TECHNIQUES FOR AUTOMATIC TEST KNOWLEDGE EXTRACTION FROM COMPILED CIRCUITS

BY

KURT HENRY THEARLING

B.S.E., University of Michigan, 1985
B.S.E., University of Michigan, 1985
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

TECHNIQUES FOR AUTOMATIC TEST KNOWLEDGE EXTRACTION FROM COMPILED CIRCUITS

Kurt Henry Thearling, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1990

In the past, research has shown that the use of high-level test knowledge can be used to greatly accelerate the test generation process. The problem was that no techniques were developed to extract this knowledge from a circuit. Typically, the only solution for a circuit designer was to manually extract the test knowledge. When designers are using sophisticated high-level synthesis tools (e.g., a silicon compiler), the designer may not be competent to extract this type of knowledge. In this thesis, solutions to the problem of automatically extracting this high-level knowledge from the structure of a compiled circuit are presented..

Two different types of knowledge are addressed. The first type of knowledge is a testability measure. We present solutions to the problem estimating the testability for circuits defined at a functional level. By using an information theoretic testability measure, the concepts of controllability and observability are captured. Instead of requiring exhaustive enumeration of the input space to compute the measure (as has been previously suggested), we presented two different methods for efficiently and accurately estimating the measure. In addition, we have present various applications of the measure, including automatic circuit partitioning and test point insertion.

The second type of knowledge is used in test generation. We describe techniques to automatically extract high-level test and DFT knowledge from the structure of compiled circuits. These techniques work autonomously and require no user intervention. This system has been implemented in a SUN workstation environment and is known as DELPHI. It operates on the high-level dataflow representation of a compiled circuit and generates the test knowledge in the form of lists of primary input assignments. Achieving both high levels of fault coverage and fast performance, DELPHI can extract test knowledge from both non-sequential and sequential circuits. When test knowledge extraction is unsuccessful, additional DFT knowledge is obtained to efficiently represent design for testability options. In those cases in which users are able to provide test knowledge, techniques to verify user-provided knowledge are described.

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Jacob Abraham for his encouragement and assistance during the development of this work. I would also like to thank Professors Janak Patel, Prith Banerjee, and Resve Saleh for serving on my committee.

I would especially like to thank my friends in the Computer Systems Group (a.k.a. the Center for Reliable and High-Performance Computing). A special thanks is owed to Marc Levitt, Paul Chen, Ralph Kling, Randy Brouwer, Tom Niermann, and Jeff Baxter.

Finally, I must thank my family for their love and support during my years as a graduate student.

1. INTRODUCTION

In recent years the complexity of VLSI circuits has continually increased. This can be credited to both better design tools and improvements in chip manufacturing technology. Since a chip manufacturer has an economic incentive not to sell defective parts, the ability to generate high-quality tests for circuits is very important. But with this increase in circuit complexity comes an increased cost paid when testing these circuits for defects. The increased complexity of the circuit designs increases the already large test generation search space (which is a known NP-Complete [1] problem). The solution to this problem is to reduce test generation complexity by directing and constraining the search.

Humans have often interacted with test generation systems to provide additional information which the test generator can use to speed up the process of producing test vectors [2]. Designers who are intimately knowledgeable about the operation of a circuit can provide valuable details which can greatly reduce the complexity of generating tests. By providing knowledge which specifies ways to justify and propagate any arbitrary value to any point in the circuit, a designer can reduce the problem of test generation to simple pattern matching and symbol replacement. In effect, this constrains the test generation search to a problem with no search (i.e., there is only one way to apply the test). Humans who “understand” the global behavior of a circuit will know how to obtain desired

values on signal paths in the circuit. If they detect a bottleneck which makes some desired value difficult to obtain, they can provide redesign options which will achieve the desired goal.

The increased use of silicon compilation [3] and other standard cell [4] design tools have changed the way many digital systems are designed. As a result, more and more systems are being designed at the functional level, with little gate level design being explicitly performed. Instead of creating a system out of gates (or even transistors), designers are using functional modules such as adders and multipliers. By understanding that designs are constructed using functional level components, appropriate techniques can be developed which efficiently represent knowledge about testing the circuit at the functional level.

In addition, silicon compilers allow users to specify their circuits at a high level, with the low level circuit details being handled by the complier. This design simplicity offers the ability to design custom circuits to persons who may not have the necessary technical background. The problem that many users now face is their own inability to deal with the problems associated with testing a circuit. End users using sophisticated design tools (especially silicon compilation tools) may be completely ignorant of the process of design and test. They understand what the circuit needs to do but they are unable to deal with the details of the design. Therefore, the silicon compiler must, in addition to generating a design, also generate tests for the circuit. Traditional approaches to test generation can be employed but as with noncompiled circuits, the increased design complexity

makes this difficult. High-level test knowledge will accelerate the process but the user cannot be relied on to provide it. Once again it falls to the silicon compiler to automatically generate the high-level test knowledge. The problem is that computers do not “understand” the operation of a circuit and to make a computer generate this type of information is difficult.

In the work presented in this thesis, new techniques are presented to generate test knowledge which will accelerate the production of individual tests for the circuit. This test knowledge is automatically extracted from the structure of a compiled circuit by a system called DELPHI, which has been implemented in a SUN workstation environment. It is not necessary for the user to interact with DELPHI during the test generation process. In the worst case, DELPHI is on its own and will get no help from the user. But this assumption may not be valid in all cases. DELPHI is flexible and will allow more technically adept users to provide test knowledge. After verifying the knowledge provided by the user, DELPHI will incorporate it into the test knowledge base. In cases in which a complete set of test knowledge is not available, DELPHI will enumerate design for testability (DFT) options which then can be used to minimize any redesign required to fully test the circuit.

Although the work presented here is aimed at designs developed using a silicon compiler, the techniques may be used to improve test generation for circuits produced by standard design methods. By automating the process of extracting test knowledge, the accuracy of the knowledge can be consistently controlled

while freeing the designer from this tedious process. In addition, facilities can be made available to automatically verify hand-generated test knowledge provided by designers. The use of automated methods to provide DFT options is also a valuable resource that can be exploited for noncompiled designs.

Traditional approaches to digital system testing have relied heavily on testability measures to aid in the process of test vector generation and DFT. At the gate level, controllability and observability measures have been used extensively. In this thesis, techniques are described to efficiently and accurately measure the testability of a digital circuit at the functional level, where traditional testability measures have little or no meaning. Instead of knowing how controllable/observable a bit in a word is, a testability measure must now indicate how easy it is to produce a specific word on an input to a functional unit. In addition, nontraditional architectural styles, such as bit-serial, need to be accommodated. In a bit-serial design the testability of a single-bit data-path is difficult to measure with traditional testability measures. Each clock cycle within a word produces data-path values with different meanings; thus, it is necessary to evaluate the testability of that data-path for each of the different clock cycles.

The creation of a testability measure which works well with new functional level design approaches is the first goal to be achieved. To achieve this goal, a functional level testability measure based on information theory has been developed. Two new techniques for estimating the measure are introduced, and examples of estimating the functional level testability are discussed. In addition

to efficient methods of estimating the measure, some applications of the testability measure are described. These testability measures are then used to assist the test knowledge extraction process carried out by DELPHI.

In Chapter 2 of this thesis, overviews of compiled circuits and the problem of testing VLSI circuitry are provided. Following that, previous work related to the developments described in this thesis are discussed. The remaining chapters of this thesis describe solutions to two problems associated with testing compiled circuits. Chapters 4 through 6 describe the development of a functional level testability measure based on the flow of information though a digital circuit. As will be shown, this measure can be accurately estimated as a low computation cost. This measure can also be applied to the problem of circuit partitioning.

Chapters 7 through 10 describe techniques to represent, extract, and verify test knowledge. In addition, methods are described to also extract design for testability knowledge which can be used when circuit constraints prevent the extraction of a complete test knowledge base. Design for testability knowledge will provide the option of efficiently adding DFT circuitry to improve test coverage for portions of the circuit where the test knowledge is not adequate. The techniques described are then applied to a number of example circuits to evaluate the performance and usefulness of the approach.

Finally, Chapter 11 summarizes the contributions presented and proposes some possible areas for future research.

2. BACKGROUND INFORMATION

The process of testing a VLSI circuit is both abstract and as physical. The abstract portion of VLSI testing relies on a model of the physical makeup of the circuit (i.e., structure of the circuit design, as it resides in a design automation system). From this abstract structure a sequence of test patterns is generated. These patterns are physically applied to a fabricated chip by a circuit tester. The results of applying these patterns to the chip determine if it is good or bad.

The primary assumption used in this thesis is that the circuit designer is not familiar with either of these processes. Since end users can now design application specific integrated circuits (ASICs) using silicon compilation tools, knowledge of the underlying circuit design process (including testing the circuit) is not necessarily understood by the designer. Although this simplifies the overall design procedure, it can introduce problems when trying to develop a strategy to test the compiled circuit. The goal is to intelligently extract knowledge which will be used to help generate test patterns. This is done without the intervention of the user so that the process will not be limited by the abilities (or inabilities) of the user.

2.1 Silicon Compilers and the Circuit Model

Before discussing the problems associated with testing a compiled circuit, it must be made clear what types of circuits are to be targeted for the work

presented in this thesis. By compiled circuits, it is meant that algorithms have been compiled directly into (digital) circuitry. Rather than executing general purpose instruction streams (e.g., a microprocessor), we are interested in compiled circuits which perform a specific set of operations. They operate on data rather than instructions plus data. This is not to say that silicon compilers cannot be used to produce instruction executing processors. Just as the “C” compiler can be used to compile a BASIC interpreter, silicon compilers can be used to produce instruction executing circuits. The work described in this thesis will be of the most benefit to users who are primarily interested in an algorithm that they have developed and implementing it in the form of an ASIC. Since they are knowledgeable about their specific algorithm but are not knowledgeable about circuit design and testing, the process of generating tests for the circuit should be automated. It is also clear that many of the techniques described in this thesis can also be used to accelerate traditional approaches to test generation.

The absence of instruction-based processing complicates the control structure for compiled circuits. In a instruction executing circuit, the control and datapaths are usually separate. In processor-like structures, this knowledge has already been shown to improve test generation performance [3]. In the compiled circuits we are concerned with, the control and data are intertwined. Therefore, it is usually not possible to partition the data and control sections into distinct sections.

There are also a number of other, more subtle differences between compiled and noncompiled circuits. The fact that a user is using a silicon compiler to design a circuit indicates that pushing the technological limit is not a primary concern. Compiled circuits tend to be much less area efficient than full custom circuits. This is due to a number of different influences, including area overhead that must be included in standard cells and inefficient area optimization algorithms. Another difference between compiled and noncompiled designs is that the structure of a compiled circuit is not as “random” as in custom circuit designs. The high-level structure of ASICs produced by a compiler are not the messy tangles of wires often seen in custom ASIC designs. This “niceness” (if such a term can be applied to a circuit) makes the problem of testing compiled circuits more readily applicable to automated test knowledge extraction.

The fact that users create designs at the functional level does not mean that the circuit is actually designed completely at that level. Typically such designs are created hierarchically, with each functional operator being composed of gate level operators, which can further be decomposed into transistor level operators (see Figure 1). Despite the fact that hierarchy exists, we are only concerned with the functional level design and do not generate the specific test values at the gate and transistor levels. As we will describe later, low-level test generators have to be used in conjunction with DELPHI to test the circuit.

Another type of hierarchy exists in the functional operators, which is due to the bit-slice design technique used to implement o perators using parameterized

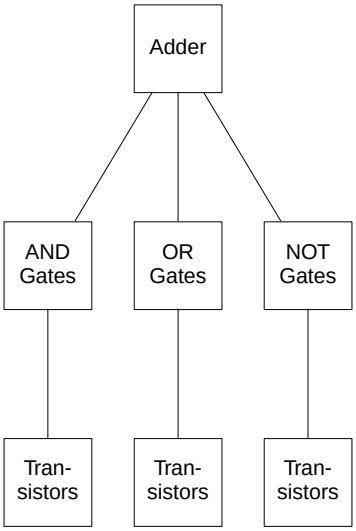

Figure 1: Example of Hierarchy

standard cells. This allows the cell designer to develop a single operator which can be used for a number of different data-path widths. Since there are only a small number of different cells that are necessary for most applications, cell designers can optimize each cell individually. This property also allows for improved designs when testing is considered. More realistic and complete fault models (see next section) can be generated by extensive evaluation of each standard cell. In addition, test knowledge extraction routines can be developed for each individual cell before any designs are actually generated. This simplifies the design process for end uses while adding slightly more work for the silicon compiler cell designer (a tradeoff we feel is appropriate). Since the added analysis is done when the cells are initially designed, the cost is incurred only once.

The circuit model in this research is the traditional dataflow graph representation used (in one form or another) by most design systems [5]. Nodes in the dataflow graph represent functional level circuit primitives such as adders, multipliers, multiplexors, logical operators, and comparison operators. Arcs between nodes represent signal paths between the functional primitives.

