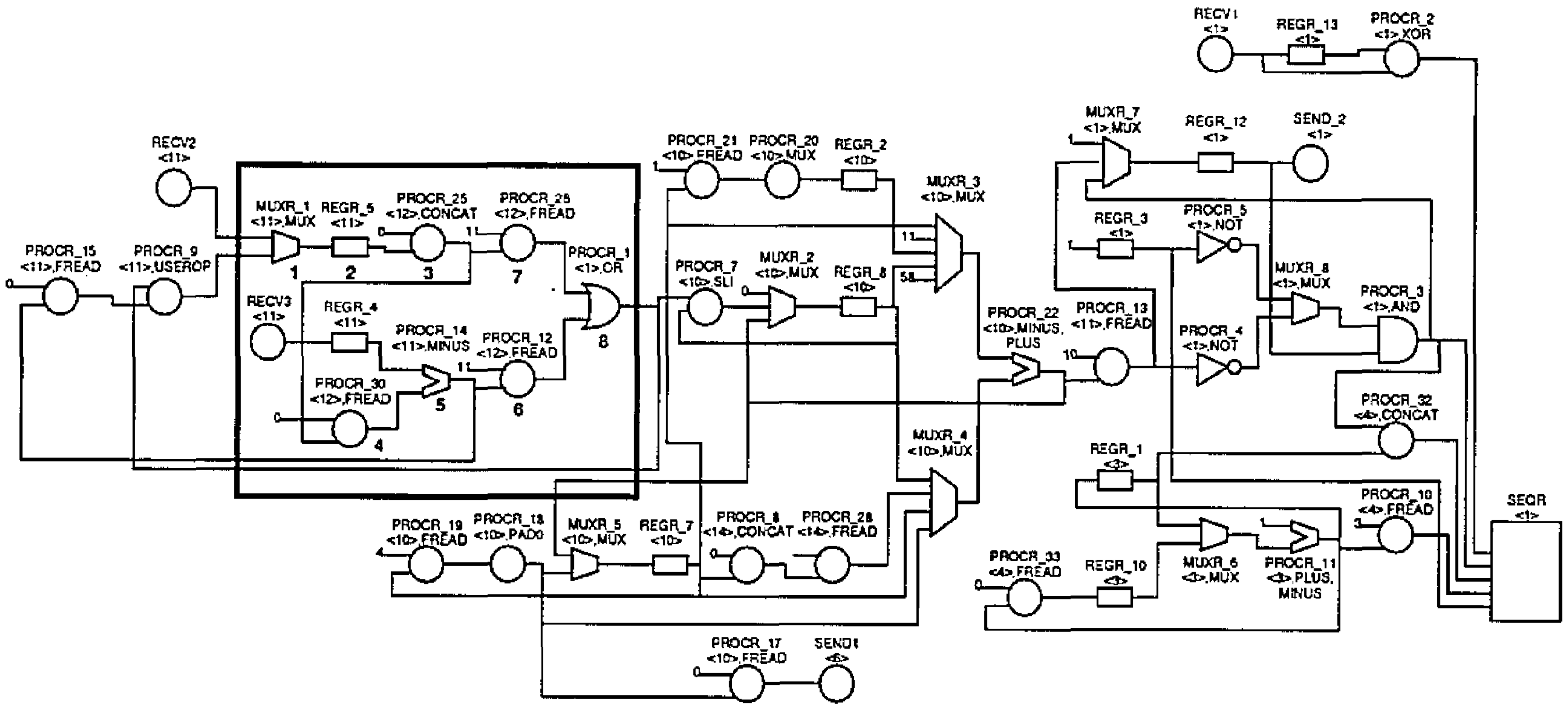


Figure 3.18. Digital filter datapath circuit.

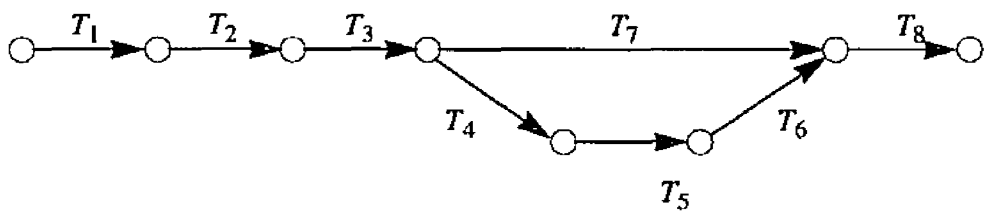


Figure 3.19. Propagation diagram for the boxed subcircuit in Figure 3.18.

inputs since both its inputs are part of connections with vertices in the propagation diagram. The only module with more than one relevant transmission function is the subtracter module PROCR_14. However, a subtracter module, like an adder, is transparent for every possible control value.

Modules PROCR_, REGR_5, and PROCR_25 are all transparent, so $T_1 \circ T_2 \circ T_3 \equiv 0$ (Property 1a). PROCR_26 and PROCR_30 are complementary truncate modules, therefore, $T_7 \# (T_4 \circ T_5 \circ T_6) \equiv 0$ if and only if $T_5 \circ T_6 \equiv 0$ (Theorem 3.5). Since PROCR_14 is a subtracter, $T_5 (V_C) \equiv 0$ for any V_C. However, PROCR_12 truncates all but the most significant bit of the output of PROCR_14, thus $T_5 \circ T_6 \not\equiv 0$, and $T_7 \# (T_4 \circ T_5 \circ T_6) \not\equiv 0$. In Sections 3.4 and 4.3 we will discuss techniques for making this circuit transparent.

3.3. Partial Transparency

To simplify test generation with precomputed modular tests, it is desirable to use fully transparent modules and circuits to propagate a test response T_R. While this is frequently not possible, it may be possible to propagate T_R along partially transparent circuit paths, an issue we explore in this section. If we cannot propagate T_R through fully transparent modules, it is useful to know which of the available partially transparent propagation paths are the most transparent.

Consider the circuit in Figure 3.20, which contains reconvergence similar to that of Fltrdp. We have identified three paths from the output of the module M_1 for analysis. PATH1 consists of the modules M_2, M_3, M_4, M_6, and M_8, and their associated interconnections. PATH2 consists of the path including modules M_5, M_7, and M_9. PATH3 simply passes through module M_8. These paths are marked on the schematic in Figure 3.20. PATH3 is clearly transparent, but PATH1 and

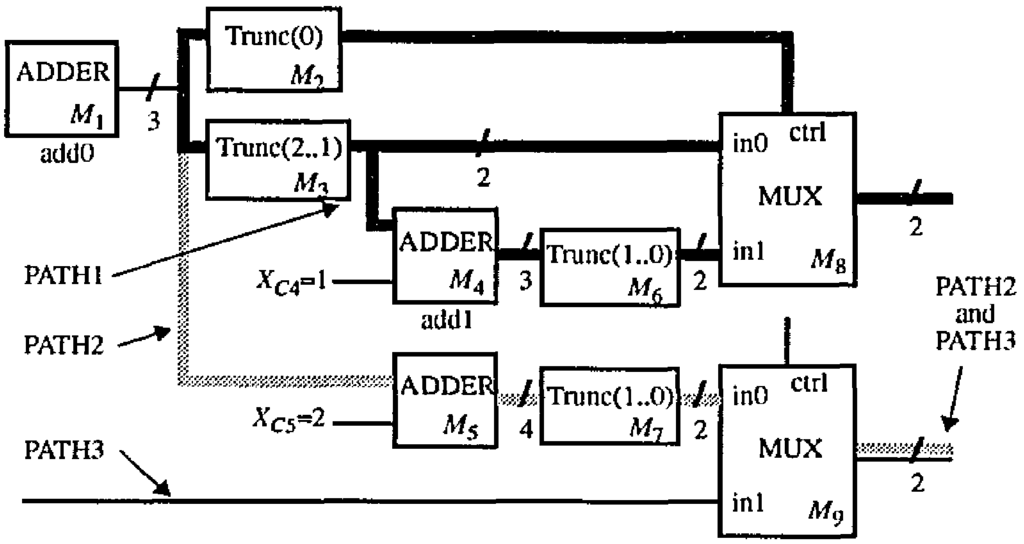


Figure 3.20. Comparison of three propagation paths.

PATH2 are not. Both PATH1 and PATH2 contain about the same number of gates from the output of module 1 to a primary output. It is not easy to determine the more transparent of these two paths by inspection. The transmission function for PATH1 is

$$T_{PATH1} = \{(0,7;0),\ (1,2;1),\ (3,4;2),\ (5,6;3)\} \tag{3.10}$$

and the transmission function for PATH2 is

$$T_{PATH2} = \{(2,6;0),\ (3,7;1),\ (0,4;2),\ (1,5;3)\}\ . \tag{3.11}$$

We now discuss a method for analyzing the relative transparency of these two paths.

In Section 3.1 we noted that if $T_{PATH1} < T_{PATH2}$, then T_{PATH1} propagates more information than T_{PATH2}. However, in this case, T_{PATH1} and T_{PATH2} are algebraically incomparable. We cannot determine the relative transparency of these two transmission functions by comparing them directly using our algebra. Instead we compute a numerical measure of their transparency which can be compared. Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. Let v_D be a value assigned to X_D in a correctly working circuit and v_D' be a value at X_D in a faulty circuit. The pair (v_D, v_D') is a discrepancy. If $N = \left|\{V(X_D)\}\right|$, then the total number of possible discrepancies is $N(N-1)$.

Definition 3.11: The fraction of discrepancies distinguished by a transmission function T is called the *transparency index* of T, and is denoted $Trans\,(T)$.

If all discrepancies are equally likely, then $Trans(T)$ is a measure of the probability that a discrepancy will be propagated through M from X_D to Z_D.

To compute $Trans(T)$, we first compute the inverse of $Trans(T)$, that is, $1 - Trans(T)$, using the structure of the transmission function. Let π be the partition embedded in transmission function T, and let B_i be the size of block i in π. Finally, let $Opacity(B_i, N)$ be the *opacity* or non-transparency index of block i, that is, the contribution of block i to the total fraction of discrepancies that cannot be distinguished by T. Since no two elements in a block can be distinguished, the contribution of block i to the number of discrepancies that cannot be distinguished is $B_i(B_i - 1)$, the number of pairs in block i. Therefore,

$$Opacity(B_i, N) = \frac{B_i(B_i - 1)}{N(N-1)} \tag{3.12}$$

If there are n blocks in π, then

$$Trans(T) = 1 - \sum_{i=1}^{n} Opacity(B_i, N) \tag{3.13}$$

As an example, let $T_1 = \{(0;1), (1;2), (2;3), (3;4), (4;5), (5;6), (6;7), (7;8)\}$. Since $B_i = 1$ for all i, $Trans(T_1) = 1$. Now consider $T_2 = \{(1, 3;0), (0, 2;1)\}$. The size of both blocks T_2 is 2, therefore, $Trans(T_2) = 1 - (1/6 + 1/6) = 2/3$. The transparency indices of the transmission functions for the modules in Figure 3.11 are listed in Table 3.2. Note that for typical datapath modules, the transparency index is either 0 or 1. Most non-transparency results from the way that these modules are connected—for example, connections to truncate modules.

We refer to the set of block sizes $(B_1, B_2, \ldots, B_n)$ for an embedded partition π, as the *partition structure* of π. A canonical list of all possible partition structures for the case where the number of possible inputs to X_D is 8, is shown in column 2 of Table 3.4. Column 1 lists the number of blocks in each partition structure, columns 3 and 4 list the means and variances of the block sizes in each partition structure, and column 5 lists the transparency index implied by the partition structure. In each partition structure in Table 3.4 the block sizes are listed in order of decreasing size. Note that the transparency index generally increases with the number of blocks in the partition structure, but not always. For instance, the transparency index of transmission functions with embedded partitions whose structure is (6,1,1) is 0.46, but the transparency index of transmission functions with structure (5,3) is 0.54. The large block (6) makes (6,1,1) non-transparent. By refer-

Module	Transmission functions	*Trans*(*T*)
(a) NAND gate	$T([0]) = \{(0,1;1)\}$ $T([1]) = \{(0;1), (1;0)\}$	0.00 1.00
(b) NOR gate	$T([0]) = \{(0;1), (1;0)\}$ $T([1]) = \{(0,1;0)\}$	1.00 0.00
(c) Adder	$T([0]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$ $T([1]) = \{(0;1), (1;2), (2;3), (3;4), (4;5), (5;6), (6;7), (7;8)\}$ $T([2]) = \ldots$	1.00 1.00 ...
(d) Multiplexer 1	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$	0.00 1.00

(e) Multiplexer 2	$T([0]) = \{(0,1;0)\}$
	$T([1]) = \{(0;0), (1;1)\}$
	$T([2]) = \ldots$
		0.00
		1.00
		1.00 ...

(f) Multiplier	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;0), (1;1), (2;2), (3;3), (4;4), (5;5), (6;6), (7;7)\}$ $T([2]) = \ldots$	0.00 1.00 ...
(g) Register	$T([0,1]) = \{(0;(0,0)), (1;(0,1)), (2;(0,2)), (3;(0,3)), (4;(0,4)),$ $(5;(0,5)), (6;(0,6)), (7;(0,7))\}$	1.00
(h) Decoder	$T([0]) = \{(0,1,2,3,4,5,6,7;0)\}$ $T([1]) = \{(0;1), (1,2,3,5,6,7;0), (4;2)\}$	0.00 0.46

Table 3.3 Transmission functions and transparency index for modules in Figure 3.11

ring to Table 3.4, we can now finally analyze the relative transparency of PATH1 and PATH2 in Figure 3.20. The transparency index of each of these paths is the same because they both have the same partition structure (2,2,2,2). Hence, $Trans(T_1) = Trans(T_2) = 0.86$. T_1 and T_2 both propagate the same amount of information, that is, the same number of discrepancies. However, since T_1 and T_2 are incomparable, each propagates a different set of discrepancies. In fact,

$$T_1\#T_2 = \{(0;(0,2)), (1;(1,3)), (2;(1,0)), (3;(2,1)),$$
$$(4;(2,2)), (5;(3,3)), (6;(3,0)), (7;(0,1))\} \equiv 0$$

Together, T_1 and T_2 propagate all information.

Incomparability is an important aspect of propagation along parallel paths. In general, if M_1 and M_2 are two modules connected in parallel, with corresponding transmission functions T_1 and T_2, then $T_1\#T_2 < T_1$ and $T_1\#T_2 < T_2$ if and only if T_1 and T_2 are incomparable, that is, M_1

Number of blocks	Partition structure	Mean	Variance	*Trans*(*T*)
8(*)	(1,1,1,1,1,1,1,1)	1.00	0.00	1.00
7(*)	(2,1,1,1,1,1,1)	1.14	0.14	0.96
6(*)	(2,2,1,1,1,1)	1.33	0.27	0.93
5(*)	(2,2,2,1,1)	1.60	0.30	0.89
4(*)	(2,2,2,2)	2.00	0.00	0.86
6	(3,1,1,1,1,1)	1.33	0.67	0.89
5	(3,2,1,1,1)	1.60	0.79	0.86
4	(3,2,2,1)	2.00	0.64	0.82
4	(3,3,1,1)	2.00	1.32	0.79
3(*)	(3,3,2)	2.67	0.34	0.75
5	(4,1,1,1,1)	1.60	1.80	0.79
4	(4,2,1,1)	2.00	1.99	0.75
3	(4,2,2)	2.67	1.32	0.71
3	(4,3,1)	2.67	2.34	0.68
2(*)	(4,4)	4.00	0.00	0.57
4	(5,1,1,1)	2.00	4.00	0.64
3	(5,2,1)	2.67	4.33	0.61
2	(5,3)	4.00	1.99	0.54
3	(6,1,1)	2.67	8.35	0.46
2	(6,2)	4.00	8.01	0.43
2	(7,1)	4.00	17.98	0.25
1(*)	(8)	8.00	—	0.00

Table 3.4 All possible structures for partitions on 8 elements, the mean and variance of the block sizes, and the transparency index of corresponding transmission functions.

and M_2 must propagate different information. Furthermore, since the transmission function of the circuit formed from series-connected modules is always greater than or congruent to the transmission function of the first module in the series (Property 9 in Table 3.1), we can determine a strategy

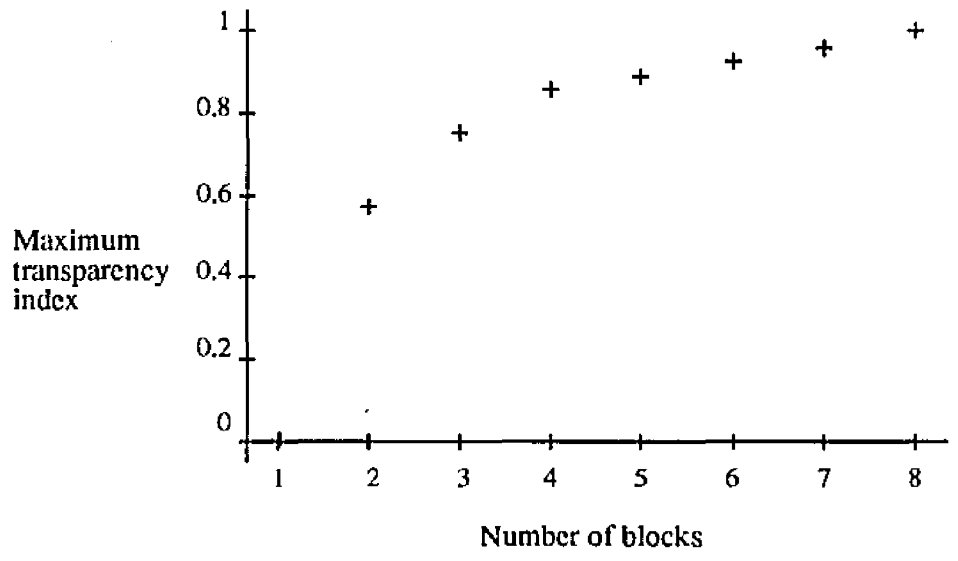


Figure 3.21. Maximum transparency index as a function of number of blocks domain size 8.

for propagation by comparing the transmission functions of the first modules along each branch of a fanout junction. Let M_1 and M_2 be two modules whose input data buses are connected to branches of a fanout junction, and let T_1 and T_2 be transmission functions for M_1 and M_2, respectively. If $T_1 < T_2$, then the path starting with M_1 should be chosen for propagation, since T_1 propagates everything that T_2 does, and more. If T_1 and T_2 are incomparable, then propagation should proceed in parallel.

Now we consider the partition structure that maximizes the transparency index. As observed above, the transparency index of a transmission function seems to increase with the number of blocks, but not monotonically. The distribution of elements among the blocks is also important. In Table 3.4, we list the partition structures in order of decreasing transparency index. For a given number of blocks, we also mark with an asterisk the partition structures leading to maximum transparency index. We plot transparency index as a function of the number of blocks for these marked partition structures in Figure 3.21. For these partition structures, transparency index does increase monotonically. Also note that in each partition structure marked with a * in Table 3.4, the block sizes are all nearly or exactly the same size. That is, the instances of maximum transparency index in Table 3.4 are those cases where the set of block sizes have minimum variance (σ^2) for a given number of blocks. Variance is used here in the classical sense to measure dispersion from the mean, that is,

$$\sigma^2 = \sum_{i=1}^{n} \frac{\left(B_i - B\right)}{n-1}$$

where B is the mean block size. Since the total number of elements to distribute among blocks is fixed, variance is reduced the closer all block sizes are to N/n. N is always a power of 2, namely $2^{|X_D|}$, but n can be any natural number and depends on the module function. As illustrated in Table 3.4, for the set of partition structures with maximum transparency for a given n, not all blocks are exactly the same size in each partition structure. However, in each case no other partition structure with the same number of blocks has lower variance. This fact is formalized in the following theorem.

Theorem 3.8: *For a given number of elements and blocks, the embedded partition of a transmission function with maximum transparency index has the minimum variance among block sizes.*

Proof: We will show that monotonically increasing the variance of partition structures monotonically decreases transparency index. We are considering transmission functions with n blocks, whose domains contain $N = \left|\{V(X_D)\}\right|$ elements. Assume without loss of generality that the block sizes in the embedded partitions for these transmission functions are ordered $B_1 \geq B_2 \geq \ldots \geq B_n$, as they are in the canonical list of partition structures shown in Table 3.4. All possible partition structures may be obtained from the minimum-variance partition structure S_{min} by transformations of the form

$$S_1 = (B_1, \ldots, B_i, \ldots, B_j, \ldots, B_n) \rightarrow (B_1, \ldots, B_i + 1, \ldots, B_j - 1, \ldots, B_n) = S_2$$

which increase variance. Consider one such transformation $S_1 \rightarrow S_2$, and let T_1 and T_2 be transmission functions whose embedded partitions have structure S_1 and S_2, respectively. By (3.12) and (3.13)

$$Trans(T_1) = 1 - \frac{B_i(B_i - 1) + B_j(B_j - 1) + C}{N(N-1)}$$

where C is the non-transparency contribution of all the blocks except i and j. Similarly,

$$Trans(T_2) = 1 - \frac{(B_i + 1)B_i + (B_j - 1)(B_j - 2) + C}{N(N-1)}$$

We want to show that $Trans(T_2) < Trans(T_1)$, that is,

$$1 - \frac{(B_i+1)B_i + (B_j-1)(B_j-2) + C}{N(N-1)} < 1 - \frac{B_i(B_i-1) + B_j(B_j-1) + C}{N(N-1)}$$

which implies

$$(B_i+1)B_i + (B_j-1)(B_j-2) > B_i(B_i-1) + B_j(B_j-1)$$

By expanding and collecting terms, we obtain

$$2B_i - 2B_j > -3 \tag{3.14}$$

The relation (3.14) is true if and only if $B_i > B_j - 3/2$, which holds since B_i was defined to be greater than or equal to B_j. □

We refer to transmission functions whose block sizes have minimum variance as *maximum transparency transmission functions.* T_{PATH1} (equation (3.10)) corresponding to PATH1 in Figure 3.20 and T_{PATH2} (equation (3.11)) corresponding to PATH2 in Figure 3.20 are maximum transparency transmission functions. It is interesting to note that for 18 of the 22 partition structures in Table 3.4, the transparency index is greater than 0.5, that is, more than half of the discrepancies are propagated. We conclude therefore that most sensitized modules are quite transparent. As another illustration of this fact, consider Figure 3.22 where we have plotted the maximum transparency index as a function of the input data bus size for transmission functions with 4 and 8 blocks. This figure shows that as the data bus size increases, the maximum transparency index possible for any transmission function with n blocks approaches an asymptotic lower bound that depends on the number of blocks. In Figure 3.22, for transmission functions with 8 blocks, this lower bound is about 0.9, and for transmission functions with 4 blocks, the lower bound is about 0.77. The following theorem formalizes this fact.

Theorem 3.9: *Let T be a maximum transparency transmission function with n blocks, the minimum possible transparency index for T is* $Trans(T) = 1 - \frac{1}{n}$.

Proof: Let π be T's embedded partition and let B_i be the size of block i in π. By definition,

$$Trans(T) = 1 - \sum_{i=1}^{n} Opacity(B_i, N) \tag{3.15}$$

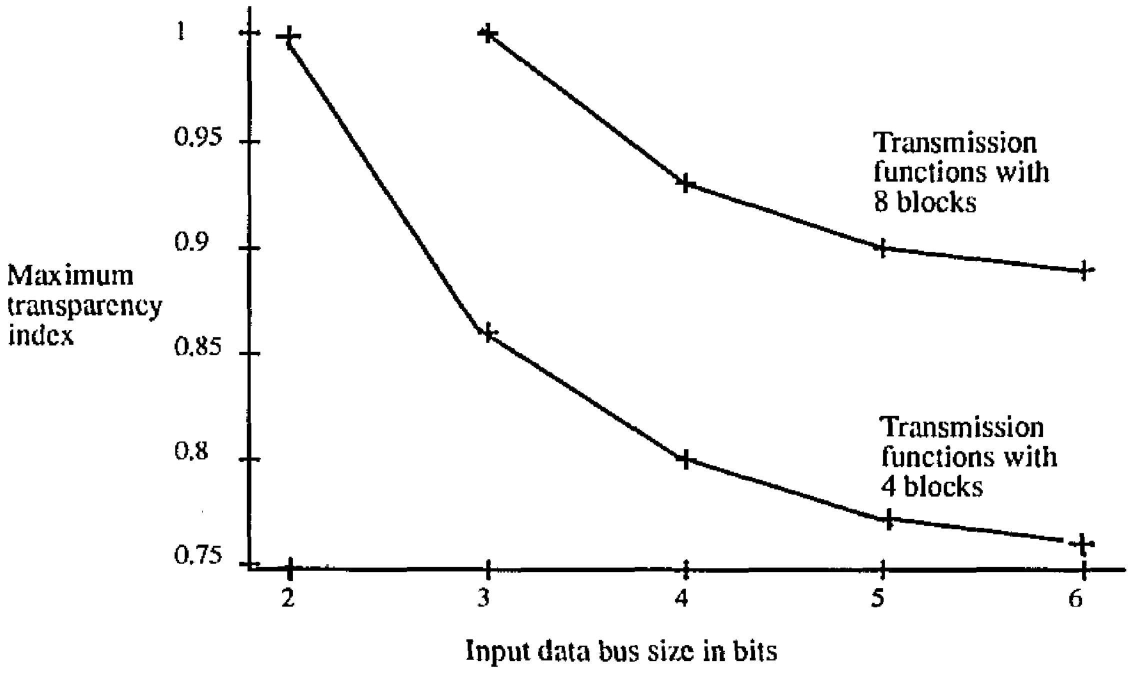


Figure 3.22. Maximum transparency index possible as a function of the input data bus size for transmission functions with 4 blocks and 8 blocks.

For maximum transparency transmission functions with n blocks, the transparency index depends only on the number of elements in the domain N, and is minimized when N becomes very large. Taking the limit of both sides of (3.15)

$$\lim_{N \to \infty} Trans(T) = \lim_{N \to \infty} \left(1 - \sum_{i=1}^{n} Opacity(B_i, N) \right)$$

$$= 1 - \sum_{i=1}^{n} \lim_{N \to \infty} Opacity(B_i, N)$$

According to Theorem 3.8, all blocks in T must be as close as possible to exactly the same size, therefore B_i is $\lceil N/n \rceil$ or $\lfloor N/n \rfloor$, since N may not be divisible by n. Suppose that N is divisible by n, so $B_i = N/n$. By equation (3.12),

$$Opacity(B_i, N) = \frac{(N/n)(N/n - 1)}{N(N-1)}$$

Expanding yields

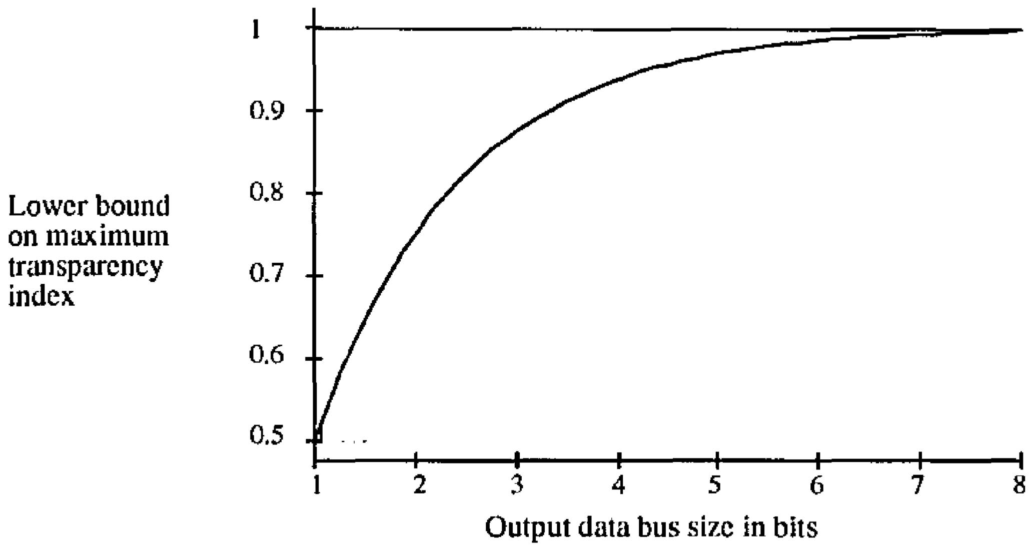


Figure 3.23. Lower bound on maximum transparency index as a function of output data bus size.

$$\frac{(1/n)^2(N^2 - nN)}{N^2 - N}$$

This form of $Opacity(B_i, N)$ for maximum transparency transmission functions is the same regardless of whether N is divisible by n. Taking the limit,

$$\lim_{N \to \infty} \frac{(1/n)^2\left(N^2 - nN\right)}{\left(N^2 - N\right)} = 1/n^2.$$

Therefore, $\lim_{N \to \infty} Opacity(B_i, N) = 1/n^2$. Finally,

$$\lim_{N \to \infty} Trans(T) = 1 - \sum_{i=1}^{n}\left(1/n^2\right) = 1 - 1/n \ \square$$

The number of blocks in a transmission function is bounded by $\left|\{V(Z_D)\}\right|$. Therefore, for a given number of output bits $|Z_D|$, a module has its maximum transparency index when its transmission function has $2^{|Z_D|}$ equal-sized blocks. Truncate modules meet this criterion. Truncate modules M_2, M_5, and M_7 in Fltrdp (Figure 3.15) each truncates a 9-bit bus to an 8-bit bus. In each case, the transparency index of the corresponding transmission function is 0.998; no module with $|X_D| = 9$ and $|Z_D| = 8$ can have a larger transparency index. Interestingly, according to Theorem 3.9, the smallest transparency index that a truncate module with an 8-bit Z_D could have is 0.996, regardless of X_D's size. Figure 3.23 shows the lower bound on maximum transparency index as a function of $|Z_D|$ for Z_D sizes up to 8 bits. Note that truncate modules with only a 4-bit

Z_D can propagate at least 90% of the discrepancies that appear at their inputs.

In summary, even when there is a very large difference between the size of the input and output buses, the fraction of discrepancies that can be propagated can still be very high. This fact can be understood in the following way. Consider a discrepancy (v_D, v_D') to be propagated through a module with a maximum transparency transmission function T. There are at most $\lceil N/n \rceil - 1$ values for v_D', such that (v_D, v_D') cannot be propagated, since there are at most $\lceil N/n \rceil$ elements in each block of T. However, there are at least $(n-1)(\lfloor N/n \rfloor)$ other values for v_D' such that (v_D, v_D') can be propagated, since there are at least $(n-1)(\lfloor N/n \rfloor)$ other values in blocks that do not contain v_D. No matter how many values conflict with v_D, there are more that do not, and the ratio of conflicting to non-conflicting values clearly depends on the number of blocks. This implies that partially transparent propagation modes are likely to be very successful at propagating T_R even though they may fail to propagate all possible discrepancies.

3.4. Sequential Transparency

In the general definition of transmission functions, modules are controlled by sequences of inputs at X_C. Control sequences $V_C = [v_{c1}, v_{c2}, \ldots, v_{ck}]$ longer than one timestep were studied by Marhöfer [66], but were not integrated into a formal theory of propagation, as we have done in Sections 3.1 and 3.2. Multi-step propagation is determined by intersecting incomparable transmission functions. The intersection of two transmission functions T_1 and T_2 is always at least as transparent as either T_1 or T_2. For parallel connections, this fact is summarized in Table 3.1 as Property 7. Intersecting T_1 and T_2 is equivalent to applying the two functions in sequence (Theorem 3.1). For a single module M, T_1 and T_2 are defined by their control sequences S_{C1} and S_{C2} respectively, that is, $T_1 = T(S_{C1})$ and $T_2 = T(S_{C2})$, where T is the transmission function for M. This implies that if $T_3 = T_1 \cap T_2$ then the control sequence for T_3 is the juxtaposition of S_{C1} and S_{C2}, that is $[S_{C1}, S_{C2}]$.

Definition 3.12: Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D, and let $T(S_C)$ be a transmission function for M such that $T(S_C) \equiv 0$. If $k = |S_C|$, then M is said to be *k-transparent*. We usually only refer to a module as k-transparent if $k > 1$.

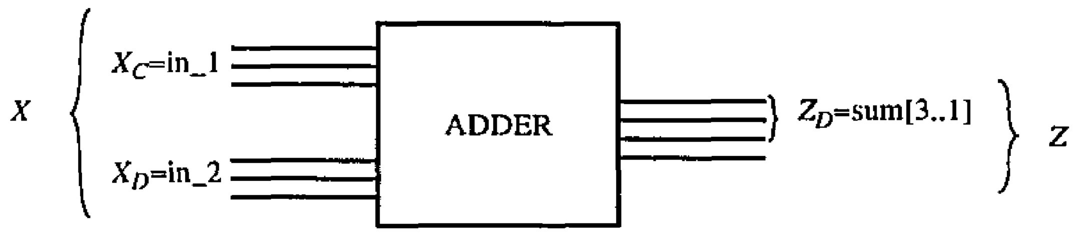


Figure 3.24. Three-bit adder module with input data bus X_D as addend, input control bus X_C as augend, and output data bus Z_D = sum[3..1].

For example, consider the adder in Figure 3.24 one final time. For this module,

$$T([0]) = \{(0,1;0),(2,3;1),(4,5;2),(6,7;3)\}$$

$$T([1]) = \{(0;0),(1,2;1),(3,4;2),(5,6;3),(7;4)\}$$

$$T([0,1]) = T([0]) \cap T([1]) = \{(0;[0,0]),(1;[0,1]),(2;[1,1]),$$
$$(3;[1,2]),(4;[2,2]),(5;[2,3]),(6;[3,3]),(7;[3,4])\} \equiv 0$$

Therefore, the adder is 2-transparent. Since it is not transparent for any control sequence of length 1, it is not 1-transparent.

If a module is k-transparent, it is also (k+1)-transparent if $k \le 2^{|X_C|} - 1$. We are usually interested in the minimum value of k. We can derive a lower bound on k which is tight in the sense that useful circuits can be constructed which meet it.

Theorem 3.10: *Let M be a module with input data bus X_D, input control bus X_C, and output data bus Z_D. Let $T(S_C)$ be a transmission function such that $S_C = [v_1, v_2, \ldots, v_k]$ is the shortest sequence for which $T(S_C) \equiv 0$. Then $max[1, \lceil |X_D| / |Z_D| \rceil]$ is a tight lower bound on k.*

Proof: Let π be the embedded partition in $T(S_C)$, and let $\pi_i = T(v_i)$, $1 \le i \le k$. Then, $\pi = \pi_1 \cap \pi_2 \cap \ldots \cap \pi_k$. Assume that $|X_D| > |Z_D|$. In order for k to be minimum, every intersection must yield the maximum number of new blocks. In this case, $|\pi_i \cap \pi_j| = |\pi_i| \times |\pi_j|$. Furthermore, each π_i must have the maximum number of blocks, $2^{|Z_D|}$. Therefore, for minimum k, $T(S_C) \equiv 0$ implies that

$$\prod_{i=1}^{k} 2^{|Z_D|} = 2^{|X_D|}$$

Thus, $2^{k|Z_D|} = 2^{|X_D|}$, which means that $k \geq \lceil |X_D|/|Z_D| \rceil$, since if $|X_D|$ is not divisible by $|Z_D|$, an extra control step will be required. If $|X_D| \leq |Z_D|$, then the minimum value for k is 1, therefore, the lower bound on k is $max\,[1, \lceil |X_D|/|Z_D| \rceil]$. □

This theorem succinctly captures the requirement mentioned above that $|X_D| \leq |Z_D|$ in order for a module to be 1-transparent. A module M can also be 0-transparent if $|X_D| \leq |Z_D|$, and $X_C = \varnothing$. Intuitively, this means that if M is not 0-transparent, then buses connected to it must meet a fundamental requirement stated in the following corollary to Theorem 3.10.

Corollary 3.3: *Let M be a module with input data bus X_D, input control data bus X_C, and output data bus Z_D. If M is not 0-transparent, then* $|X_C| \geq \log_2 (\lceil |X_D|/|Z_D| \rceil)$.

In other words, M's control bus must be big enough so that the minimum number of control values can be applied.

Recall from the discussion in Subsection 3.2.2 that the truncate module 6 (PROCR_12) in Divfilt (Figure 3.18) blocks propagation through the boxed subcircuit. PROCR_12 truncates the 12-bit output of the subtracter PROCR_14 to one bit. According to Theorem 3.10, the shortest control sequence that can propagate all discrepancies through a module with $|X_D| = 12$ and $|Z_D| = 1$ has length 12. Of course, PROCR_12 has no natural control input that can be used for propagation. Additional inputs shown in Figure 3.18 reflect parameters used by the synthesis tool. PROCR_12 can be modified to improve propagation, we discussed in the next chapter. Alternatively, we can treat modules 5 and 6 as a single module: a subtracter with only the most significant bit (borrow) used as output. Such a module is 2^{11}-transparent. All possible values must be applied sequentially to X_C to propagate all discrepancies from X_D to Z_D. Thus, circuits can have transparent modes of operation that are totally impractical.

Next, we consider the combination of modules in series when the minimum-length control sequence is longer than one. The control sequences of the individual modules on the propagation path must be modified to form a control sequence for the entire path. This fact is summarized in the following theorem.

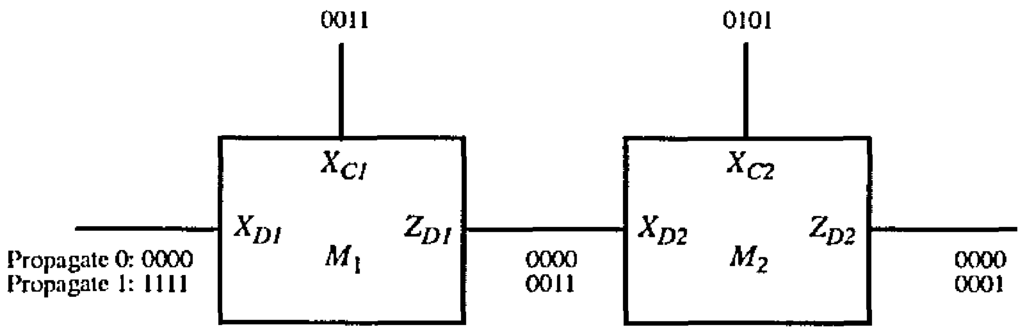


Figure 3.25. Two 2-transparent modules connected in series.

Theorem 3.11: *Let M_1 and M_2 be modules whose minimum-length control sequences have length k_1 and k_2, respectively. Then the minimum length of a control sequence for the series combination of M_1 and M_2 is $k_1 k_2$.*

Proof: To propagate a single vector v_{D2} through M_2, it must be held constant for k_2 timesteps. Therefore, each control value v_{C1} assigned to X_{C1} must be repeated for at least k_2 steps. Since there are at least k_1 of these, each value v_{D1} at X_{D1} will take at least $k_1 k_2$ steps to propagate through the series combination. □

The circuit in Figure 3.25 shows two k-transparent modules are connected in series. Both M_1 and M_2 could be 3-bit adders with the three most significant bits of the sum forming Z_D, as in Figure 3.24. The sensitizing propagation mode for each module is [0,1]. Note that in Figure 3.25, each timestep of the sequence V_{C1} applied to X_{C1} is repeated $k_2 = 2$ times, and that the entire sequence $V_{C2} = [0, 1]$ applied to X_{C2} is repeated $k_1 = 2$ times. The 2-transparent transmission function for M_1 and M_2 is:

$$T([0,1]) = \{(0;[0,0]), (1;[0,1]), (2;[1,1]), (3;[1,2]), (4;[2,2]), (5;[2,3]), (6;[3,3]), (7;[3,4])\}$$

This transmission function must be converted to two 4-transparent forms:

$$T_1 = \{(0;[0,0,0,0]), (1;[0,0,1,1]), (2;[1,1,1,1]), (3;[1,1,2,2]), (4;[2,2,2,2]), (5;[2,2,3,3]), (6;[3,3,3,3]), (7;[3,3,4,4])\}$$

and

$$T_2 = \{(0;[0,0,0,0]), (1;[0,1,0,1]), (2;[1,1,1,1]), (3;[1,2,1,2]), (4;[2,2,2,2]), (5;[2,3,2,3]), (6;[3,3,3,3]), (7;[3,4,3,4])\}$$

for modules M_1 and M_2, respectively. The resultant transmission function is:

$$T_1 \circ T_2 = \{ (0;[0,0,0,0]), (1;[0,0,0,1]), (2;[0,1,0,1]), (3;[0,1,1,1]), (4;[1,1,1,1]), (5;[1,1,1,2]), (6;[1,2,1,2]), (7;[1,2,2,2]) \}$$

As in the case of serially connected modules, the combination of k-transparent modules in parallel does have an effect on the propagation modes. In order to satisfy the requirements of the parallel connection operation, the number of timesteps for all modules must be the same. Thus the length of all control sequences must be normalized. The only constraint is that all elements of the new sensitizing sequence appear in the original order. For example, suppose that module M has sensitizing sequence $V_C = [0,1,2]$. Both sequences $V_{C1} = [0,0,1,2]$ and $V_{C2} = [0,1,1,2,2]$ are equivalent to V_C in their ability to sensitize M since both preserve the order of the values.

The length of the shortest normalized control sequence is the least common multiple of the lengths of the individual control modes. As an example, let T_1 and T_2 be ambiguity sets for modules M_1 and M_2, each of which has a single-bit output. Let

$$T_1([0,1]) = \{(0,1;[0,0]), (2,3;[0,1]), (4,5;[1,0]), (6,7;[1,1])\}$$

$$T_2([1,2,3]) = \{(0;[0,0,0]), (1,2;[0,0,1]), (3,4;[0,1,0]), (5,6;[0,1,1]), (7;[1,0,0])\}.$$

Then

$$T_1'([0,0,0,1,1,1]) = \{(0,1;[0,0,0,0,0,0]), (2,3;[0,0,0,1,1,1]), (4,5;[1,1,1,0,0,0]), (6,7;[1,1,1,1,1,1])\}$$

$$T_2'([1,1,2,2,3,3]) = \{(0;[0,0,0,0,0,0]), (1,2;[0,0,0,0,1,1]), (3,4;[0,0,1,1,0,0]), (5,6;[0,0,1,1,1,1]) (7;[1,1,0,0,0,0])\}.$$

The parallel combination of T_1 and T_2 is

$$T_1 \# T_2 = \{(0;[0,0,0,0,0,0]), (1;[0,0,0,0,1,1]), (2;[0,0,0,2,3,3]), (3;[0,0,1,3,2,2]), (4;[2,2,3,1,0,0]), (5;[2,2,3,1,1,1]), (6;[2,2,3,3,3,3]), (7;[3,3,2,2,2,2])\}.$$

3.5. Summary

In this chapter, we have presented a general theory of propagation through modular bus-structured circuits. We showed how circuits can be modeled as series and parallel connections of modules and analyzed using this theory. We applied the theory to analysis of transparency and demonstrated the analysis on some example circuits. Finally, we identified some of the fundamental properties and limitations of transparency in modular circuits. We will discuss some further applications of the theory in the next chapter.

The propagation theory formalizes intuitive concepts of information propagation in bus-structured circuits, such as propagation along parallel, partially transparent paths, so that analysis of propagation paths in circuits can be automated. This is a necessary step in generating tests for circuits with an irregular bus structure, which was our goal. We also obtained some novel results. For example, we showed that even when there is a very large difference between the sizes of the input and output buses of a module M, the fraction of discrepancies that can be propagated through M can still be very high. The theory is general enough that other applications for it may also be possible.

CHAPTER IV
TEST PACKAGE PROPAGATION

In this chapter, we discuss the representation and propagation of test package information. The theory and methods described here directly address the deficiencies of *PathPlan* and other test generators that use precomputed tests. We also show how to modify circuits to increase their transparency and thereby improve their ability to propagate test packages.

4.1. Symbolic Propagation

As discussed in Chapter II, test packages are information objects containing test, propagation and control information for modules and circuits. The basic form of a test package for a MUT M is $(T_S;T_R)$, where T_S is a precomputed test stimulus sequence for M, and T_R is the fault-free response from M when T_S is applied. T_S is frequently decomposed into spatial vector sequence components $A_1, A_2, \ldots, A_n$, that correspond to the natural inputs of the MUT. The central problem in test generation using precomputed tests is to produce vector sequence signals at the primary inputs that

- Propagate to the inputs of the MUT and match $A_1, A_2, \ldots, A_n$
- Establish fully or partially transparent propagation paths from the output of the MUT to primary outputs so that all errors that can be produced by the MUT are observable.

We want to propagate vector sequences symbolically whenever possible, as we did in *PathPlan*. A symbolic vector sequence A can only be propagated by *PathPlan* from the data input X_D to the data output Z_D of module M if the module function implemented by M leaves A unchanged or inverted at Z_D; *PathPlan* avoids the creation and manipulation of more complex expressions of

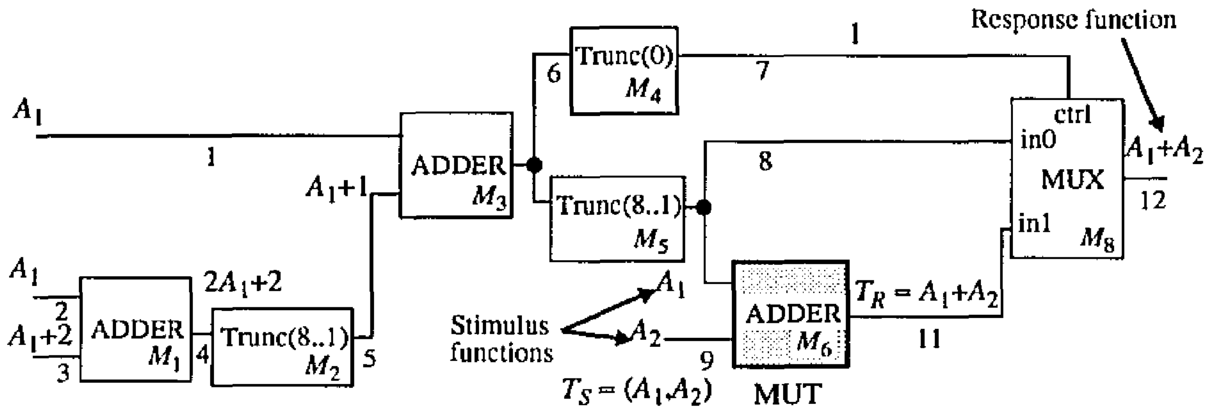


Figure 4.1. Propagating a symbolic test package ($T_S;T_R$) through the Fltrdp datapath circuit

vector sequence symbols. However, as we saw when testing Fltrdp (shown again in Figure 4.1), this approach is often overly restrictive. Recall from our discussion in Chapter II that *PathPlan* cannot simultaneously propagate T_S to module M_6 (the MUT) of Fltrdp, and errors through the multiplexer module M_8 using T-mode propagation. Nevertheless, if the appropriate signal assignments are made, M_6 can be tested. Let $T_S = (A_1, A_2)$ and $T_R = A_1 + A_2$, so that the fault test package for M_6 is $T_1 = (A_1, A_2; A_1 + A_2)$. A_1 is required on bus 8 and A_2 on bus 9. If we assign the symbolic vector sequence expressions A_1 to buses 1 and 2, $A_1 + 2$ to bus 3, and A_2 to bus 9, as shown in Figure 4.1, then the symbolic expression propagated to buses 8 and 9 match T_S, and T_R is propagated to the primary output, bus 12.

4.1.1 Symbolic Expressions

To improve upon *PathPlan*, signals must be propagated as symbolic expressions, as demonstrated in Figure 4.1. These symbolic expressions are functions composed of individual module functions; they are another form of the propagation expressions (PEs) discussed in Chapter III, where function composition due to circuit structure was written to emphasize its series/parallel nature. The vector sequence data represented by symbols in a PE constitute its domain. PEs propagated from primary inputs to MUT inputs are *stimulus functions* and PEs propagated from the MUT outputs to any other point in the circuit are *response functions*. Stimulus and response functions are determined by control signals applied to the X_Cs of modules. Different control signals imply different functions, as shown in Chapter III, and in most cases, if control signals are unassigned or unknown, the function is undetermined.

To successfully instantiate a test package $(T_S;T_R)$, the codomains of the stimulus functions must be components of T_S. Often, we can determine whether vector sequence components $A_1, \ldots, A_n$ of T_S are propagated to the inputs of the MUT by matching $A_1, \ldots, A_n$ symbolically to PEs, as shown in the Fltrdp example above. Otherwise, the PEs can be evaluated using vector sequence data and matched to the components of T_S numerically. This amounts to moving down the hierarchy to the level of the individual vectors and bits of the vector sequences. In the Fltrdp example above, let A_1 and A_2 be given by

$$A_1 = \begin{bmatrix} 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&1&0&1 \end{bmatrix}$$

To match T_S numerically, we propagate one vector in one timestep, that is, one vector per clock cycle. For example, at timestep 1 we attempt to propagate the vector 0 to bus 8 to match the first vector of A_1 and at timestep 2 we attempt to propagate the vector 10101010_2 to bus 8 to match the second vector of A_1. For Fltrdp, we are able to match PEs symbolically to T_S, despite the fact that there is no T-mode path from primary inputs to the inputs of the MUT.

As demonstrated by *PathPlan*, we can sometimes determine symbolically whether all errors are propagated from the output of the MUT to primary outputs. For example, if a PE at a primary output is equivalent to T_R (I-mode propagation), or is only the logical inverse of T_R (T-mode propagation), then all errors are implicitly propagated. However, in many cases we cannot tell by analyzing symbolic expressions whether all errors are propagated from the outputs of the MUT to primary outputs. We will discuss response functions and error propagation further below. First we discuss the problem of representing symbolic expressions so that they can be easily matched to T_S.

4.1.2 Symbolic Expression Simplification

As they propagate through a circuit model, PEs grow as a result of module function composition. In the Fltrdp circuit of Figure 4.1, if PEs are allowed to grow without modification, then the output of module 1 would be $A_1 + (A_1 + 2)$, and the output of module M_2 would be $(A_1 + (A_1 + 2))/2$, since truncating the least significant bit of a bus is equivalent to dividing signal values on the bus by 2. Continuing in this way, the input to module M_6 (the MUT) on bus 8 is

$$\frac{\frac{A_1 + (A_1 + 2)}{2} + A_1}{2} = \frac{1}{2}\left(\frac{1}{2}(A_1 + (A_1 + 2)) + A_1\right) \qquad (4.1)$$

The test package value to be matched at bus 8 is A_1. We want to determine symbolically if (4.1) is equivalent to A_1 using an algorithm.

To solve this problem, we turn to the field of computer algebra [4, 22, 74], where mathematical formulas are manipulated symbolically. Maple [38] and Mathematica [102] are examples of programs that embody these techniques. To manipulate symbolic formulas algorithmically, it is usually necessary to convert them to a standard or canonical form. Arbitrary formulas are then simplified to match the standard form. The simplification algorithms of a particular computer algebra program like Mathematica [102] are among its most fundamental components. Unfortunately, there is no general agreement on what the standard form should be. However, we can follow some typical guidelines to obtain a standard form for symbolic expressions that allow us to symbolically match PEs to components of T_S [22].

First we consider the requirements of a canonical symbolic expression; a representation of a propagation function. Let F be a set of functions and E be a set of expressions. The elements of E are *canonical* if there is a bijective mapping from F to E, that is, two different expressions in E always correspond to different functions in F. We need to define the set of functions to be considered and the corresponding set of canonical expressions, and design a simplification algorithm that always converts arbitrary expressions representing functions in F to canonical form.

In order to define the set of possible functions F, we consider the types of module functions used in typical circuits. We consider a restricted set of module functions first, the arithmetic functions. Assume that circuits are formed from adders, subtracters, multipliers, and truncate modules, whose corresponding module functions are the integer operations: +, −, ×, and division by 2^n (truncating n lower order bits is equivalent to division by 2^n), respectively. The set F contains the functions computed at various points in a circuit formed by interconnections of these modules, as illustrated by Fltrdp in Figure 4.1. We can define corresponding canonical representations with the following features:

- All expressions are fully expanded, that is, the distributive law $A(B+C) \rightarrow AB+AC$ cannot be applied to any subexpression to get a new representation
- No equivalent terms are present, that is, no terms with the same set of vector sequence symbols
- No zero terms are present, that is, no terms with a coefficient of 0.

The canonical expressions have the same form as multi-variable polynomials with rational coefficients [22]. Each term of the expression can contain multiple vector-sequence "variables" as well as a rational coefficient, in this case, a fraction with denominator 2^n. A typical "multi-variable" PE is $(1/2)A_1A_2 + A_1A_3$, where A_1, A_2, and A_3 are vector sequences and multiplication is represented by juxtaposition. Two different expressions in this form represent different functions. Moreover, all expressions that can be created by function composition in the circuits described above can be converted to this form by the following technique.

Algorithm for creating canonical forms

1. Distribute multiplication over addition $A(B+C) \rightarrow AB+AC$
2. Collect equivalent terms by adding coefficients
3. Eliminate any resulting zero terms

As an example, expression (4.1) can be simplified as follows:

$$\frac{1}{2}\left(\frac{1}{2}((A_1+1)+(A_1+1))+A_1\right)$$

$$= \frac{1}{4}A_1 + \frac{1}{4} + \frac{1}{4}A_1 + \frac{1}{4} + \frac{1}{2}A_1 \quad \text{Distribute multiplication over addition}$$

$$= A_1 + \frac{1}{2} \quad \text{Add like terms}$$

Individual vectors in a vector sequence are treated as integers, therefore fractions that do not reduce to an integer only have meaning as coefficients, and $A_1 + \frac{1}{2}$ becomes A_1. By applying the steps above, we have simplified (4.1) to a canonical expression that matches the required test package component. In Figure 4.1, the simplifications were applied as soon as possible at the output of each module.

In general, PEs in a modular circuit are more complex than in the example above. For instance, AND, OR, and NOT (word) gates introduce Boolean operations, giving rise to Boolean expressions. The most common canonical representation for a Boolean function is the sum-of-minterms or disjunctive normal form [55]. Boolean PEs can usually be simplified to reduce the number of both terms and literals. Most PEs contain a mixture of arithmetic and Boolean operations, as well as other higher-level operations. Such PEs may not have a canonical form that can be produced by a fixed set of simplification steps. However, in many cases the basic simplification steps for arithmetic expressions given above can still be recursively applied to arithmetic subexpressions, treating other functions as symbols. Similarly, as a separate step, the Boolean simplification operation can be applied to Boolean subexpressions, treating other functions as symbols. For example, the PE $(1/2)\,(((A+0) \vee (B \wedge A)) + A)$ is simplified by the following steps:

$$\frac{1}{2}(((A+0) \vee (B \wedge A)) + A)$$

$$= \left(\frac{1}{2}((A+0) \vee (B \wedge A)) + \frac{1}{2}A\right) \qquad \text{Distribute multiplication over addition}$$

$$= \left(\frac{1}{2}(A \vee (B \wedge A)) + \frac{1}{2}A\right) \qquad \text{Eliminate zero terms}$$

$$= \frac{1}{2}A + \frac{1}{2}A \qquad \text{Absorption (Boolean simplification)}$$

$$= A \qquad \text{Add like terms}$$

In this instance, we are able to reduce a mixed arithmetic/Boolean expression to its simplest form. More research is needed to formally characterize all of the possible PE functions and to develop simplification algorithms that guarantee a canonical form. However, by combining arithmetic and Boolean simplification algorithms heuristically, as shown above, many PEs can be simplified and matched symbolically to components of T_S. It is important to note that in all cases, even when PEs cannot be matched symbolically, they can always be evaluated numerically to check for equivalence as we showed above for Fltrdp.

Finally, we consider the cost of symbolic PE manipulation relative to the cost of propagating the individual test vectors of T_S separately to check for numerical equivalence. In symbolic propagation, a large number of test vectors, representing many clock cycles of data, are concurrently processed. This feature is directly exploited by *PathPlan*. By using simple propagation modes, it avoids manipulation of expressions and the corresponding processing costs. The various simplification steps mentioned above add to the computation time required for signal propagation. The cost is low if expressions are small, as in the Fltrdp example. Since PEs grow as a result of function composition, the longer the propagation path, the longer they can grow. PEs can be kept small if they are only propagated through a few modules, or if they can be simplified to a few symbols at the output of each module, never having a chance to grow large.

There is no bound on the size of expressions that can ultimately be simplified and successfully matched symbolically to a component of T_S. For example, suppose that the output of a subtracter module is connected to input X_1 of a MUT M, and suppose that A_1 is the component of T_S required at X_1. Let E_1 is an arbitrarily long expression being propagated in the circuit containing M. If $A_1 + E_1$ and E_1 are two inputs to the subtracter, then the output value (vector sequence) of the subtracter is $A_1 + E_1 - E_1 = A_1$, which matches the required signal for X_1. On the other hand, the cost of propagating E_1 and simplifying it at the output of each module on a propagation path to X_1 may be excessive, and negate the advantage of symbolic propagation. Therefore, the sizes of PEs that a test generator propagates symbolically should to be bounded in order to control cost. When expressions containing more than a fixed number of terms and symbols are created, the test generator can revert to evaluating stimulus functions to propagate individual test vectors numerically. In general, symbolic propagation and matching will be cost-effective when PEs representing stimulus functions are short and vectors sequences long. This will be the case for circuits composed of very large modules and regular buses, e.g., microprocessors in a microcontroller.

4.2. Hierarchical Error Propagation

Error propagation along transparent paths can sometimes be analyzed by simply checking the form of symbolic PEs. This is easy when errors are propagated along I-mode or T-mode paths as in *PathPlan*. We can also use the symbolic techniques discussed in Chapter III to identify cases where a combination of partially transparent paths may be combined to form a fully transparent distributed path. However, for a given fault model, the errors produced by a particular MUT are sometimes propagated along only partially transparent single or distributed paths. In these cases, successful error propagation depends on the types of errors to be propagated—it cannot be determined simply by analyzing the form of the symbolic PEs, the functions they represent must be evaluated.

In this section, we present a hierarchical method of error propagation. At the lowest level of abstraction, response functions are evaluated in a special way for the set of errors that can appear at the output of the MUT. This detailed method allows us to determine when all errors are successfully propagated through irregular circuits, but it is still more efficient than propagating individual single-bit errors as in classical test generation. At the highest level of abstraction, response functions with special properties are represented symbolically and propagated along transparent paths in a circuit model using high-level module functions. This abstract method allows us to efficiently propagate errors along all types of transparent paths, not just T-mode paths.

4.2.1 Response Sets

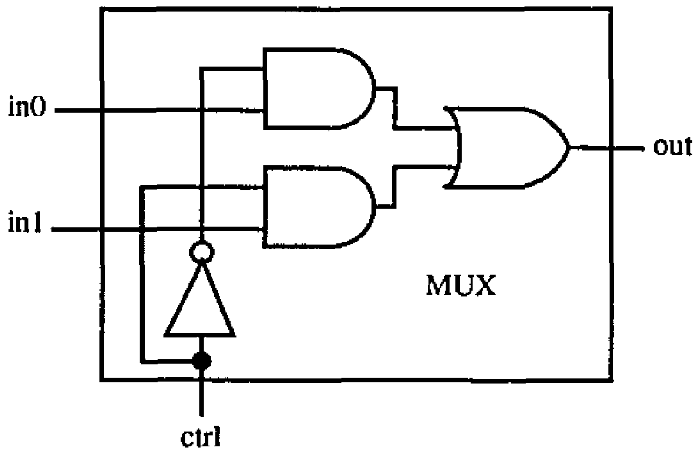
To analyze test response propagation in circuits with irregular buses, we can evaluate response functions for the set of values in their domain. Let $\{(T_R, T_{R1}), (T_R, T_{R1}), \ldots, (T_R, T_{R1})\}$ be the set of test response discrepancies to be propagated, where T_{Ri} is the response of the MUT to T_S when some fault i is active. For a given fault model, the set of all possible vectors that can appear in T_R or any T_{Ri}, that is, the domain of the response function, is called the *response set*. An element of the response set can be a correct response, a faulty response, or a correct response to some vectors in T_S and a faulty response to others.

Consider the two-input, 4-bit multiplexer shown in Figure 4.2b, which is constructed from four single-bit multiplexers (Figure 4.2a). Assuming that the SSL fault model is used, the test stimulus vector ctrl = 0, in0[3..0] = 15, and in1[3..0] = 0, produces the response set

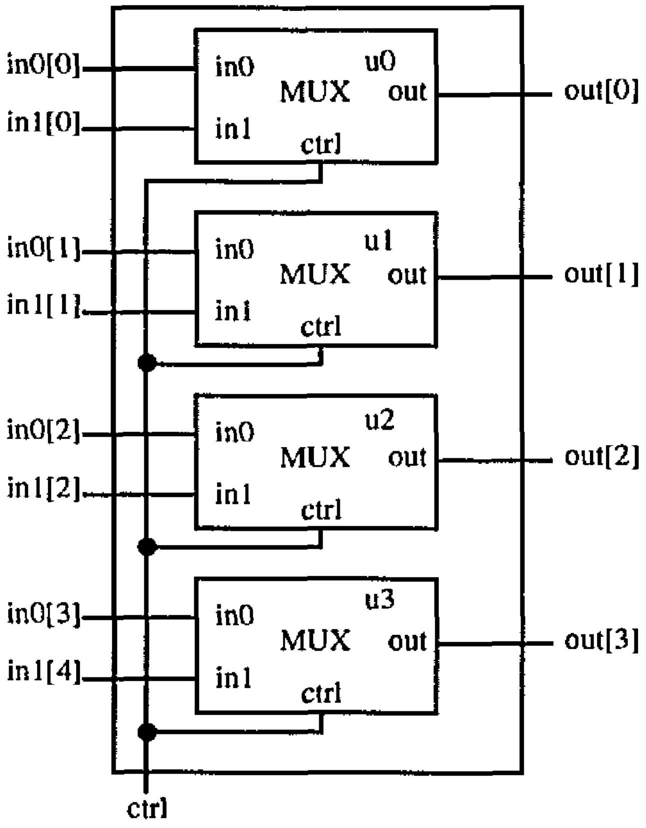
$$\Omega_{R1} = \{0, 7, 11, 13, 14, 15\}. \tag{4.2}$$

The correct response is out = 15; the other responses are due to SSL faults propagated to each individual output bit, as well as a stuck-at-1 fault on ctrl. Since there is no fanout within this module except for ctrl, all faults on lines other than ctrl produce independent errors at the output.

It can easily be shown that the test package



(a) Basic cell: two-input, single-bit multiplexer



(b) Four basic cells combined to form a two-input, 4-bit multiplexer

Figure 4.2. Implementation of a two-input, 4-bit multiplexer: (a) basic cell, and (b) four basic cells combined to form the multiplexer.

$$(T_S;T_R) = \left(\begin{bmatrix} 0 & 1 & 0 & 1 \\ 15 & 15 & 0 & 0 \\ 0 & 0 & 15 & 15 \end{bmatrix} ; \begin{bmatrix} 15 & 0 & 0 & 15 \end{bmatrix} \right)$$

detects all SSL faults in the multiplexer. The first row of T_S corresponds to input data bus ctrl, the second row corresponds to in0, and the third row corresponds to in1. The response set for this test package is

$$\Omega_R = \{0, 1, 2, 4, 7, 8, 11, 13, 14, 15\} \tag{4.3}$$

While 0 is an error response to the first vector, it is the correct response to the third vector.

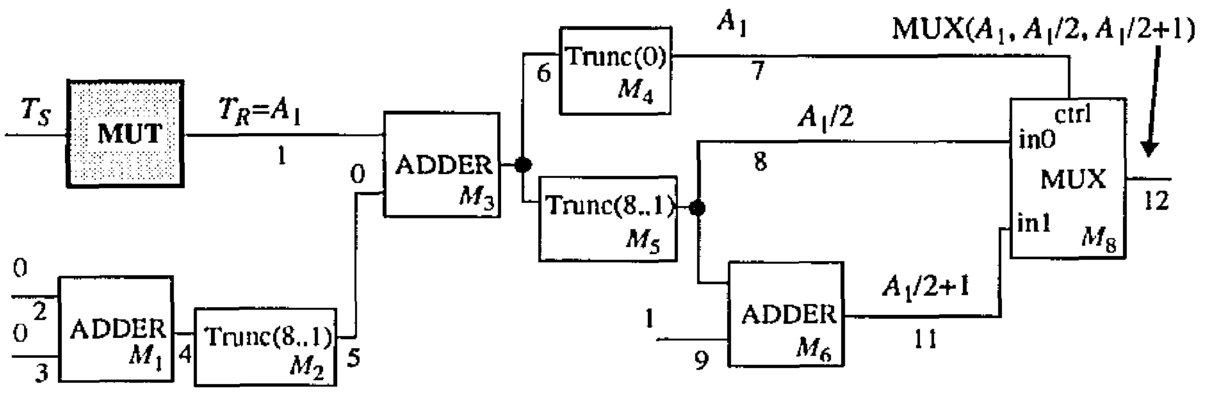
The size of Ω_R is only 10, while the number of SSL faults is 48. Therefore, evaluating the response function for Ω_R is more efficient in this case than propagating individual single-bit errors due to faults. For most modules, errors produce the same response for a given test sequence T_S. In addition, the size of the response set is bounded by the size of the bus at the output of the MUT, but often depends on the length of T_S rather than the size of the output bus or the number of faults in the MUT. For example, if we construct a two-input, 8-bit multiplexer from single-bit, multiplexers a test stimulus sequence similar to the one for the 4-bit multiplexer (four vectors) produces a response set of size 18. The module has 96 possible faults and 256 possible outputs.