In an effort to reduce any ambiguity due to a misunderstanding of terms, a few will be defined here. The functional level circuit primitives will be referred to as functional operators, functional units, as well as functional primitives. Signal paths will be also referred to as nets, and nets are composed of individual lines. Primary inputs and primary outputs are interfaces between the entire circuit and the outside world (i.e., a tester). An input or output (not primary) will refer to the inputs or outputs on an individual functional unit. Figure 2 illustrates these terms.

Two silicon compilation systems were used as vehicles for the research presented in this thesis. The first was the Bit-Serial Silicon Compiler (BSSC) developed at the General Electric Corporate Research and Development Center [6,7,8]. The BSSC (as its name implies) generates pipelined circuits using a bit-serial architecture from a “C-like” high-level specification. A list of the functional operators available to designers using the BSSC is listed in Table 1 [9]. The BSSC was later modified to handle digit-serial architectures [10,11]. Although this change affects the details of the low-level implementation, the functional level representation is unchanged. The digit-serial compiler can generate bit-serial architectures, by setting the size of the digit to one bit.

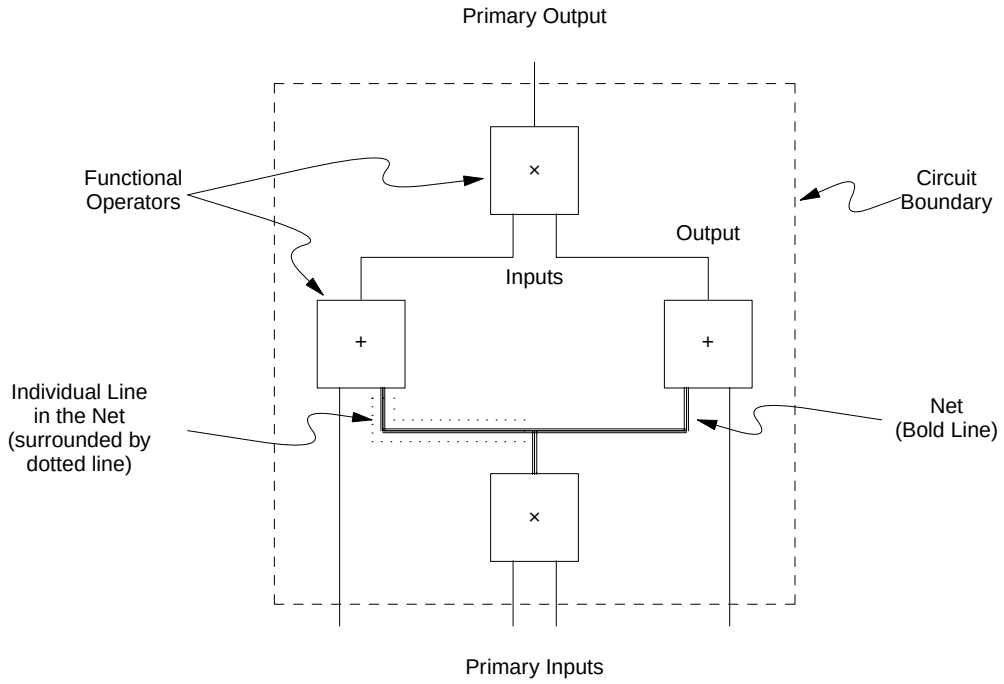

Figure 2: Definition of Terms

Table 1: The BSSC Operators

Operators		
Arithmetic	Logical	Comparison
Addition	And	=
Subtraction	Or	≠
Multiplication	Not	>
Negation	Multiplexor	≥
Divide by 2	Delay	<
	Shift Left	≤
	Shift Right	

An example circuit designed using the BSSC is shown in Figure 3 (the pipeline latches are not shown but are included in each signal path). It has eight-bit data-paths and is composed of approximately 50,000 transistors. The functional specification used to generate this circuit is listed in Figure 4.

This circuit is a modification of a design used to implement a solid-state circuit breaker. Due to the proprietary nature of the original circuit, the design was modified slightly (overall features of the design remain). In the figure, Σ is addition, $\times$ is multiplication, Δ is a 1-word delay, and & is a bitwise AND operator. The triangles represent comparison operators, with the symbol inside the triangle determining the type of comparison. The trapezoidal figures represent multiplexors. The operation of the circuit is as follows: The inputs X1 and X2, along with Y1 and Y2, are added together. The results of these additions are then multiplied together. The inputs X3 and Y3 are also multiplied together. To save silicon area, a single multiplier is used and is time-multiplexed between the two sets of data using the control input S. While the result of the multiplication is greater than threshold T (stored as a constant) and the control input K is true, the input Z is summed. The summation is performed using feedback through a delay operator. If either the multiplication result is less than T, or K is false, then the sum is initialized to the value on the R input. The result of the multiplication is fed to the primary output O1 while the running sum of Z is fed to the output O2.

The second compilation system that was used as a research vehicle was the Flexible Architecture Compilation Environment (FACE), also developed at the

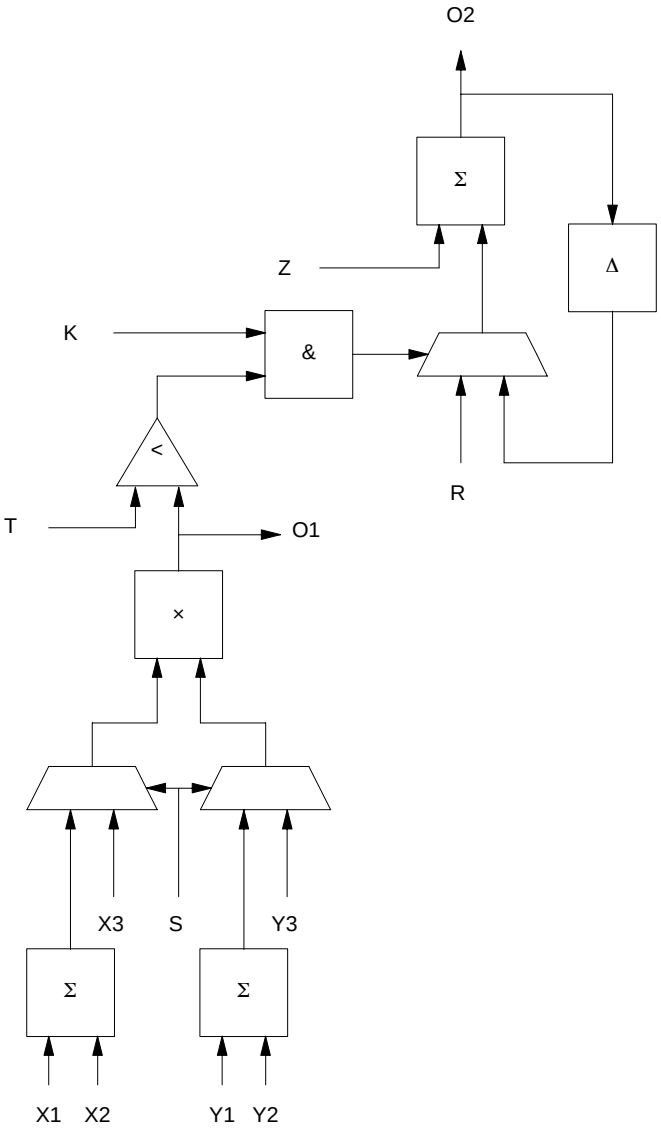

Figure 3: Example BSSC Generated Circuit

```
symbolic processor

     brkr(in:X1,X2,X3,Y1,Y2,Y3,Z,T,S,K,R;out:O1,O2) ;

     wordsize 8 ;

     signal   S1,S2,S3,S4,S5,S6,S7 ;

     specification

          S1 := X1 + X2 ;
          S2 := Y1 + Y2 ;
          S3 := S ? S1 : X2 ;
          S4 := S ? S2 : Y2 ;
          O1 := S3 * S4 ;
          S5 := O1 > T ;
          S6 := S5 & K ;
          S7 := S6 ? R : O2[-1] ;
          O2 := Z + S7 ;

     end brkr ;
```

Figure 4: BSSC Circuit Specification

General Electric Corporate Research and Development Center [12,13,14]. FACE is a more general compilation tool than the BSSC and is actually a collection of tools which can be used to build a wide variety of silicon compilers. A special language called SMALL (System Modeling Analysis Language) has been developed to allow users to easily specify their designs [15]. A graphical interface called VISAGE has also been created which allows users to interact with their design in a visual environment [16]. Some of the early research described in this thesis was carried out at the General Electric Corporate Research and

Development Center and used the FACE system to investigate test knowledge extraction algorithms.

An example circuit designed using FACE is shown in Figure 5. This circuit is a 102,000 transistor pipelined video processor which was fabricated by General Electric [17]. Each clock cycle produces three results for the equation:

$$R(x) = A \times B \times C \times D \times E(x) + (1 - A) \times F \times G(x)$$

where x is either 1, 2 or 3 and A, B, C, D, E, F, and G are circuit inputs. As will be shown later, this circuit is fully testable using the techniques described in this thesis.

FACE circuit

Figure 5: Example FACE Generated Circuit

2.2 The Testing Problem

The aim of VLSI test generation is to detect physical failures (faults) in a circuit by applying a test vector to the primary inputs of the circuit. By monitoring the primary outputs and comparing the observed values with the expected values, some of the faults can be detected. The problem is to determine a (small) set of test vectors which can detect all faults in a circuit.

Since every possible type of physical failure cannot be considered individually, a fault model is used to simplify the test generation process. Historically, the first fault model used was the *single stuck-at* [18] which models failures at the gate level by clamping an input or output of a gate to either 0 or 1. A more general gate level fault model is the *single module fault* which models a fault as any arbitrary change in the truth table for a single gate. It should be noted that the single stuck-at fault model is a subset of the single module fault model. Moving up in the hierarchy to the functional level is usually achieved by combining the fault models for the gates composing the functional operator into one compacted list [19]. In some cases the structure of the functional operator design can greatly simplify the functional level fault model [20].

Traditionally the process of VLSI test generation is separated into two steps. The first step is to justify a set of specific Boolean values (either a 0 or 1) to the inputs of a gate internal to a circuit by setting (some of) the primary inputs. The second step is to propagate the output of the gate to a primary output. The order in which a test generation algorithm performs these steps, as well as how each

step is performed, are active areas of research.

The Boolean stimuli applied to the inputs of the gate will be referred to as *local test values* while the Boolean stimuli applied at the circuit's primary inputs are referred to as *global test values.* The goal of gate level test generation is to determine the global test values for a particular gate's local test values.

In recent years there has been an on-going research effort to move test generation up one level in the design hierarchy. Instead of trying to justify a set of Boolean values to an internal gate, functional level testing attempts to justify a set of integers (or some other functional level number representation, e.g., normalized finite precision fractions) to the inputs of a functional level operator [21]. Once that has been done, the output of the functional operator is propagated to a primary output. Typically these functional level test vectors are independently generated using a hierarchical level test generator, which uses the gate and transistor levels to produce the test. They can also be precomputed by the standard cell designers and stored in the cell database. In either case, more detailed and accurate fault models can be tailored for each available functional operator. Using the same scheme as the gate level tests, the integer value stimuli applied to the inputs of a functional operator will be referred to as local test values while the integer value stimuli applied at the circuit's primary inputs are referred to as the global test values.

In the work presented in this thesis, the actual generation of the local functional level test values is not performed. It is assumed that tests for a specific

functional operator will be provided by another source (either on the fly or from a precomputed database). The problem is to determine a global test which will produce a specified local test.

2.3 Test Knowledge

The work presented in this thesis attempts to solve the problem of functional level testing at a meta-level. Instead of looking at specific test values, the problem of applying arbitrary functional level test values is examined. The way in which this type of information is represented is referred to as test knowledge. Even if this test knowledge cannot be obtained for the entire circuit, it will still provide valuable information for the test generation process. In addition, the failure to extract test knowledge will be used to generate lists of DFT options which will remedy any problems in test knowledge extraction.

Two types of test knowledge are testability measures and test generation knowledge. A testability measure describes how information/data flows through the circuit and is used to make graph traversal decisions during test generation. An example of the use of a testability measure would occur when trying to obtain a specific value on either one of two data-paths in a circuit. The testability measure would indicate which data-path has more information flowing to it (from the primary inputs) and thus the decision would be to choose the one with the largest testability measure.