4.2.2 Evaluating Response Functions

Next we show how to analyze error propagation by evaluating the response function, denoted P_R, for the vectors in the response set. As an illustration, let the MUT be the multiplexer shown in Figure 4.2b, and let the module M connected to its output be a modulo 16 incrementer. That is, $V(Z_D) = V(X_D) + 1 \pmod{16}$. The response set is given by (4.3); therefore, the response function evaluated is

Response	0	1	2	4	7	8	11	13	14	15
P_R	1	2	3	5	8	9	12	14	15	0

Assume that the response function from the output of a MUT to some other point Z in a circuit containing the MUT is always represented as a propagation function of the form defined in Chapter III. Let Ω_R be the MUT's response set; then $P_R = \{(\alpha_i;\beta_i)\}$, $1 \le i \le n$, where α_i is the set of elements of Ω_R that produce the same output β_i at Z, and $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is a parti-



P_{R1} = {(0;0), (1;1), (5;5), (6;6)}
P_{R7} = P_3°P_4 = {(0,6;0), (1,5;1)}
P_{R8} = P_3°P_5 = {(0,1;0), (5,6;2)} Ω_R = {0,1,5,6}
P_{R11} = P_3°P_5°P_6 = {(0,1;1), (5,6;3)} A_1 = [0 5]
P_{R12} = (P_3°P_4#P_3°P_5#P_3°P_5°P_6°P_7)°P_8 = {(0;0), (1;1), (5;3), (6;2)} ≡ 0

Figure 4.3. Response functions evaluated at various points on the propagation path to the output in Fltrdp. P_{Ri} denotes the response function on bus i.

tion of Ω_R. Continuing the example above, we represent the fully evaluated response function at the output of the incrementer as

$$P_R = \{ (0;1),\ (1;2),\ (2;3),\ (3;4),\ (4;5),\ (5;6),\ (7;8),\ (8;9),\ (10;11),\ (11;12),\ (12;13),\ (13;14),\ (14;15),\ (15;0)\ \} \equiv 0 \qquad (4.4)$$

In some cases, we can determine from the form of the response function whether all test response errors are distinguished at Z. If two elements of the response set, v_1 and v_2, are contained in the same block of the response function, they cannot be distinguished. On the other hand, if only one element of the response function is in each block, that is, if $P_R \equiv 0$ as in (4.4), then clearly all errors are distinguished. In this case, the circuit from the output of the MUT to Z is said to be *transparent relative to the response set.*

Figure 4.3 shows response functions evaluated at various points in Fltrdp for error propagation analysis. Each bus in the schematic is identified by a number below it; the output of the MUT is bus 1. The evaluated response function for each bus i is denoted P_{Ri}, and is listed below the schematic, together with the associated PE for bus i in the series-parallel form presented in Chapter III. As usual, P_i is the propagation function of module M_i. Above each bus in the schematic is a simplified symbolic expression in the form described above (Subsection 4.1.1). The test package $(T_S;T_R)$ is successfully instantiated at the MUT and T_R is the vector sequence [0 5], denoted by the symbol A_1. The response set $\Omega_R = \{0, 1, 5, 6\}$ contains two faulty values, 1 and 6, as well as the two correct values, 0 and 5, which appear in different vectors (timesteps) of $T_R = A_1$.

Although several response functions are greater than 0, the response function P_{R12} at the output (bus 12) is congruent to 0. We have shown here that Fltrdp is transparent relative to Ω_R. Recall that in Chapter III, we could not show that the path from bus 1 to bus 12 was fully transparent because, due to reconvergent fanout, the proof required explicit evaluation of series and parallel connection operations for transmission functions with $2^{17} = 131,072$ elements. However, response functions are usually much smaller than transmission functions. In general, it is much easier to show relative transparency than full transparency.

Figure 4.3 also demonstrates how fully evaluated response functions can be propagated in a circuit model as composite signal values. Note first that the response function at the output of the MUT is always 0. In Fltrdp, $P_{R1} = \{(0;0), (1;1), (5;5), (6;6)\} = 0$. The response function at the output of any other module M_i on the propagation path is computed by applying the module function for M_i to response functions at its inputs. Let $P_R = \{(\alpha_j;\beta_j)\}$ be a response function at the input to module M_i. The module function for M_i is used to compute a new value of β_i for each block of P_R. For instance, the response function $P_{R11} = \{(0,1;1), (5,6;3)\}$ at the output bus 11 of module M_6 in Fltrdp is computed from the response function $P_{R8} = \{(0,1;0), (5,6;2)\}$ on bus 8 by adding 1 to each β_i. If the updated values β_i and β_j for two blocks $(\alpha_i;\beta_i)$ and $(\alpha_j;\beta_j)$ are the same, then the blocks are combined to form $(\alpha_i \cup \alpha_j;\beta_i)$. For example, when $P_{R1} = \{(0;0),$ $(1;1), (5;5), (6;6)\}$ is propagated through the truncate module M_5, the result is $\{(0;0), (1;0), (5;2),$ $(6;2)\}$, which is rewritten as $P_{R8} = \{(0,1;0), (5,6;2)\}$.

When T_R is propagated along two or more paths to a point of reconvergence, such as module M_8 in Figure 4.3, then the response functions of these paths are combined by the parallel connection operation. Therefore, P_{R7}, P_{R8}, and P_{R11} in Figure 4.3 are combined to form

$$P_{R7}\#P_{R8}\#P_{R11} = \{(0;(0,0,1)), (1;(1,0,1)), (5;(1,2,3)), (6;(0,2,3))\} \tag{4.5}$$

The individual components of each β_i in (4.5) are then applied to their corresponding buses to compute the response function at the output of the multiplexer. For example, block 1 of (4.5) is

(0;(0,0,1)), therefore, $\beta_1 = (0,0,1)$. The first component 0 of β_1 corresponds to the control input of M_8 (bus 7). The second component 0 corresponds to the in0 input of M_8 (bus 8), and the third component corresponds to the in1 input of M_8 (bus 11). Therefore, this block is propagated to the output of module M_8, the primary output of the circuit, as (0;0).

We have achieved our goal of developing an error propagation technique for irregular buses. We can analyze error propagation along a distributed propagation path, at least for the case where the distributed path is transparent relative to the response set of the MUT. Previously reported test generators such as *PathPlan* and ARTEST [58] cannot analyze such distributed propagation paths. These test generators can only propagate errors over fully transparent single (not distributed) propagation paths, that is, paths whose *transmission function* is congruent to zero.

4.2.3 Error Propagation Analysis

Next, we show how to analyze test response error propagation when the (possibly distributed) propagation path is not transparent relative to the response set of the MUT. Let P_R be a response function on response set Ω_R from the output of the MUT to some point Z in a circuit. All the test response errors for a particular MUT can often still be propagated, even if $P_R > 0$. For instance, if two elements v_1 and v_2 in Ω_R are both correct or both faulty, it does not matter whether they are combined in the same block and therefore indistinguishable. Whereas, previously we have been analyzing cases where all possible pairs of responses must be distinguished, here we are concerned with distinguishing discrepancies of the form (v, v^e), $v, v^e \in \Omega_R$, where v is a correct response and v^e is a error response due to faults in a particular MUT exposed by some vector in T_S. All errors produced by the MUT can be propagated to Z if all discrepancies of the form (v, v^e) can be distinguished at Z.

If the test stimulus sequence T_S consists of only one vector, then there is only one correct response v. All errors exposed by T_S are propagated to Z if v is not combined in one block of P_R with any other response v^e in the response set. The requirement that the correct response v be separated from faulty responses in P_R is called the *propagation condition*.

If T_S consists of more than one vector, then the corresponding response set for the MUT is the union of the response sets for each vector in T_S. Any element of the response set can be a correct response, a faulty response, or a correct response to some vectors in T_S and a faulty response

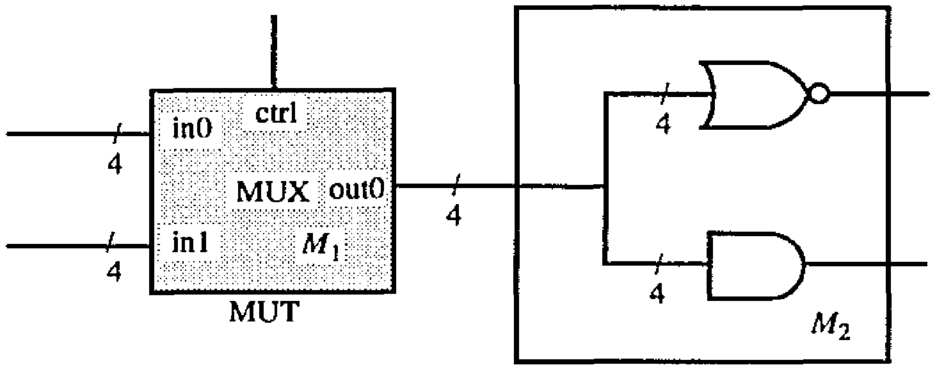


Figure 4.4. Four-bit 2-input multiplexer connected to a module detecting all-zero and all-ones vectors (M_2).

to others. As we showed above, the test package

$$(T_S;T_R) = \left(\begin{bmatrix} 0 & 1 & 0 & 1 \\ 15 & 15 & 0 & 0 \\ 0 & 0 & 15 & 15 \end{bmatrix};\begin{bmatrix} 15 & 0 & 0 & 15 \end{bmatrix}\right)$$

detects all SSL faults in the 4-bit multiplexer of Figure 4.2b. Let Ω_{Ri} be the response set of vector i in T_S, then Ω_{R1} is given by equation (4.2); $\Omega_{R1} = \{0,7,11,13,14,\underline{15}\}$. Similarly, $\Omega_{R2} = \{\underline{0},1,2,4,8,15\}$, $\Omega_{R3} = \Omega_{R2}$, and $\Omega_{R4} = \Omega_{R1}$. The correct response in each set is underlined. The union of these response sets, $\Omega_R = \{0,1,2,4,7,8,11,13,14,15\}$, is the response set for T_S.

The *order* of a response set is the number of test vectors in the corresponding test stimulus T_S. Response sets Ω_{Ri}, $1 \le i \le 4$, given above are first-order response sets and Ω_R has order 4. Note that there is often considerable overlap between first-order response sets, therefore it is much simpler to evaluate the response function for the combined response set than for each individual first-order response set. Let P_R be the response function at some point Z in the circuit being tested. All errors exposed by T_S are propagated to Z if the propagation condition is met for all the first-order response sets corresponding to the individual vectors in T_S.

Suppose that the 4-bit multiplexer of Figure 4.2b is part of the circuit shown in Figure 4.4. The output of module M_2 is 10_2 if its input is the all-0 vector, 01_2 if its input is the all-1s vector, and 00_2 otherwise. Therefore, the response function for the test package given above is $P_R = \{(0;2), (1,2,4,7,8,11,13,14;0), (15;1)\}$. Clearly, module M_2 is not transparent, or even transparent

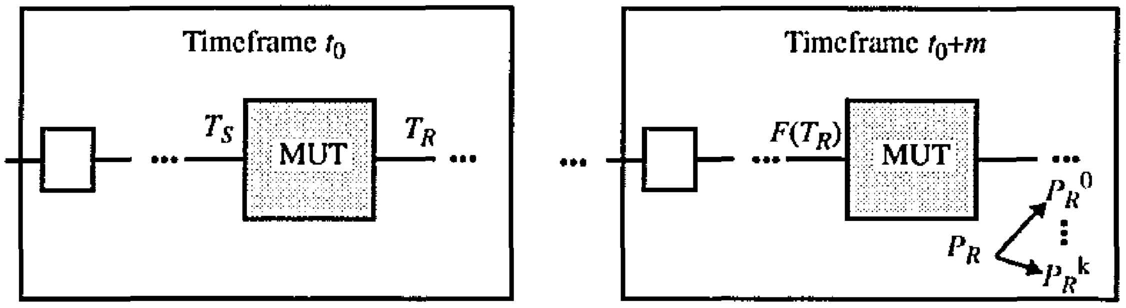


Figure 4.5. Test response T_R propagated through MUT in timeframes later than when T_S is applied.

relative to Ω_R. However, it does meet the propagation condition for vectors 1 and 4 in T_S since 15 is not combined with any other response. It also meets the propagation condition for vectors 2 and 3 in T_S, since 0 is not combined with any other response. Therefore, all errors are propagated to the output of M_2.

As discussed in Chapter II, if the MUT is on a global feedback path, then T_R may have to be propagated through the MUT in timeframes later than when T_S is applied to the inputs of the MUT. The faults in the MUT may affect the propagation of errors produced when the MUT is tested in timeframe t_0 The situation is illustrated in Figure 4.5, which depicts a test package $(T_S;T_R)$ successfully instantiated at the MUT at timeframe t_0, and a symbolic expression $F(T_R)$ containing T_R propagated to the input of the MUT in timeframe t_0+m. The response function at the output of the MUT is P_R. To analyze error propagation in this case, the test generator must have detailed knowledge of the faulty behavior of the MUT. Each fault f_i, $1 \leq i \leq k$, in the MUT can alter its function and lead to a different response function at its output in timeframe t_0+m. Let P_{Ri}, $1 \leq i \leq k$, be the response function associated with fault f_i, and let P_{R0} be the fault-free response function. This is depicted in Figure 4.5. As an example, suppose that for a given T_S, the MUT has two first-order response sets $\Omega_{R1} = \{\underline{0},1\}$ and $\Omega_{R2} = \{\underline{5},6\}$ leading to an order-2 response set $\Omega_R = \{0,1,5,6\}$. At the output of the MUT in timeframe t_0+m, let $P_{R0} = \{(0;1), (1;2), (5;6), (6;7)\}$ be the fault-free response function, $P_{R1} = \{0,1;1), (5;4), (6;5)\}$ be the response function due to fault 1, and $P_{R2} = \{(0;1), (1,5;2), (6;3)\}$ be the response function due to fault 2.

To determine whether all errors are propagated, all P_{Ri}s must be analyzed to determine if they satisfy the propagation condition. If so, then all errors are propagated. On the other hand, if the propagation condition is not met in the fault-free response function P_{R0}, then some errors are not propagated. If the propagation condition is satisfied by P_{R0}, but not by some of the faulty versions, then further analysis is required. In the example above, the propagation condition is definitely met by P_{R0} since it is transparent relative to Ω_R. The propagation condition is also met by P_{R2}, but not by P_{R1} since responses 0 and 1 are contained in the same block. We must analyze P_{R1} further as follows. If fault f_1 causes the correct response 0 to become 1 in Ω_R, then the error (0,1) is not propagated. However, if fault f_1 does not cause this error (and perhaps others), then all errors are propagated through the MUT. This analysis requires detailed fault models for the MUT, and is necessary if propagation through the MUT cannot be avoided. However, such detailed models of modules to be tested are often unavailable when precomputed testing is used.

In summary, test response propagation in irregular bus-structured circuits can be analyzed by evaluating response functions for the response set of the MUT when T_S is applied. This method satisfies our goal of analyzing such circuits. For large modules, the response set is usually small relative to the total number of single-bit errors that must be propagated using low (gate-) level methods. The response set for a MUT with an n-bit output bus is also usually small relative to the total number of possible errors for an n-bit bus. Therefore, fully transparent propagation paths are rarely needed, although they simplify analysis when they exist.

The response functions are represented for analysis as pairs of the form $(\alpha_i;\beta_i)$, $1 \le i \le n$, where $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ is a partition of the response set and β_i is the output of the response function for the subset α_i. They can be computed by propagating them as complex signals in a circuit model. If a response function P_R is congruent to zero, then the propagation path represented by P_R is transparent relative to the response set and all errors are propagated. This case is easy to analyze. On the other hand, if $P_R > 0$, then the possible errors may still be propagated, however we must analyze them according to the propagation condition, that is, the requirement that all correct responses are in separate blocks of P_R from faulty responses. Finally, we can analyze whether test response errors are propagated through the MUT in timeframes other than when T_S is applied by evaluating several response functions corresponding to faulty versions of the MUT. However, this analysis requires a detailed fault model for the MUT, which may not be available. Analysis of error propagation using response functions is implemented in our test generation tool *MATSim*, discussed in Chapter V. Next, we discuss propagation in regular or nearly regular circuits.

4.2.4 Symbolic Error Propagation

Although error propagation in complex bus-structured circuits can be analyzed by evaluating response functions, in many cases such detailed analysis is unnecessary. In addition, for some modules, the data necessary to construct response sets may be unavailable. Since most standard datapath modules are fully transparent for some modes of operation (see Table 3.2), most non-transparency is due to bus truncation or random logic blocks. When a circuit contains only regular buses and no random logic blocks, such as the Encode circuit in Figure 2.2, a more abstract method of error propagation analysis can be used. Even in Fltrdp, where irregular buses do exist, there are some transparent paths. This fact is exploited by *PathPlan* and ARTEST. However, there are many transparent propagation paths that are not T-mode paths, so they cannot be analyzed by *PathPlan*, and ARTEST uses an ad hoc method of error propagation that requires mixing abstraction levels. In this subsection, we discuss hierarchical representations of response functions as symbolic signal values and show how module functions can be systematically constructed to propagate these symbols.

Symbolic Types. We will now assign symbols to various groups of propagation functions to classify them at a very high level. For example, all transmission functions with the same transparency index might be placed in a single group identified by a "type" symbol. For purposes of error propagation in test generation, we are interested in symbolic representations of response functions. Let R be the symbol associated with all response functions P_R such that $P_R \equiv 0$. For example, if the bus X_D connected to the output of the MUT is 2 bits wide, then R contains all response functions of the form {{(0;0), (1;1), (2;2), (3;3)}, {(0;0), (1;1), (2;2)}, ...}. In other words, R contains all fully transparent response functions as well as all possible response functions that are transparent relative to some smaller set $\Omega_R \subset \{V(X_D)\}$. Let Z be a bus on the propagation path from the output of the MUT. If the response function at Z has type R, then all possible errors can be observed at Z; the path to Z is transparent.

Similarly, let C be the symbol associated with the set of all response functions P_R such that $0 < P_R < 1$. For example, if the bus X_D connected to the output of the MUT is 2 bits wide, then C contains all response functions of the form {{(0,1;0), (2;2), (3;3)}, {(0;0), (1,2;1), (3;3)},

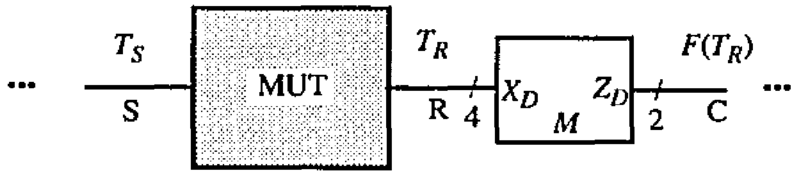


Figure 4.6. Use of symbolic values to represent propagation functions.

...}. If a bus Z on the propagation path from the output of the MUT has a response function of type C, then we cannot determine whether all test response errors exposed by T_S are propagated to Z without analyzing the fully evaluated form of the response function as described above.

Since stimulus functions that propagate T_S do not carry errors, we classify them separately; let S be the symbol associated with any stimulus function. Finally, both stimulus and response functions may be undetermined in some circuit states; in these cases, no information is propagated. Let X be the symbol associated with any propagation function P such that $P \equiv 1$. The four symbols X, S, R, and C represent our basic set of propagation function types that are useful for testing using precomputed tests.

Figure 4.6 illustrates the use of symbolic type to represent propagation functions for testing with precomputed tests. In this figure, the test package $(T_S;T_R)$ is successfully instantiated at a MUT. Symbolic expressions of vector sequences are propagated to the inputs of the MUT and matched to T_S. The test response T_R is produced at the output of the MUT and propagated through a module M with function F as a symbolic expression denoted $F(T_R)$. These symbolic expressions represent propagation functions. Below each bus is the symbol that represents the type. The stimulus function that matches T_S has type S. The response function at the output of the MUT has type R. However, the output data bus Z_D of M is smaller than its input data bus X_D; M is not transparent. Therefore, the type of the response function at the output of M is C.

Symbolic Signals. X, S, R, and C can also be defined using P-sets and U-sets as symbolic signal values representing the information that can be propagated by the corresponding response function type. For example, let Z be a bus on the propagation path from the output of the MUT and let P_R be the response function at Z. Then R is the symbolic signal value representing the set of discrepancies {(0,{1,2,3}, (1,{0,2,3}), (2,{0,1,3}), (3,{0,1,2})}, where the first element of each pair is a

Symbol	Definition	Interpretation	Two-bit example
X	$\{(V_n, V_n)\}$	Unknown signal	{({0,1,2,3},{0,1,2,3})}
S	$\{(v_i, v_i), v_i \in V_n\}$	Stimulus signal	{(0,0), (1,1), (2,2), (3,3)}
R	$\{(v_i, V_n - v_i), v_i \in V_n\}$	Response signal	{(0,{1,2,3}), (1,{0,2,3}), (2,{0,1,3}), (3,{0,1,2})}
C	$\{(v_i, V_n), v_i \in V_n\}$	Corrupted signal	{(0,{0,1,2,3}), (1,{0,1,2,3}), (2,{0,1,2,3}), (3,{0,1,2,3})}

Table 4.1 Definitions and examples of the elements of R_4.

vector propagated to Z in a correctly working circuit and the second element is the set of possible values propagated to Z in a faulty circuit. If P_R has type R, then all these discrepancies can be propagated to Z; the two definitions for R are therefore equivalent. In general, let $V_n = \{0, 1, \ldots, 2^n - 1\}$ be the set of all possible vectors that may be applied to an n-bit bus Z, then R = $\{(v_i, V_n - v_i), v_i \in V_n\}$.

Similarly, S can be defined as the set $\{(v_i, v_i), v_i \in V_n\}$. In this case, if S is propagated to a bus Z, then Z has the same value in a faulty circuit as it does in a correctly working circuit, since stimulus functions do not propagate errors. X can be defined as $\{(V_n, V_n)\}$ which indicates that the value of a bus could be any value in either the correct or the faulty circuit, that is no information is propagated. Finally, C can be defined as $\{(v_i, V_n), v_i \in V_n\}$, which indicates that while the value of a bus in the good circuit is known exactly, the value in the faulty circuit is unknown. In other words, we cannot tell whether errors are propagated. We refer to the set {X,S,R,C} as R_4. Table 4.1 summarizes the interpretation of R_4 as a set of symbolic signal values, and displays an example of each value for 2-bit buses.

We can also define less abstract symbolic signal values than S and R. For instance, consider the set of signals on a 2-bit bus. Let

$$R^{LO} = \{(0, \{1,3\}), (1, \{0,2\}), (2, \{1,3\}), (3, \{0,2\})\}$$

and

$$R^{HI} = \{(0, \{2,3\}), (1, \{2,3\}), (2, \{0,1\}), (3, \{0,1\})\}$$

R^{LO} and R^{HI} are both contained in R, and together, they contain all possible discrepancies. They form a complete set of less abstract symbolic error values than R. Now, R^{LO} is not blocked by a truncate module M_1 that removes the most significant bit of a 2-bit bus and R^{HI} is not blocked by a truncate module M_2 that removes the least significant bit of a two-bit bus. However, R is blocked by both M_1 and M_2. Therefore, there is a useful hierarchy of symbolic signal values with R_4 at the top and fully evaluated response functions at the bottom. Less abstract values are used when greater precision is required, and more abstract values are used when greater efficiency is desired. By refining R into some less abstract values, error information can be propagated through circuits with a small number of truncated buses.

Signal Value Algebras. Symbolic signal values can be propagated through a circuit model using high-level module functions. Together the functions and signal value set form a symbolic signal value algebra. Lee and Patel introduced a symbolic signal-value (type) algebra in [58]. Their types are constructed in ad hoc fashion, and as a result, the associated module functions cannot be rigorously derived from basic operations on the underlying signal sets. The module functions are implemented using rule-based methods, which are often inefficient and difficult to prove correct. Moreover, some module functions cannot be closed for the given type set and consequently the algebra is not well-defined. By treating symbolic response function types as symbolic signal values using P-sets and U-sets, and using the operation extension methods we presented in Chapter II, we can easily construct well-defined and useful algebras based on R_4.

We will illustrate the construction of module functions using the addition operation, denoted *ADD*. Some examples of the operation applied to elements of R_4 are as follows:

1. $ADD(\mathrm{X},\mathrm{X}) = ADD(\{v_i + v_j, \text{ for all } i, j\}, \{v_i + v_j, \text{ for all } i, j\}) = \mathrm{X}$, assuming + is closed and onto for $v_i + v_j$.

2. $ADD(\mathrm{S}, \mathrm{S}) = \{ (v_i + v_j, v_i + v_j) \text{ , for all } i, j\} = \{ (v_h, v_h), h = 1, \ldots, k\} = \mathrm{S}$

3. $ADD(\mathrm{S}, \mathrm{R}) = (v_i + v_j, \{v_i + v_h\}) = \mathrm{R}$, for all i, j and h such that $v_h \neq v_j$, since $\{v_i + v_h\}$ cannot contain $v_i + v_j$ if $(v_i + v_p) \neq (v_i + v_q)$ for all $p \neq q$.

Note that $ADD(\mathrm{R}, \mathrm{R}) = \{v_i + v_j, \{v_h + v_k\} \} = \mathrm{C}$, for all i, j, h, k, such that $v_i \neq v_h$ and $v_j \neq v_k$. Thus adding two response signals results in a corrupted output—the algebra is extremely pessimistic about the propagation of errors. As previously noted, R is blocked (becomes C) whenever a

ADD	X	S	R	C
X	X	X	X	X
S	X	S	R	C
R	X	R	C	C
C	X	C	C	C

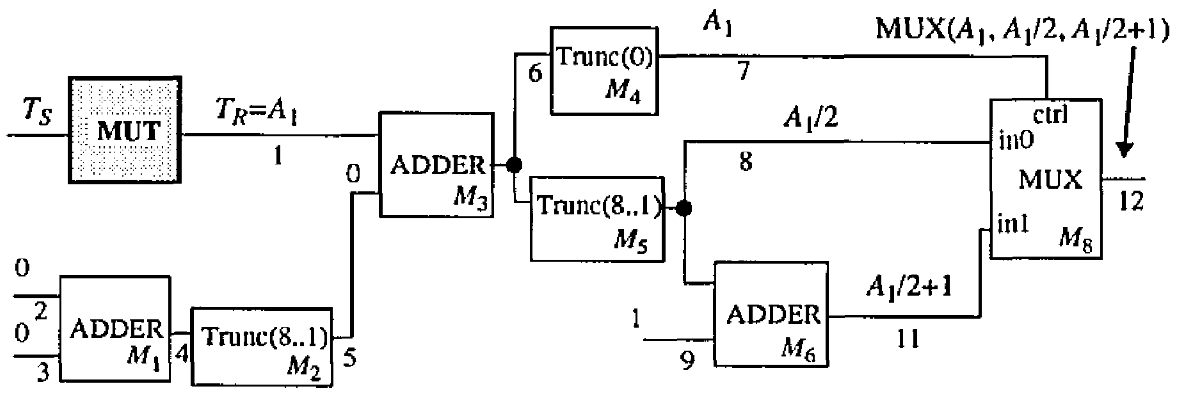
Table 4.2 The *ADD* operation and R_4.

AND	X	S	R	C
X	X	X	X	X
S	X	S	R	C
R	X	R	R	X
C	X	C	X	C

Table 4.3 The *AND* operation and B_4.

module function is not transparent, despite the fact that many specific discrepancies may still be propagated.Table 4.2 shows the *ADD* operation table for R_4 obtained by combining all pairs of elements in R_4, as we did above for *ADD*(X, X), *ADD*(S, S), and *ADD*(S, R). Finally, note that all subsets of R_4 containing X, but not containing the error signal value R, are closed subalgebras of (R_4, *ADD*). In particular, the smallest closed algebra is based on the singleton set {X}. Algebras based on {X, S}, {X, C}, and {X, S, C} are also closed.

In many cases, we can develop a very efficient implementation for the high-level module functions by properly encoding the symbolic values as vectors of Boolean values. There is no guarantee that a good encoding will be found, however—it depends on the underlying algebra. To determine a good encoding, one strategy is to look for similarities with other algebras that already have good encodings. The *ADD* module function for R_4 is very similar to the *AND* operation for a four-element Boolean algebra B_4. Table 4.3 shows the *AND* function table for the four element Boolean algebra. Here the elements of the algebra have been assigned symbols from R_4. X is the zero element and S is the unit element in Table 4.3. R and C are intermediate values. Tables 4.2 and 4.3 are identical except for the elements shaded in Table 4.3. Therefore, we define an encoding



P_{R1} = R
P_{R7} = C
P_{R8} = C
P_{R11} = C
P_{R12} = C

A_1 = [0 5]

Figure 4.7. Response functions represented as symbolic signal values on a propagation path to the output of Fltrdp. P_{Ri} denotes the response function at bus i.

for R_4 by slightly modifying the standard encoding for the four element Boolean algebra B_4 to aid in handling the differences between (R_4, *ADD*) and (B_4, *AND*). B_4 requires only two bits. X is encoded as 00_2, S is 11_2, R can be 01_2, and C can be 10_2. *AND* is then computed using logical AND ($\wedge$) for individual bits independently (word-wise AND). For R_4, we first encode the underlying sets: $V_n \to 00_2$, $v_i \to 11_2$, and $(V_n - v_i) \to 01_2$. Next we concatenate the codewords of each element in a pair. This implies that X = 0000_2, S = 1111_2, R = 1101_2, and C = 1100_2. The output of the *ADD* module function can be computed by taking the word-wise *AND* of the inputs, and treating the case where either input is *R* separately. This can be easily programmed as follows.

```
ADD( input $t_1$, $t_2$: symbolic signal values)
{
    $t_{out}$ = $t_1 \wedge t_2$
    if ($t_{out}$ == R && $t_1 \vee t_2$ == R)
        $t_{out}$ = C;
    return( $t_{out}$ );
}
```

Figure 4.7 shows response functions represented as symbolic signal values at points along a propagation path from the output of the MUT to the primary output in Fltrdp. As in Figure 4.3, each bus in the schematic is identified by a number below it; the output of the MUT is bus 1. Here the response functions for each bus i is denoted P_{Ri} and is listed below the schematic together with the associated symbolic signal value. PEs not listed are assumed to be stimulus functions with the corresponding symbolic signal value S. Above each bus in the schematic is a simplified symbolic expression in the canonical form described in Subsection 4.1.2 The test package $(T_S;T_R)$ is successfully instantiated at the MUT and T_R is the vector sequence [0 5], denoted by the symbol A_1. Note that R is propagated through the adder (module M_3) but is immediately corrupted by the two truncate modules M_4 and M_5. Thus error propagation at this level of abstraction is more pessimistic than the analysis shown in Figure 4.3.

Although, R_4 is sufficient to correctly represent the propagation of error signals through the set of modules in Fltrdp, R cannot be propagated through other typical module functions. For instance, R cannot be propagated through any input of an *AND* gate, regardless of the other input. To address this issue, we add some less abstract values to R_4 in order to propagate R through datapath modules other than adders. Let 0_n be the n-bit all-0 vector, that is, {(0,0)}, let 1_n be the n-bit all-1 vector, that is, $\{(2^n-1, 2^n-1)\}$, and let $\hat{S}$ be the same as S without 0_n and 1_n, that is $\hat{S} = S - \{0_n, 1_n\}$. We will construct an algebra from the set {*ADD, SUBTRACT, MULTIPLY, AND, OR, NOT, XOR*} of module functions, and the set $R_4 + \{0_n, 1_n, \hat{S}\}$ of signal values. We call this signal set R_7; it has the property that it is closed for all of the above operations and R appears as an output in all functions, that is, R is propagated when the function is transparent.

Consider the function tables for *AND, MULTIPLY, ADD/SUBTRACT/XOR*, and *OR*, shown in Tables 4.4–4.7. These tables were constructed using the algebra extension techniques developed above for (R_4, ADD). Note that when inputs are elements of R_4 = {X, S, R, C}, R does not appear as an output in any of these tables. By adding $\hat{S}$, R can be propagated through the *MULTIPLY* function, since *MULTIPLY*($\hat{S}$, R) = R. Similarly, 1_n is needed to propagate R through the *AND* function, and 0_n is needed to propagate R through the *OR* function.

In each function in Tables 4.4–4.7, a subfunction consisting of four symbols is enclosed in a box. In the function table for *ADD/SUBTRACT/XOR* (Table 4.4) and the table for *MULTIPLY* (Table 4.5), the behavior of the set $\hat{R}_4$ = {X, $\hat{S}$, R, C} is exactly the same. Therefore, we can implement a single module subfunction for $\hat{R}_4$ and {*ADD,SUBTRACT,XOR,MULTIPLY*}, and construct module functions *ADD, SUBTRACT, XOR*, and *MULTIPLY* for R_7 by treating the ele-

AND	X	Ŝ	R	C	S	0_n	1_n
X	X	X	X	X	X	0_n	X
Ŝ	X	S	C	C	S	0_n	Ŝ
R	X	C	C	C	C	0_n	R
C	X	C	C	C	C	0_n	C
S	X	S	C	C	S	0_n	S
0_n	0_n	0_n	0_n	0_n	0_n	0_n	0_n
1_n	X	Ŝ	R	C	S	0_n	1_n

Table 4.4 Function table for the AND operation.

MULTIPLY	X	Ŝ	R	C	S	0_n	1_n
X	X	X	X	X	X	0_n	X
Ŝ	X	Ŝ	R	C	S	0_n	Ŝ
R	X	R	C	C	C	0_n	R
C	X	C	C	C	C	0_n	C
S	X	S	C	C	S	0_n	S
0_n	0_n	0_n	0_n	0_n	0_n	0_n	0_n
1_n	X	Ŝ	R	C	S	0_n	1_n

Table 4.5 Function table for the MULTIPLY operation.

ments of $R_7 - \hat{R}_4$ as special cases. Similarly, since the behavior of $\hat{R}_4$ in the tables for *AND* (Table 4.6) and *OR* (Table 4.7) is the same, we can implement a single module function for $\hat{R}_4$ and $\{AND, OR\}$, and construct module functions AND and OR for R_7 by treating the elements of $R_7 - \hat{R}_4$ as special cases in each function.

To implement the algebra R_7, we begin by encoding the elements of R_7 as we did for R_4, namely, X = 0000_2, S = 1111_2, R = 1101_2, and C = 1100_2. For R_7, we encode the elements of the subset $\hat{R}_4$ similarly, but add an extra 0 bit, thus: X = 00000_2, Ŝ = 01111_2, R = 01101_2, and C = 01100_2. For the elements of $R_7 - \hat{R}_4$, we set the most significant bit to 1.

ADD	X	Ŝ	R	C	S	0_n	1_n
X	X	X	X	X	X	X	X
Ŝ	X	Ŝ	R	C	S	Ŝ	S
R	X	R	C	C	R	R	R
C	X	C	C	C	C	C	C
S	X	S	R	C	S	S	S
0_n	X	Ŝ	R	C	S	0_n	1_n
1_n	X	S	R	C	S	1_n	Ŝ

Table 4.6 Function table for the ADD, SUBTRACT, and XOR operations.

OR	X	Ŝ	R	C	S	0_n	1_n
X	X	X	X	X	X	X	1_n
Ŝ	X	S	C	C	S	Ŝ	1_n
R	X	C	C	C	C	R	1_n
C	X	C	C	C	C	C	1_n
S	X	S	C	C	S	S	1_n
0_n	X	Ŝ	R	C	S	0_n	1_n
1_n	1_n	1_n	1_n	1_n	1_n	1_n	1_n

Table 4.7 Function table for the OR operation.

We encode 1_n as all 1s, that is, $1_n = 11111_2$, 0_n as all 0s except for the most significant bit, that is, $0_n = 10000_2$. Finally, S is encoded as 10001_2.

Algorithms for implementing $\{ADD, SUBTRACT, XOR, MULTIPLY\}$ for $\hat{R}_4$ and $\{AND, OR\}$ for $\hat{R}_4$ appear in Figure 4.8. As in the module function *ADD* for R_4, t_1 and t_2 are the encoded versions of the symbolic signal values. Sample algorithms for implementing the module functions are given in Figures 4.9 and 4.10. Figure 4.9 shows the module function for *ADD*. Evaluations functions for *MULT* and *XOR* are similar. Figure 4.10 shows the module function for *AND*. The module function for *OR* is similar.

```
{ADD,SUBTRACT,XOR,MULTIPLY}( input $t_1$, $t_2$: symbolic signal value)
{
    $t_{out} = t_1 \wedge t_2$;
    if ($t_{out}$ == R && $t_1 \vee t_2$ == R)
        $t_{out}$ = C;
    return( $t_{out}$ );
}
```

```
{AND,OR}( input $t_1$, $t_2$:symbolic signal value )
{
    if ($t_1$ == R) $t_1$ = C;
    if ($t_2$ == R) $t_2$ = C;
    $t_{out} = t_1 \wedge t_2$;
    return( $t_{out}$ );
}
```

Figure 4.8. Algorithms for subfunctions *{ADD,SUBTRACT,XOR,MULTIPLY}* and *{AND,OR}*.

```
ADD( input $t_1$, $t_2$:symbolic signal value )
{
    /* Convert $1_n$, $0_n$ and S to $\hat{S}$ and compute subfunction */
    if ($t_1 \wedge$ 10000) $t_1' = \hat{S}$;
    else $t_1' = t_1$;
    if ($t_2 \wedge$ 10000) $t_2' = \hat{S}$;
    else $t_2' = t_2$;

    $t_{out}$ = {ADD,SUBTRACT,XOR,MULTIPLY}( $t_1'$, $t_2'$ );

    /* Special cases */
    if (($t_{out}$ == $\hat{S}$) && ($t_1 \wedge$ S == S || $t_2 \wedge$ S == S)) $t_{out}$ = S;

    if ($t_1$ == $0_n$) $t_{out} = t_1$;
    if ($t_2$ == $0_n$) $t_{out} = t_2$;

    if ($t_1$ == $1_n$ && $t_2$ == $1_n$) $t_{out} = \hat{S}$;

    return( $t_{out}$ );
}
```

Figure 4.9. *ADD* module function for R_7.

ARTEST's Error Value Set. Now let us analyze a similar set of symbolic signal values used for propagating error information: the "typing" scheme given in [58] for the test generator ARTEST that we discussed in Chapter II. The set of signal values used by ARTEST is summarized in Table 4.8, which repeats Table 2.2. The functions associated with this set are not discussed in [58],

Error signal	Symbol	Interpretation
(X,X)	X	Unassigned
(X,V)	CF	Constant faulty
(X,U)	VF	Variable faulty
(V,X)	CG	Constant good
(V,V)	C	Constant good and gaulty
(V,V')	CGCFE	Constant good and constant faulty effect
(V,U)	CGVF	Constant good and variable faulty
(V,U')	CGVFE	Constant good and variable faulty effect

Table 4.8 The set of symbolic error signals or "types" used in ARTEST [58].

```
AND( input $t_1$, $t_2$:symbolic signal value )
{
    /* Convert $1_n$, $0_n$ and S to $\hat{S}$ and compute subfunction */
    if($t_1$ == S) $t_1' = \hat{S}$;
    else $t_1' = t_1$;
    if($t_2$ == S) $t_2' = \hat{S}$;
    else $t_2' = t_2$;

    $t_{out}$ = {AND,OR}( $t_1'$, $t_2'$ );

    return( $t_{out}$ );
}
```

Figure 4.10. *AND* module function for R_7.

and as mentioned earlier, are ad hoc.

Although the ARTEST type set contains several more values than R_4, there are few practical differences in its ability to represent abstract values. ARTEST was designed to support fault propagation from the output of a MUT, where the faulty value is not known, but is known to be faulty (denoted U'). This technique is used by ARTEST to generate tests for datapath circuits. It is assumed in [58] that datapaths are controlled by control units which are tested by a separate algorithm. The interface between datapath and control circuits is assumed to be neither directly controllable nor observable (e.g., by a scan chain). The control unit is tested by a conventional (low-level) test generation algorithm, and faults that are propagated to the interface are injected into the datapath circuit and propagated by the datapath test generation algorithm. The faulty values injected in this way are known, and are denoted V'. Thus, test response data generated at the output of a MUT is assigned type CGVFE and test response data injected from the control unit interface is assigned type CGCFE.

Since there is no apparent difference between "unknown" and "unassigned," U is equivalent to X. In this case, X and VF are equivalent to X in R_4 and CG and CGVF are equivalent to C in R_4. The type C in ARTEST is equivalent to S in R_4, and type CGVFE is equivalent to R in R_4. Only ARTEST types CF and CGCFE are unique to ARTEST. Type CF is an unassigned fault site, and its use is not clear. CGCFE is not an abstract value, rather it is a name given to the class of all possible low-level faults. That is, $CGCFE = \{(v_i, v_j)\}$, where $v_i \neq v_j$. In this case, knowledge of both correct and faulty values is required for propagation through modules. Since each element of CGCFE is not individually named, both correct and faulty values must be computed, implying that the class is superfluous.

We conclude that the ARTEST type set is roughly equivalent to R_4 in its ability to propagate error information. As we showed above, the level of abstraction represented by R_4 is extremely pessimistic and is insufficient to propagate signals through many module functions. The less abstract signal values added to R_7 are needed to propagate R through *MULTIPLY*, *AND*, and *OR*. Since R is blocked for *MULTIPLY*, *AND*, and *OR*, propagation of CGVFE will likewise be blocked for these module functions. To propagate R through these modules, ARTEST must perform a more detailed, low-level analysis for error propagation; the method for this is not discussed in [58].

4.3. Design for Transparency

In many circuits, error information is unavoidably lost as T_R is propagated from the output of the MUT to primary outputs. In this case, an alternate test package with a slightly different response set can sometimes be used to test the MUT. Often however, the constraints imposed by the propagation path block the alternatives as well. This is the case when only part of the function computed by a subcircuit M is used in the larger circuit containing M. In Fltrdp (Figure 4.1), for instance, truncate module M_2 is used to divide the output of module M_1 by two. When module

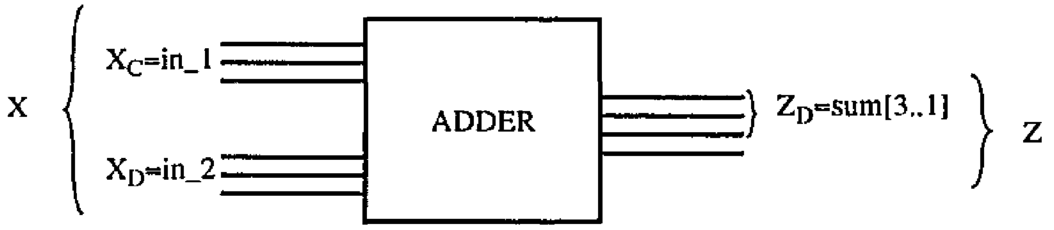


Figure 4.11. Three-bit adder module with input data bus X_D as addend, input control bus X_C as augend, and output data bus Z_D = sum[3..1].

M_1 is the MUT, some test response errors are blocked by module M_2 regardless of which test package we use. As discussed in Chapter II, the faults producing errors that cannot be propagated are redundant in these circuits, but since the test has been precomputed, we may not be able to separate detectable faults from redundant faults in T_R. Therefore, we want to increase transparency to improve testability. In this section, we present some examples of how the propagation theory of Chapter III can be applied to increase transparency.

One technique for designing circuits that can easily propagate T_R, is to route test points to primary outputs, perhaps through multiplexers [83]. Another technique is to use a hierarchical form of scan or boundary scan to ensure adequate observability [9]. These techniques lead to high routing overhead and highly constrained design styles. Both approaches provide additional paths for routing T_R. An alternative approach is to modify modules on the normal data paths of the circuit to make them more transparent, that is, to synthesize transparent modules. Some examples of places where specially designed transparent modules might be useful are: points of reconvergence as in Fltrdp (module M_8) and Divfilt (module 8 in Figure 3.18), truncate modules, and decoders and random logic modules where $|X_D| > |Z_D|$.

Let M be a module with $|X_D| > |Z_D|$. Then M cannot be designed to be 1-transparent. Our method is to synthesize a module M' to replace M, where M' has two modes of operation: *normal mode*, where M' behaves as M, and *test mode* where M' is k-transparent. Before discussing some methods for synthesizing M', we consider modules that are naturally k-transparent. As discussed in Section 3.4., the adder module in Figure 4.11 is transparent for the sequence $S(X_C) = [0, 1]$. The combination of adder module M_1 in Fltrdp (Figure 4.1) with truncate module M_2 produces an 8-bit version of the adder similar to the 3-bit adder in Figure 4.11 (the least significant bit is trun-

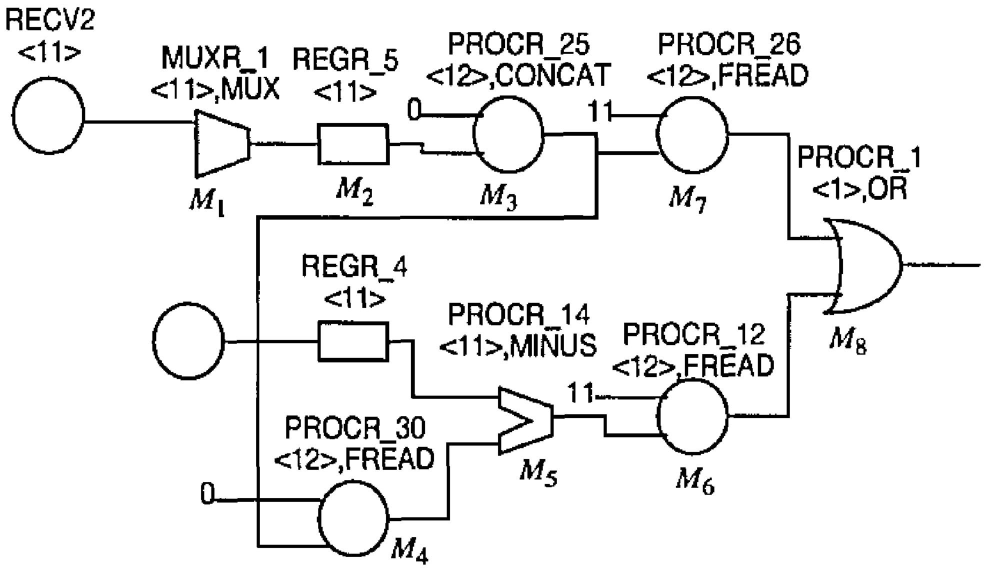


Figure 4.12. Smalldp, a datapath subcircuit of Divfilt.

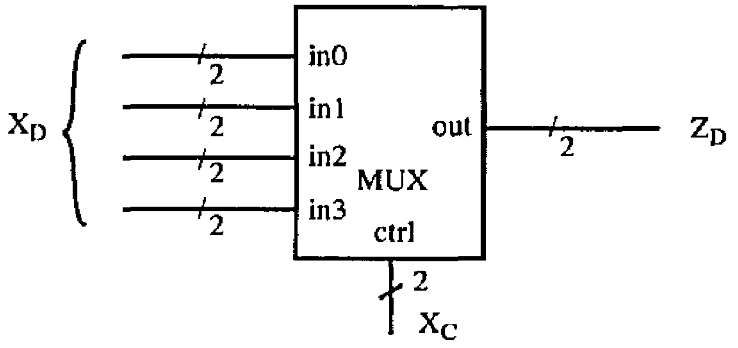


Figure 4.13. Pass-through multiplexer, an example of an optimal k-transparent module.

cated from the sum). It can easily be shown that this combination is also transparent for $S(X_C) = [0, 1]$. Now consider the subcircuit of Divfilt (Figure 3.18) shown in Figure 4.12. As discussed in Section 3.4, the combination of modules M_5 and M_6 is 2^{11}-transparent, therefore, even in cases where a module is naturally k-transparent, the required control sequence length may be impractical and modifications to improve transparency become attractive.

The multiplexer shown in Figure 4.13 is an example of a k-transparent module where k is always minimum, that is, $k = \left\lceil \frac{|X_D|}{|Z_D|} \right\rceil$ (see Theorem 3.10). In a sense, this module is an optimal k-transparent module since its only function is to propagate information on X_D to Z_D, $|Z_D|$ bits at a time. We can use this module as a k-transparent replacement for truncate modules. In normal mode, information is transferred from one input, e.g. in0, to Z_D. In test mode, all inputs are selected in sequence. We refer to a multiplexer used as a k-transparent replacement for truncation as a *pass-through multiplexer.* The pass-through multiplexer can be used in place of truncate module M_6 to improve the transparency of Divfilt. In this case, test response errors can be propagated through module M_6 in 12 steps. This is a considerable reduction in test time compared with the 2^{11} steps required to propagate through the subtracter and truncate module together using the natural k-transparency of the combination. However, the multiplexer requires 36, two-input gates, causing a gate count overhead of about 20% (there are about 148 gates in the original circuit). Thus there is a time-space trade-off associated with k-transparent design.

Pass-through multiplexers provide an effective method for dealing with non-transparency caused by truncate modules. However, we are also interested in systematic methods for modifying arbitrary combinational modules. Recall from the discussion in Chapter III that k-transparency is determined for a module M by intersecting k transmission functions for M, each associated with a different control value. If the resulting transmission function is congruent to zero, then M is k-transparent. For example, for the adder in Figure 4.11,

$$T([0]) = \{(0,1;0),(2,3;1),(4,5;2),(6,7;3)\}$$

$$T([1]) = \{(0;0),(1,2;1),(3,4;2),(5,6;3),(7;4)\}$$

$$\begin{aligned} T([0,1]) = T([0]) \cap T([1]) = \{ & (0;[0,0]),(1;[0,1]),(2;[1,1]), \\ & (3;[1,2]),(4;[2,2]),(5;[2,3]),(6;[3,3]),(7;[3,4])\} \cong 0 \end{aligned}$$

The general procedure for synthesizing k-transparent replacement modules consists of the following steps

1. Identify data bus ports X_D and Z_D from the propagation paths of the circuit containing the original module
2. Identify k transmission functions whose intersection is congruent to zero
3. Identify a control bus port X_C in $X - X_D$, and add one bit to act as a switch to change from normal mode to test mode
4. Assign control values to each transmission function

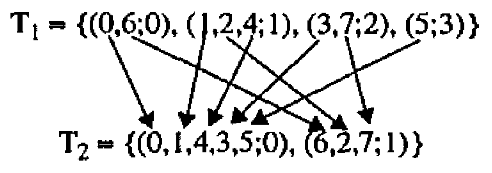

T1 = {(0,6;0), (1,2,4;1), (3,7;2), (5;3)}
T2 = {(0,1,4,3,5;0), (6,2,7;1)}

Figure 4.14. Construction of orthogonal transmission function T_2.

5. Synthesize the module

We want to minimize k, so each intersection should produce the maximum number of new blocks, as discussed in the proof of Theorem 3.10. This requires a special set of transmission functions.

Definition 4.1: Let T_1 and T_2 be transmission functions for a module M with m and n blocks, respectively. Then T_1 and T_2 are *orthogonal* if there is no transmission function T_3 for M with m blocks such that $T_3 \cap T_2 < T_1 \cap T_2$, and no transmission function T_4 for M with n blocks such that $T_1 \cap T_4 < T_1 \cap T_2$.

Note that if $T_1 \cap T_2 \equiv 0$, then T_1 and T_2 are also complements, but that they can be orthogonal without being complements.

The transmission function associated with each control value for a pass-through multiplexer is orthogonal to the transmission function for the other control values. In addition, for any transmission function T_1, we can easily construct an orthogonal transmission function T_2 with n blocks by distributing the contents of each block of T_1 among the n blocks of T_2. For example, let $T_1 = \{(0,6;0), (1,2,4;1), (3,7;2), (5;3)\}$. Then we can create a transmission function $T_2 = \{(0,1,4,3,5;0), (6,2,7;1)\}$ with two blocks as shown in Figure 4.14. In this case, $T_1 \cap T_2 = \{(0;(0,0)), (1,4;(1,0)), (2;(1,1)), (3;(2,0)), (7;(2,1)), (5;(3,0))\}$.

We intersect orthogonal transmission functions to design k-transparent modules with minimum control sequence length k. The next theorem formalizes the requirement, and follows directly from the definition of orthogonality and the proof of Theorem 3.10.

Theorem 4.1: *Let $T_0 > 0$ be a transmission function for a module M with input data bus X_D, input control bus X_C, and output data bus Z_D. Let $T_1, T_2, \ldots, T_{k-1}$ be a set of transmission functions such that $T_0 \cap T_1 \cap \ldots \cap T_{k-1} \equiv 0$. Then k is minimum if and only if for all $i \geq 1$, T_i has $2^{|Z_D|}$ blocks and is orthogonal to $T_0 \cap \ldots \cap T_{i-1}$.*

The sequence length k in Theorem 4.1 depends on the initial transmission function T_0. If T_0 is a maximum-transparency transmission function with $2^{|Z_D|}$ blocks, then k is optimal. Otherwise, k may be greater than $\left\lceil \frac{|X_D|}{|Z_D|} \right\rceil$, but will still be minimum for the given T_0.

We have developed two specific methods for obtaining a set of k orthogonal transmission functions to make a module k-transparent: the *orthogonal transmission function* (OTF) method and the *embedded multiplexer* (EM) method. Using the OTF method, we first obtain an initial transmission function T_I, for a module M by identifying a "natural" transmission function for M based on its normal function. Alternatively, we can select an arbitrary maximum-transparency transmission function with $2^{|Z_D|}$ blocks. Orthogonal transmission functions are then constructed to intersect with T_0 as discussed above.

Figure 4.15 demonstrates how a module with a k-transparent test mode can be specified for synthesis using the OTF method In this figure, we show a block symbol for a random logic module dec1.1 together with a definition for the module written in the Verilog hardware description language. The module dec1.1 has a 6-bit data bus input ha (X_D), a test mode input tm (X_C), and a 3-bit data bus output ta (Z_D). The minimum control sequence length that can make dec1.1 transparent is 6/3 = 2.

Module definitions of the kind in Figure 4.15 serve as input to a number of logic synthesis programs that automatically construct gate-level models and map the gates and interconnections to a physical implementation. The input and output ports are specified at the top of the HDL description (lines 3–5). The **always** statement beginning on line 8 specifies that this is a definition of a combinational module. When the test mode input is 0, dec1.1 executes its normal mode function. This function is specified by a transmission function in the form of a case statement in the Verilog description (lines 10–19). Each line in the case statement specifies a set of inputs at ha that produce a specific output at **ta**, that is, a block of the transmission function. For instance, the first block specifies that **ha** inputs 1, 4, 10, 40, 42, 58, 59, and 61 produce the ta output 0. When the test mode input **tm** is 1, the module executes an orthogonal transmission function. Note that every element of a single block in the transmission function for the normal mode is in a different block in the orthogonal transmission function. Only two transmission functions are required for transparency, thus the lower bound is met for dec1.1.

```
module dec1.1 (tm, ha, ta);

input [6:0] ha;     \
input tm;            } I/O ports
output [2:0] ta;    /
reg [2:0] ta;

always begin @(ha)
    if (tm == 0)
        case (ha)
            1,4,10,40,42,58,59,61: ta = 0;  <-- block
            5,6,9,25,28,32,53,55: ta = 1;
            2,11,19,31,38,57,39,43: ta = 2;          Transmission function
            8,15,30,37,49,50,51,56: ta = 3;          representing normal mode
            13,29,33,34,48,44,45,47: ta = 4;
            3,12,14,16,20,21,22,35: ta = 5;
            17,18,23,24,26,41,63,52: ta = 6;
            7,27,36,46,60,62,54,0: ta = 7;
    endcase
else if (tm == 1)
    case (ha)
            1,5,2,8,13,3,17,7: ta = 0;
            4,6,11,15,29,12,18,27: ta = 1;
            10,9,19,30,33,14,23,36: ta = 2;          Orthogonal transmission
            40,25,31,37,34,16,24,46: ta = 3;         function for test mode
            42,28,38,49,48,20,26,60: ta = 4;
            58,32,57,50,44,21,41,62: ta = 5;
            59,53,39,51,45,22,63,54: ta = 6;
            61,55,43,56,47,35,52,0: ta = 7;
    endcase

end

endmodule
```

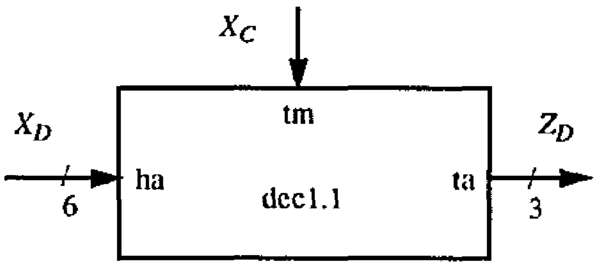


Figure 4.15. Orthogonal transmission function (OTF) method applied to random logic block.

The second, EM, method combines the definition of the normal function for a module M with the definition of a pass-through multiplexer. This method is illustrated in Figure 4.16, where we show the definition in Verilog and block symbol for the module dec1.2. The normal mode for this module has the same function as dec1.1 shown in Figure 4.15, but the k-transparent test mode is implemented using the EM method. This method requires an extra bit in X_C for dec1.2. The first bit is the test mode switch **tm** as in dec1.1. If **tm** is 0, then the module implements the normal mode function. If **tm** is 1, then the test mode is selected. If tc is 0, then bits 3 through 5 of ha are

```
1  module dec1.2 (tm, tc, ha, ta);
2
3  input [6:0] ha;      \
4  input tm;             \  I/O ports
5  input tc;             /
6  output [2:0] ta;     /
7  reg [2:0] ta;
8
9  always begin @(ha)
10     if (tm == 0)
11         case (ha)
12         1,4,10,40,42,58,59,61: ta = 0;      \
13         5,6,9,25,28,32,53,55: ta = 1;        \
14         2,11,19,31,38,57,39,43: ta = 2;       \  Transmission function
15         8,15,30,37,49,50,51,56: ta = 3;       /  representing normal mode
16         13,29,33,34,48,44,45,47: ta = 4;     /
17         3,12,14,16,20,21,22,35: ta = 5;     /
18         17,18,23,24,26,41,63,52: ta = 6;   /
19         7,27,36,46,60,62,54,0: ta = 7;    /
20     endcase
21 else if (tm == 1)                          \
22     begin                                   \
23         if (tc == 0)                         \  Definition of pass-through
24             ta = ha[5:3];                    /  multiplexer
25         else if (tc == 1)                   /
26             ta = ha[2:0];                  /
27     end                                   /
28
29 end
30
31 endmodule
```

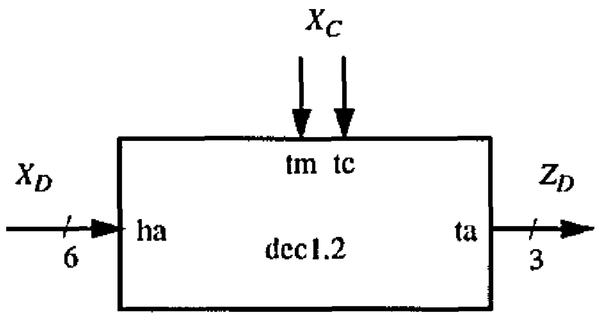


Figure 4.16. Embedded multiplexer (EM) method applied to random logic block.

propagated to the output ta; if tc is 1, then bits 0 through 2 of ha are propagated to the output ta. The **if** statement in lines 23–26 implements the pass-through multiplexer.

Some synthesis results for dec1.1 and dec1.2 are summarized in Table 4.9, together with gate counts for the versions of the 11-bit ripple-carry subtracter used in Divfilt and discussed above. Clearly, modifying a module to include a k-transparent test mode can significantly increase its size. The Finesse automatic logic synthesis program from Cascade Design Automation [27] was used to obtain the results for dec1.1 and dec1.2 using the standard parameters of the program; no effort was made to minimize the overhead—we are only interested in the relative performance of the OTF and EM methods. Note that the function performed in normal mode by dec1.1 and dec1.2

Module	Design strategy	Gate count in basic module	Gate count in modified module
11-bit ripple-carry subtracter	ad hoc	55	91
dec1.1	OTF	52	98
dec1.2	EM	52	68

Table 4.9 Summary of synthesis results for k-transparent modules.

is a maximum transparency transmission function. We take advantage of this fact for dec1.1, so we only add the function required to implement one orthogonal transmission function. Despite this, dec1.1 is considerably larger than dec1.2, which uses a complete pass-through multiplexer for the test mode function. Apparently, for dec1.1, few gates can be reused as part of the test mode function, therefore, the EM method is better since the pass-through multiplexer is very efficient for implementing k-transparency. The OTF method may be better when more gates can be reused to implement the orthogonal function. The two methods may also be combined; this has the potential to improve overall efficiency.

In spite of the fact that k-transparent replacement modules are often much larger than unmodified modules, the overall increase in the size of a circuit that incorporates the modules can be very low. Recall that the Smalldp datapath in Figure 4.12 is transparent from the primary input RECV2 to the inputs of module M_8, the OR gate, but that reconvergence at the OR gate blocks the transparent path. In Figure 4.17, we show a 2-transparent replacement module for the OR gate. This module was designed by the OTF method. Here T2 acts as the test mode switch, and T1 toggles between two orthogonal transmission functions. The k-transparent replacement module increases the gate count of the entire circuit by only 2%.

Another application for orthogonal transmission functions is to construct partially transparent modules. As discussed earlier, it may not be necessary to synthesize a completely transparent module in order to improve transparency enough to allow propagation of specific response functions. The synthesis of partially transparent modules is an important topic for further research.

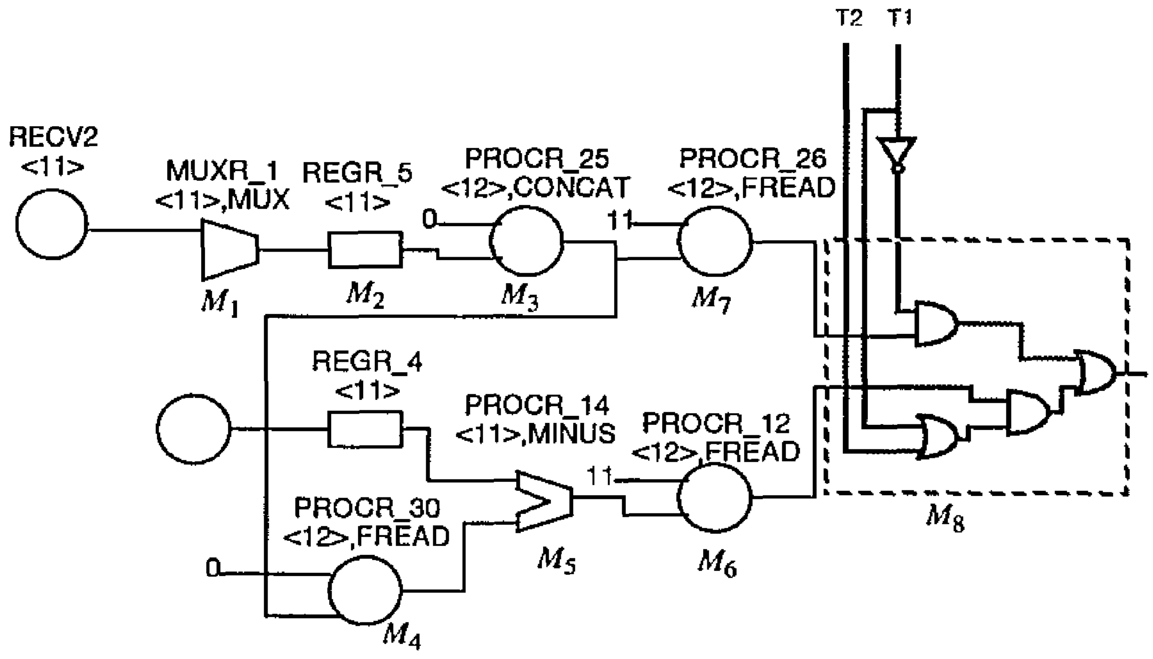


Figure 4.17. Smalldp with k-transparent replacement for module M_8.

4.4. Summary

In this chapter, we analyzed the hierarchical propagation of test package information. A test stimulus sequence T_S is represented as a vector sequence and we want to propagate it symbolically whenever possible (high-level propagation of stimulus signals), as in *PathPlan*. In many cases, symbolic vector sequences must be propagated as expressions representing the composition of module functions in order to justify T_S. To match symbolic expressions propagated to the input of the MUT to components of T_S, the expressions must be simplified to a canonical form. If test package data cannot be propagated symbolically, the vectors that make up the vector sequences can be propagated individually and matched numerically to the vectors in T_S (lower-level propagation of stimulus signals).

Expressions propagated in a circuit represent functions. The functions applied to the inputs of the MUT are called stimulus functions. The functions propagated from the output of the MUT to some other point Z in a circuit are called response functions. In some cases, response functions can be analyzed symbolically to determine if all test response errors are propagated. However, when T_R is only propagated along partially transparent paths, we must evaluate the response functions for the set of values, good and faulty, that can appear at the output of the MUT when T_S is successfully instantiated (low-level propagation of errors). The fully evaluated form of these response functions is considered to be a propagation function of the kind we discussed in Chapter III. If a response function P_R is congruent to zero, then all pairs of values that can be generated by faults in the MUT are propagated—including all errors. If $P_R > 0$, then all errors may still be propagated if the good value for timestep t in T_S never occupies the same block in P_R as a faulty value for timestep t. This requirement is called the propagation condition.

Response functions can be represented symbolically and propagated using high-level module functions (high-level propagation of errors). We assigned symbolic type names to groups of response functions. The same names are used for the corresponding set of symbolic signal values that represent the information type propagated by the response functions. For example, the type R represents all transparent and partially transparent response functions, and the corresponding symbolic signal value R represents the set of all possible discrepancies.

Since error propagation is blocked by non-transparent modules in some cases, we are interested in methods for increasing transparency to improve testability. We have shown examples of how this can be done by modifying modules to have a k-transparent test mode. We presented two methods for modifying modules: the OTF method and the EM method. In the OTF method, we identify orthogonal transmission functions whose intersection is congruent to zero and synthesize a module with a test mode that implements the orthogonal transmission functions. The EM method adds the function of a pass-through multiplexer to the module function. The EM method is more efficient unless the orthogonal transmission functions can reuse most of the normal function of the module.

The techniques described in this chapter directly address the required modifications to *PathPlan* discussed in Chapter II. Propagation of symbolic expressions removes the T-mode restriction of *PathPlan*. Analysis of response functions can be used to deal with circuits containing an irregular bus structure. The propagation methods described in this chapter are implemented in our test generation tools described in Chapter V.

CHAPTER V
TEST GENERATION AND SIMULATION

This chapter describes the design of two new test-processing programs *MATSim* and *PathPlan2*, that use precomputed tests for modules. These tools extend *PathPlan* and address its limitations discussed in Chapter II, but are not based directly on it. *MATSim* is a novel simulator that implements the test package propagation methods discussed in Chapter IV. *PathPlan2* is a test generator that uses *MATSim* to propagate signals.

The relationship between *MATSim* and *PathPlan2* is shown in Figure 5.1. *PathPlan2* generates stimulus signals to apply to the primary inputs of a circuit to be tested. *MATSim* simulates a circuit for a given set of inputs, generates reports on the circuit state, and analyzes error propagation. It propagates test package information at multiple levels of abstraction. Vector sequences used by *MATSim* as stimulus signals at the primary inputs, and response vector sequences propagated by *MATSim* to primary outputs form a circuit test package, which can be converted to a standard test program by expanding vector sequences in a straightforward way.