Test generation knowledge, on the other hand, specifies how to apply arbitrary test values to internal nodes in a circuit by setting the primary inputs. These arbitrary test values are referred to as *symbolic tests.* Consider the example in Figure 6. We are interested in testing component C1 by applying test vectors on inputs I1 and I2. Note that the values of specific test vectors are not being

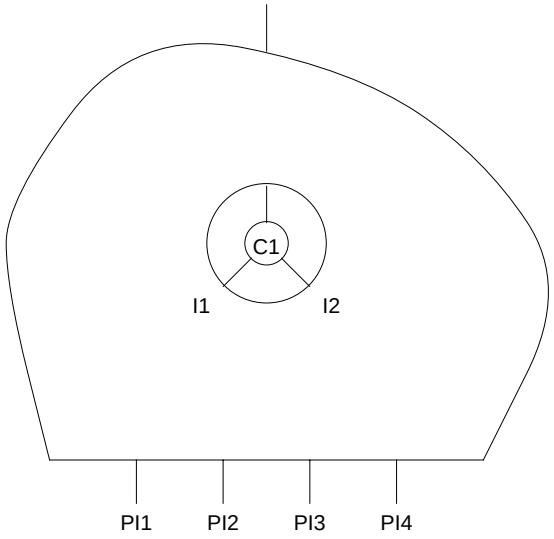

Figure 6: An Example of Test Generation Knowledge

considered at this time and we are only interested in how to apply arbitrary test vectors to the inputs I1 and I2 of component C1. Let us refer to these arbitrary test vectors for I1 and I2 as the symbolic values N_1 and N_2, respectively. Suppose that the following primary input setting produces the symbolic values N_1 and N_2 on inputs I1 and I2:

$$PI1 = 12, \quad PI2 = N_2 + 1, \quad PI3 = -11, \quad \text{and } PI4 = N_1.$$

A hierarchical test generator or precomputed set of test vectors for C1 would then be used to obtain the specific tests required to fully test C1. If one such test has $N_1 = 3$ and $N_2 = 0$, a simple variable replacement of PI1 = 12, PI2 = 1, PI3 = -11, and PI4 = 3 would produce the desired test vectors on the inputs of component

C1. Although this is a very simple example, it is meant only as a general description of the use of test generation knowledge. A thorough discussion of test generation knowledge is described in Chapter 7.

In some circumstances, tests for all parts of a circuit cannot be generated due to constraints provided by the structure of the circuit. In these cases the traditional approach has been to use *Design for Testability* (DFT) techniques to enhance the testing of the circuit. Some examples of DFT techniques include full scan, partial scan, test points, and other ad-hoc approaches [22,23].

During the extraction of test generation knowledge from the structure of the circuit, a complete set of test knowledge may not be produced. In those cases, DFT options are collected so that an intelligent redesign for testability can be carried out automatically. These DFT options are referred to as DFT knowledge, which are used after the test generation knowledge is extracted to overcome any structural constraints which prevent a complete set of test generation knowledge from being extracted. As will be shown, simple techniques are used to minimize the cost of adding DFT circuitry by using the extracted DFT knowledge. In the work presented here, we will concentrate on minimal scan latch insertion although the techniques described would also work well with other techniques.

3. PREVIOUS RESEARCH

The testing problem has been actively researched since the late 1960's [24]. Work in this area has centered on deterministic test pattern generation algorithms at the gate level. This section of the thesis describes previous research which has attempted to accelerate the test generation process.

3.1 Testability Measures

There has been a great deal of previous work performed in the area of testability measures. Most of these measures have focused primarily on bit-level analysis. Examples of bit-level 0/1 testability measures are SCOAP [25,26] and CAMELOT [27]. Extensions of this work to the functional level have also been attempted and a summary of various functional level testability measures can be found in [28]. Fong [29,30] and Takasaki, et al. [31] have each derived extensions of the SCOAP 0/1 testability measures for the functional level. Expensive computational requirements limit the use of these measures. In TMEAS [32], Stephenson developed a probabilistic measure of testability for both controllability and observability at the functional level.

There has also been work performed in the application of information theory to digital system testing. Agrawal [33] looked at applying information theory to test pattern generation. In this work, random pattern testing using equiprobable

outputs is proposed. By choosing test patterns which maximize information on the output, the author shows that these test vectors have the highest probability of fault detection.

The first application of information theory as a testability measure was performed by Dussault [34]. This work presents observability and controllability measures based on information theory for gate level circuits. Aside from exhaustive enumeration, random sequence application is the only possible estimation technique mentioned. A major problem with this work is that it computes absolute instead of relative values for the measures. Two different circuits may produce similar values using the measures described in [34] despite the fact that one may be much more difficult to test than the other. In the work presented in this thesis, a relative testability measure is described which will allow accurate comparison of testability values.

The measure presented in [35] is most closely related to the work described in this thesis. Major differences arise in the methods used to calculate the measure as well as applications of the measure once it is computed. We are proposing techniques to efficiently and accurately estimate the measure. Previous methods (for example, the method presented in [35]) requires exhaustive analysis (which has a computational complexity that grows exponentially with the number of inputs). Designs with more than a small number of input lines (say, greater than 24 bits) are computationally intractable and thus preclude the computation technique presented in [35]. In addition, the work presented in [35] considers

only a controllability measure and does not derive a similar measure for observability (which we have developed).

3.2 Testing Compiled Circuits

Previous research in the area of testing compiled circuits has been somewhat limited. Two projects aimed specifically at testing compiled circuits are the SILC silicon compiler developed at GTE [36,37,38] and the CATREE system developed at the University of Waterloo [39,40,41]. In the SILC system, behavioral specifications of the desired circuit are provided by a user. A testability evaluator then determines which of the system components (FSMs, PLAs, etc.) are difficult to test and the “TESTPERT” program modifies the low level structure of the components to increase testability. These modifications consist of BIST and scan-path techniques as well as the addition of a special test bus and test controller. There is no feedback path provided for additional design exploration or user input. A hierarchical ATPG system is used to generate tests for the circuit components and can request modifications to the circuit to improve testability.

The CATREE system uses a partitioning algorithm to insert DFT circuitry into the design of a compiled circuit. The partitions are generated using a mincut technique using a binary tree representation of the circuit. Design for testability circuitry is added to the edges of the partitions and estimates of the area, delay, and testability of the partitions are determined. CATREE is limited to the addition of BIST and scan-paths to a design and does not consider test pattern

generation. No fault coverage estimates are provided. It does however include testing in the design feedback loop. This allows various implementations in the design search space to be looked at from a testing point of view.

Although not specifically aimed at test generation in a silicon compilation environment, Breuer and colleagues at USC [42,43] have developed a number of techniques to automatically evaluate and design testable logic. The TDES system developed by Abadir and Breuer [42] attempts to match local circuit structures with special testability design methodologies (TDMs). When a match is found, the TDM can be used to improve testability.

There has also been work in the area of automated built-in self-test (BIST) insertion in VLSI circuits [44,45,46]. Although this work has merit, its applicability for most users is limited.

3.3 High-Level Testing

The use of high-level knowledge (typically at the register-transfer level) has previously been shown to increase test generation speed. The reasons for the test acceleration are obvious. High-level knowledge replaces the justification and propagation of zeros and ones through gates/transistors by the propagation of integers (or, in some cases, fixed precision rational numbers) through algebraic equations. For example, trying to propagate two thirty-two bit numbers through a gate level representation of an array multiplier is going to be more complicated than simply multiplying the two numbers together.

Early work by Breuer and Friedman [21] showed that high-level representations could help in test generation by quickly propagating tests through a circuit. Recently Kunda et al. [47] described a system which uses high-level primitives for both justification and propagation of tests through a circuit.

The work presented in this thesis is one more move up in the hierarchy of the circuit. Instead of using knowledge to justify and propagate through a single functional level made up of gates, we are extracting knowledge which will allow us to justify and propagate through an entire circuit made up of the functional level modules. High-level test knowledge has previously been shown to drastically accelerate the test generation process. In [2], Brahme and Abraham show that a speed improvement of several orders of magnitude can be achieved by providing high-level circuit knowledge to a test generation system. Unlike the work presented in this thesis, Brahme and Abraham manually extract test knowledge from a circuit and insert it into a test generation program.

Murray and Hayes [48] have done work on precomputation of tests for functional level modules. Although their work is not strictly at the functional level, the work they have done makes a strong case for the use of high-level test knowledge to facilitate the propagation and justification of precomputed test vectors.

3.4 Artificial Intelligence and Knowledge-Based Testing

In recent years the application of artificial intelligence (AI) has been suggested as one way to improve test generation techniques [49]. A number of rule-

based expert systems have been developed to aid in test pattern generation and design for testability (e.g., [50,51]). Some researchers have claimed that hierarchy is a form of artificial intelligence and that the use of hierarchy in test generation is an AI technique [52,53].

A limited amount of work in the area of automatic test knowledge extraction has been performed. In [54], Shirley describes the “Designed-Behavior Test Generator” (DB-TG) which can be used to extract high-level test knowledge from instruction executing processors. The knowledge is obtained by simulating the instruction set on the circuit and recording the effects. The results of the simulations are then used to aid in the justification and propagation of test vectors through the circuit. Although this work is very applicable to the testing of instruction based processors, it is not appropriate for most compiled circuits. Since the class of circuits we are considering carry out a specific algorithm rather than executing instructions (as will be described in the next section), the simulation of instructions is not possible. In [55], Anirudhan and Menon describe another approach to extract high-level test knowledge from instruction executing circuits using a hierarchical model for the circuit. The interaction between data-path and control path circuitry is incorporated into the test generation process. Most recently the work of Roy and Abraham [56] described techniques to extract test knowledge from the data-path portions of instruction executing processors. The generation of knowledge for testing control signals was not discussed.

The use of knowledge to assist DFT has also been researched. Wu's DFT advisor [57] and HHB's "Knowledge Based Compiler" [58] both use circuit test knowledge to minimize the amount of additional DFT circuitry necessary. Both of these approaches make use of the knowledge during the low-level test generation phase. In our approach, we use the knowledge before the low-level test generation process is begun. As a result, the DFT strategy that is chosen will be fully reflected in the knowledge base and can be used to its full potential.

4. MEASURING INFORMATION FLOW IN A DIGITAL CIRCUIT

A testability measure indicates the relative difficulty of obtaining an arbitrary value on an internal line in a circuit. This is directly related to the flow of information through a circuit. The more information that flows onto a line from the primary inputs, the easier it will be to achieve a desired test on that line. Testability measures are used in a number of applications. As will be shown later, the measures described in this thesis will be used as heuristics to help guide the search during test knowledge extraction in DELPHI. An automatic partitioning algorithm (for testability) using the information theoretic testability measures will be described and evaluated. Finally, testability measures can be used in the evaluation of various circuits early in the design phase. Since the information theoretic testability measure is defined at a higher (functional instead of gate) level, it can be used early in the design, before the detailed circuit structure has been synthesized. This will allow a design automation environment to identify potential testability problems while initially exploring the design space.

4.1 Fundamental Mathematical Properties

Two mathematical properties that we will use extensively in our testability measure are *information* and *mutual information.* We will first discuss the exact computation of these properties and later describe simple estimation techniques.

Information (or entropy) is a measure of the *randomness* of a signal [59]. It is minimized when there is no variance in the signal values (i.e., the signal is a constant) and is maximized when all of the signal values are equally likely. From classical information theory [59], the information contained in a signal X (in bits) is defined as:

$$H(X) = \sum_{a \in X} p(a) \log \frac{1}{p(a)}$$

where $p(a)$ is the probability mass function over the set X and log is over the base two. If an n-bit signal path is influenced by j (binary) primary input lines, the exact entropy calculation will have a time complexity of order $O(2^j)$ and a space complexity of $O(2^n)$.

Mutual information is a measure of "the amount of information provided about [signal] X by [signal] Y [59]." The mutual information between two signals X and Y is defined as [59]:

$$I(X;Y) = \sum_{a \in X, b \in Y} p_{X,Y}(a,b) \log \frac{p_{X,Y}(a,b)}{p_X(a)\, p_Y(b)}$$

For n-bit signal paths, the calculation of the mutual information between an internal signal path and a primary output has a time complexity of order $O(2^k)$, where k binary primary input lines influence the primary output. The space complexity of the calculation will be of order $O(2^{2n})$.

Since information and mutual information are strongly influenced by a number of factors (including data-path width, circuit depth, etc.), two different systems could produce similar values for both $H(X)$ and $I(X;Y)$. Thus, we are interested in using relative instead of absolute values for both information and mutual information. In the case of information, we are concerned with how much information is transferred from the primary inputs to the node of interest. Consider two circuits each containing a node whose entropy is equal to H. In the first circuit, this node is influenced by m primary inputs. In the second circuit, the node of interest is influenced by $2m$ primary inputs. It is obvious that it will be more difficult to obtain a desired value on the node in the second circuit as opposed to the first simply due to the fact that there is a larger input space to be searched.

In [60], a measure referred to as the the *Information Transmission Coefficient* (ITC) is developed. The ITC is defined as "a measure of the fraction of information that can be transmitted through a function." It is computed by taking the ratio of the entropy on the inputs of a function and the entropy of the outputs of the function. The value of a function's ITC depends only on the mapping of the function, the input probability distribution and the word length. The value for an ITC is bounded below by zero (i.e., no information has made it to the outputs, e.g., a constant output) and above by one (i.e., no information has been lost, e.g., the function is one-to-one).