5.1. Multiple-Abstraction Test Package Simulation

The Multiple-Abstraction Test Package Simulator (*MATSim*) is an event-driven simulator similar to a fault simulator. It has two principle characteristics. First, it analyzes primary input sequences to determine whether a given test package $TP_1 = (T_S;T_R)$ is instantiated at the current MUT, and whether all test response errors (T_R, T_{Ri}) are propagated to a primary output, an activity we call test package simulation. Second, it is a "multiple-abstraction" simulator, which means that events from several different abstraction levels can be processed together. For example, at a

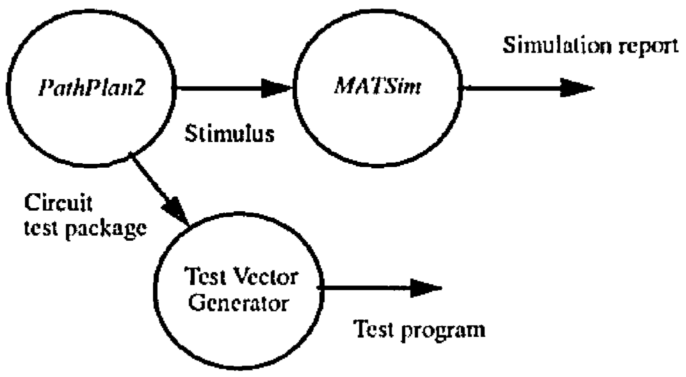


Figure 5.1. Relationship of *MATSim* to *PathPlan*

particular time instance, the simulator may be processing signals consisting of symbolic expressions, integers, and Boolean constants. It uses the hierarchical error propagation analysis method discussed in Chapter IV to determine if all test response errors are successfully propagated to primary outputs and tabulates test coverage, that is, the number of test vectors successfully applied from each T_S, rather than fault coverage. Conventional fault simulators propagate error signals, but link the errors directly to faults rather than to successfully instantiated test packages.

MATSim supports a variety of module primitives, including adders, multiplexers, and typical gate-level primitives such as AND, OR, and NOT. Gate-level modules may be word gates or single-bit gates. Future versions of *MATSim* will support user-defined functional modules; however, all functions must be decomposed into primitive functions, since the hierarchical signal value sets, discussed in Chapter IV, are designed for a fixed set of module functions.

Abstraction and hierarchy have been used in fault simulation previously [77, 81, 89, 88]. The Multiple Abstraction Rule-Based Simulator (MARS) developed by Singh [89] propagates fault-free signals at several levels of abstraction. Module functions are implemented by rules stored in a database. However, only gate-level error signals are processed by MARS. The CHIEFS concurrent, hierarchical fault simulator [77] also uses separate functional and structural models for each module in the structural hierarchy. When the inputs to a module M in CHIEFS are updated, a functional model is used to produce new outputs for M. As in MARS, only gate-level error signals are processed. CHIEFS traverses the hierarchy of structural models within M down to the gate level to update fault lists. Other hierarchical fault simulators such as CHAMP are concerned with coverage of switch-level faults [81]. These fault simulators also do not process error signals more abstract than those due to SSL faults. Finally, Lee has implemented an architectural-level fault simulator ARSIM to speed up simulation for hierarchical circuits with precomputed tests for modules [88]. However, ARSIM does not propagate signals at multiple levels of abstraction. It propagates the set of symbolic error signal values discussed earlier for ARTEST [58], and relies on lower-level fault simulators to analyze fault coverage when these symbolic signals are blocked by non-transparent modules.

5.1.1 Dimensions of Abstraction

As we saw in Chapter I, a large number of test generation tools use hierarchy in an attempt to speed up test generation. Each tool exploits a different view of circuit hierarchy to gain performance advantages. However, hierarchy and abstraction in circuit design, while widely used, are poorly quantified. There is no standard method for using hierarchy in design, and therefore no optimal way to exploit it in test generation. Although most modern CAD tools allow circuits structure to be captured hierarchically, not all tools support the use of behavioral models for higher-level modules. In order to describe the hierarchy of abstractions that *MATSim* can exploit, we have developed a multidimensional view of the abstraction hierarchy relevant to test generation. It is illustrated in Figure 5.2 by a Y-chart similar to that used by Gajski [37] to define relevant dimensions of abstraction in layout synthesis.

The three axes of the Y-chart are labeled information, time, and function. Abstraction in the information dimension ranges from bits to words to multi-word packets (vector sequences). The time dimension ranges from gate delays to clock cycles to instruction cycles. Finally, the third dimension deals with the hierarchy of functions. Consider a ripple-carry adder module. It is composed of full adders, which in turn are composed of logic gates. The adder is itself only one component of the datapath for a computer. The structural composition hierarchy has a corresponding functional composition hierarchy ranging from Boolean functions through arithmetic operations to computer instructions. The functional hierarchy defines the meaning of the signals being processed. For instance, the adder module is designed to perform either signed or unsigned addition on integers modulo *n* for some *n*. A single-bit AND gate performs the AND operation on Boolean signals. The search for effective methods of exploiting the functional hierarchy motivates most

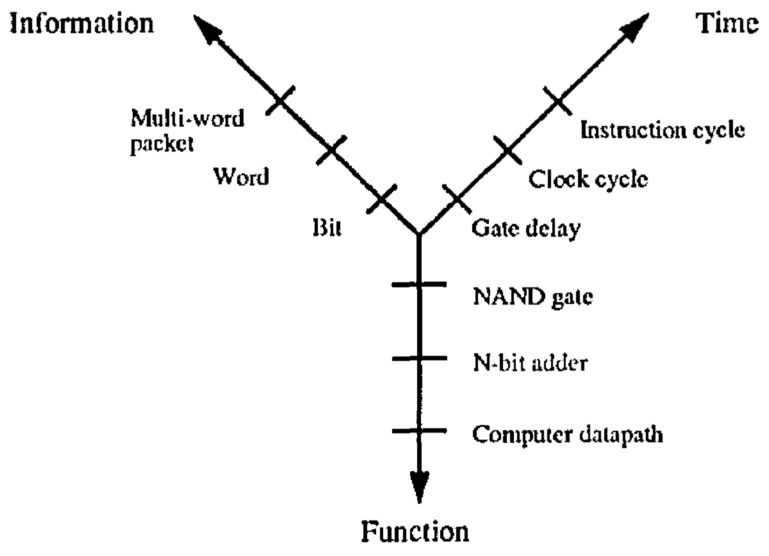


Figure 5.2. Dimensions of hierarchy and abstraction.

Abstraction Dimension / Layer	Information	Time	Function	Example
Symbolic	High	High	High	Symbolic vector sequence expressions, symbolic error signal values
Vector	Medium	Medium	Medium	Response functions
Bit	Low	Low	Low	Signals in classical test generators and fault simulators

Table 5.1 Combinations of abstractions forming layers.

research in hierarchical test generation.

Each axis can be divided into three ranges: low, medium, and high abstraction. We call a point in this three-dimensional "abstraction space" a *layer*. The three primary layers of interest in this thesis are called symbolic, vector, and bit. For the symbolic layer, all dimensions are high abstraction, for the vector layer, all dimensions are medium abstraction, and for the bit layer, all dimensions are low abstraction. These layers are depicted in Table 5.1. Intermediate layers can also be defined that mix the abstractions in other ways, for instance single-bit signals (low-level information abstraction) can be propagated through large modules with very abstract models (high-level function abstraction).

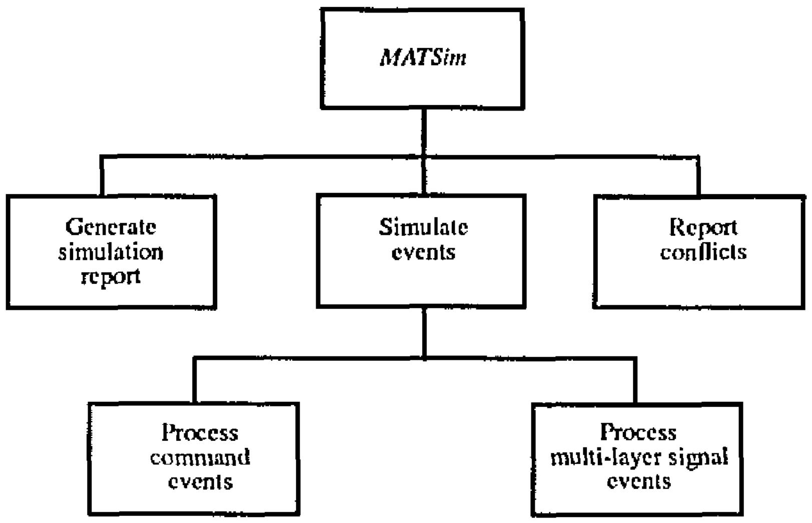


Figure 5.3. Structure of *MATSim*

MATSim assigns every signal to a layer. Two layers are currently implemented, symbolic and vector, which implement the test package propagation methods discussed in Chapter IV. The hierarchical error propagation analysis method discussed in Chapter IV is implemented using two layers. Fault-free test package data, including T_S, T_R, and various control values for sensitizing modules are propagated as symbolic expressions of vector sequences; they are the *symbolic-layer signals*. Symbolic signal values from the set R_7 = {X, S, R, C, 0_n, 1_n, $\hat{S}$ } are assumed by symbolic-layer signals. These signals are propagated along transparent single paths in the circuit model. If error propagation is blocked at the symbolic layer by non-transparent modules, all of the individual vectors of the MUT's response set are evaluated at the vector layer and propagated numerically. We examine this two-layer hierarchical technique for error propagation further below.

5.1.2 Structure of *MATSim*

MATSim has three functions as shown in Figure 5.3: to simulate events, to report simulation results, and to detect and respond to conflicts. *MATSim* simulates events representing changes in signals at multiple levels of abstraction as discussed above. Simulator commands to print the state of the circuit at a particular timestep or to stop at a particular timestep are also represented as events. Since *MATSim* implements signal propagation for the test generator *PathPlan2*, it must detect and respond to conflict. In test generation, once a signal has been assigned to a bus Z at time

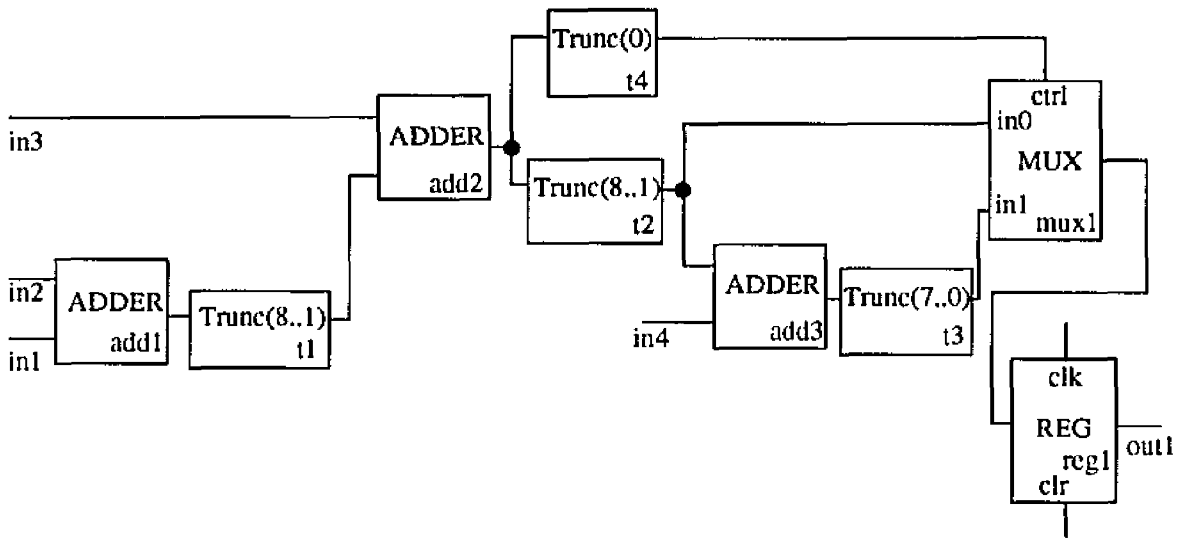


Figure 5.4. The Fltrdp datapath circuit

t, no other signal value can be assigned to Z at time *t* on the same layer. Any other signal propagated to Z denotes a conflict which, when detected, causes the test generator to backtrack.

Verilog Parser and Circuit Data Structure. *MATSim* reads circuit descriptions written in netlist form using a stylized subset of the Verilog simulation language [91]. The netlist may be hierarchical as discussed in Chapter I. An important feature of the netlists is that they are executable by a standard Verilog digital simulator, so the results of *MATSim* can be correlated with results from other simulators.

The Fltrdp circuit is shown again in Figure 5.4, where the modules have been assigned symbolic names for use in a netlist. The Verilog netlist for Fltrdp is shown in Figure 5.5. The module types addmod, fread1, etc., refer to module definitions. *MATSim* uses modified version of a circuit data structure (Netstruct) developed at Carnegie Mellon University for representing circuits designed by high-level synthesis. The routines that access Netstruct, and the Verilog parser that produces it are part of the AutoCircuit synthesis tool [32].

The Verilog language was designed to describe both circuit structure and function, as well as the simulation environment, including the stimulus vectors. However, the semantics of Verilog do not support assignment of symbolic expressions. To support such expressions, *MATSim* reads a separate file containing a list of stimulus events with signals represented as symbolic expressions. There is also a separate file for test package descriptions. The test package descriptions define the

```
module main(Clock,reset,in1,in2,in3,in4,fr1,fr2,fr3,ctrl,out1);
/// class: Structure
        /*{portclass: CLOCK}*/  input [0:0] Clock;
        /*{portclass: ASYNC_CTRL}*/    input [0:0] reset;
        /*{portclass: DATA}*/   input [7:0] in1;
        /*{portclass: DATA}*/   input [7:0] in2;
        /*{portclass: DATA}*/   input [7:0] in3;
        /*{portclass: DATA}*/   input [7:0] in4;
        /*{portclass: DATA}*/   input [7:0] fr1;
        /*{portclass: DATA}*/   input [7:0] fr2;
        /*{portclass: DATA}*/   input [7:0] fr3;
        /*{portclass: DATA}*/   input [2:0] ctrl;
        /*{portclass: DATA}*/   output [7:0] out1;
        /*{portclass: DATA}*/   wire [8:0] add1out;
        /*{portclass: DATA}*/   wire [8:0] add2out;
        /*{portclass: DATA}*/   wire [8:0] add3out;
        /*{portclass: DATA}*/   wire [7:0] t1out;
        /*{portclass: DATA}*/   wire [7:0] t2out;
        /*{portclass: DATA}*/   wire [7:0] t3out;
        /*{portclass: DATA}*/   wire [0:0] t4out;
        /*{portclass: DATA}*/   wire [7:0] mux1out;

        addmod add1( in1, in2, add1out, ctrl );
        fread1 t1( add1out, fr2, t1out, ctrl );
        addmod add2( in3, t1out, add2out, ctrl );
        fread1 t2( add2out, fr2, t2out, ctrl );
        addmod add3( t2out, in4, add3out, ctrl );
        fread1 t3( add3out, fr1, t3out, ctrl );
        fread2 t4( add2out, fr3, t4out, ctrl );
        muxmod mux1( t2out, t3out, mux1out, t4out );
        regr_1 reg1( mux1out, out1, ctrl, Clock, reset );
endmodule
```

Figure 5.5. Verilog netlist for Fltrdp (Figure 5.4).

symbols used in the stimulus file, as well as provide the underlying test vector data.

Discrete-Event Simulator. *MATSim* is a discrete-event simulator and its overall structure is shown in Figure 5.6. An *event* is an action which is scheduled to occur at a specific time instance *t*. The action may be a command to the simulator, for instance to print the current state to a file. However, the most common action is to assign a new signal *v* to a bus *Z* in the circuit at time *t*, so the most common event may be expressed as a triple (*v*,*Z*,*t*). The bus *Z* is a primary input or output, or an input or output of some module. *MATSim* labels events according to their source. Initial events are produced by reading the file of stimulus events written by the user, or created by *PathPlan2*. Signal assignments made as a result of initial events cannot be changed in the same clock cycle. All other events are generated during simulation, and corresponding assignments can be changed an arbitrary number of times unless they are marked as "frozen." All signal values are marked frozen at the end of a simulation pass (line 9) in Figure 5.6. This is the mechanism used by

```
MATSim
{
        current_time = init_time;
        while (TRUE) {
                result = process_events_at_current_time( current_time );
                if (result == assignment_conflict)
                        return( assignment_conflict );
                if (result == quit || num_current_events == 0) {
                        freeze values;
                        return( quit );
                }
                current_time = current_time + timestep;
        }
}
```

Figure 5.6. Main routine for *MATSim*.

PathPlan2/*MATSim* to detect conflicts.

Events are scheduled by entering them into a timing wheel [2]. An event $E = (v, Z, t)$ inherits the layer assignment of v. In *MATSim*, the layer associated with E determines the update routine that handles it. The routine *process_events_at_current_time* retrieves all of the events scheduled for the current time and initiates the appropriate action for each, depending on event type and layer. If the event is not a command, *process_events_at_current_time* calls the layer-specific update routine. An update routine implements an event by assigning v to Z, evaluating module functions whose inputs are connected to Z, and scheduling new events produced by the evaluation of module functions; it updates the state of the circuit.

A skeleton update routine is shown in Figure 5.7. Specialized versions of this routine exist for each layer. The update method is based on the one-pass evaluation strategy with suppression of multiple signal changes scheduled for the same time [2]. As noted above, the routine implements the value changes specified by an event $E = (v, Z, t)$. First, it assigns the value v (denoted event_value in Figure 5.7) to the bus Z (denoted event_bus in Figure 5.7). If another value has previously been assigned to the port and frozen, then there is a conflict. Let $p_1, p_2, \ldots, p_k$ be the set of module inputs (input ports) that are connected to Z. If M is a module with input port p_i $(1 \le i \le k)$, then M's module function is evaluated at the current layer. If the value of a module function changes as a result of new inputs, then this change becomes a new event. Since real circuits compute module functions after a finite delay δ, the new event is scheduled for the future time instance

```
1   update( input event, time )
2   {
3       assign event_value to event_port;
4       if (there is a conflict) return( conflict );
5       for (every port fport on fanout list of event_port) {
6           assign event_value to fport;
7           if (there is a conflict) return( conflict );
8           fmod = module with port fport;
9           update_list = evaluate( fmod, event_layer );
10          for (every event update_event in update_list) {
11              process update event according to layer;
12              if (update_event_value != update_event_lsv) {
13                  update_event_time = update_event_delay + time;
14                  if (update_event_time == update_event_lst)
15                      cancel_scheduled_event( update_event );
16                  schedule( update_event );
17                  update_event_lsv = update_event_value;
18                  update_event_lst = update_event_time;
19              }
20          }
21      }
22      return( no_conflict );
23  }
```

Figure 5.7. Update routine.

$t + \delta$. Each module function evaluation can produce multiple events since the module can have multiple outputs and the MUT can produce error propagation signals (discussed below) on two layers in addition to T_R, the fault-free response of M to T_S. Each layer-specific update routine handles the events returned from the module evaluation routine differently (line 11 in Figure 5.7). However, each update routine compares new signals to be scheduled to previous signals on the same layer scheduled for the same bus. The new signal is only scheduled if it differs from the old. If two events on the same layer are scheduled for the same port at the same time, then the most recently scheduled event is kept and the other canceled.

MATSim currently supports a small set of module primitives including: adders, multiplexers, inverters, OR gates, AND gates, subtracters, bus-truncate modules, and bus-concatenate modules. This set can easily be expanded to include other primitive modules such as RAMs, ROMs, decoders, encoders, multipliers, and XOR gates. Each primitive module has an associated evaluation routine for the symbolic and vector layers.

5.1.3 Simulation of Fault-Free Signals

In this subsection, we describe how *MATSim* processes fault-free signal components. All fault-free signals are symbolic expressions of vector sequences, and so are processed at the symbolic layer. Because vector sequences are capable of representing signals from bits to multi-word packets, expressions based on vector sequences can represent the full information hierarchy. As pointed out in Chapter IV, it is sometimes necessary to propagate the individual vectors of a vector sequence numerically, for instance when symbolic expressions cannot be simplified to match T_S. In *MATSim*, T_S can be propagated to the MUT using either symbolic references to vector sequences as in *PathPlan*, or as individual vectors as in ARTEST. The individual vectors of T_S and T_R are also handled at the symbolic layer in the current version of *MATSim*. We discussed symbolic expressions and how they are simplified in Chapter IV. Here we give some additional implementation details for *MATSim*.

Symbolic expressions can be considered as trees, where the root and interior nodes are operators and the leaves are vector sequences. In *MATSim*, vector sequences are stored once, and the leaves of expression trees contain pointers to the vector sequence data. Symbolic expressions are frequently divided by a constant, as a result of shifting by truncation, or by propagation through a division circuit. As discussed in Chapter IV, division by a constant is represented by multiplication by a rational coefficient. For simplicity therefore, all constants in *MATSim* are represented by rational numbers and implemented by integer pairs numerator/denominator. For example, the constant 0 is represented by 0/1, and 1 is represented by 1/1. Symbolic expressions are manipulated by well-known computer algebra techniques [54], so that they are always maintained in exact and reduced form. Subtraction is implemented in *MATSim* as addition of a negative number. The sign of a vector sequence is associated with the numerator of its rational coefficient. Therefore, $A - B$ is represented as $A + (-1/1)B$.

An example of the propagation of expressions through Fltrdp is shown in Figure 5.8, which repeats Figure 4.1. The symbolic test program shown in Figure 5.8 successfully instantiates test package (A_1, A_2; $A_1 + A_2$) at the adder module add3. The output from *MATSim* when simulating this example is shown in Figure 5.9. The names of the buses are printed on the left hand side of each line, followed by the symbolic expression assigned to each bus as a fault-free signal value.

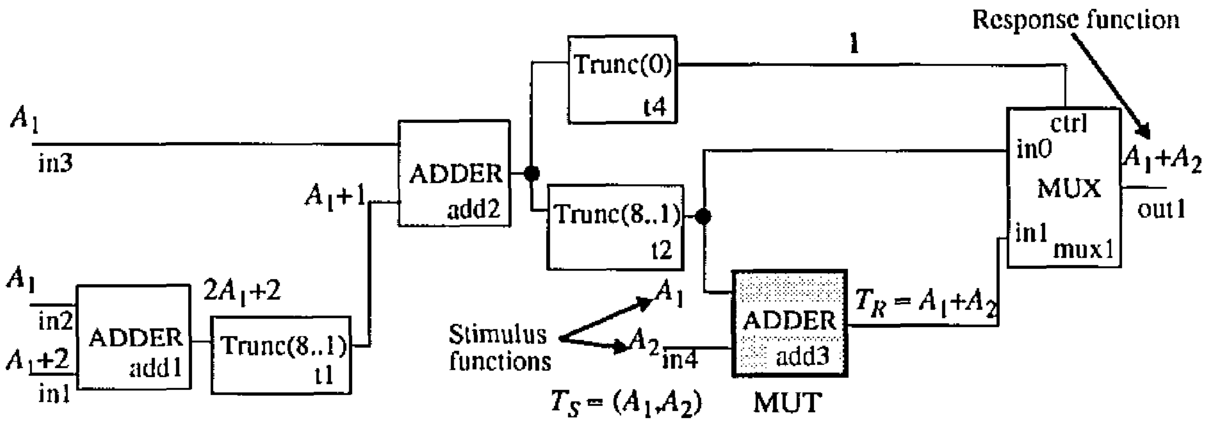


Figure 5.8. Propagating a test package (T_S;T_R) through the Fltrdp datapath circuit

```
Circuit State at time 300

Module: main
    main.in1    ( + ( A1 ) ( 1/1 ) ), error component = S
    main.in2    ( + ( A1 ) ( 1/1 ) ), error component = S
    main.in3    ( A1 ), error component = S
    main.in4    ( A2 ), error component = S
    main.fr1    ( 0/1 ), error component = 0
    main.fr2    ( 1/1 ), error component = S
    main.fr3    ( 0/1 ), error component = 0
    main.ctrl   ( 0/1 ), error component = 0
    main.out1   ( + ( A1 ) ( A2 ) ), error component = R
Module: add1
    add1.in1    ( + ( A1 ) ( 1/1 ) ), error component = S
    add1.in2    ( + ( A1 ) ( 1/1 ) ), error component = S
    add1.out1   ( + ( * ( 2/1 ) ( A1 ) ) ( 2/1 ) ), error component = S
Module: t1
    t1.in1      ( + ( * ( 2/1 ) ( A1 ) ) ( 2/1 ) ), error component = S
    t1.in2      ( 1/1 ), error component = S
    t1.out1     ( + ( A1 ) ( 1/1 ) ), error component = S
Module: add2
    add2.in1    ( A1 ), error component = S
    add2.in2    ( + ( A1 ) ( 1/1 ) ), error component = S
    add2.out1   ( + ( * ( 2/1 ) ( A1 ) ) ( 1/1 ) ), error component = S
Module: t2
    t2.in1      ( + ( * ( 2/1 ) ( A1 ) ) ( 1/1 ) ), error component = S
    t2.in2      ( 1/1 ), error component = S
    t2.out1     ( A1 ), error component = S
Module: add3
    add3.in1    ( A1 ), error component = S Objective: ( A1 )
    add3.in2    ( A2 ), error component = S Objective: ( A2 )
    add3.out1   ( + ( A1 ) ( A2 ) ), error component = R
Module: t4
    t4.in1      ( + ( * ( 2/1 ) ( A1 ) ) ( 1/1 ) ), error component = S
    t4.in2      ( 0/1 ), error component = 0
    t4.out1     ( 1/1 ), error component = S
Module: mux1
    mux1.in1    ( A1 ), error component = S
    mux1.in2    ( + ( A1 ) ( A2 ) ), error component = R
    mux1.out1   ( + ( A1 ) ( A2 ) ), error component = R
    mux1.ctrl   ( 1/1 ), error component = S
```

Figure 5.9. Example of *MATSim* output of circuit state information; add3 is the MUT

Expressions are printed by *MATSim* in prefix notation, except for rational constants, which are written in infix notation as discussed above. For example, $A_1 + 1$ on the line for bus main.in2 is written as $(+\ (A_1)\ (1/1))$.

In Chapter IV we showed how symbolic expressions are simplified to a canonical form to match them to T_S. *MATSim* simplifies expressions after each module function is evaluated. Since symbolic expressions can contain both Boolean and arithmetic operations, there are two algorithms for simplifying them. The set of arithmetic simplification steps is:

A1. Distribute multiplication over addition, e.g., replace $A(B+C)$ by $AB+AC$

A2. Apply the associative law to remove parentheses and reduce the number of tree levels

A3. Combine rational expressions forming coefficients or constants into a single rational number

A4. Add like terms.

A5. Eliminate unit coefficients.

A6. Eliminate terms with zero coefficients

As discussed in Chapter IV, the key simplification steps for arithmetic expressions are A1, A4, and A6; the remaining simplifications steps are intermediate steps required in *MATSim*. After expanding (step A1), for instance, expressions frequently have a tree form where the child Y of a node X is the same as X. For example, let $A+1$ and $B+1$ be inputs to an adder. The result at the output of the adder is $(A+1)+(B+2)$. Each parenthesis implies a node in the expression tree. We reduce the size of the tree by applying the associative rule so that $(A+1)+(B+2)$ becomes $A+B+3$.

The only Boolean simplification step currently implemented in *MATSim* is involution. Even-length chains of the Boolean NOT operation are eliminated, that is, $NOT(NOT(X))$ is replaced by X. Both the Boolean and arithmetic simplification algorithms are applied recursively. Each searches for subexpressions of the proper type to simplify. Arithmetic simplification steps treat Boolean operations as symbols, and vice versa. For example, the expression $(1/2)(((A+0) \wedge B)+C)$ is simplified to $(1/2)(A \wedge B)+(1/2)C$.

After simplifying an expression using the general techniques A1–A6, *MATSim* performs some ad hoc context-specific checks to see if the expression can be simplified further. For instance, if an even (odd) expression is identified at a 1-bit port, it is simplified to 0 (1). An example of this is shown in Figure 5.9. The input data to module t4 (t4.in1) is odd because (+ (* (2/1) (A1)) (1/1)) denotes $2 \times A1 + 1$, an odd number. Since t4 truncates all but the least significant bit, the result is 1 at the output of module t4 (t4.out).

To reduce memory overhead, expressions are reused as much as possible. When updating a module, new expressions are constructed by creating a root node (an operator) and adding pointers to subexpressions representing the arguments. During simplification, various parts of an expression E are copied and/or rearranged, so some parts of E are pointers to copies generated elsewhere, and some are unique to E. It is obvious that simplification rearranges expressions, but it frequently also results in the generation of multiple copies of the expressions at the inputs. To see why, consider the expression $\frac{1}{2}(A+B)$, created when $A+B$ is propagated through a truncate module which deletes the least significant bit of a bus carrying $A+B$. As a prerequisite to further simplification, this expression must be expanded. However, if $A+B$ is modified, ports to which it is assigned will be incorrect. Therefore, a new expression must be generated by copying $A+B$ and using it as the basis for modification. Uncontrolled processing of expressions uses memory rapidly. Therefore, *MATSim* implements extensive memory management routines to control the growth of expression storage.

5.1.4 Error Propagation

In this section, we discuss how *MATSim* propagates error information hierarchically at two levels of abstraction, symbolic and vector. Response functions can be represented as symbolic signal values (symbolic-layer signals) to analyze error propagation in circuits with transparent propagation paths. On the other hand, in order to analyze error propagation in irregular circuits, we must evaluate the response function from the output of the MUT to primary outputs for all possible responses from the MUT (the response set). *MATSim* propagates vector-layer signals for this analysis.

MATSim uses the set of symbolic signal values $R_7 = \{X, S, R, C, 0_n, 1_n, \hat{S}\}$ for symbolic-layer error signal propagation. These are implemented exactly as discussed in Chapter IV. An example of their use appears in Figure 5.9. Error propagation signals are labeled "error component" in this listing. Every fault-free symbolic expression has a corresponding symbolic-layer "error component," even signals propagated from primary inputs as stimulus or control signals. These signals, main.in1, for example, have error component S or 0. The value R appears at the output of the MUT when T_S is successfully instantiated at the inputs tot the MUT. In Figure 5.9, module add3 is the MUT. The test package objectives: A1 assigned to add3.in1, and A2 assigned to add3.in2 are satisfied by the assignment to primary inputs. Thus, the signal value at the output of add3 is R, which is subsequently propagated to a primary output.

The processing of error propagation signals at the symbolic layer is the default in *MATSim*. A global switch determines when vector-layer processing should be used. This switch is set by the test generator *PathPlan2* when propagation is blocked at the symbolic layer, that is, when R cannot be propagated along any path in the circuit from the current state. It can also be set manually when *MATSim* is used as a stand-alone simulator.

Error propagation at the vector-layer is analyzed by evaluating the response function for the set of vectors in the MUT's response set at the output of every module on a propagation path from the output of the MUT to a primary output. This is implemented in *MATSim* by propagating response functions as complex signals representing propagation functions $P = \{(\alpha_i;\beta_i)\}$, $1 \le i \le m$, from the output of the MUT when T_S is successfully instantiated and vector-layer processing is enabled. A special data structure is used to represent the response functions. Let Ω_{Rj}, $1 \le j \le n$, be the (first-order) response set for *j*th test vector in test stimulus sequence T_S, and let P_{Rj} be the corresponding (first-order) response function at some point *Z* on the propagation path. Each Ω_{Rj} has one correct value and one or more faulty values. Recall from the discussion in Chapter IV that to determine if all errors are propagated, we must analyze each P_{Rj} to determine whether the correct value is in a different block from any faulty value, that is whether P_{Rj} satisfies the propagation condition.

MATSim combines all the first order response functions P_{Rj}, $1 \le j \le n$, into one order-*n* response function P_R. Each block of the composite response function P_R has one current value β_i, but several response subsets α_{ij}, $1 \le j \le n$, one for each first-order response function on Ω_{Rj} containing a block whose current value is β_i. The index *j* is referred to as the timestep since it indicates which vector of T_S produces Ω_{Rj}. We also separate correct responses from faulty responses for each timestep to facilitate analysis of the propagation condition, as we will show below.

The data structure for a single block is depicted in Figure 5.10. The current value β_i of the block appears in the box at the top. To the left of each current value is a list of timesteps with cor-

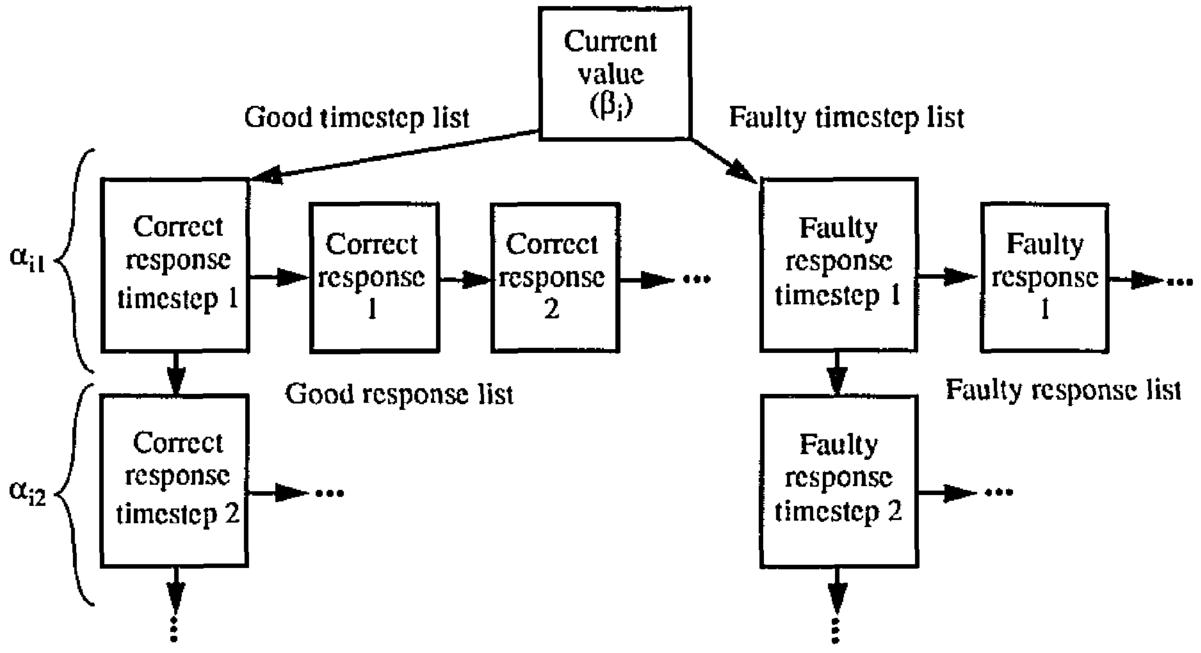


Figure 5.10. Data structure for one block of a response function.

rect MUT responses whose current value is β_i, and to the right is a similar list of timesteps with faulty MUT responses whose current value is β_i. Thus, each α_{ij} is split between the good timestep list and the faulty timestep list. Each timestep in the good and faulty lists contains a set of responses associated with the timestep. A complete response function is represented by a linked list of these blocks.