Since the primary inputs for a circuit are directly controllable, the information on m input lines is equal to m bits. Returning to the example of the two circuits

each with a signal path whose entropy was equal to H, the ITC for the first circuit would thus be $ITC_1 = \frac{H}{m}$. On the other hand, the ITC for the second circuit would be $ITC_2 = \frac{H}{2m} = \frac{ITC_1}{2}$. The relative differences between two ITC values indicate more than just a relative difference in the amount of information flowing to different points in a circuit. Since most compiled circuits have constant width datapaths, the (absolute) maximum amount of information flowing to two different signal paths is the same. Differences in the actual values can be attributed to two sources. Obviously if the functional operators that the information flows through are not identical, different degrees of transmission will be involved. The other influence is due to the number of primary inputs influencing the nodes of interest. Since the ITC ratio is obtained by dividing the signal path entropy by the number of primary input bits, the larger the number of primary inputs the smaller the ITC value. Thus the ITC value gives a good indication of the relative sizes of the input spaces that potentially need to be searched.

In the case of mutual information, we are concerned with how much information at a particular node is conveyed to a primary output. Thus, if a primary output whose entropy value is H has a mutual information of MI with an internal circuit node, the ratio $\frac{MI}{H}$ indicates what percentage of information at the node of interest is directly transferred to the output. This will be used as part of our observability measure (the computation of the observability measure is described later in this thesis).

4.2 Estimation of Information Flow

Obviously the exponential growth in the computation of these properties precludes their use on most circuits. Since the testability measures that will use these properties are a means and not an end in the process of testing a circuit, it is not imperative that the exact value of the properties be obtained. It is desirable to find a good approximation in a short period of time. To achieve these goals, two simple estimation techniques were developed.

4.2.1 Estimation technique 1 : sampling

The first technique used to estimate relative information and mutual information relies on sampling each signal path of interest while a series of random input vectors are applied to the circuit. These samples can then be used to determine the information content on various signal paths. To implement this technique, a functional level simulation of the circuit is performed and the values at each signal path are sampled over a series of predefined intervals.

A number of experiments were carried out on various circuits. The experiments looked at different sampling techniques, including fixed and variable width data-paths. In the largest circuit used in the experiments, the video processor shown in Figure 5 was evaluated. Entropy values converged very quickly and reasonable results were achieved in as few as 1000 samples. The results of this experiment are illustrated in Figure 7 for increasing data-path widths. The regular spacings of the entropy values shown in this figure are to be expected and are a

Figure 7: Sampling Entropy for CMC Circuit

consequence of sampling the distribution to estimate the entropy [61]. In purely arithmetic circuits, an increase of one bit in the data-path width will, in the limit, add one bit to the entropy for that signal path (i.e., in the limit additional bits provide little information). For purely logical circuits, the entropy will be proportional to the number of bits in the data-path (i.e., if a circuit with an n-bit data-path has an entropy of H_n, the same circuit with an m-bit data-path will have an entroy of $H_m = \frac{m}{n} H_n$). These simple relationships can be used to simplify the estimation of entropy values by using simulations of smaller data-path widths (which are computationally simpler) to estimate entropy values for circuits with larger data-path widths. If it is not possible to make the distinction between purely arithmetic and purely logical behavior for a particular circuit, a linear extrapolation can be used based on two estimates.

For example, if the data-path widths for the circuit in Figure 5 were increased to 32 bits, it would be very difficult to estimate the entropy values using sampling due to the exponentially increasing computational requirements. Instead, a smaller data-path width (say 14 bits) could be sampled and that estimate could be used to compute the estimate for a 32 bit data-path. Since this circuit is composed entirely of arithmetic operators, the entropy value for the 32-bit circuit would equal the entropy value for the 14-bit circuit plus 18 bits.

As would be expected, the errors associated with the sampling estimates for the ITC value decline as the number of samples is increased. Figure 8 illustrates the relationship between the percent error for various data-path widths and

number of samples.

The same sampling technique can be applied to the estimation of mutual information. Experiments were again run on the circuit in Figure 5 and the mutual information between internal nodes and the output was obtained. Figures 9 and 10 illustrate the results for nodes A and U, respectively. As can be seen, the mutual information values for the various data-path widths converge in a reasonable number of samples. They all converge to the same values for the same reason that there was an even spacing between the entropy estimates. This is due to the fact that, in the limit, additional bits in the data-path (for purely arithmetic circuits) provide little information and thus cannot provide additional mutual information. Once again, if the circuit is purely logical, the mutual information is proportional to the data-path width. As with the entropy estimation, a linear extrapolation technique can be used in cases where it is not possible to categorize a circuit as purely arithmetic or logical.

From these estimates of information and mutual information, the desired ITC and MITC ratios can be obtained. In the following section, another estimation technique will be described.

4.2.2 Estimation technique 2 : composition

The second approximation technique that has been developed is a composition technique which can be computed in a time proportional to the number of components in the circuit. The first step in the process is to obtain the

Figure 8: Sampling Error for Various Data-Path Widths for ITC

Figure 9: Sampling Mutual Entropy for Node A

Figure 10: Sampling Mutual Entropy for Node U

information and mutual information ratios for each of the individual components in the circuit.

The ITCs for each of these individual components are listed in Table 2. The assumptions made are that each signal path has a word length of 8 and the input distributions are even (i.e., the inputs are directly controllable). These values can be obtained either by simulation or analytical techniques. In both the addition and subtraction operators, two distinct cases are considered. In the first case (Addition_1 and Subtraction_1), overflow is allowed and the result of an overflow is mapped back onto the value space. The overflow values are used in the ITC calculation. In the second case (Addition_2 and Subtraction_2), whenever an overflow is observed, those values are thrown away and are not used to compute the ITC value. The specific implementation will determine which case to use.

Table 2: ITC Values for the BSSC Operators

Operator	ITC	Operator	ITC
Addition_1	0.500	Addition_2	0.498
Subtraction_1	0.500	Subtraction_2	0.498
Multiplication	0.461	Divide by 2	0.875
Negation	1.000	Delay	1.000
And	0.406	Or	0.406
Not	1.000	Multiplexor	0.471
Shift Left	0.875	Shift Right	0.875
=	0.004	≠	0.004
>	0.063	≥	0.063
<	0.063	≤	0.063

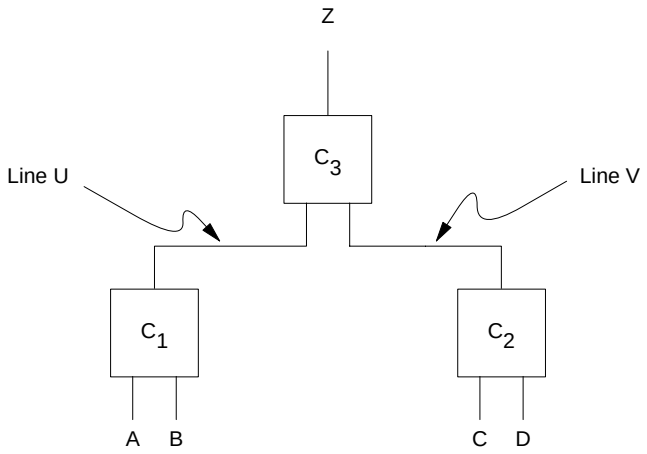

Figure 11: Simple Example Circuit

Now consider the circuit in Figure 11. The blocks C_1, C_2, and C_3 can be any of the functional components available to a designer. Let ITC_{C_1}, ITC_{C_2}, and ITC_{C_3} be the ITC values for components C_1, C_2, and C_3, respectively. Since components C_1 and C_2 are fed by the primary inputs, the ITC value for the signal paths U and V are simply the ITC values for components C_1 and C_2 (e.g., $ITC_U = ITC_{C_1}$ and $ITC_V = ITC_{C_2}$). We will differentiate between the ITC for a signal path and the ITC for a component. The ITC for a signal path is defined as the ratio of the entropy for signals on that signal path and the entropy for the primary input lines that influence that signal path. The ITC for a component is independent of the circuit containing the component and is defined as the ratio of the entropy of the signal on the output of that component and the entropy of the signals on the input of that component. In the case of the component ITC, it is

assumed that the inputs are directly controllable.

We estimate the ITC for the signal path Z by the product of the ITC for component C_3 (i.e., ITC_{C_3}) and the weighted average of ITCs for the input signal paths. In this case, the weighted (where the weight is the signal path width) average of the ITCs on the input signal paths is $\frac{ITC_U + ITC_V}{2}$, assuming that all of the signal paths have the same width. More formally, the relationship between the ITC on the output signal path and the ITCs on the input signal paths for a particular component is described by the following formula:

$$ITC_{out} = ITC_{comp} \times \sum_{i=1}^{n} \frac{w_i}{W} ITC_i$$

where ITC_{comp} is the component ITC for the component of interest, ITC_i is the ITC value for input i, n is the number of input signal paths for the component of interest, w_i is the data-path width for input i, and $W = \sum_{i=1}^{n} w_i$.

For example, suppose that component C_1 is a multiplier and components C_2 and C_3 are adders (thus, $Z = A \times B + C + D$). From Table 2, $ITC_{C_1} = .461$ and $ITC_{C_2} = ITC_{C_3} = .500$. Thus, $ITC_U = .461$ and $ITC_V = .500$. The weighted average of the inputs to component C_3 are $\frac{.461 + .500}{2} = .481$. Multiplying this value by the ITC for component C_3 produces an ITC value for signal path Z of .222.

Although this composition technique is only an approximation, our experience has found that it reasonably estimates the true ITC value. A number of

example circuits have been analyzed to verify the approximation.

Figure 12 contains the results of estimating the signal path ITCs using the composition technique. The number inside each operator is the component ITC for that functional unit (from Table 2) while the numbers next to each of the signal paths represent the ITCs for data on those paths. These values accurately reflect the actual CTM values. For example, the estimated CTM value for outputs O1 and O2 are .169 and .363, respectively. Using exhaustive simulation, values of .1507 and .3187 are obtained. The estimates are reasonably close but required much less computation that the exhaustive simulation values.

The same composition techniques can be used in the estimation of mutual information. Table 3 lists the *mutual information transmission coefficient* (MITC) values for the BSSC operators. Note that unless explicitly stated, the MITC values are equal for all of the component inputs. As with the ITC, the MITC is bounded below by zero (no direct information transfer to the output) and above by one (all information on that signal path is directly transferred to the output).

Mutual Information, unlike the ITC estimate (which, as we will show later, has a direct meaning as a testability measure), will not be used by itself as a testability measure. We therefore will not introduce techniques to use composition to directly calculate MITC values for signal paths in a circuit. Instead, the composition technique will be used to calculate the observability testability measure based on the individual component MITC values. This is discussed in Chapter 5.

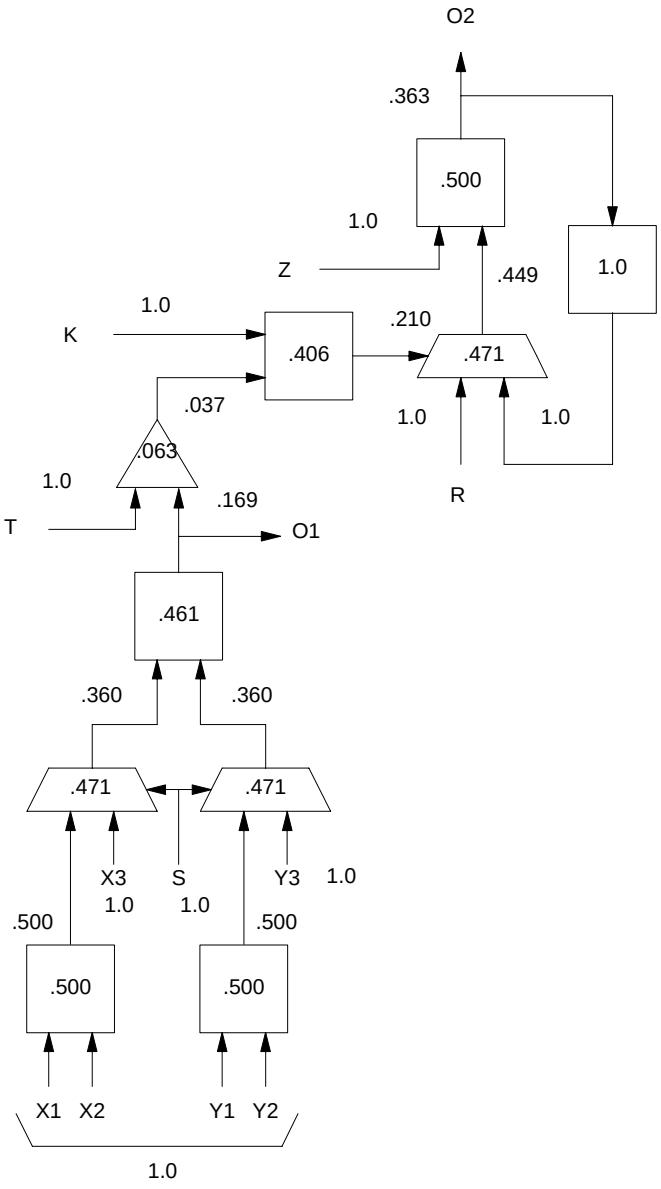

Figure 12: ITC Values for BSSC Circuit Using Composition

Table 3: MITC Values for the BSSC Operators

Operator	MITC	Operator	MITC
Addition_1	0.000	Addition_2	0.045
Subtraction_1	0.000	Subtraction_2	0.045
Multiplication	0.117	Divide by 2	0.875
Negation	1.000	Delay	1.00
And	0.384	Or	0.384
Not	1.000		
Multiplexor Data	0.377	Multiplexor Control	0.000
=	0.000	≠	0.000
>	0.279	≥	0.279
<	0.279	≤	0.279

5. THE TESTABILITY MEASURES

The testability measures described in this thesis rely on the fact that as data values flow through a circuit, the amount of information that they can convey is reduced. As a result, the entropy of signals along a data-path monotonically decreases as they pass through the components in the circuit. If we assume that the primary inputs are fully controllable (i.e., we can achieve any desired value on one of the inputs), this reduction of entropy introduces difficulty as one tries to achieve specific data values at noninput nodes in the circuit. Thus, as the entropy is reduced as data values flow through the circuit, the controllability of those data values is reduced.

The ITC measure in functional level circuit testing plays much the same role that controllability does in gate level circuit testing. If the ITC between the primary inputs and a particular signal path is close to one, it is obvious that the data on that path are strongly controlled by the primary inputs. On the other hand, if the ITC between the primary inputs and a particular signal path is close to zero, it will be difficult to obtain a desired value on that signal path (over the space of possible values). Thus, the ITC between primary inputs and a word value signal path is analogous to the controllability of a binary value signal path. In this thesis, the ITC between the primary inputs and a specific signal path is referred to as the CTM (controllability testability measure).

With regard to observability, the mutual information shared between an internal signal path and a primary output represents the amount of information from that signal path that is transferred *directly* from the internal signal path to the primary output. The word *directly* is important here because not all information transferred to an output is a direct transfer. Looking back at the MITC values listed in Table 4, the MITC value for Addition_1 (addition with overflow) is equal to zero. This means that none of the information on the output is due strictly to either one of the particular inputs. Instead, all of the information on the output is due to the interaction of the two inputs. This makes sense because any desired output and can be obtained as an arbitrary value on the first input so long as the appropriate value is applied to the second input. The value of zero for the adder's MITC does not mean that information is not transferred to the output.

Instead, it means that it is necessary to control the second input (to some degree) to transfer information from the first input to the output.

To determine how much information is due to the interaction of the inputs, simply add all of the individual MITC values for the component inputs and subtract that value from one. In the case of Addition_1, this value is equal to $1.0 - 2 \times 0.0 = 1.0$. For a multiplier, this value is equal to $1.0 - 2 \times 0.117 = 0.766$. We refer to this quantity as *shared information* (SI). A listing of the shared information (SI) for the BSSC operators is shown in Table 4.

The shared information is assumed to be distributed among the the component inputs, with each input receiving a share proportional to its data-path width. The degree to which an input can take advantage of its portion of the shared information is determined by the degree to which the other inputs can be controlled. The sum of a signal path's MITC and appropriate share of the SI is

Table 4: SI Values for the BSSC Operators

Operator	SI	Operator	SI
Addition_1	1.000	Addition_2	0.910
Subtraction_1	1.000	Subtraction_2	0.910
Multiplication	0.766	Divide by 2	NA
Negation	NA	Delay	NA
And	0.232	Or	0.232
Not	NA	Multiplexor	0.258
=	1.000	≠	1.000
>	0.442	≥	0.442
<	0.442	≤	0.442

referred to as the OTM (observability testability measure). The following formula uses the composition MITC values to calculate the OTM values for input signal paths for components in a circuit:

$$OTM_i = OTM_{out} \times \left[MITC_{comp} + SI_{comp} \times \sum_{j \in \xi - i} \frac{w_j}{W} \times CTM_j \right]$$

where OTM_{out} is the OTM value on the output signal path, $MITC_{comp}$ is the component MITC, SI_{comp} is the SI for the component of interest, ξ is the set of inputs, w_i is the data-path width for input i, and $W = \sum_{i=1}^{n} w_i$. In cases of fanout, the signal path OTM for the fanout point is set to the maximum of the OTMs for the signal paths fed by fanout point (see Figure 13).

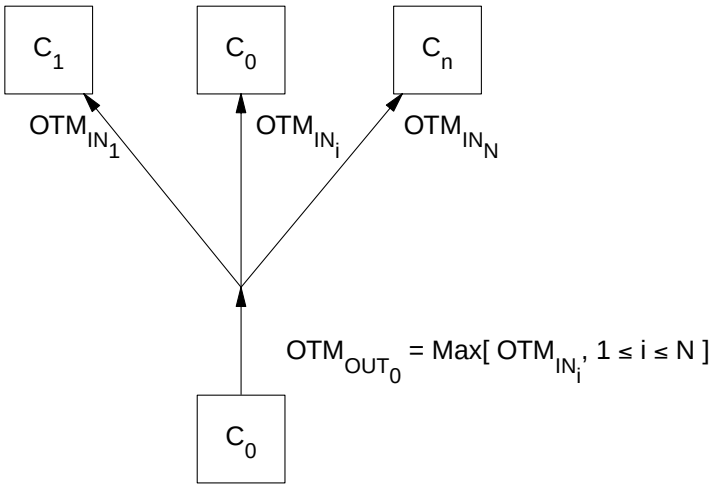

Figure 13: OTM Assignment for Fanout Points

Let us return to the example circuit shown in Figure 4. Since Z is a primary output, it has an OTM value of 1.0 Again we assume that component C_1 is a multiplier and components C_2 and C_3 are adders. From Table 3, $MITC_{C_1}$ = 0.00 and $MITC_{C_2}$ = ITC_{C_3} = .117 From this we compute SI_{C_1} = 1.00 and SI_{C_2} = SI_{C_3} = .766. We have previously determined that OTM_V = .461 and OTM_V = .500. For the signal path U we determine that

$$OTM_V = OTM_Z \times [MITC_{mult} + SI_{mult} \times 0.5 \times CTM_V]$$

$$= 1.0 \times [.117 + .766 \times 0.5 \times .500]$$

$$= 0.309$$

We similarly determine that OTM_U = 0.294. We then recursively apply this calculation to components C_1 and C_2 to obtain OTM_A = OTM_B = 0.155 and OTM_C = OTM_D = 0.147.

The techniques for calculating OTM values have so far exclusively used composition techniques. We have also developed a technique to obtain OTM values using the sampled mutual information ratio estimates discussed in Section 4.1. This approach is very similar to the composition OTM calculation. Let MI_{out} be the mutual information ratio on the output of a component. Let MI_i be the mutual information ratio for input i for the same component. The following recursive formula is used to compute the OTM values using the sampled estimates for the mutual information ratios:

$$OTM_i = \frac{OTM_{out}}{MI_{out}} \times \left[MI_{out} + SI \times \sum_{j \in \xi - i} \frac{w_j}{W} \times CTM_j \right]$$

where $SI = MI_{out} - \sum_{j \in \xi} MI_j$, ξ is the set of inputs for the component of interest, w_i is the data-path width for input i, and $W = \sum_{i=1}^{n} w_i$.

Referring back to [26], the measure which we refer to as CTM is equivalent to the “F” measure described by Fung et al. There is no equivalent for the measure we refer to as OTM.

6. AN APPLICATION OF THE TESTABILITY MEASURE

Partitioning is often used to aid in testing a circuit [62,63]. By partitioning a circuit, increased controllability and observabilty can be achieved through the addition of test points and/or scan paths. Partitioning is also used to add BIST to a VLSI circuit. In the past a number of techniques have been proposed [64,65] to automate this process. The problem with this body of work is that there is little attempt to understand the reasons for using partitioning. As data flows through a circuit, a compression of the state space takes place and some data values are difficult to produce on certain lines. This indicates that information has been lost. Thus it may be difficult to produce desired test vectors on the inputs to some functional units. To alleviate this problem, the circuit is partitioned and additional controllability can be added. The same is true for observability. Data on the output of a particular functional unit (i.e., the result of a test) may not always flow to one of the circuit outputs. This is a loss of (mutual) information. Partitioning can be used to increase observability and thus increase the mutual information between the functional unit and an output.

By using the information theoretic testability measure described in this proposal for partitioning, the underlying motivation for partitioning can be embodied. The basic idea is to use the measure to determine the amount of information compression occurring in a circuit and then using that knowledge to perform

partitioning of the circuit.

In the special case in which testing is performed through the application of (pseudo-) random test vectors, information flow partitioning would seem especially appropriate. When random patterns are applied to test a circuit (via a BIST structure or external test source), these random vectors flow through the circuit and their distribution changes. Eventually the distribution becomes so skewed that the vectors are no longer suitable for testing/exercising subsequent circuit components. The problem is to determine the sets of components that skew the distribution beyond an acceptable level, and partition appropriately.

An algorithm to partition a circuit based on the information flow test measure is listed in Figure 14. The basic premise of the partitioning algorithm is to first push the limits of the partition until signal paths have a controllability less than the desired threshold (Step 2). The partition limits are then pulled back until each path in the partition has an observability greater than the desired threshold (Step 3). This is repeated until the entire circuit has been partitioned.

As an experiment, the circuit shown in Figure 4 was automatically partitioned using the information flow partitioning algorithm (see Figure 15). The circuit was then fault simulated using random pattern test input vectors. The results of this experiment for various partitioning threshold values are listed in Table 5.

A comparison of the fault coverage is listed for each of the two different partitions. In the table, the threshold values are represented as controllability/observability. Since the first partition has higher values for both controllability

Step 1: Levelize the circuit signal paths. Primary inputs are assigned level 0. Assign the value of the maximum level to the variable max_levels.

Step 2: Starting with the level 1 signal paths, compute the CTM values for subsequent signal paths until the CTM falls below a user specified controllability threshold. Collect those paths where the CTM falls below the threshold and call this set C_part.

Step 3: Starting with level 0 signal paths, compute the OTM values between that signal path and the signal paths contained in C_part. For each of the signal paths set its OTM value to be the maximum of the computed OTM values between that path and the paths contained in C_part. Remove from C_part any signal paths which are not among those producing a maximum OTM value. Next, collect each of the paths whose OTM values are less than a user specified observability threshold and call this set O_part. Remove from C_part any signal path not used to obtain the OTM values for the paths in O_part and save them in CO_part. For each path in C_part, replace that signal path by the signal path(s) feeding the component whose output is that path. Recompute the OTM values for the paths in O_part using the new paths in C_part. Repeat until each OTM value is greater than the observability threshold.

Step 4: Set each of the paths contained in CO_part to level 0 and relevelize the circuit (ignoring the previously partitioned components). Steps 2, 3, and 4 are then repeated until the entire circuit has been partitioned.

Figure 14: Algorithm to Partition Circuit

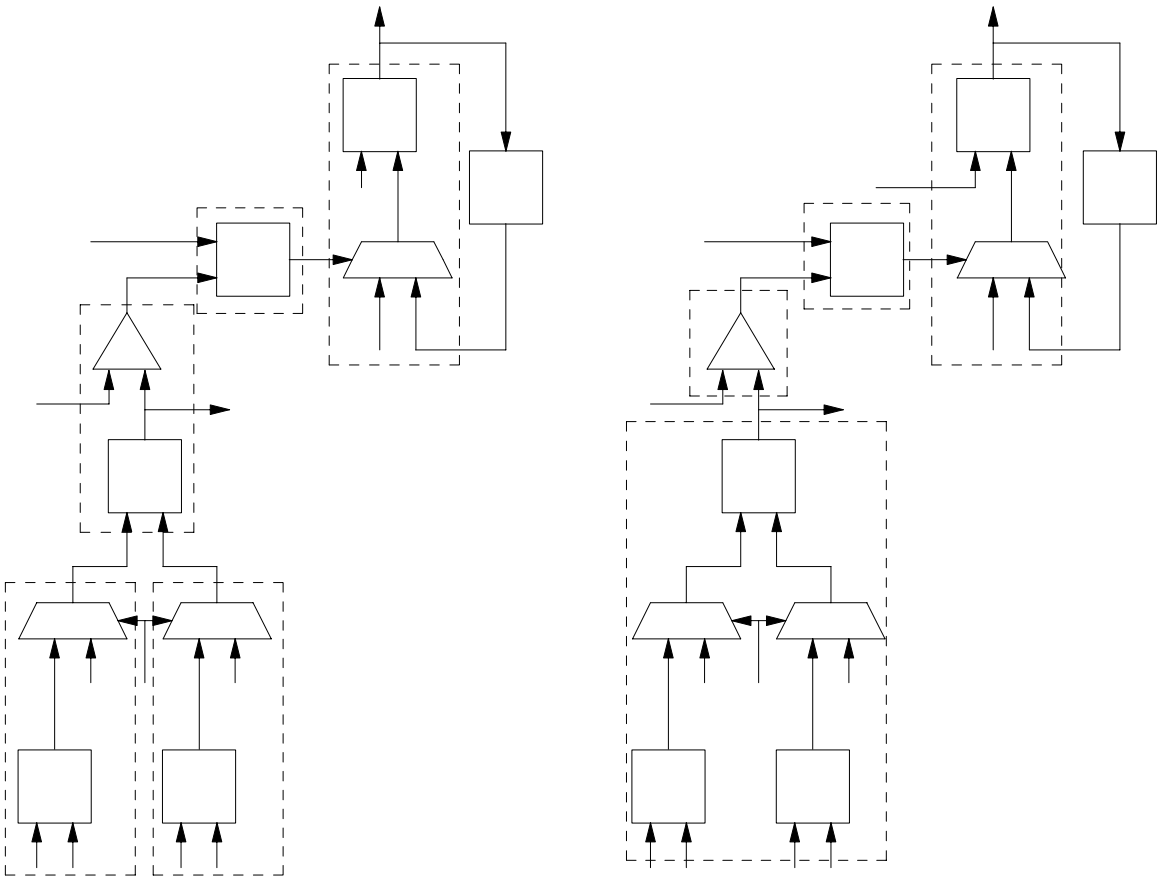

Figure 15: BSSC Circuit Partitions

Table 5: Fault Coverage for 0.4/0.2 and 0.3/0.1 Partitions

Number of Test Vectors	Threshold = 0.4/0.2	Threshold = 0.3/0.1
500	.64	.51
750	.83	.67
1000	.94	.78
1250	.96	.88
1500	.97	.93
2000	.98	.96
3000	.99	.98