In order to demonstrate the use of this data structure, we will examine a response function for a two-input, 4-bit multiplexer, which we also used as an example in Chapter IV (Figure 5.11). The set of first-order response sets for this module is as follows, with the correct response in each first-order response set underlined.

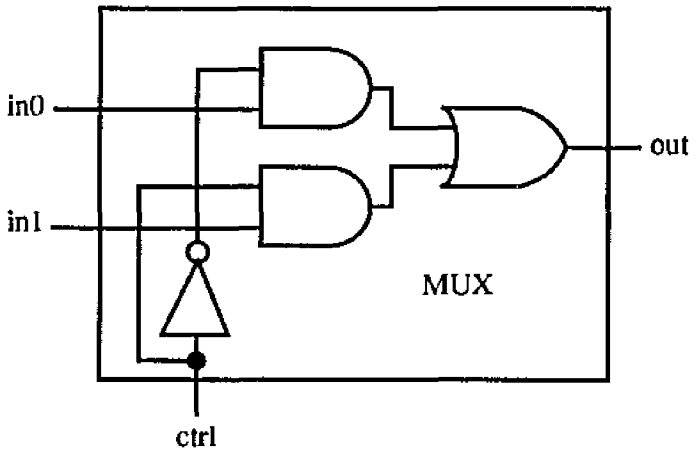
$\Omega_{R1} = \{0,7,11,13,14,\underline{15}\}$

$\Omega_{R2} = \{\underline{0},1,2,4,8,15\}$

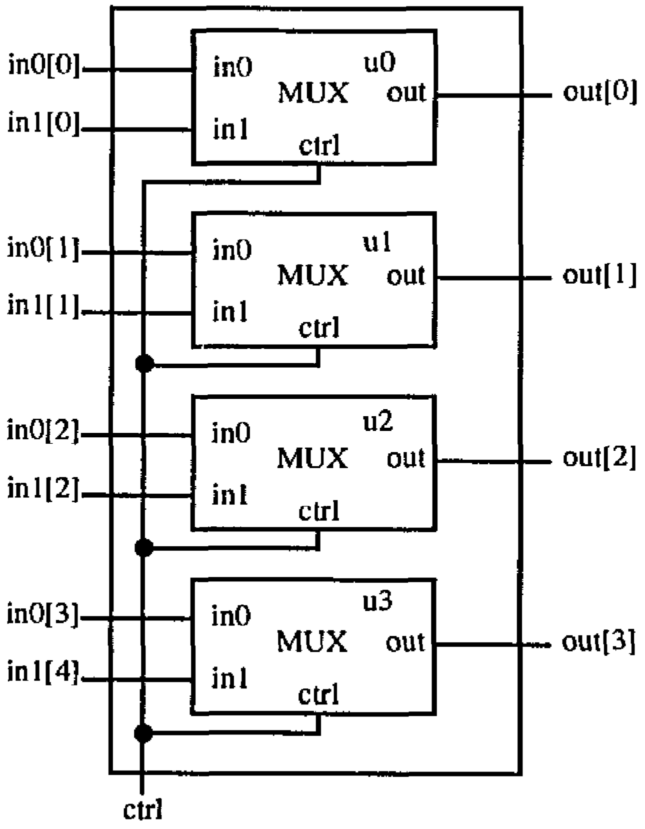
$\Omega_{R3} = \Omega_{R2}$

$\Omega_{R4} = \Omega_{R1}$

These lead to a combined order-4 response set of $\Omega_R = \{0,1,2,4,7,8,11,13,14,15\}$. The response function on Ω_R at the output of the MUT is $P_R = \{(0;0), (1;1), (2;2), (4;4), (7;7),(8;8), (11;11), (13;13), (14;14), (15;15)\}$. The corresponding data structure for P_R, which is to be assigned to the output of the MUT, is shown in Figure 5.12. Each block of P_R is shown in a shorthand notation



(a) Basic cell: two-input, single-bit multiplexer



(b) Four basic cells combined to form a two-input, 4-bit multiplexer

Figure 5.11. Implementation of a two-input, 4-bit multiplexer.

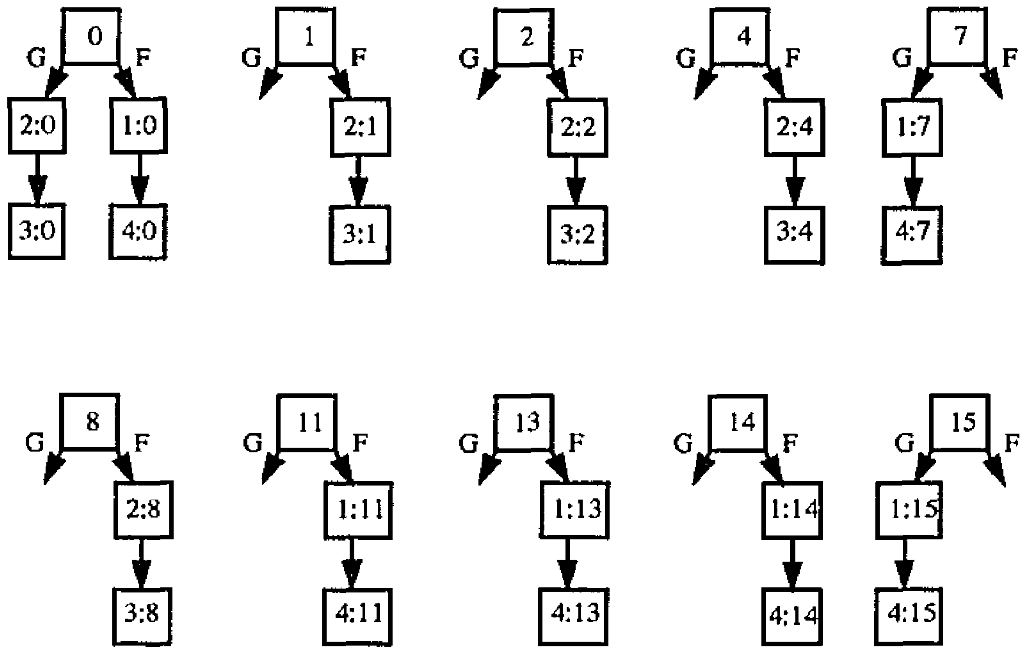


Figure 5.12. Response function data structure at output of MUT.

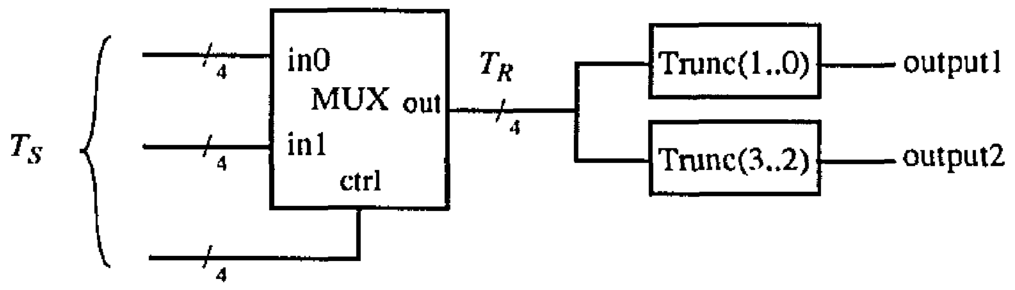


Figure 5.13. Example circuit for propagation of response functions, a multiplexer connected to two complementary fanout modules.

that corresponds to Figure 5.10. The current value β_i for each block i is in a box at the top of Figure 5.12. To the left of each current value is the list of timesteps with correct responses (marked G for good), to the right is the list of timesteps with faulty responses (marked F for faulty). Each box in a timestep list has the form $j{:}\alpha_{ij}$, where j is the timestep and α_{ij} is the list of MUT responses at j.

Now consider the propagation of P_R through a circuit. Let the output of the two-input, four-bit multiplexer be connected to two truncate modules in parallel, as shown in Figure 5.13.

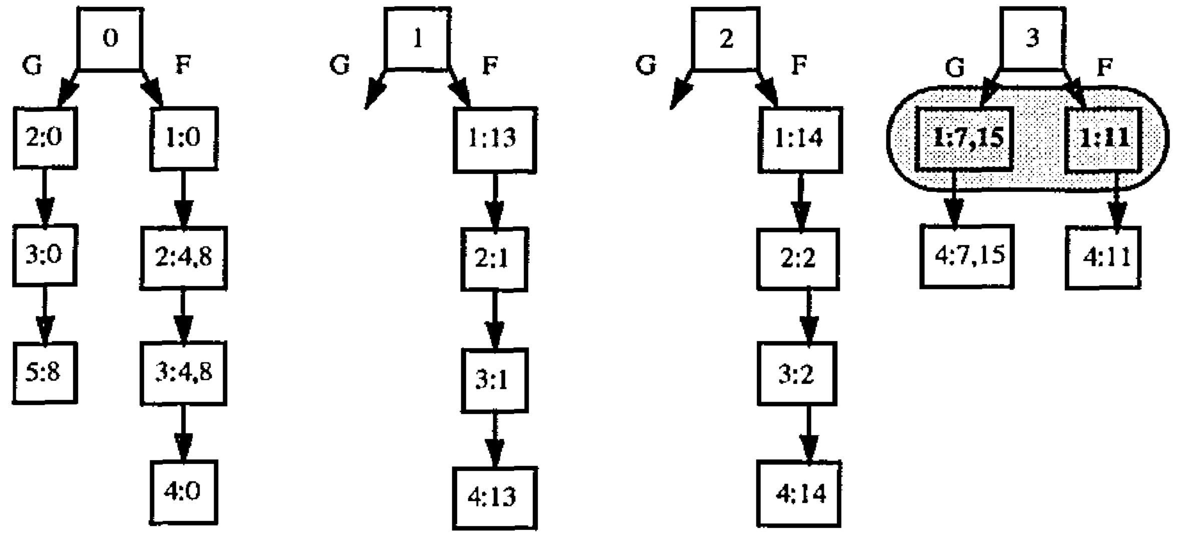


Figure 5.14. Response function data structure at output1 (Figure 5.13).

The response function for truncate module Trunc(1..0) is P_{R1} = {(0,4,8;0), (1,13;1), (2,14;2), (7,11,15;3)} and the response function for truncate module Trunc(3..2) is P_{R2} = {(0,1,2;0), (4,7;1), (8,11;2), (13,14,15;3)}. Computing the output of a module for response functions at the module's inputs is a two-step process. The first step is to compute the new current values for each block using the module function. The second step is to combine blocks whose new current values are the same. These two steps implement the series connection operation discussed in Section 3.1. The resulting response functions for buses output1 and output2 of the circuit in Figure 5.13 are shown in Figures 5.14 and 5.15, respectively. If response functions are incident on more than one input of a module, then they are combined using the parallel connection operation discussed in Section 3.1. before computing the output of the module.

Since module functions need only compute new current values for blocks, the cost of propagating a vector-layer (evaluated) response function through a module is proportional to the number of its blocks. Blocks are combined as they are propagated through non-transparent modules. Therefore, propagation performance at the vector layer increases as propagation paths become less transparent. Consider propagation through a truncate module that removes one bit of the input bus. The number of blocks can be reduced by as much as a half. Propagation of vector-layer response functions is slowest along fully transparent paths, however these paths are easily analyzed using symbolic-layer error propagation techniques.

We take advantage of the response function data structure given above to evaluate the

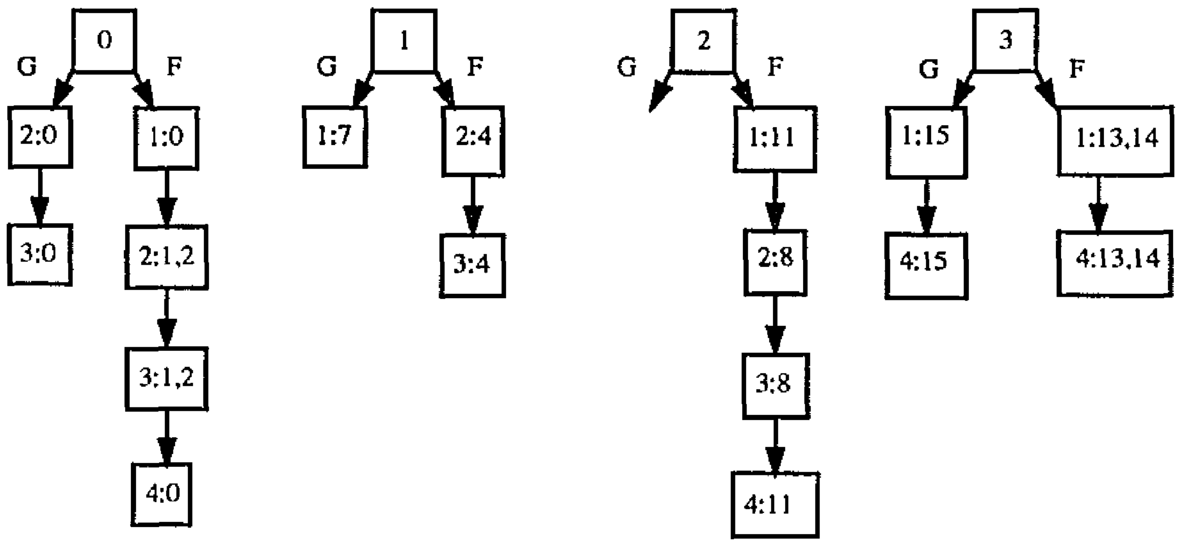


Figure 5.15. Response function data structure at output2 (Figure 5.13).

propagation condition, that is, the requirement that correct responses be separated from faulty responses in a response function P_R. The propagation condition is satisfied by P_R if for each block, there are entries in only one response list for each timestep. For example, we can easily see that the propagation condition is not met for the response function shown in Figure 5.14, since there are entries (shown shaded) in both the good and faulty timestep lists for block 3 at timestep 1.

Propagation functions P_{Ri}, $1 \le i \le k$, can be propagated in parallel along k partially transparent paths to primary outputs. These response functions are combined by *MATSim* using the intersection (parallel connection) operation to create a single response function $P_R = P_{R1} \# P_{R2} \# \ldots \# P_{Rk}$ for the set of primary outputs. In order to determine whether all errors are propagated, *MATSim* applies the propagation condition to P_R. Similarly, *MATSim* can analyze a set of k response functions propagated sequentially through a k-transparent module or subcircuit. Again, the set of response functions propagated to a primary output or set of primary outputs over multiple time instances are combined using the intersection operation and the resultant response function analyzed for the propagation condition.

For efficiency, we have combined the procedure for analyzing the propagation condition with intersection of propagation functions. The algorithm is shown in Figure 5.16. Let $\Re$ be the set of response functions to be analyzed. Response functions in $\Re$ may be a set of response functions

```
analyze_propagation_condition( input $\Re$:set of response functions )
{
    collision_list = $\varnothing$; new_collision_list = $\varnothing$;
    for (all response functions $P_{Ri}$ in $\Re$) {
        for (each block B in $P_{Ri}$) {
            for (each correct response timestep $t_C$ in good timestep list of B) {
                for (each faulty response timestep $t_F$ in
                    faulty timestep list of B) {
                    if ($t_C$ == $t_F$) {
                        response_list = faulty response list for $t_F$;
                        if ($t_F$ is in collision_list) {
                            get old_response_list for $t_F$ from collision_list;
                            response_list = response_list $\cap$ old_response_list;
                            if (response_list $\neq$ $\varnothing$)
                                add response_list to new_collision_list;
                        }
                    }
                }
            }
        }
        if (new_collision_list == $\varnothing$) return( SUCCESS );
        else {
                empty collision_list;
                collision_list = new_collision_list;
                empty new_collision_list;
        }
    }
}
return( FAILURE );
```

Figure 5.16. Algorithm for analyzing the propagation condition for a set of response functions.

propagated serially in time over a single bus, or a set of response functions propagated to different outputs, or both. For each block in each response function P_{Ri} in $\Re$, the algorithm *analyze_propagation_condition* checks to see if correct and faulty responses occupy the same timestep t. If they do, the propagation condition is not satisfied for P_{Ri}, but it still may be for $P_R = P_{R1}\#P_{R2}\#...\#P_{Rk}$ or $P_R = P_{R1} \cap P_{R2} \cap ... \cap P_{Rk}$. When correct and faulty responses both occupy timestep t in a block of one response function P_{Ri}, the list of faulty responses is stored as a block associated with t in a list called response_list in Figure 5.16. For example, if correct response 0 occupies timestep 2 with faulty responses 4 and 8, then response_list will contain the pair (4,8) associated with timestep 2, denoted (2:(4,8)).

Non-empty response_lists created by analyzing a set of response functions for the propagation condition are stored according to timestep in a list called collision_list in Figure 5.16. For example, suppose response_lists (2:(4,8)) and (2:(5,7)) are created by analyzing response functions P_{R1} and P_{R2} respectively. Each is added to collision_list, which then contains a list of pairs associated with timestep 2, denoted (2:(4,8), (5,7)). When a set of blocks associated with a particular timestep is retrieved from collision_list, it is referred to as old_response_list in Figure 5.16. In step 13 of *analyze_propagation_condition*, when it is determined that the propagation condition has not been met at timestep t, an old_response_list associated with timestep t in collision_list is retrieved. It is intersected block by block with the current response_list. If there are no elements in common, then all discrepancies associated with timestep t are distinguished, otherwise, the intersection is added to new_collision_list to be compared with another response function. If, after examining each response function in $\mathfrak{R}$, new_collision_list is empty, then all errors are propagated, otherwise new_collision_list becomes collision_list.

The procedure *analyze_propagation_condition* follows the propagation function intersection operation, but only lists of faulty responses are actually intersected. Given an ordered list of response functions, the procedure terminates successfully when the shortest sequence of intersections is determined to be sufficiently transparent to satisfy the propagation condition. All response functions must be intersected to prove that the sequence is not sufficiently transparent.

As an example of how *analyze_propagation_condition* works, let $\mathfrak{R} = \{P_{R1}, P_{R2}\}$, where P_{R1} is the response function in Figure 5.14, and P_{R2} is the response function in Figure 5.15. After processing P_{R1}, collision_list contains {(2:(4,8)), (3:(4,8)), (1:(11)), (4:(11))} since faulty values 4 and 8 conflict with correct value 0 in timesteps 2 and 3 of block 1, and faulty value 11 conflicts with correct value 15 in timesteps 1 and 4 of block 3. Next, the procedure processes P_{R2}. In this case, correct response 0 conflicts with faulty responses 1 and 2 in timesteps 2 and 3 of block 1. Intersecting these blocks with corresponding blocks in the collision_list results in the empty set, so no blocks are added to a new_collision_list. Continuing in this way, we see that all intersections result in the null set, so new_collision_list is empty at line 21 and *analyze_propagation_condition* returns SUCCESS; all errors are propagated.

Next we consider the complexity of *analyze_propagation_condition*. Let $|\mathfrak{R}|$ the number of response functions in $\mathfrak{R}$ and V be the number of test vectors (timesteps) in stimulus sequence

T_S. The maximum number of blocks that a response function can have depends on the width W_{max} of the largest bus in the circuit; in the worst case, every response function in $\Re$ will have $2^{W_{max}}$ blocks. Amortized over all blocks, the number of timesteps to analyze in any particular block is $O\left(V/2^{W_{max}}\right)$. In the worst case, the size of Ω_R is $O\left(2^{W_{max}}\right)$, so the worst-case cost of the intersection operation is $O\left(2^{W_{max}}\right)$. This implies that as the problem size grows in terms of $\Re$, W_{max}, and V, the worst-case complexity of *analyze_propagation_condition* is

$$O\left((|\Re|)\left(2^{W_{max}}\right)\left(V/2^{W_{max}}\right)\left(2^{W_{max}}\right)\right) = O\left(2^{W_{max}}|\Re|V\right)$$

As noted in Chapter IV, for many modules, the size of Ω_R is $O(V)$, not $O\left(2^{W_{max}}\right)$, therefore we expect the average-case complexity to be $O\left(|\Re|V^2\right)$. Finally, in the optimal case, no intersections are required, and propagation is determined by the first response function in $\Re$ to be analyzed. However, all blocks and timesteps in the first response function must still be examined by any algorithm analyzing the propagation condition. Therefore, the lower bound complexity of the problem of analyzing the propagation condition for a set of response functions $\Re$ is $\Omega(V)$. If any of the response functions in $\Re$ is congruent to zero, then the propagation condition is met without executing *analyze_propagation_condition.*

5.1.5 Summary of *MATSim*

Let M be a MUT and let $TP_1 = (T_S;T_R)$ be a test package for M. *MATSim* can determine whether all errors that can be produced by M are propagated to primary outputs. *MATSim* needs no structural model for M, and TP_1 can be based on any appropriate fault model. It can calculate fault coverage if the faults that cause each faulty vector in the response set for M are provided. Conventional fault simulators on the other hand require more detailed structural models for modules and explicit fault models. As discussed in Chapter II, for many modules, e.g., embedded RAMs, the use of precomputed tests is not only appropriate, but necessary. The propagation of precomputed tests for embedded RAMs can easily be analyzed by *MATSim*, but not by conventional fault simulators.

MATSim propagates test package information hierarchically at the symbolic and vector layers. Symbolic-layer propagation is significantly faster than the bit-layer methods used by standard (gate-level) fault simulators, since far fewer module evaluations are needed on any path through the circuit. Modules with precomputed tests are usually connected by multi-bit bus structures that are well-suited for the symbolic-layer signal abstraction supported by *MATSim*. Since high-level functional circuit behavior is readily apparent to the designer, symbolic test programs can often be easily generated for these circuits manually, or by the combination of manual and automated methods used by *PathPlan2* (discussed below).

MATSim implements propagation of symbolic expressions, response functions, and hierarchical error signal values within the framework of a conventional event-driven simulator. Some or all of the features discussed above can therefore be added to existing commercial or experimental simulators. *MATSim* accepts both combinational and sequential circuits described in the conventional HDL Verilog, which is also accepted by several commercial fault simulators.

However, only a small number of primitive modules are currently implemented by *MATSim*. More primitives can be added, but a general method for modeling large modules such as microprocessors is needed for *MATSim*. The models must implement the module functions that propagate symbolic error signals as discussed in Chapter IV. These module functions can be difficult to construct for arbitrary modules. However, functional models for arbitrary modules can be implemented by decomposing the modules into dataflow graphs of primitives. This method is already used in the AutoCircuit Verilog parser [32] which *MATSim* employs, so *MATSim* can be modified to include the capability.

5.2. *PathPlan2*

The *PathPlan2* test generation algorithm is the successor to our original test generator *PathPlan*. It implements the extensions to *PathPlan* that we identified in Chapter II and uses *MATSim* to propagate signals. In this section, we discuss the overall design of *PathPlan2*, the use of test packages, the test generation algorithm, and some experimental results.

5.2.1 Design Philosophy

PathPlan2 is intended to propagate generate tests for circuits by propagating test packages $(T_S;T_R)$ for modules at two levels of abstraction: symbolic and vector. The primary emphasis is on symbolic-layer propagation of fault-free signals such as T_S and T_R as expressions, and hierarchical propagation of test response errors at the symbolic and vector layers. It uses a test generation algorithm with forward-only signal propagation similar to PODEM [40]. Like PODEM, the

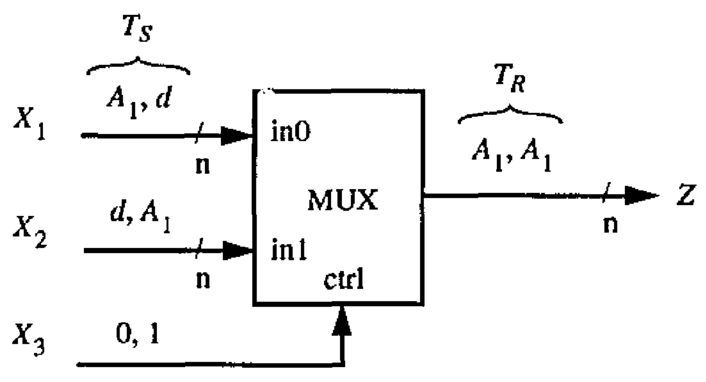


Figure 5.17. Bus assignment example

basic structure of *PathPlan2* is simple: generate new objectives, that is, internal signal values to be justified, relate the new objectives to the primary inputs, and perform forward implication (simulation) using *MATSim*. Objectives in *PathPlan2* are T_S, and control signals for modules on the propagation path. *PathPlan2* relates new objectives on module ports in the circuit to primary inputs by topological backtrace from the site of the objective to some primary input. All of these tasks are greatly complicated compared to gate-level test generators such as PODEM due to *PathPlan2*'s use of high-level functional modules and symbolic data.

5.2.2 Test Packages in *PathPlan2*

As in *PathPlan*, every module M in the circuit has two types of test packages. The fault test package (FTP) contains precomputed tests for M and propagation test packages (PTPs) are used to determine values for signal propagation and are associated with all modules other than the MUT. Test packages for *PathPlan2* are similar in format to those used by *PathPlan*. Variables are unified with values to be propagated, and vector sequences must be matched exactly to the current circuit state using the instantiation procedure discussed in Chapter II. For example, an FTP for the multiplexer shown in Figure 5.17 might be

$$(T_S;T_R) = ([(A_1, d), (d, A_1)], [0, 1]; (d, d))$$

where the d's are don't care values. *PathPlan2* uses the values in this test package that are not don't cares as objectives for the corresponding bus.

When propagating the test response T_R *Pathplan2* does not use PTPs to compute or assign module output values as in *PathPlan*. Instead, module output signals are computed using simulation by *MATSim*. Module inputs determined by instantiation provide objectives for *PathPlan2*.

The PTP $((\alpha, d), 0; d)$ for the multiplexer in Figure 5.17 specifies that T_R is to be propagated from input in0 (X_D), and that 0 is an objective for ctrl.

PTPs are also used by the backtrace procedure to transfer objectives from module outputs to module inputs. Suppose that the objective A_1 (component of T_S) is to be transferred from the output of the multiplexer in Figure 5.17 to input in0 on a path from the MUT to a primary input. The PTP $TP_1 = ((\alpha, d), 0; \alpha)$ can be used to transfer this objective. First, A_1 is assigned to the variable α throughout the test package using unification as discussed in Chapter II. Then all the inputs specified in TP_1 become new objectives for *PathPlan2* to satisfy. Note that in contrast to *PathPlan*, during backtracing no signal values are actually assigned to buses except at primary inputs.

5.2.3 Test Generation Algorithm

Figure 5.18 describes *PathPlan2*'s procedure for testing a module M. The goal of this procedure is to propagate precomputed test stimuli to a module M and to propagate all errors produced by M to a primary output. Stimulus and response sequences are stored in a test package $(T_S; T_R)$. The assignment of a symbolic vector sequence component v_k of the stimulus sequence T_S to a corresponding a input bus k of M represents an objective that *PathPlan2* attempts to satisfy by making assignments to primary inputs and propagating them using *MATSim*. The assignment of symbolic vector sequences as control signals to sensitize modules to propagate T_R are similarly treated as objectives by *PathPlan2*. Success is achieved when these objectives are met and when error propagation signals that contain all errors in M's response set reach primary outputs. *PathPlan2* terminates unsuccessfully if the components of T_S cannot be propagated to M, or if the error information in the error propagation signals cannot be propagated to primary outputs.

The algorithm works at two levels of abstraction, the symbolic layer and the vector layer, corresponding to the two levels of abstraction currently supported by *MATSim*. As noted, *MATSim* currently treats all fault-free signals, including individual vectors as symbolic-layer signals. The FTPs used by *PathPlan2* describe the format for the components of T_S. If an individual component v_k is a symbolic reference to a vector sequence, then *PathPlan2* will propagate expressions using *MATSim*. If v_k is a single vector, then *PathPlan2* will propagate vectors at this same level of abstraction. However, *MATSim* uses the same symbolic-layer update routines to propagate the sig-

```
test_one_module
{
    error_propagation_layer = symbolic
    perform initial implication;
    if (successful) return( SUCCESS );
    while (TRUE) {
        if (there are more objectives) {
            get new objective $(k, v_k)$ ;
            $(j, v_j)$ = backtrace( $(k, v_k)$ );
            push( PI_stack, $(j, v_j)$ );
            imply forward;
            if (successful) return( SUCCESS );
        } else if (PI_stack empty)
            return( FAILURE );
        if (symbolic-layer error propagation is blocked) {
            error_propagation_layer = vector;
            imply forward;
            if (successful) return( SUCCESS );
        }
        while (test is not possible) {
            error_propagation_layer = symbolic;
            $(j, v_j)$ = pop( PI_stack );
            imply forward;
            if (there is an untried alternative $v_j'$) {
                $v_j = v_j'$;
                push( PI_stack, $(j, v_j)$ );
                imply forward;
                if (successful) return( SUCCESS );
            } else if (PI_stack empty)
                return( FAILURE );
            if (symbolic-layer error propagation is blocked) {
                error_propagation_layer = vector;
                imply forward;
                if (successful) return( SUCCESS );
            }
        }
    }
    return( FAILURE );
}
```

Figure 5.18. The main *PathPlan2* algorithm

nals; the vectors are treated as trivial expressions.

When T_S is successfully matched at the inputs to the MUT, *PathPlan2* initiates error propagation in parallel with the propagation of fault-free signals. It automatically controls the abstraction level for error propagation. It tries first to propagate symbolic error signal values from the set R_7 (symbolic-layer signals). If propagation is blocked along all paths to a primary output, then *PathPlan2* automatically switches to the vector layer and attempts to propagate errors by evaluating response functions using the same circuit state previously used to propagate signals at the symbolic layer.

We now examine the test generation procedure of Figure 5.18 in more detail. *PathPlan2* generates (bus, value) pairs (k, v_k) as objectives and uses a backtrace procedure to assign v_k to a primary input (PI). The initial assignment is saved on a stack PI_stack and forward implication is performed using symbolic-layer error propagation. If forward implication is successful, the algorithm terminates successfully. Otherwise, the algorithm switches to the vector layer for error propagation. If a test is not possible, alternatives are tried for each PI assignment in the stack. Note that there may be many alternatives for each PI, whereas there are just two in PODEM. The algorithm terminates unsuccessfully when all possible alternatives have been tried for each element in the stack. The test for success in symbolic-layer error propagation in lines 12 and 28 is simply a check to see if the symbolic error signal value propagated to a primary output (PO) is R $\in R_7$. The test for success in the case of vector-layer error propagation (lines 18 and 34 of Figure 5.18) uses *analyze_propagation_condition* (Figure 5.16). Let the *test frontier* be defined as the set of lines carrying error propagation signals incident on modules whose output is unassigned. The state of a circuit determined by *PathPlan2* during a test generation pass, but before it terminates successfully, is called a *partial test*. For any partial test, propagation is blocked (line 15) at the symbolic layer when the test frontier is empty. Propagation may still succeed at the vector layer if there has been no conflict.

When generating new objectives (line 8 in Figure 5.18), *PathPlan2* attempts first to satisfy the requirements of the MUT. When the input stimulus sequence has been matched at the inputs to the MUT and a test response T_R has been produced at the outputs of the MUT, *PathPlan2* generates objectives for propagating T_R using PTPs. *PathPlan2* selects a module M on the test frontier and searches the PTPs for M to determine a relevant input control port X_C and value $V(X_C)$ that will make M transparent for T_R on input data port X_D.

PathPlan2 initiates backtracking (discussed below) at line 20 when a test is not possible given the current state of the circuit, that is, when

1. Propagation is blocked at the vector layer, implying that error propagation on both the symbolic and vector layers have been tried unsuccessfully
2. There is a conflict, implying that two different values are scheduled to be assigned to the same bus at the same layer
3. The inputs to the MUT have all been assigned, but do not match T_S
4. A new objective cannot be generated

Propagation is blocked at the vector layer when analysis of the response functions at the test frontier using *analyze_propagation_condition* shows that all errors are not propagated. To determine conflict, *MATSim* checks for frozen values. The majority of assignments to the MUT are not tests, in contrast to the typical gate-level case. Therefore, if all of the MUT inputs have been assigned, but the values are not in T_S, then a test is not possible from the current state, and backtracking must be initiated. Finally, *PathPlan2* backtracks when, given the current partial test, no test packages for any module can be applied to obtain a new objective.

To backtrack, *PathPlan2* returns to a previous decision point by popping the old alternative off PI_stack and generating a new alternative. The algorithm returns to the symbolic layer to attempt error propagation. It may switch again to the vector layer again to complete propagation if the symbolic error signal R cannot be propagated through any module at the test frontier. Although it is easy to generate all possible alternatives for a vector, there is no unambiguous alternative for arbitrary symbolic expressions. Therefore, alternatives are not tried exhaustively for symbolic expressions. Instead, *PathPlan2* associates strategies for generating alternatives directly with the test package component v_k of the objective (k, v_k) that is backtraced to the primary input. Examples of such strategies include: increment, decrement, and multiply-by-two. For instance, if the most recently assigned PI is in1 and has signal value A, and the alternate generation strategy is increment, then on backtrack, the new assignment is A+1. Backtrack termination strategies also include placing limits on the number of backtracks.