and observability, the partitioning algorithm reaches a threshold quicker and thus the first partition has a finer grain when compared to that for the second partition. As can be seen, the circuit partitioned with a threshold value of 0.4/0.2 has a higher fault coverage (as would be expected with finer grained partitioning). In both cases, a high degree of fault coverage has been achieved in a reasonable number of test vectors.

The second experiment in automatic partitioning took the first partition and modified one of the threshold values. Partitions with thresholds of 0.0/0.2 (partitioning based exclusively on observability) and 0.4/0.0 (partitioning based exclusively on controllability) were obtained (see Figure 16). The results of this experiment are listed in Table 6.

As would be expected, the coarser partitions produced test coverage values slightly less than the finer grained partition. But in both cases reasonable results were obtained.

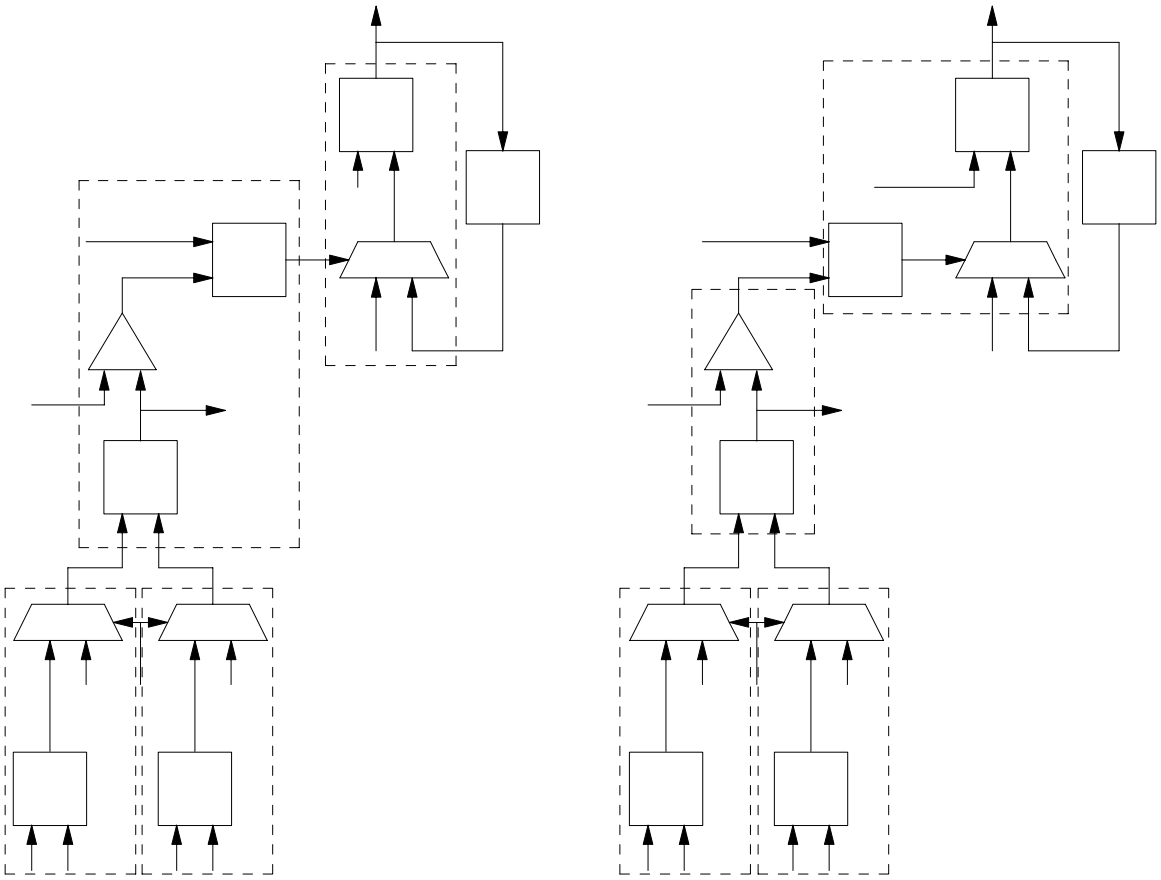

Figure 16: Second Example Partitions

A similar approach can be used to add test points to the circuit. In this test strategy, controllability test points are added first. Then observabilty test points are added. This can be done separately or in conjunction with an ATPG system.

Table 6: Fault Coverage for 0.0/0.2 and 0.4/0.0 Partitions

Number of Test Vectors	Threshold = 0.0/0.2	Threshold = 0.4/0.0
500	.61	.56
750	.77	.69
1000	.90	.85
1250	.94	.91
1500	.96	.94
2000	.98	.96
3000	.99	.98

7. REPRESENTING TEST KNOWLEDGE

7.1 Test Generation Knowledge

As stated earlier, test generation knowledge specifies primary input settings which will achieve a desired set of values on some of the internal nets in the circuit. The values incorporated into test generation knowledge are divided into two groups: 1) *Specified Constants*, and 2) *Symbolic Values*. Specified constants are simply what their name implies, constant values. For example, the numbers 5, -88, and 1024 are all specified constants. On the other hand, a symbolic value is a variable that can take on any value in the circuit's number system. Symbolic values will be represented by the symbol N (or, as N_i, where i is an integer, when multiple symbolic values are needed). For example, if N is a symbolic value included in a piece of test generation knowledge for a circuit using a 16-bit two's complement integer number system, N could represent any valid number in that representation scheme. In addition to simple symbolic values such as N, there are also algebraic combinations of constants and symbolic values. Some examples of such a combination would include $N + 1$, $3 \times N$, and $2 \times N - 3$, as well as $N_1 + 3 \times N_2$.

There is also one additional type of value used during test knowledge extraction, but not included in the final test knowledge base. This is the *symbolic*

arbitrary constant (usually referred to as C or C_i) which is used to represent an arbitrary constant value. This is a hybrid of the concepts of symbolic variables and specified constants. It is used to represent a constant value on a net when that value is not yet known (i.e., before all of the primary inputs feeding that net are assigned). In more formal terms, C is a existential quantifier while N is a universal quantifier.

Test generation knowledge is represented in the form of a list of ordered pairs (or triples, if the circuit is sequential). In representing test generation knowledge, the first entry in the pair/triple is the "name" of the primary input being assigned. The second entry is the value assigned to that primary input and can be either a symbolic value or a specified constant. For sequential circuits, the third entry is a relative time reference which indicates the time offset from zero. The type of circuit will determine the number of entries in the list.

For example, the following list specifies (for a nonsequential circuit) that the primary input B is set to the value 4 and the primary input R is set to the symbolic value N – 1:

{{B,4},{R,N – 1}}.

The following would be a valid assignment of test knowledge for a sequential circuit:

{{A,0,-1},{G,2,0},{A,N,0}}.

This specifies that during the first time step (i.e., T_{-1}), the primary input A is set to the value 0. During the next time step (i.e., T_0), the primary input G is set to the

value 2 and the primary input A is set to the symbolic value N. The convention of using a negative time offset is somewhat arbitrary and does not affect the results of the knowledge extraction process. The convention is used only so that the desired effect will occur at time T_0 in all cases.

Four types of test generation knowledge are extracted from the structure of the compiled circuit. They are:

1) The primary input assignments which justify a symbolic value of N to a specified circuit net.

2) The primary input assignments which justify the symbolic values $N_1, \ldots, N_I$ to the inputs of a specified functional unit in the circuit (where I is the number of inputs for the unit being referred to).

3) The primary input assignments which will propagate some form of the output of a specified functional unit to a primary output.

4) The primary input assignments which will both propagate the output of a specified functional unit to a primary output as well as justify the symbolic values $N_1, \ldots, N_I$ to the inputs of the functional unit.

As an example of test generation knowledge, consider the circuit in Figure 17. This circuit computes the value (A+B) × C if the D input is set to zero and (A+B) × B if the D input is set to one. Using this circuit as an example, the four

types of test generation knowledge will be discussed. In this circuit, the valid numbers are positive fractions ranging from 0 to 1 with 16-bit resolution. Adders wrap around in case of overflow. Multipliers truncate low-order bits to keep a product 16-bits. First consider the justification of the symbolic value N to net N7 (i.e., the output net). This is an example of Type 1 knowledge. One possible set of primary input assignments to justify N to net N7 would be

{{A,N},{B,0},{C,1},{D,1}}.

The only other valid instances of type 1 knowledge for net N7 are

{{A,N – 1},{B,1},{D,0}}

and

{{A,0},{B,1},{C,N},{D,1}}.

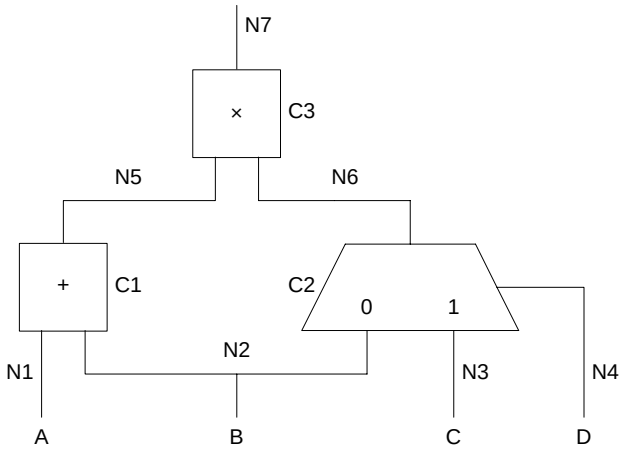

Figure 17: Test Generation Knowledge Example Circuit

Now try to justify the symbolic values N_1 and N_2 to the first and second inputs of functional unit C3, respectively, (i.e., nets N5 and N6). This is an example of type 2 knowledge and corresponds to the application of symbolic test values to the inputs of functional unit C3. One valid assignments for type 2 knowledge for functional unit C3 is

$$\{\{A,N_1\},\{B,0\},\{C,N_2\},\{D,1\}\}.$$

Type 3 knowledge is used to transfer the data on the output of a functional operator to a primary output. As an example of type 3 knowledge, consider functional unit C1. If the output value (net N5) is represented as as a symbolic variable N, the following input settings will produce the value N on the primary output:

$$\{\{B,1\},\{D,0\}\}$$

and {{C,1},{D,1}}. Notice that the first knowledge assignment has the B input set to 1. This would most likely be in conflict with the input settings required to achieve the value N on the output of functional unit C1 since it is influenced by the B input. Therefore, type 3 knowledge assignments usually avoid settings which could be in conflict with type 1/2 knowledge (this will be discussed further in Chapter 8) The second assignment does not have these problems and it is easy to see that the value N is propagated to the primary output (net N7). Since the only symbolic values considered for type 3 knowledge occur on internal circuit nets, the list of primary input settings contains only specified constants and no symbolic values.

It should be noted that the propagation of a symbolic value from a functional unit's output to a primary output need not be exact. Depending on the type of functional unit(s) that the symbolic value is being propagated through, some modification of the symbolic value may be tolerated. Consider the very simple example of an adder functional unit (Figure 18). The goal is to propagate the symbolic value N from the first input to the output. An obvious solution to this problem is to set the other input to the value 0 (i.e., C = 0). But what if it is impossible to set the other input to 0? What if it were held at a constant value of 10? If the adder operated with wraparound in the case of overflow, there would be no information loss and the value N could be recovered on the output by subtracting 10. If there were no wraparound when overflow occurs, there would be some

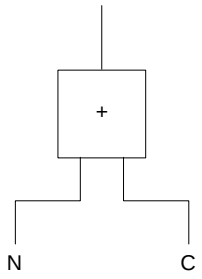

Figure 18: Propagation of a Symbolic Value Through an Adder

loss of information[1] of the input value state space it would not be possible to recover the value of N. The goal is to minimize amount of information loss between the functional unit output and a primary output by choosing appropriate primary input settings.

Returning to the example in Figure 17, a type 3 knowledge assignment of {{C,3},{D,1}} would produce a value of $3 \times N$ on the primary output. Depending on the way overflow is handled by the multiplier (functional unit C3), these primary input assignments may be satisfactory.

Finally let us consider a complete symbolic test for functional unit C1 (both justification of symbolic test values and propagation of the result to a primary output). This is type 4 knowledge and one valid assignment would be:

$$\{\{A,N_1\},\{B,N_2\},\{C,1\},\{D,1\}\}.$$

This will justify both the values N_1 and N_2 on the inputs of functional unit C1 and propagate result to the primary output.

As has been stated previously, there are often a number of valid knowledge assignments which will propagate and/or justify the desired values correctly. We are only interested in determining one specific assignment for any piece of knowledge. Extraction of equivalent pieces of knowledge is unnecessary to solve the problem which is being attacked.

---

[1] In this case, the fraction of the state space that is lost is $\frac{10}{2^n}$, where the data-paths are n bits wide.

The four types of test generation knowledge make up a hierarchical representation scheme (Figure 18). In a circuit free of reconvergent fanout, type 2 knowledge is simply the the union of pieces of type 1 knowledge, and type 4 knowledge is simply the union of type 2 and type 3 knowledge. Without reconvergent fanout, there can be no conflicting input settings for any of the internal data-paths. In circuits without reconvergent fanout, the circuit topology can provide constraints which preclude the simple creation of one type of knowledge through combinations of other types of knowledge lower in the hierarchy. In Section 7.2, the extraction and combination of test generation knowledge will be discussed in detail.

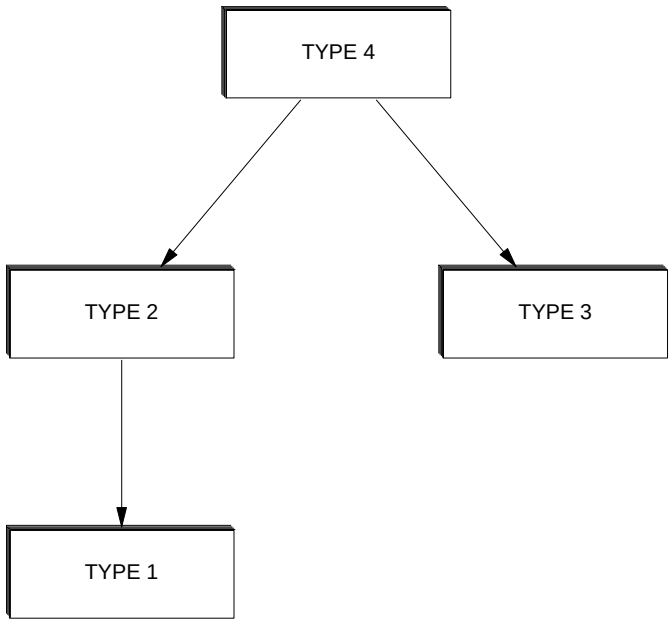

Figure 18: Test Generation Knowledge Hierarchy

7.2 Design for Testability Knowledge

DFT knowledge is produced when it is impossible to generate symbolic tests for some of the functional operators in a circuit. This knowledge is used to intelligently add partial scan-paths [66] to the circuit to make it fully testable. For example, consider the example in Figure 19. In this circuit the second input for adder C2 is fed by the square of the B input (i.e., this circuit computes $A + B^2$). The squaring function is performed by multiplier C1, with both inputs to the multiplier fed by the B input. Obviously, the justification of an arbitrary symbolic test to the second input of C2 is impossible. Only symbolic values that are squares can be justified. Unless the tests for C2 are restricted to vectors whose second input value is a square, it will be impossible to apply all of the test vectors to C2. Note

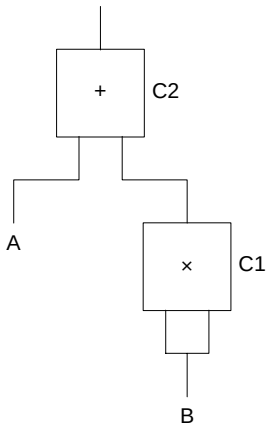

Figure 19: Example Circuit Generating DFT Knowledge

that falling to a gate level in the design will not solve this problem. The functionality of the multiplier is the same at the gate level. However, some faults (detected by tests requiring nonsquares) will now be undetectable.

To alleviate this problem, DFT techniques are used to make the circuit more controllable (i.e., it will be easier to justify a value to some point in the circuit) or more observable (i.e., it will be easier to propagate a value from some point in the circuit to a primary output). DELPHI uses a partial scan approach (rather than full scan) to add a scan latch only where necessary. This will help minimize the area overhead for DFT.

There are often many possible locations in the circuit to add the necessary scan latches and the goal is to minimize the total number of additional scan latches used (and thus minimize the additional area required for DFT). Consider another example circuit in Figure 20. This circuit computes the value A if A > B and the value A × B otherwise. The problem is that it is impossible to justify symbolic tests to both of the inputs of multiplier C2 (unless symbolic test N_2 is always less than N_1). There are four possible locations to add a single scan latch to solve this problem. Those locations are marked by the dotted boxes in Figure 20 and are labeled S1, S2, S3, and S4. If a scan latch is added at location S1, simply set the scanned value to N_1. The test vector for the second input is achieved by setting the A input to 0 (assuming that 0 is the smallest number in the representation scheme) and the B input to symbolic test N_2. This type of knowledge is represented in a list format similar to that for test generation knowledge. In

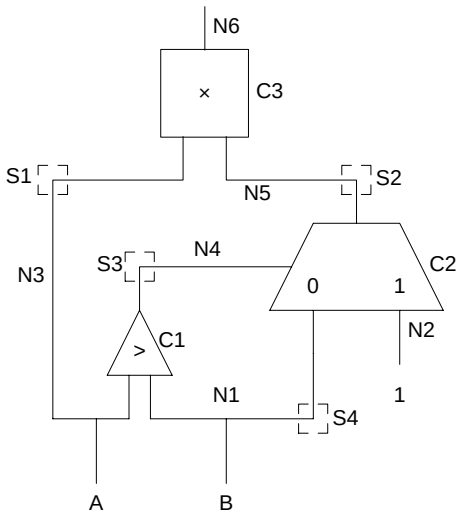

Figure 20: Second Example Circuit Generating DFT Knowledge

addition to the primary input and scan latch settings, the location of the scan latch must be specified. The location is simply the component input line which is to be put in the scan path. In the case of scan latch S1, this is the line in net N3 connected to component C3. This complete piece of DFT knowledge is represented as

$$\{\{N3,C3\},\{\{Scan,N_1\},\{A,0\},\{B,N_2\}\}\}.$$

The first sublist specifies the location of the scan latch and the second sublist contains the input settings to achieve the desired symbolic test (the assignment for “Scan” specifies the value needed to be scanned into the latch).

Notice that there are three other ways to add a scan latch to the circuit and achieve the desired symbolic tests. In addition to the previous piece of DFT

knowledge, three more pieces are generated (corresponding to locations S2, S3, and S4, respectively):

$$\{\{N5,C3\},\{\{Scan,N_2\},\{B,N_1\}\}\}$$

$$\{\{N4\},\{\{Scan,0\},\{A,N_1\},\{B,N_2\}\}\}$$

$$\{\{N1,C2\},\{\{Scan,N_2\},\{A,N_1\},\{B,1\}\}\}$$

Note that in the second piece of DFT knowledge listed above, there is no component specified in the location sublist. Since the net has only two connections, it is not necessary to specify the line to which the scan latch is added since there is only one line.

In addition to justifying symbolic tests, DFT knowledge is also used to propagate the results of symbolic tests. Consider the circuit in Figure 21. This circuit computes the value $A \times (1 - A) \times (C + D)$ where input values are positive fractions (i.e., they range from 0 through 1). The problem occurs when the symbolic test value on net N7 is propagated to the primary output (through multiplier C4). Since the numbers are positive fractions, the quantity $A \times (1 - A)$, which is fed to the other input of the multiplier, has a maximum value of .25. Thus, three-quarters of the bits on N7 are lost when passing through C4. To alleviate this problem, DFT techniques are used. Four possible locations for scan latch insertion are available, and they are labeled S1 - S4 in the figure. The DFT knowledge is used to propagate the symbolic test result on net N7 to a primary output. The four pieces of DFT knowledge are

$$\{\{N7\},\{\{\}\}\}$$

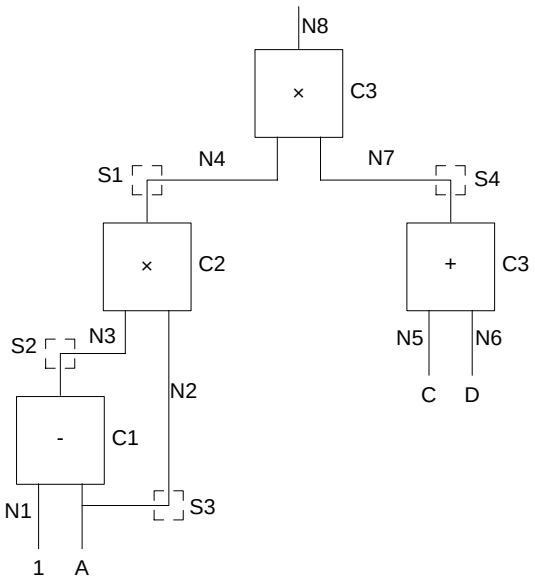

Figure 21: Example Circuit Requiring DFT Knowledge for Test Propagation

{{N4},{{Scan,1}}}

{{N3},{{Scan,1},{A,1}}}

{{N2,C2},{{Scan,1},{A,0}}}

Notice that the first piece of DFT knowledge simply adds a scan latch on the net whose value is to be propagated (and thus needs no additional primary inputs settings). This is obviously the “greedy” way to solve this problem but it is not necessarily the only way (as indicated by the three other pieces of DFT knowledge).

Fanout often offers multiple scan path locations (see Figure 22). Assuming that latches exist on all lines in the net shown in the figure, there are are four

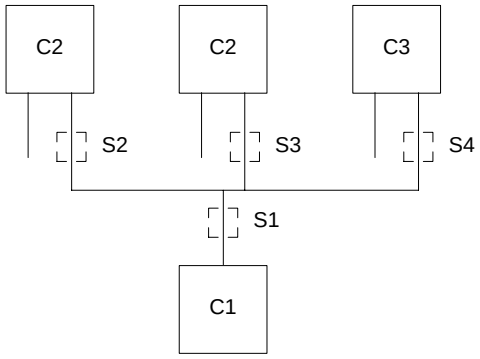

Figure 22: Fanout and Scan Paths

possible locations to add a scan to this net. If this were represented in the form of DFT knowledge, each location would generate its own piece of DFT knowledge:

{{N1,C1},{{}}}

{{N1,C2},{{}}}

{{N1,C3},{{}}}

{{N1,C4},{{}}}

The ability to choose from various scan locations is used to minimize the total scan cost (i.e., the amount of additional area required to implement the scan path). In the next section the techniques used to automatically extract test knowledge from a circuit are described.

8. EXTRACTING TEST KNOWLEDGE

The extraction of test knowledge takes the data flow graph representing the circuit structure and generates test knowledge for the nets and functional operators in the circuit. Each type of functional unit has a unique routine associated with it which is used during the traversal of the structure of the circuit. For example, when an adder is encountered during the extraction of test knowledge, a routine customized for addition is used. In some sense this mimics an object-oriented programming approach towards operating on elements in the circuit. Since there are only a limited number of functional operators available to a user, it is reasonable to provide each operator with a customized traversal routine.

In some cases the desired test knowledge for all components in a circuit cannot be extracted. In those cases, two options are available. The first possibility is to retreat to low-level (gate/transistor level) test generation using available high-level test knowledge from other parts of the circuit. Specific, rather than symbolic, test values are used in this circumstance. Note that will not necessarily completely solve the problem of testing the circuit. Some tests may not be possible even at the gate/transistor level due to inherent physical constraints provided by the circuit. The second possibility is to use the partial test knowledge that has been extracted to intelligently add DFT circuitry. As will be shown in the examples in the next section, this option is very powerful.

For the purposes of knowledge extraction, circuits are segregated into two distinct classes. This first class of circuits is referred to as *nonsequential* and includes the circuits traditionally called combinational. Since many compiled circuits are actually pipelined, they are not strictly combinational. But they also do not exhibit (feedback) sequential behavior in that all inputs for a particular result are applied at the same time. Thus, nonsequential simply means that all test knowledge refers to a single time step. Circuits which require input assignments for a single computational result to be applied during different time steps will be called *sequential*.