The algorithm *test_one_module* discussed above generates tests for acyclic combinational circuits or circuits with full scan. We now outline how *PathPlan2* can be extended to handle sequential circuits with feedback loops. The values assigned to ports of the circuit model become arrays of values indexed by time. These arrays are allocated in fixed-sized blocks. The extended version of *PathPlan2* searches for a test generation solution in time and space up to the limits of

```
seq_test_one_module
{
    initialize max_span, $t_0$, and max_time;
    do {
        allocate value arrays of size max_span;
        if (test_one_module() == SUCCESS) return( SUCCESS );
        increment max_span, $t_0$, max_time;
    } while (max_span <= SPAN_LIMIT);
    return( FAILURE );
}
```

Figure 5.19. Top level algorithm for sequential test generation.

these arrays. New larger arrays are dynamically allocated to continue the search. The top-level algorithm is given in Figure 5.19. The variable *max_span* controls the size of the value arrays. The variable t_0 is the timestep in which the test is instantiated, nominally in the center of the value array. The variable *max_time*, is the maximum timestep for either propagation or backtrace.

One difficulty in sequential test generation using precomputed tests for modules, is the propagation of error signals through the MUT in timeframes other than t_0. The symbolic error signal R cannot be propagated through the MUT since we cannot be certain that error information is not masked. To analyze error propagation using response functions, multiple versions of response functions are needed, one for each fault in the MUT, as discussed in Chapter IV. The propagation condition must be analyzed for all of these response functions, which complicates the analysis by a factor proportional to the number of faults in the MUT. In addition, a detailed fault model is needed for the MUT, which may not always be available.

The ability to generate tests for sequential circuits is an important requirement of commercial test generators, since most practical circuits have memory elements and feedback. The two extensions to the basic *PathPlan2* algorithm discussed above can be added to convert *PathPlan2* to a sequential circuit test generator. However, our main objective in implementing the current version of *PathPlan2* is to demonstrate new signal propagation and analysis techniques, based on our propagation theory, that allow precomputed test packages to be propagated in circuits with complex bus structures.

5.2.4 Example

Next we present an example of how *PathPlan2* generates a test for a module. We use the

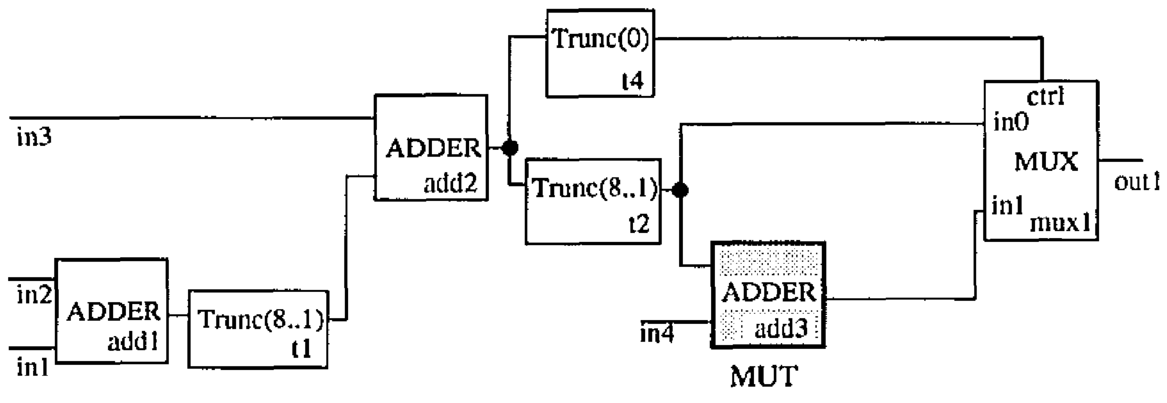


Figure 5.20. Module add3 in Fltrdp to be tested by *PathPlan2*.

version of Fltrdp shown in Figure 5.20. The MUT is add3, and the FTP we want instantiate is $TP_1 = (A_1, A_2; d)$. In the following, to demonstrate signal assignment we use the statement $Z := v$ to indicate that value v has been assigned to bus Z. As before, the statement $V(Z) = v$ indicates that the value v has been propagated to bus Z.

To instantiate TP_1, *PathPlan2* begins by backtracing objectives to the nearest primary input (lines 8–11 in Figure 5.18) and simulating using *MATSim*. Therefore, it makes the assignment in4 $:= A_2$ first. This satisfies the objective specified in TP_1, namely, V(add3.in2) $= A_2$. Next *PathPlan2* transfers the objective V(add3.in1) $= A_1$ to the other primary inputs. Recall that PTPs are used in determining how objectives are transferred from module outputs to module inputs during backtrace. For modules t1 and t2 we use the PTP $TP_2 = (2\alpha;\ \alpha)$, where α is a variable. In other words, to obtain a particular value v at the output of a truncate module such as t1, the input should be $2v$. For modules add1 and add2, we use the PTP $TP_3 = (\alpha,\ \alpha;\ 2\alpha)$, where again, α is a variable. In other words, for a particular value v, to obtain a output of $2v$, v must be applied to both inputs. Using these PTPs, we find *PathPlan2* makes the assignments in3 $:= A_1$, in2 $:= A_1$, in1 $:= A_1$ in this order.

With these assignments, TP_1 is successfully instantiated, and mux1.ctrl = 0 because the least significant bit of an even number is 0. Since *PathPlan2* begins by processing error signals at the symbolic layer, the error signal at the output of add3 is R. However, the error signal cannot be propagated through mux1 with the circuit in this state. Switching error processing to the vector layer (lines 15–18 in Figure 5.18) does not help since mux1.in1 is disabled; the transmission func-

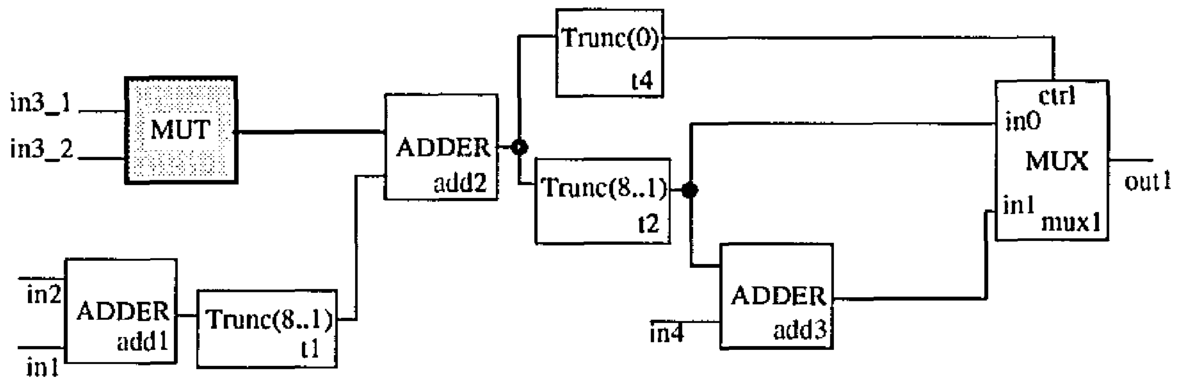


Figure 5.21. MUT connected to Fltrdp tested by *PathPlan2*.

tion of mux1 is 1. Therefore, *PathPlan2* backtracks (lines 21–23) by choosing an alternative for the most recent assignment in1 := A_1. We assume here that the backtrack strategy associated with each value is increment, so the new assignment to in1 is $A_1 + 1$. The output of add1 becomes $2A_1 + 1$, but the output of the truncate module t1 is A_1 which also fails. When *PathPlan2* backtracks a second time however, we obtain in1 := $A_1 + 2$, and the output of add1 is $2A_1 + 2$, which becomes $A_1 + 1$ at the output of t1. This results in TP_1 being successfully instantiated and also in V(mux1.ctrl) = 1, which propagates R to the primary output. The final symbolic test program is

in1 := $A_1 + 2$
in2 := A_1
in3 := A_1
in4 := A_2
V(out1) = $A_1 + A_2$

Recall (Section 2.5.) that *PathPlan* cannot generate a test for add3 because it cannot propagate arbitrary symbolic expressions.

Next, we show how *PathPlan2* switches layers to complete error signal propagation when propagation is blocked at the symbolic layer. Consider the new version of Fltrdp in Figure 5.21. In this version, an arbitrary MUT is connected to add2. We assume that the FTP for this module is $TP_1 = (A_1, A_2; A_3)$, and that the test has two test vectors with first-order response sets $\Omega_{R1} = \{0, 1\}$ and $\Omega_{R2} = \{5, 6\}$ respectively. We also assume that the correct value in Ω_{R1} is 0 and the correct value in Ω_{R2} is 5.

To generate a test, *PathPlan2* first tries to instantiate TP_1. Since the MUT is connected directly to primary inputs, *PathPlan2* immediately makes the assignments in3_1 := A_1 and in3_2 := A_2. The FTP TP_1 is instantiated and the symbolic-layer error value R is produced at the output of the MUT. To propagate the test response through add2, *PathPlan2* uses the PTP (α, 0; α), which results in the assignments in2 := 0 and in1 := 0 after backtrace. At this point, the test frontier consists of the inputs to truncate modules t2 and t4. Since R cannot be propagated through truncate modules, propagation is blocked.

Next, *PathPlan2* switches to the vector layer. The response function at the output of the MUT is P_R = {(0;0), (1;1), (5;5), (6;6)}, which is propagated to mux1.ctrl and mux1.in0. In order to propagate this response through mux1, *PathPlan2* uses the PTP (α_1, α_2, $\alpha_2 + 1$), where α_1 and α_2 are variables. This test package specifies that redundant error information can be propagated through all inputs of the multiplexer simultaneously as long as the two data inputs differ—in this case, they are made different by adding 1 to the value at port in1. After backtrace, *PathPlan2* makes the assignment in4 := 1. The final symbolic test program is

in1 := 0
in2 := 0
in3_1 := A_1
in3_2 := A_2
in4 := 1
V(out1) = $MUX(A_3, A_3/2, A_3/2 + 1)$

Recall that ARTEST [58] cannot propagate error information through circuits with irregular buses of the kind illustrated by Fltrdp in Figure 5.21.

5.2.5 Summary of *PathPlan2*

PathPlan2 automates the generation of test programs that employ precomputed tests and several levels of abstraction. It can generate either symbolic or vector-level test programs depending on the abstraction level in the test packages used. Since *PathPlan2* uses a test generation algorithm similar to PODEM that only makes assignments to primary inputs, it needs no implication procedure other than *MATSim*.

In addition, *PathPlan2* is unique as a test generation algorithm making assignments only to primary inputs, in that it supports symbolic-layer signal assignments and backtracking. *PathPlan2* uses a variety of alternative-generating strategies associated with each primary input as discussed above in order to backtrack. In addition, *PathPlan2* uses PTPs during backtrace to transfer some functional constraints and requirements to primary inputs. For instance, in the Fltrdp example (Figure 5.21), we showed how *PathPlan2* transferred an objective of $2A_1$ at the output of an adder to A_1 on both inputs to the adder using PTPs. *PathPlan2* cannot currently take into account conflicting constraints from different paths, although this capability can be added.

The current version of *PathPlan2* also does not implement the sequential circuit test generation algorithm discussed above. The ability to generate tests for sequential circuits is important, and future versions of *PathPlan2* will include this feature. However, as mentioned, the objective here is to demonstrate that *PathPlan2* can generate tests for circuits with distributed partially transparent paths, since this is a key limitation of *PathPlan*, ARTEST, and all other previously reported hierarchical test generation methods using precomputed tests. We demonstrated above how *PathPlan2* propagates precomputed test responses on partially transparent paths in Fltrdp. We will discuss further examples below.

5.2.6 Experimental Results

PathPlan2 and *MATSim* are implemented in C and C++. Together with the Verilog parser and related library functions, they contain roughly 30K lines of code. Here we report the results of using *PathPlan2* to generate tests for some medium-sized datapath circuits. The objective of this experiment is to further demonstrate the ability of *PathPlan2* to generate tests for circuits with distributed partially transparent paths, and to compare the performance of *PathPlan2* to the performance of *PathPlan* for the circuits that *PathPlan* can also handle. Since *PathPlan2* is a more complicated program than *PathPlan*—it is designed to generate tests for more general types of circuits—it might be expected that *PathPlan* is more efficient for the smaller domain of circuits that it handles. However, we have found that the performance of the two programs is similar.

The experimental results are shown in Table 5.2. CPU time measurements were taken on an Sun 4 workstation with 32 megabytes of memory. The simple datapath circuits Fltrdp, Vertdp, and Rowdp were used as benchmark circuits in the original work on *PathPlan* and are discussed in Chapter II. Performance results for *PathPlan* were not measured in terms of CPU time, since this depends on the implementation of the algorithm, the quality of the compiler, the machine used to execute the program, etc. Instead, module evaluations are counted. Subsequent performance mea-

Circuit	Number of modules	Number of gates (est.)	*PathPlan2/MATSim*			*PathPlan*		
			Module evaluations	CPU (sec)	Test coverage (%)	Module evaluations	CPU (sec)	Test coverage (%)
Fltrdp	6	254	255	1.12	100	57	–	70
Vertdp	5	138	29	0.25	100	21	–	100
Rowdp	6	220	100	0.59	100	33	–	100
Smalldp	8	250	180	1.07	100	–	–	–
Mdp	7	792	439	3.21	100	–	–	–

Table 5.2 Performance comparison of *PathPlan2/MATSim* and *PathPlan*.

surements on other test generators, particularly ARTEST, have been limited to CPU time. Therefore, we have provided measurements of both types for *PathPlan2*. The "test coverage" column lists the percent of tests in the test packages that were successfully applied.

We assume direct control of all control signals for the datapaths in this experiment. Typically, datapath circuits such as these are controlled by a separate control unit. The output of the control unit is latched in a register and is part of a scan chain. This is a modest design-for-testability strategy implemented in nearly all machines of this type used in commercial circuits. A sequential version of *PathPlan2* can also generate the control signals in many cases if there are no conflicts. This will make test generation times longer, but will not invalidate the results presented here.

As shown in Table 5.2, the performance in terms of module evaluations for *PathPlan2* and *PathPlan* are comparable (within an order of magnitude) for Fltrdp, Verdp, and Rowdp, despite the fact that *PathPlan2* is more complicated and general. The circuit Smalldp is a modified version of the datapath circuit shown in the box in Figure 3.18. In its original form, the circuit is untestable; some changes were made to make it possible for *PathPlan2* to instantiate all test packages. Despite these changes, the circuit still contains bus truncations that make it impossible for either *PathPlan* or ARTEST to test it. Finally, Mdp is a small computer datapath circuit similar to one used as a benchmark in ARTEST. Since this implementation of *PathPlan2* assumes direct access to control signals, an assumption not made by ARTEST, we cannot directly compare the performance of the two programs. Nevertheless, the performance of *PathPlan2* in generating a test for this circuit appears to be similar to its performance in the other cases listed.

5.3. Summary

We have presented two new test generation tools *PathPlan2* and *MATSim*. *PathPlan2* is a hierarchical test generation algorithm similar to PODEM in that it uses forward-only implication. *MATSim* analyzes error propagation for circuits with precomputed tests for modules. It does not require low gate-level models for modules being tested, these are implicit in the precomputed tests. It propagates signals at two levels of abstraction, and can analyze error propagation along distributed partially transparent paths.

PathPlan2 generates tests at two levels of abstraction. It assigns signal values to primary inputs and uses *MATSim* to propagate these signals to the inputs of the MUT and to the control inputs of modules on the propagation path from the output of the MUT to primary outputs. Precomputed tests stored in test packages can be represented symbolically and propagated as symbolic expressions, or numerically and propagated as vectors. Error signals are propagated symbolically by default, but when propagation is blocked at the high level, *PathPlan2* automatically switches to propagation at the vector level to complete the analysis. Hence, it can analyze propagation in circuits with complex and irregular bus interconnection structures, a capability lacked by previous high-level test generators using precomputed tests

CHAPTER VI
CONTRIBUTIONS AND FUTURE WORK

This chapter reviews the major contributions of the thesis and briefly discusses some future directions for research.

6.1. Thesis Contributions

Hierarchical test generation using precomputed tests is an important method for computing tests for complex digital circuits. Many circuits are composed of well-defined modules from proprietary design libraries and have precomputed tests stored for them. It is often inconvenient or impossible to generate new tests for these modules when they are included in a larger circuit. This thesis has presented the theory and tools we developed for generating tests for circuits using precomputed tests for modules. The major contributions are as follows:

- We designed and implemented one of the first test generation programs *PathPlan* designed specifically to use precomputed tests and symbolic propagation techniques
- We developed a new, general theory of signal propagation for modular, bus-structured circuits
- We developed an efficient, hierarchical method for analyzing error propagation when testing modular circuits with complex, irregular buses.
- We proposed a novel, logic design method for increasing the transparency of modules to improve the testability of circuits
- We designed and implemented a pair of test generation tools *MATSim* and *PathPlan2* with the unique capability of generating tests for modular circuits with complex, irregular buses using precomputed tests and hierarchical signal propagation

PathPlan generates tests for circuits by propagating symbolic representations of precomputed test data, stored in units called test packages $(T_S;T_R)$, through a circuit model. The test stimulus T_S is propagated to a module under test (MUT) and the test response T_R is propagated to primary outputs. Both T_S and T_R are represented using a flexible and hierarchical notation in which signals are sequences of vectors. Only simple transformations of T_S and T_R during signal propagation are allowed by *PathPlan*. This approach cannot handle circuits with complex, irregular bus structures including truncation and reconvergent fanout—a restriction also shared by all published test generation techniques developed since *PathPlan* that use symbolic and high-level signals. However, *PathPlan* is very effective for testing embedded RAMs and other large modules in circuits with regular buses that cannot be tested using conventional techniques. It also demonstrates that symbolic propagation can provide substantial performance benefits over conventional techniques. In some cases, *PathPlan* can generate tests up to three orders of magnitude faster than conventional techniques. This is important when the precomputed tests are long, as they may be for embedded modules such as CPUs in microcontrollers. The overall philosophy of *PathPlan* has been implemented by a major electronics company, and it forms the basis for the other contributions of this thesis.

The main limitation of *PathPlan* and other test generators using precomputed tests is the pessimistic methods they use for analyzing the propagation of errors at high levels of abstraction. The problem of hierarchical error propagation has been poorly understood. To expand the range of circuits that can be tested using precomputed tests and hierarchical propagation techniques, we developed a theory of complex signal propagation for modular circuits. To characterize the propagation of vector sequences through module functions, we introduced the concept of propagation functions. A propagation function is a representation of a module's input-output behavior in which disjoint subsets (a partition) of the module's input signal domain are mapped to signal values at the module's output ports. Signal propagation in multi-module circuits is characterized by expressions in a propagation algebra whose elements are propagation functions expressed as partitions and whose operators correspond to series-parallel connections. The propagation algebra is governed by a number of common algebraic laws and properties, which we have identified. Special versions of propagation functions called transmission functions represent the information transmission properties of modules and circuits alone, independent of the information to be propagated.

We defined a circuit to be transparent if its input signals can propagate through it without loss of information. This property is explicitly represented in our propagation algebra as the zero propagation function. We then studied the central problem of propagating errors, that is, discrepancies between faulty and fault-free signals. If there is a transparent path from the output of the module under test (MUT) to primary outputs, then all errors can be propagated along this path; it is not necessary to analyze the propagation of individual errors. Therefore, the knowledge that transparent paths exist can be exploited to improve test generation speed. On the other hand, if a circuit has poor transparency, it may be impossible to propagate all errors to primary outputs. Our techniques exactly quantify the capacity of modules to propagate errors. When factors that contribute to poor transparency are detected, circuit modifications can be made to improve transparency.

Subcircuits with no single transparent path can still be transparent since partially transparent circuit paths in space (connected in parallel) or in time (sequences of propagation modes for the same path), can be combined to provide fully transparent information transmission. Previous hierarchical test generation approaches based on precomputed tests do not take this composite transparency into account. The transparency of such combinations is explicitly represented in our propagation algebra by the intersection/parallel connection operation.

We derived several theorems describing the effect on transparency of circuit or module structure. For example, we determined the effect of relative input and output port sizes on the transparency of modules (Theorem 3.3 and Corollary 3.1). We derived necessary and sufficient conditions for transparency in series connections (Theorem 3.4 and Corollary 3.2) and parallel connections (Theorem 3.5). We demonstrated that when the output data port of a module is much smaller than the input data port, the amount of error information propagated through the module is limited only by the size of the output port. Moreover, the amount of information that can be propagated in this case is surprisingly large (Theorems 3.8 and 3.9). To deal with incomplete transparency, we introduced the concept of k-transparency, that is, the propagation of data signals along partially transparent circuit paths in k timesteps using k different control signals in sequence to make the paths transparent. We also obtained bounds on the minimum value for k (Theorem 3.10).

We employed the propagation theory to obtain an efficient hierarchical method for analyzing the propagation of test response errors for specific modules. We developed a high-level method for representing response functions as symbolic signal values. These signals can be efficiently propagated through transparent circuit paths using high-level symbolic module functions. We have developed a method for rigorously constructing these functions from the signal specifications. Our method is more general than previous, ad hoc methods of defining and using high-level error propagation signals, and supports a full hierarchy of symbolic signals. At the lowest level in this hierarchy, error propagation is analyzed by evaluating response functions from the output of the MUT to primary outputs when T_S is successfully applied to it. The response functions are represented in the partition format developed in our propagation theory, and can be efficiently propagated as complex signals through circuits with irregular buses. Since some circuits are opaque to propagation, we have also devised design techniques to improve circuit transparency. In contrast to conventional approaches that use additional buses or scan latches to bypass opaque modules in circuits, our methods focus on making opaque modules more transparent. We developed two methods based on our propagation theory for specifying k-transparent modules that can be synthesized using conventional logic synthesis techniques.

We implemented our method for hierarchical error signal propagation in a pair of programs called *MATSim* and *PathPlan2*. *MATSim* is a novel simulator that propagates test package data at two levels of abstraction, symbolic and vector. Fault-free stimulus signals in *MATSim* are represented by symbolic expressions that are simplified using techniques derived from symbolic computer algebra. These techniques have been specialized by us for use in symbolic simulation. *MATSim* also uses our hierarchical error propagation techniques to propagate T_R. *PathPlan2* is our new hierarchical test generation algorithm. It extends the capabilities of *PathPlan* and uses *MATSim* to propagate T_S and T_R. *PathPlan2* can generate tests for circuits with irregular buses not handled by previously published high-level test generators using precomputed tests for modules. It is a more powerful and general program than *PathPlan*, nevertheless, its performance is at least as good.

6.2. Future Work

We conclude this section with some suggestions for extending the results of this thesis.

Theory. Our general theory of propagation can be used to analyze signal transmission through circuits composed of combinational modules and simple transparent sequential modules such as registers and latches. However, many circuits contain complex sequential library modules described by behavioral models. To handle such modules, the propagation theory can be expanded to analyze sequential modules, or alternatively, methods can be developed to automatically decompose behavioral descriptions of large sequential modules into high-level combinational and transparent sequential components for use in analyzing signal propagation. Similar decomposition methods are widely employed in high-level synthesis programs such as AutoCircuit.

The simulation approach employed in *MATSim* can form the basis of a general theory of symbolic simulation. Such a theory would be useful in several CAD areas including testing and design verification. The theory would fully characterize the canonical forms for expressions created while propagating symbolic values in typical circuits. Exact and complete algorithms for simplifying expressions are also desirable. We plan to further develop context-specific simplification techniques for expressions used as signals in circuits with buses of fixed width. *MATSim* currently simplifies known even (odd) expressions to 0 (1) when they are propagated on 1-bit buses.

Test Generation. *PathPlan2* can generate tests for acyclic circuits with irregular buses using precomputed tests for modules. We plan to implement the sequential extensions discussed in Section 5.2. and employ *PathPlan2* in testing microcontrollers composed of large library modules and circuits designed by AutoCircuit. When generating tests for sequential circuits, T_R is sometimes propagated through a MUT M in timeframes later than when T_S is applied to it. In order to propagate test response errors through M, individual faults in M must be linked to faulty responses in M's response set and multiple versions of response functions must be propagated through M using detailed fault models. We plan to add this feature to *MATSim* and study the performance of various implementation methods.

MATSim currently analyzes the propagation of test response errors as discrepancies, but does not relate these discrepancies to individual faults. The same fault f_i in a MUT M may produce several different error vectors contained in the response set. Therefore, when one error caused by f_i cannot be propagated, f_i may be covered by another error that is propagated. Adding some redundant vectors can improve fault coverage when test response errors must be propagated through non-transparent circuit paths. On the other hand, typical non-transparent modules such as truncate modules may block certain errors no matter how many redundant vectors are added. The effectiveness of such redundant test vectors in precomputed library tests for modules needs to be investigated.

Design for Testability. One promising area of future research is design for *k*-transparency. Our experiments suggest that the overhead of using *k*-transparent modules in a circuit is comparable with other design-for-testability techniques such as scan design. However, our current design methods produce *k*-transparent versions of modules that are much larger than the originals since these methods focus on minimizing the sequence length *k*, not the module size. Techniques for optimizing module size and trade-offs between module size and sequence length need to be developed. Methods are also needed for designing partially *k*-transparent modules and analyzing the relationship between module size and the amount of transparency (transparency index). Our theoretical results (Section 3.3), suggest that some modules can be made quite transparent with small modifications.

The relationship between transparency index and the propagation of typical faults in practical circuits requires further study. Increasing transparency may provide diminishing returns for propagating these faults. Instead, there may be specific errors which should always be propagated. Finally, it would be useful to compare the overhead of *k*-transparent design techniques with other, more common design techniques such as scan design and the routing of test points to determine when each is most appropriate. Transparency should be enhanced only when necessary and always in the most efficient way. The combination of a powerful hierarchical test generation program that processes precomputed test sets, and design techniques that provide sufficient transparency only when needed, should lead to a comprehensive testing method based on precomputed tests that simultaneously enhances the productivity of designers and improves the quality of the circuits they design.

APPENDICES

APPENDIX A

Mathematical Concepts

This appendix reviews mathematical concepts that are used in the propagation theory described in Chapter III.

A.1. Partition Theory

A relation $\equiv$ on a set S is an equivalence relation if and only if it is

- *reflexive*, that is, $x \equiv x$ for all $x \in S$
- *transitive*, that is, if $x_1 \equiv x_2$ and $x_2 \equiv x_3$ then $x_1 \equiv x_3$ for all $x_1, x_2, x_3 \in S$
- *symmetric*, that is, $x_1 \equiv x_2$ if and only if $x_2 \equiv x_1$ for all $x_1, x_2 \in S$.

A partition π on a set S is a collection of disjoint subsets of S called *blocks*, whose union is S. If $s \in S$, then $B_\pi(s)$ is the block that contains s. Two elements $s_1, s_2 \in S$ are equivalent, denoted $s_1 \equiv s_2(\pi)$, if and only if $B_\pi(s_1) = B_\pi(s_2)$. If R is an equivalence relation on S, then the set of equivalence classes defines a partition π on S, and vice versa. The partition consisting of all singleton blocks is the *zero partition* and the partition consisting of a single block containing all elements of S is called the *unit partition*.

Let π_1 and π_2 be partitions on a set S. The *intersection* $\pi_1 \cap \pi_2$ is the partition on S such that for any two elements $s, t \in S$, $s \equiv t(\pi_1 \cap \pi_2)$ if and only if $s \equiv t(\pi_1)$ and $s \equiv t(\pi_2)$ [44]. This operation can be computed by intersecting (using set intersection) each block of π_1 with every block of π_2. Let $S = \{0, 1, 2, 3, 4\}$, $\pi_1 = \{\{0, 1\}, \{2, 3\}, \{4\}\}$ and $\pi_2 = \{\{0, 1\}, \{2\}, \{3, 4\}\}$, then $\pi_1 \cap \pi_2 = \{\{0, 1\}, \{2\}, \{3\}, \{4\}\}$.

Let π_1 and π_2 be partitions on a set S. The *union* $\pi_1 \cup \pi_2$ is the partition on S such that two elements of S, s and t are equivalent, ie. $s \equiv t(\pi_1 \cup \pi_2)$, if and only if there exists a chain in S, $s = s_0, s_1, \ldots, s_n = t$ for which either $s_i \equiv s_{i+1}(\pi_1)$ or $s \equiv t(\pi_2)$, $0 \le i \le n-1$ [44]. The union operation can be computed by the procedure shown in Figure A.1. This procedure chains

```
Compute_Union( $\pi_1$, $\pi_2$ )
{
    $\pi_1 \cup \pi_2 = \varnothing$;
    for ($s \in S$) $B_1(s) = B_{\pi_1}(s) \cup B_{\pi_2}(s)$;
    $i = 1$;
    do {
        for ($s \in S$)
            $B_{i+1}(s) = B_i(s) \cup \{B | B$ is a block of $\pi_1$ or $\pi_2$ and $(B \cap B_i(s)) \neq \varnothing\}$;
        if ($B_{i+1}(s) == B_i(s)$) add $B_i(s)$ to $\pi_1 \cup \pi_2$ and delete all $s \in B_i(s)$ from $S$;
        $i = i + 1$;
    } while ($S \neq \varnothing$);
    return( $\pi_1 \cup \pi_2$ );
}
```

Figure A.1. Algorithm for computing partition union.

together blocks from π_1 and π_2 that have elements in common. Let $S = \{0, 1, 2, 3, 4\}$, $\pi_1 = \{\{0, 1\}, \{2, 3\}, \{4\}\}$, and $\pi_2 = \{\{0, 1\}, \{2\}, \{3, 4\}\}$, as in the intersection example above, then $\pi_1 \cup \pi_2 = \{\{0, 1\}, \{2, 3, 4\}\}$.

If π_1 and π_2 are two partitions on a set S, we say that π_2 *is greater than or equal to* π_1, denoted $\pi_1 \leq \pi_2$, if and only if every block of π_1 is contained in π_2. A binary relation R on a set S is called a *partial ordering* of S, denoted $(S, \leq)$, if and only if R has the following properties:

1. Reflexive: $a R a$ for all $a \in S$,
2. Transitive: $a R b$ and $b R c$ implies $a R c$,
3. Antisymmetric: $a R b$ and $b R a$ implies $a = b$.

The set of all partitions on a set S, together with the ordering relation $\leq$ is a partial ordering [44].

Let $(S, \leq)$ be a partially ordered set, and let P be a subset of S; then an element $s \in S$ is the *least upper bound* (*lub*) of P if and only if for every $p \in P$, $p \leq s$, and for every $p \in P$, $p \leq s'$ implies that $s \leq s'$. Similarly, an element $s \in S$ is the *greatest lower bound* (*glb*) of P if and only if for every $p \in P$, $s \leq p$, and for every $p \in P$, $s \leq p$ implies that $s' \leq s$.

A *lattice* is a partially ordered set that has a *lub* and a *glb* for every pair of elements. The set of all partitions on a set S is a lattice, where for any two partitions π_1 and π_2, $glb(\pi_1, \pi_2) = \pi_1 \cap \pi_2$, and $lub(\pi_1, \pi_2) = \pi_1 \cup \pi_2$ [44]. A lattice can also be characterized as an algebra, $L = (S, \cap, \cup)$, where S is a nonempty set of lattice elements, and $\cap$ and $\cup$ are binary operations satisfying four basic postulates for any $x, y, z \in S$.

1. $x \cap x = x$ and $x \cup x = x$
2. $x \cap y = y \cap x$ and $x \cup y = y \cup x$
3. $x \cap (y \cap z) = (x \cap y) \cap z$ and $x \cup (y \cup z) = (x \cup y) \cup z$
4. $x \cap (x \cup y) = x$ and $x \cup (x \cap y) = x$.

These four postulates are known respectively as the *idempotent*, *commutative*, *associative*, and *absorption* laws. If $L = (S, \cap, \cup)$ is a finite lattice, then it has a *least* and a *greatest* element denoted 0 and 1 respectively. Thus for all $s \in S$, $s \le 1$, $0 \le s$, $s \cap 1 = s$, and $s \cup 0 = s$. The zero partition and the unit partition given above are the least and greatest elements respectively in the lattice of partitions.

A *Boolean algebra* is a lattice L that also obeys the distributive law, that is., for $x, y, z \in L$ $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$ and $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$, and in which each element has a unique complement, that is, for all $x \in L$, there is a unique element $x' \neq x$ such that $x \cap x' = 0$ and $x \cup x' = 1$. The lattice of partitions lacks these properties and is consequently not as special and well-studied as the Boolean algebras. Nevertheless, a number of useful theorems have been derived for general lattices [14, 42].

Let $L_1 = (S_1, *, +)$ and $L_2 = (S_2, *, +)$ be two algebras with sets S_1 and S_2 respectively, and binary operations * and +. L_1 is *homomorphic* to L_2 if and only if there exists an onto mapping $h: S_1 \to S_2$, such that $h(x*y) = h(x) * h(y)$ and $h(x+y) = h(x) + h(y)$, for any two elements $x, y \in S_1$. In other words, both L_1 and L_2 have the same behavior with respect to the operations * and +; the homomorphism is said to preserve the operations. The algebras L_1 and L_2 are said to be *isomorphic* if h is also one-to-one. In this case, L_1 and L_2 are identical except for the names of the elements.

We can depict the ordering relation in a lattice L by means of a graph, called a *Hasse diagram* [55], whose vertices are elements of L. Vertex a is drawn in a higher level than vertex b whenever $b < a$, that is, $b \le a$ and $a \neq b$ (in the case of module functions, $a \not\equiv b$). Vertices a and b are adjacent if there is no element c, such that $b < c < a$. If $\Omega = \{1, 2, 3, 4\}$, then the set of all partitions on Ω, S_Ω, is

$\pi(1) = \{\{1,2,3,4\}\}$ $\pi_1 = \{\{1,2,3\}, \{4\}\}$ $\pi_2 = \{\{1,2,4\}, \{3\}\}$

$\pi_3 = \{\{1,3,4\},\{2\}\}$	$\pi_4 = \{\{2,3,4\},\{1\}\}$	$\pi_5 = \{\{1,2\},\{3,4\}\}$
$\pi_6 = \{\{1,3\},\{2,4\}\}$	$\pi_7 = \{\{1,4\},\{2,3\}\}$	$\pi_8 = \{\{1,2\},\{3\},\{4\}\}$
$\pi_9 = \{\{2,3\},\{1\},\{4\}\}$	$\pi_{10} = \{\{1,3\},\{2\},\{4\}\}$	$\pi_{11} = \{\{2,4\},\{1\},\{3\}\}$
$\pi_{12} = \{\{1,4\},\{2\},\{3\}\}$	$\pi_{13} = \{\{3,4\},\{1\},\{2\}\}$	$\pi(0) = \{\{1\},\{2\},\{3\},\{4\}\}$

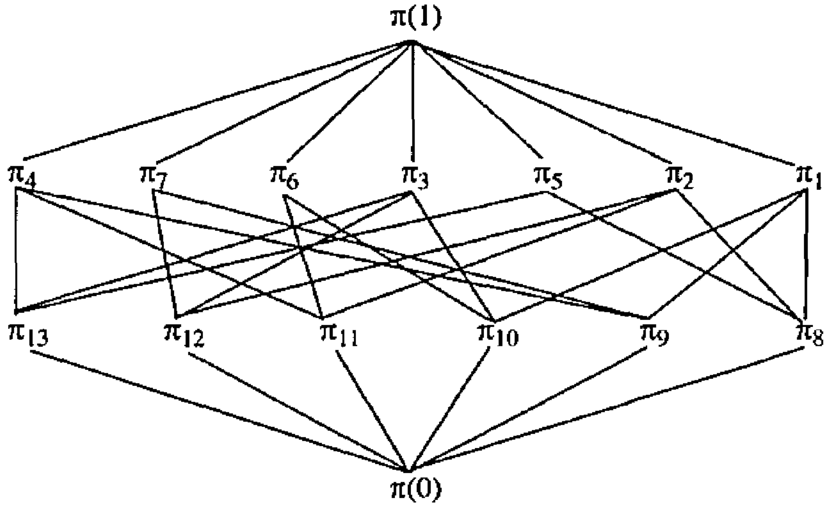


Figure A.2. Hasse diagram for the lattice of partitions π_Ω on $\Omega = \{1, 2, 3, 4\}$.

The partitions $\pi(0)$ and $\pi(1)$ are the zero and unit partitions, respectively. The Hasse diagram for $L_\Omega = (S_\Omega, \cap, \cup)$, is shown in Figure A.2. An interior vertex a represents an element π_a which is the intersection of two elements $\pi_b, \pi_c \in S_\Omega$, where $\pi_a < \pi_b$, and $\pi_a < \pi_c$. The elements π_b and π_c are themselves represented by two vertices above and adjacent to a. For example, in Figure A.2, π_{12} is adjacent to superior vertices π_7 and π_3 and the intersection of the partitions π_7 and π_3 is

$$\pi_7 \cap \pi_3 = \{\{1,4\},\{2,3\}\} \cap \{\{1,3,4\},\{2\}\} = \{\{1,4\},\{3\},\{2\}\} = \pi_{12}\,.$$

The vertext π_2 is also adjacent to π_{12}; the intersection of any two of π_7, π_3, and π_2 is π_{12}. Note that π_7, π_3, and π_2 are all at the same level in the Hasse diagram and are thus incomparable.

A.2. Series-Parallel Graphs

A graph G is a set of nodes $N = \{n_1, n_2, \ldots, n_{m_1}\}$ and edges $E = \{e_1, e_2, \ldots, e_{m_2}\}$. Each edge is a pair $e_i = \{n_j, n_k\}$ of nodes, which implies that e_i links n_j and n_k. A graph is

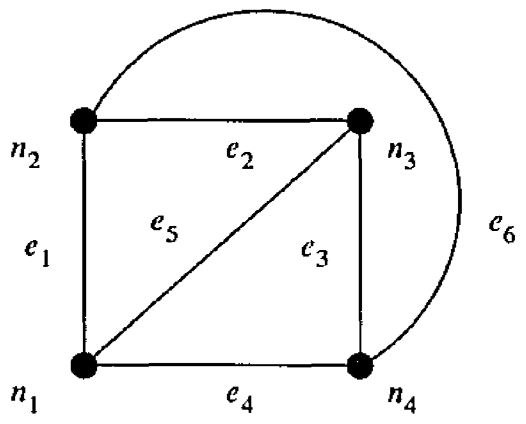


Figure A.3. Example of a non-confluent graph.

directed if every edge is an ordered pair, $e_i = (n_j, n_k)$, which implies that the edge begins on n_j and ends on n_k. Two nodes are *adjacent* if there is an edge between them and two edges are adjacent if they share a node. A *circuit* is a sequence of adjacent nodes and edges $\left(n_{k_1}, e_{i_1}, n_{k_2}, e_{i_2}, \ldots, e_{i_m}, n_{k_1}\right)$ that begins and ends on the same node. Each e_{i_j} specifies which edge connects adjacent nodes n_{k_j} and $n_{k_{j+1}}$ in the circuit (since more than one edge may connect them). A circuit C imposes a direction on its edges: each e_{i_j} begins on n_{k_j} and ends on $n_{k_{j+1}}$. Two edges e_i and e_j are *confluent* if there do not exist two circuits C_1 and C_2, each containing edges e_i and e_j, such that the direction imposed on exactly one of the two edges is reversed between C_1 and C_2 [23]. A graph is confluent if all its edges are confluent.

Consider the graph G in Figure A.3, and consider the circuits $C_1 = (n_1, e_1, n_2, e_2, n_3, e_3, n_4, e_4, n_1)$, and $C_2 = (n_1, e_1, n_2, e_6, n_4, e_3, n_3, e_5, n_1)$. The direction of both C_1 and C_2 through edge $e_1 = \{n_1, n_2\}$ is the same, namely (n_1, n_2). However, the direction of C_1 through e_3 is (n_3, n_4), while the direction of C_2 through e_3 is (n_4, n_3). Therefore, G is not confluent. In electrical circuit theory, where edges represent resistors or batteries, G is called a Wheatstone bridge. No graph with an embedded Wheatstone bridge is confluent [23]. On the other hand, directed graphs are all confluent, because the directions of the edges are already fixed; circuits must conform to the fixed direction in order to be valid. If a graph has no circuit, then it is trivially confluent.

According to Duffin [23], every edge in a confluent graph is part of a series-parallel connection. Most datapath circuits can be modeled as directed graphs, since most modules are unidirectional. Even circuits with tristate buses can often be modeled as directed graphs for particular operation cycles [12]. Therefore, most datapath circuits can be analyzed as a set of edges connected in series and parallel.

APPENDIX B

Propagation Algebra

The propagation algebra $(S, °, \#)$ describes the behavior of propagation functions in the same way that a Boolean algebra describes Boolean logic. The properties which define this algebra are listed in Table B.1 which repeats Table 3.1. In this appendix, we show that the transparency algebra is consistent, that is, there is at least one system that has these properties. Any other algebra that also has these properties is propagation algebra. All properties in Table B.1, except those marked with a *, are independent, that is, they cannot be derived from any other properties. We will indicate how this is proved. Note that only three properties are marked. This is due to the fact that there is no unique inverse for either # or °, a key property used in deriving new algebraic theorems. In defining the properties in Table B.1 we make use of the following two relations.

Definition B.1: if a, and b are in S, then $a \le b$ if and only if $a\#b \equiv a$.

Definition B.2: if a and b are in S, then $a \equiv b$ if and only if $a \le b \cap b \le a$.

B.3. Consistency

To show that the propagation algebra is consistent, we will prove that the properties of the algebra are satisfied by the set of all propagation functions and the series and parallel connection operations introduced in Chapter III. We begin by showing that Definition A.1 is equivalent to the definition of partial ordering given earlier. Definition A.2 was discussed in Chapter III. Let S be the set of all propagation functions, and let $a, b \in S$. In the propagation algebra, the relation $a \le b$ implies that every α of a is contained in some α of b.

$$\begin{array}{lll} a\#b \equiv a & \text{if and only if} & (\alpha_{ai} \cap \alpha_{bj}) = \alpha_{ai} \quad \text{for all } i \text{ and } j \\ & \text{if and only if} & \alpha_{ai} \subseteq \alpha_{bj} \qquad\qquad\quad \text{for all } i \text{ and } j. \\ & \text{if and only if} & a \le b \end{array}$$

Name		Property
Closure	1	$a\#b$ is in S whenever a and b are.
Identity	2a 2b	There is an element 1 in S such that $1\#a = a$ for every element a in S. There is an element 0 in S such that $0\ a = a$ and $a°0 = a$ for every element a in S.
Distributivity	3	$a°(b\#c) = (a°b)\#(a°c)$ whenever $a, b, c, (b\#c), (a°b), (a°c), a°(b\#c)$, and $(a°b)\#(a°c)$ are in S.
Commutativity	4	$a\#b \cong b\#a$ whenever $a, b, a\#b$ and $b\#a$ are in S.
Idempotence	5	$a\#a \cong a$ whenever a and $a\#a$ are in S.
Absorption	6	$a\#(a°b) \cong a$ whenever $a, b, a°b$, and $a\#(a°b)$ are in S.
Associativity	7a 7b	$a\#(b\#c) \cong (a\#b)\#c$ whenever $a, b, c, (a\#b), (b\#c), a\#(b\#c)$, and $(a\#b)\#c$ are in S. $a°(b°c) = (a°b)°c$ whenever a, b, c, $(a°b), (b°c), a°(b°c)$, and $(a°b)°c$ are in S.
Miscellaneous	8* 9* 10*	$a\#b \leq a$ and $a\#b \leq b$ $0\#a \cong 0$ $a \leq a\ b$

Table B.1: Properties of transparency algebra.

We will analyze each of the properties in order.

1. **Closure.** $a\#b$ is in S whenever a and b are.

   It is clear that # is closed, since it is based on partition intersection.

2. **Identity.**

   a. There is an element 1 in S such that $1\#a \cong a$ for every element a in S.

   b. There is an element 0 in S such that $0°a = a$ and $a°0 = a$ for every element a in S.

   For part a,

$$\begin{aligned} 1\#a &= \{\left(0, 1, \ldots, 2^{|Z_D|} - 1; \left(0, 1, \ldots, 2^{|Z_D|} - 1\right)\right)\} \# \{(\alpha_1;\beta_1), \ldots, (\alpha_n;\beta_n)\} \\ &= \{\left(\alpha_1; (0, \beta_1), (1, \beta_1), \ldots, \left(2^{|Z_D|} - 1, \beta_1\right)\right), \\ &\qquad \left(\alpha_2; (0, \beta_2), (1, \beta_2), \ldots, \left(2^{|Z_D|} - 1, \beta_2\right)\right), \\ &\qquad \vdots \\ &\qquad \left(\alpha_n; (0, \beta_n), (1, \beta_n), \ldots, \left(2^{|Z_D|} - 1, \beta_n\right)\right)\} \\ &\cong a \end{aligned}$$

For part b,

$$0°a = \{(0;0), (1;1), \ldots, \left(2^{|Z_D|}-1;2^{|Z_D|}-1\right)\} \quad \{(\alpha_1;\beta_1), \ldots, (\alpha_n;\beta_n)\}$$

If α_i contains j, then $0 \; a$ contains $(j;\beta_i)$ for all $j \in \alpha_i$. This implies that $0°a$ combines $(\alpha_i;\beta_i)$ for all i.

Finally, consider

$$a°0 = \{(\alpha_1;\beta_1), \ldots, (\alpha_n;\beta_n)\} \quad \{(0;0), \ldots, \left(2^{|Z_D|}-1\right)\}$$

If $\beta_i = j$, then $(\alpha_i;j)$, $(\alpha_i;\beta_i) \in a \; 0$ for all i. This implies that $a°0 = a$.

3. **Distributivity.** $a°(b\#c) = (a°b)\#(a°c)$ whenever a, b, c, $(b\#c)$, $(a°b)$, $(a°c)$, $a°(b\#c)$, and $(a°b)\#(a°c)$ are in S.

$$\begin{aligned}
b\#c &= \{(\alpha_{b1}\cap\alpha_{c1};(\beta_{b1},\beta_{c1})), \ldots, \left(\alpha_{bn_b}\cap\alpha_{cn_c};\left(\beta_{bn_b},\beta_{cn_c}\right)\right)\} \\
a°(b\#c) &= \{(\alpha_{a1};(\beta_{bi},\beta_{cj})\,|\beta_{a1}\in\alpha_{bi}\cap\alpha_{cj}), \ldots\} \\
a°b &= \{(\alpha_{a1};\beta_{bi}|\beta_{a1}\in\alpha_{b1}), \ldots\} \\
a°c &= \{(\alpha_{a1};\beta_{ci}|\beta_{a1}\in\alpha_{c1}), \ldots\} \\
(a°b)\#(a°c) &= \{(\alpha_{a1}\cap\alpha_{a1};(\beta_{bi},\beta_{cj})\,|\beta_{a1}\in\alpha_{bi}\cap\alpha_{cj}), \ldots\} \\
&= \{(\alpha_{a1};(\beta_{bi},\beta_{cj})\,|\beta_{a1}\in\alpha_{bi}\cap\alpha_{cj}), \ldots\}
\end{aligned}$$

4. **Commutativity.** $a\#b \equiv b\#a$ whenever a, b, $a\#b$ and $b\#a$ are in S.
   This is true since partition intersection is commutative.
5. **Idempotence.** $a\#a \equiv a$ whenever a and $a\#a$ are in S.

   This is true since partition intersection is idempotent
6. **Absorption.** $a\#(a°b) \equiv a$ whenever a, b, $a°b$, and $a\#(a°b)$ are in S.

$$\begin{aligned}
a°b &= \{(\alpha_{a1};\beta_{bi}|\beta_{a1}\in\alpha_{bi}), \ldots\} \\
a\#(a°b) &= \{(\alpha_{a1}\cap\alpha_{a1};(\beta_{a1},\beta_{bi})\,|\beta_{a1}\in\alpha_{bi}), \ldots\} \\
&\equiv a
\end{aligned}$$

7. **Associativity.**
   a. $a\#(b\#c) \equiv (a\#b)\#c$ whenever a, b, c, $(a\#b)$, $(b\#c)$, $a\#(b\#c)$, and $(a\#b)\#c$ are in S.

b. $a°(b°c) = (a°b)°c$ whenever a, b, c, $(a°b)$, $(b°c)$, $a°(b°c)$, and $(a°b)°c$ are in S.

These follow directly from the definitions of # and .

8. $a\#b \leq a$ and $a\#b \leq b$.

$a\#b \leq a$	if and only if	$a\#b\#a \cong a\#b$	definition A.1
	if and only if	$a\#a\#b \cong a\#b$	commutation (property 4)
	if and only if	$a\#b \cong a\#b$	idempotence (property 5)
$a\#b \leq b$	if and only if	$a\#b\#b \cong a\#b$	definition A.1
	if and only if	$a\#b \cong a\#b$	idempotence (property 5)

9. $0\#a \cong 0$.

This is clear from property 8.

10. $a \leq a°b$.

To prove this note that $a \leq a°b$ if and only if $a\#(a°b) \cong a$ by definition A.2, and $a\#(a°b) \cong a$ is true by property 6.

B.4. Independence

We have shown that the propagation algebra is consistent with respect to the algebra formed by propagation functions and the series and parallel connection operations. We have also shown that properties 8, 9, and 10 are not independent; they can be derived from some of the other properties. The other nine properties are independent. To show this, we need to demonstrate for each property p_i, that there are systems in which property p_i is not true, but all other properties are true. We will demonstrate this for one property–closure. The proofs for the other properties are similar, but lengthy. Consider the algebra defined by the set $\{0,1\}$ and operators # and as given in Figure B.4. It is clear that the algebra is not closed. Tables B.2 through B.4 demonstrate that all other properties are true.

#	0	0
0	X	0
1	0	1

°	0	1
0	0	1
1	1	1

Figure B.4. Algebra for proving that closure is independent.

a	b	c	$a° (b\#c)$	$(a°b) \# (a°c)$	$a\#b$	$b\#a$	$a\#a$	$a\# (b°c)$
0	0	0	–	—	–	–	–	–
0	0	1	0	0	–	–	–	–
0	1	0	0	0	0	0	–	0
0	1	1	1	1	0	0	–	0
1	0	0	–	1	0	0	1	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	1	1	1
1	1	1		1	1	1	1	1

Table B.2: Truth table proving that closure is independent.

a	b	c	$a\# (b\#c)$	$(a\#b) \#c$	$a° (b°c)$	$(a°b) °c$	$0°a$
0	0	0	–	–	0	0	0
0	0	1	–	–	1	1	0
0	1	0	–	–	1	1	0
0	1	1	0	0	1	1	0
1	0	0	–	–	1	1	1
1	0	1	0	0	1	1	1
1	1	0	0	0	1	1	1
1	1	1	1	1	1	1	1

Table B.3: Truth table proving that closure is independent.

a	b	c	$a°0$	$1\#a$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Table B.4: Truth table proving that closure is independent.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. M. S. Abadir and M. A. Breuer. "Test Schedules for VLSI Circuits Having Built-in Test Hardware," *IEEE Trans. Comput.*, Vol. 35, pp. 361–367, April 1985.

2. M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*, Computer Science Press, New York, NY, 1990.

3. Advanced Micro Devices. "The Am2910, A Complete 12-bit Microprogram Sequence Controller," in *AMD Data Book*, Sunnyvale, CA, AMD Inc., 1978.

4. A. Akritas. *Elements of Computer Algebra with Applications*, John Wiley and Sons, New York, NY, 1989.

5. V. D. Agrawal, K. T. Cheng, and P. Agrawal. "CONTEST: A Concurrent Test Generator for Sequential Circuits," *Proc. 25th Design Automation Conf.*, pp. 84–89, June 1988.

6. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

7. H. Ando. "Testing VLSI With Random Access Scan," in *Comcon 80*, 1980.

8. D. R. Barstow. *Knowledge Based Program Construction*. Elsevier North Holland, Inc., New York, NY, 1979.

9. F. Beenker et al. "A Testability Strategy for Silicon Compilers," in *Proc. Int. Test Conf.*, pp. 660–668, 1989.

10. R. G. Bennetts et al. "A Modular Approach to Test Sequence Generation for Large Digital Networks," *Digital Processes*, pp. 3–23, 1975.

11. M. Bershteyn. "Sequential Test Generation Tool for Embedded Cells," in *Proc. Workshop on Hierarchical Test Generation*, Blacksburg, VA, August 8-11, 1993.

12. D. Bhattacharya and J. P. Hayes. *Hierarchical Modeling for VLSI Circuit Testing*, Kluwer Academic Publishers, Boston, MA, 1990.

13. D. Bhattacharya, B. T. Murray, and J. P. Hayes. "High-Level Test Generation for VLSI," *IEEE Computer*, Vol. 22, pp. 16–24, April 1989.

14. G. Birkhoff. *Lattice Theory*, American Mathematical Society, Providence, Rhode Island, 1960.

15. D. Brahme and J. A. Abraham. "Functional Testing of Microprocessors," *IEEE Trans. Comput.*, Vol. 33, pp. 475–485, 1984.

16. M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, Rickville, Maryland, 1976.

17. R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, Vol. 35, pp. 677–691, August 1986.

18. S. J. Chandra and J. H. Patel. "A Hierarchical Approach to Test Vector Generation" in *Proc. 24th Design Automation Conf.*, pp. 495–501, 1987.

19. C-H Chen, T. Karnik, and D. G. Saab. "Structural and Behavioral Synthesis for Testability Techniques," *IEEE Trans. Computer-Aided Design*, Vol. 13, pp. 777–785, June 1994.

20. D. R. Coelho. *The VHDL Handbook*, Kluwer Academic Publishers, Boston, MA, 1989.

21. D. Cohen. "On Holy Wars and a Plea for Peace," *IEEE Computer*, Vol. 14, pp. 48–54, October 1981.

22. J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*, Academic Press, San Diego, CA, 1988.

23. R. J. Duffin. "Topology of Series-Parallel Networks," *Journal of Mathematical Analysis and Applications*, Vol. 10, pp. 303–318, 1965.

24. J. R. Durbin. *Modern Algebra*, John Wiley & Sons, New York, NY, 1979.

25. E. B. Eichelberger and T. W. Williams. "A Logic Design Structure for LSI Testability," in *Proc. 14th Design Automation Conf.*, pp. 462–468, 1977.

26. M. Emori et al. "ASIC CAD System Based on Hierarchical Design-for-Testability," in *Proc. IEEE Int. Test Conf.*, pp. 404–409, 1990.

27. *Epoch Finesse User and Reference Manual*, Cascade Design Automation, Bellevue, WA, 1993.

28. K. E. Erickson. "A New Operation for Analyzing Series-Paralled Networks," *IRE Trans. Circuit Theory*, pp. 124–126, March 1959.

29. F. J. Ferguson and J. P. Shen. "A CMOS Fault Extractor for Inductive Fault Analysis," *IEEE Trans. Computer-Aided Design*, Vol. 7, pp. 1181–1194, November, 1988.

30. S. Freeman. "Test Generation for Data-Path Logic: The F-Path Method." *IEEE Journal of Solid-State Circuits*, Vol. 23, pp. 421–427, April 1988.

31. A. D. Friedman. "Easily Testable Iterative Systems," *IEEE Trans. Comput.*, Vol. 22, No. 12, pp. 1061–1064, December, 1973.

32. T. E. Fuhrman. "Industrial Extensions to University High Level Synthesis Tools: Making It Work in the Real World," in *Proc. 28th Design Automation Conf.*, pp. 520–525, 1991.

33. H. Fujiwara. *Logic Testing and Design for Testability.* The MIT Press, Cambridge, MA., 1985.

34. H. Fujiwara and T. Shimono. "On the Acceleration of Test Generation Algorithms," in *Proc. 13th Fault-Tolerant Computing Symp.*, pp. 98–105, June 1983.

35. S. Funatsu, N. Wakatsuki, and T. Arima. "Test Generation Systems in Japan," in *Proc. 12th Design Automation Conf.*, pp. 77–84, 1980.

36. S. Gai, F. Somenzi, and E. Ulrich. "Advances in Concurrent Multilevel Simulation," *IEEE Trans. Computer-Aided Design*, Vol. 6, pp. 1006–1012, November 1987.

37. D. D. Gajski. "Silicon compilation," *VLSI Systems Design*, pp. 48–64, November 1985.

38. K. O. Geddes. "On the Design and Performance of the Maple System," in *Proc. 1984 MACSYMA Users' Conference*, pg. 199, 1984.

39. A. Ghosh, S. Devadas, and A. R. Newton. "Sequential Test Generation at the Register-Transfer and Logic Levels," in *Proc 27th Design Automation Conf.*, pp. 580-586, 1990.

40. P. Goel. "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits." *IEEE Trans. Comput.*, Vol. 30, pp. 215–222, March, 1981.

41. P. Goel. "Test Generation Costs Analysis and Projections," in *Proc. 17th Design Automation Conf.*, pp. 77–84, 1980.

42. G. Grätzer. *General Lattice Theory*, Academic Press, New York, NY., 1978.

43. F. Harary. *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

44. J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

45. J. P. Hayes. "A Calculus for Testing Complex Digital Systems," in *Proc. 10th Fault-Tolerant Computing Symp.*, pp. 115–120, Kyoto, Japan, 1980.

46. J. P. Hayes. "Digital Simulation with Multiple Logic Values," *IEEE Trans. Computer-Aided Design*, Vol. 5, pp. 274–283, April 1986.

47. J. P. Hayes. "Fault Modeling." *IEEE Design and Test*, pp. 88–95, April 1985.

48. J. P. Hayes. *Introduction to Digital Logic Design*, Addison-Wesley, Reading, MA, 1993.

49. E. V. Huntington. "Sets of Independent Postulates for the Algebra of Logic," *Trans. American Mathematical Society*, Vol. 5, pp. 288–309, 1904.

50. O. H. Ibarra and S. K. Sahni. "Polynomially Complete Fault Detection Problems." *IEEE Trans. Comput.*, Vol. 24, pp. 242–249, March 1975.

51. V. Immaneni and S. Raman. "Direct Access Test Scheme—Design of Block and Core Cells for Embedded ASICs," in *Proc. IEEE Int. Test Conf.*, pp. 488–492, 1990.

52. T. Kirkland and M. R. Mercer. "A Topological Search Algorithm for ATPG," in *Proc. 24th Design Automation Conf.*, pp. 502–508, 1987.

53. K. Knight. "Unification: A Multidisciplinary Survey." *ACM Computing Surveys*, Vol. 21, pp. 23–124, March 1989.

54. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Second Edition, Addison Wesley, Reading, MA, 1981.

55. Z. Kohavi. *Switching and Finite Automata Theory, 2nd Edition*. McGraw-Hill, New York, NY, 1978.

56. J-H. Kong and S. A. Szygenda. "MixMOS: a mixed-level simulator for digital MOS circuits using a new algebraic approach," *Computer-Aided Design*, Vol. 22, pp. 618–632, December 1990.

57. B. Krishnamurthy. "Hierarchical Test Generation: Can AI Help?," in *Proc. Int. Test Conf.*, pp. 694–700, 1987.

58. J. Lee and J. H. Patel. "An Architectural Level Test Generator for a Hierarchical Design Environment," in *Proc. 21st Fault-Tolerant Computing Symp.*, pp. 44–51, June 1991.

59. J. Lee and J. H. Patel. "An Architectural Level Test Generator Based on Nonlinear Equation Solving," *Journal of Electronic Testing*, Vol. 4, pp. 137–150, April 1993.

60. J. Lee. *Architectural Level Test Generation and Fault Simulation*. Ph.D. Thesis, University of Illinois, November 1992.

61. T-C Lee, N. K. Jha, and W. H. Wolf. "Behavioral Synthesis of Highly Testable Data Paths Under the Non-Scan and Partial Scan Environments," in *Proc. 30th Design Automation Conf.*, pp. 292–297, 1993.

62. Y. H. Levendel and P. R. Menon. "Test Generation Algorithms for Computer Hardware Description Languages," *IEEE Trans. Comput.*, Vol. 31, pp. 577–588, July 1982.

63. C. J. Lin and S. M. Reddy. "On Delay Fault Testing in Logic Circuits." *IEEE Trans. Computer-Aided Design*, Vol. 6, pp. 694–703, September 1987.

64. T. Lin and S. Y. H Su. "Functional Test Generation of Digital LSI/VLSI Systems Using Machine Symbolic Execution Technique," in *Proc. IEEE Int. Test Conf.*, pp. 660–668, 1984.

65. M. Majewski and S. Pichumani. "The Use of Silicon Compilation in the Design of a Gaussian Filter and a Template Matching Processor." *VLSI Systems Design*, pp. 20–25, October 1987.

66. M. Marhöfer. "An Approach to Modular Test Generation Based on the Transparency of Modules," in *Proc. IEEE CompEuro 87*, pp. 403–406, May 1987.

67. R. Marlett. "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," in *15th Design Automation Conf.*, pp. 335–339, June 1978.

68. B. T. Murray and J. P. Hayes. "Hierarchical Test Generation Using Precomputed Tests for Modules," in *Proc. IEEE Int. Test Conf.*, pp. 221–229, 1988.

69. B. T. Murray and J. P. Hayes. "Hierarchical Test Generation Using Precomputed Tests for Modules," *IEEE Trans. Computer-Aided Design*, Vol. 9, pp. 594–603, 1990.

70. B. T. Murray and J. P. Hayes. "Test Propagation Through Modules and Circuits," in *Proc. IEEE Int. Test Conf.*, pp. 748–757, 1991

71. M. Mukaidono. "A Set of Independent and Complete Axioms for a Fuzzy Algebra (Kleene Algebra)," in *Proc. Eleventh International Symposium on Multiple-Valued Logic*, pp. 27–34, 1981.

72. T. Niermann and J. H. Patel. "HITEC: A Test Generation Package for Sequential Circuits," in *Proc. European Design Automation Conf.*, pp. 214–218, Feb. 1991.

73. T. Ogihara et al. "Testable Design and Support Tool for Cell Based Test," in *Proc. IEEE Int. Test Conf.*, pp. 1056–1071, 1990.

74. R. Pavel, M. Rothstein, and J. Fitch. "Computer Algebra," *Scientific American*, pp. 136–152, December 1981.

75. S. Rao, B. Pan, and J. R. Armstrong. "Hierarchical Test Generation for VHDL Behavioral Models," in *Proc. European Design Automation Conf.*, Feb. 1993.

76. G. D. Robinson. "Hitest—Intelligent Test Generation," In *Proc. IEEE Int. Test Conf.*, pp. 311–323, 1983.

77. W. A. Rogers, J. F. Guzolek, and J. Abraham. "Concurent and Hierarchical Fault Simulation," *IEEE Trans. Computer-Aided Design*, Vol. 6, pp. 848–862, September, 1987.

78. J. P. Roth. *Computer Logic, Testing, and Verification*, Computer Science Press, Potomac, Maryland, 1980.

79. J. P. Roth. "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, Vol. 10, pp. 278–291, July 1966.

80. K. Roy and J. A. Abraham. "High Level Test Generation Using Data Flow Descriptions," in *Proc. IEEE European Design Automation Conf.*, pp. 480–484, 1990.

81. D. G. Saab et al. "CHAMP: Concurrent Hierarchical And Multilevel Program for Simulation," in *Proc. Int. Conf. Computer-Aided Design*, pp. 246–249, 1988.

82. K. Sakashita et al. "Cell-Based Design Method," in *Proc. IEEE Int. Test Conf.*, pp. 909–916, 1989.

83. A. Samad and M. Bell. "Automating ASIC Design-for-Testability—the VLSI Test Assistant," in *Proc. IEEE Int. Test Conf.*, pp. 819–828, 1989.

84. T. M. Sarfert et al., "Hierarchical Test Pattern Generation Based on High-Level Primitives," in *Proc. IEEE Int. Test Conf.*, pp. 1016–1026, Sept. 1989.

85. M. H. Schultz, E. Trischler, and T. M. Sarfert. "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System." *IEEE Trans. Computer-Aided Design*, Vol. 7, pp. 126–137, January 1988.

86. C. E. Shannon, *Collected Works*, IEEE Press, New York, NY, 1993.

87. M. Shirley et al. "A Synergistic combination of Test Generation and Design for Testability," in *Proc. IEEE Int. Test Conf.*, pp. 701–711, 1987.

88. M. H. Shirley. *Generating Circuit Tests by Exploiting Designed Behavior.* Ph.D. Thesis, MIT Artificial Intelligence laboratory, December 1988.

89. N. Singh. *An Artificial Intelligence Approach to Test Generation.* Kluwer Academic Publishers, Boston, 1987.

90. F. Somenzi et al. "Testing Strategy and Technique for Macro-Based Circuits." *IEEE Trans. Comput.*, Vol. 34, No. 1, pp. 85–90, January 1985.

91. E. Sternheim, R. Singh, and Y. Trivedi. *Design with Verilog HDL*, Automata Publishing Co., Cupertino, CA, 1990.

92. J. H. Stewart. "Future Testing of Large LSI Circuit Cards" in *Proc. Semiconductor Test Symp.*, pp. 6–17, 1977.

93. C-C Su and C. R. Kime. "Multiple Path Sensitization for Hierarchical Circuit Testing," in *Proc. IEEE Int. Test Conf.*, pp. 152–161, 1990.

94. S. M. Thatte and J. A. Abraham. "Test Generation for Microprocessors." *IEEE Trans. Comput.*, Vol. 29, pp. 429–441, June 1980.

95. D. E. Thomas et al. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

96. E. G. Ulrich and T. G. Baker. "Concurrent Simulation of Nearly Identical Digital Networks," *IEEE Computer*, Vol. 7, pp. 39–44, April 1974.

97. E. G. Ulrich et al. "The Comparative and Concurrent Simulation of Discrete-Event Experiments," *Journal of Electronic Testing*, Vol. 3, pp. 107–118.

98. R. L. Wadsack. "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits." *The Bell System Technical Journal*, Vol. 57, pp. 1449–1474, May-June 1978.

99. J. A. Waicukauski et al. "Fault Simulation for Structured VLSI," *VLSI Systems Design*, pp. 20–32, December, 1985.

100. T. W. Williams. "Sufficient Testing in a Self-Testing Environment," in *Proc. IEEE Int. Test Conf.*, pp. 167–172, 1984.

101. T. W. Williams and K. P. Parker. "Testing Logic Networks and Design for Testability," *IEEE Computer*, Vol. 12, pp. 9–21, No. 9, October 1979.

102. S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, MA, 1988.

103. R. Woudsma and A. Delaruelle. "The Design of DSP Components for the CD Digital Audio System Using Silicon Compilation Techniques," In *Proc. IEEE Custom Integrated Circuit Conf.*, San Diego, 1989.