Before the general structure traversal techniques are discussed, some formal descriptions of symbolic variables and their flow through a circuit will be introduced.

Definition 1: The value on a w-bit data-path is a *symbolic variable* if and only if that data-path can assume all 2^w possible values.

The set of all symbolic variable values will be referred to as ξ. By definition the cardinality of ξ is 2^w for a w-bit data-path. Most compiled circuits have a constant data-path width throughout the circuit and this convention will be assumed in the work presented in this thesis. In cases of underflow and overflow, some form of truncation must be performed.

Consider the simple case in which a data-path is fully controllable (i.e., it is a primary input). It is obvious that a value on the data-path is a symbolic variable since any arbitrary value can be achieved. This process becomes more

complicated when trying to achieve a symbolic variable on data-paths that are not directly controllable.

To produce a symbolic variable on the output of a functional operator, several options are available. If none of the inputs are symbolic variables, difficulties may arise. In this case there must be enough variation in the input values such that when combined by the functional operator a symbolic variable will be produced. An example of this would be a adder operator with both inputs limited to the values $0, \ldots, 2^{w-1}$. By definition, neither input is a symbolic variable. But by carefully choosing the input pair values, a symbolic variable can be achieved on the output. The problem with this scheme lies in representing all of the different possible distributions of values for a data-path. It would be nearly impossible to generate a systematic procedure to combine distribution representations on input lines and to generate a distribution representation for the output. Since there are 2^{2^w} possible distributions for a w-bit data-path, a double exponential number of input combinations would need to be considered. Therefore, the creation of a symbolic variable from of the combination of nonsymbolic variables will not be considered a viable option.

All the other options available when trying to produce a symbolic variable on the output of a functional operator rely on at least one symbolic variable on an input. The goal is then to transfer one of the symbolic values on the inputs to the output. As will be shown later, it is sufficient to have a single symbolic variable available on an input line. If available, multiple symbolic variables on the inputs

will be used to simplify the search for solutions to the test knowledge extraction problem. But before this problem is examined, a few terms need to be defined.

Definition 2: Let $F[X_1,\ldots,X_I]$ be a functional operator with inputs X_1 through X_I. Let C be an ordered list of constant values $\{C_1,\ldots,C_I\}$. One (and only one) of the constants is set to a marker value which is represented by the "•" symbol. We say that F_C is the *restriction* of F on C if $F_C[X] = F[C_1,\ldots,C_{i-1},X,C_{i+1},\ldots,C_I]$, where $C = \{C_1,\ldots,C_{i-1},\bullet,C_{i+1},\ldots,C_I\}$.

The effect of the restriction is a transformation from an operator with I inputs to an operator with 1 input. All inputs but one are fixed and thus the functionality of the restricted operator is a subset of the original operator. Figure 23 illustrates this transformation. For example, let F be a multiply operator with two inputs such that $F[X_1, X_2] = X_1 \times X_2$. Choosing $C = \{\bullet, 2\}$ yields the restriction $F_C[X] = X \times 2$.

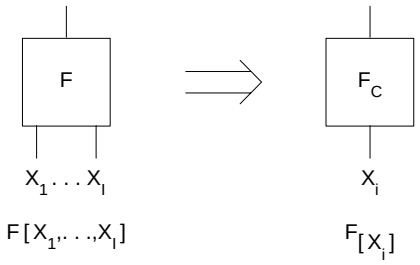

Figure 23: Restriction of F on C

Definition 3: Let X be a symbolic variable. A functional operator F is *symbolic variable preserving* (SVP) if there is a C such that the output of $F_C[X]$ is a symbolic variable.

Theorem 1: A functional operator F is symbolic variable preserving if and only if there is a C such that F_C is bijective (i.e., one-to-one and onto).

Proof: Part 1: If. Since F_C is onto, each value on the output must be mapped to from the input. Thus, by definition, the output is a symbolic variable. Also, since both the input and output are symbolic variables, F_C must be SVP.

Part 2: Only If. Subpart 1 - Onto. If F_C were not onto, there would have to some subset of the range (output) not mapped to by the domain (input). Thus, some output values would not be achievable. Since the definition of a symbolic variable requires every value to be achievable, the output is not a symbolic variable. This implies that F_C is not symbolic variable preserving. Thus F_C must be onto.

Part 2: Only If. Subpart 2 - One-to-one. Since F_C produces only a single output per input setting, the cardinality of the range must be less than or equal to the cardinality of the domain. If F_C were not one-to-one, it would be necessary that two values from the domain map to a single value in the range (i.e., the cardinality of the range would be strictly less than the cardinality of the domain). Since the input is a symbolic variable, the domain is all of ξ. This would indicate that the range was a subset of ξ and violates the requirement that F be onto. Thus F_C must be one-to-one.

The implication is that the restriction of an operator must be bijective in order to propagate a symbolic variable through that operator. Returning to the information theoretic framework developed in Chapter 4, there is no loss of information through the restriction F_C. This indicates that the information transmission coefficient for F_C is 1.0 and the mutual information transmission coefficient between the selected input line of F and the output is 1.0. Thus, if there is a symbolic variable preserving restriction of F, it is possible to transfer a symbolic variable from the input to the output. The concept will be used to pass symbolic variables from one data-path to another in a circuit. The key is in determining the values of C such that F_C will be symbolic variable preserving.

Consider the circuit in Figure 24. The goal is to obtain a symbolic variable on line G by appropriately setting the primary inputs. One way to achieve this goal is to first achieve a symbolic variable on line A. Then, by producing an appropriate constant value on line B, the symbolic variable on line A will “flow” to line C. This process is repeated until the symbolic variable appears on line G. The type of functional operators on the symbolic variable flow path will determine the type of constants that will propagate the symbolic variable. The ability to achieve these constant values will be determined by by the circuitry feeding the specified data-paths.

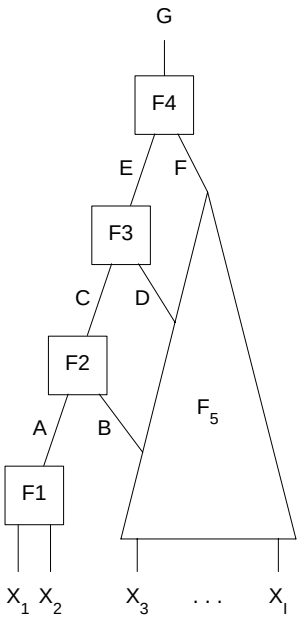

Figure 24: Flow of Symbolic Values Through a Circuit

8.1 Traversal Routines for Individual Operators

The algorithms for each of the operator types available to users of the BSSC and FACE compliers will be described in this subsection. For each operator four types of routines are described. One routine is used to justify a symbolic value to the output of the operator by setting the operator inputs. Another routine operator is used to propagate a symbolic value on an input to the output. There are also routines for justifying arbitrary symbolic constants and specified constants.

8.1.1 Add operator

Theorem 2: If F is an adder (with wraparound), the restriction F_C is symbolic variable preserving for all values of C.

Proof: Under the restriction, $N = F_C[N'] = (N' + C) \bmod 2^w$, where w is the data-path width. Let n'_1 and n'_2 be two arbitrary instances of N' such that $n'_1 > n'_2$. When the value of N' is n'_1, the output of the adder is $n_1 = (n'_1 + C) \bmod 2^w$. When the value of N' is n'_2, the output of the adder is $n_2 = (n'_2 + C) \bmod 2^w$. The difference between these values is

$$\begin{aligned} n_1 - n_2 &= (n'_1 + C) \bmod 2^w - (n'_2 + C) \bmod 2^w \\ &= ((n'_1 + C) - (n'_2 - C)) \bmod 2^w \\ &= (n'_1 + C - n'_2 - C) \bmod 2^w \\ &= (n'_1 - n'_2) \bmod 2^w \\ &\neq 0 \end{aligned}$$

Thus, the two different input values map to different output values, regardless of the value of C (i.e., F_C is one-to-one). Since the cardinality of the domain equals the cardinality of the range, F_C is bijective. By definition, an adder is symbolic variable preserving for any value of C.

The fact that an adder is SVP for any value of C means that there are two ways to justify a symbolic value N to the output of an adder. Simply apply a constant value to one input and a symbolic value to the other input. If the adder wraps around in case of overflow, the constant can be any arbitrary value. On the other hand, if an overflow does not wraparound, the only valid assignment for the other input is zero.

Since it does not matter which input has the symbolic variable and which input has the constant, which goal is achieved first? Since any arbitrary constant can be used, it is trivial to achieve this goal. It is more difficult to achieve the symbolic constant so that this goal will be attempted first. The next thing that must be done is to order the operator inputs for the symbolic variable search. At this point the testability measures described earlier are put to use. Since the controllability testability measure (CTM) measures the flow of information through the circuit, it will be used to order the symbolic variable search. The input with the highest CTM will be tried first and if it fails, the other input will then be attempted. The algorithm to perform the symbolic value justification for an adder is as follows:

Goal: Justify N to the output of an adder.

Step 1: Order the inputs by CTM measure.

Step 2: Attempt to achieve N′ on the input with the highest CTM measure by achieving N′ on the output of the operator feeding the input. If the input is a primary input, store the input name and the value N′ in the test generation knowledge base.

Step 3: If Step 2 fails, attempt to achieve N′ on the input with the lowest CTM measure by achieving N′ on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Achieve constant C on the other input. The type of constant necessary (either arbitrary symbolic or specified) will depend on the implementation of the adder.

Step 6: Replace N′ by N – C in the knowledge base.

To propagate a symbolic value N from one input to the output, simply achieve a constant C on the other input. If the adder wraps around in case of overflow, C can be any arbitrary constant. If the adder does not wrap around, the only possible value for C is zero.

To justify an arbitrary symbolic constant C to the output of an adder, achieve arbitrary symbolic constants C_1 and C_2 on the two inputs. Then set the arbitrary constant C to $C_1 + C_2$. To achieve a specified constant (call this specified value S), achieve an arbitrary symbolic constant C on the input with the smallest CTM value. Achieve the specific constant $S - C$ on the other input.

8.1.2 Subtract operator

Since subtraction is isomorphic to addition, the traversal routine is quite similar. The only major difference is that the inputs are not symmetric (as they are in addition). To put it in the framework of addition requires the second input be negated. This effect simply requires the test knowledge extraction system to keep track of the negated input and make any necessary corrections (as in the algorithms described in this section). Since subtraction and addition are isomorphic, subtraction is bijective which implies that it is also symbolic variable preserving. The algorithm to perform the symbolic value justification for a subtraction operator is as follows:

Goal: Justify N to the output of an subtractor.

Step 1: Order the inputs by the CTM measure.

Step 2: Attempt to achieve N′ on the input with the highest CTM measure by achieving N′ on the output of the operator feeding the input. If the input is a primary input, store the input name and the value N′ in the test generation knowledge base.

Step 3: If Step 2 fails, attempt to achieve N′ on the input with the lowest CTM measure by achieving N′ on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Achieve constant C on the other input. The type of constant necessary (either arbitrary symbolic or specified) will depend on the implementation of the adder.

Step 6: If the symbolic variable is being subtracted from a constant, replace N′ by $C - N$ in the knowledge base. If the constant is being subtracted from the symbolic variable, replace N′ by $N + C$ in the knowledge base.

To propagate a symbolic value N from one input to the output, simply achieve a constant C on the other input. If the subtractor wraps around in case of overflow, C can be any arbitrary constant. If the subtractor does not wrap around, the only possible value for C is zero.

To justify an arbitrary symbolic constant C to the output of an adder, achieve arbitrary symbolic constants C_1 and C_2 on the two inputs. Then set the arbitrary constant C to $C_1 - C_2$. To achieve a specified constant (call this specified value S), achieve an arbitrary symbolic constant C on the input with the smallest CTM value. Achieve the specific constant $(C + S)$ or $(C - S)$ on the other input (depending on which input has the arbitrary constant and which one has the specific constant).

8.1.3 Multiply operator

Theorem 3: If F is a multiplier (with wraparound), the restriction F_C is symbolic variable preserving if and only if C is odd.

Proof: Under the restriction, $N = F_C[N'] = (N' \times C) \bmod 2^w$, where w is the data-path width.

Case 1: C is an even number. Rewrite C as $2 \times C'$. Choose two different instances of the symbolic variable N' (call them n'_1 and n'_2) such that $n'_1 < n'_2$ and that $n'_2 = n'_1 + 2^{w-1}$. When the value of N' is n'_1, the output of the multiplier is $n_1 = (n' \times C) \bmod 2^w$. When the value of N' is n'_2, the output of the multiplier is

$$
\begin{aligned}
n_2 &= (n'_2 \times\ ) \bmod 2^w \\
&= ((n'_1 + 2^{w-1}) \times C) \bmod 2^w \\
&= ((n'_1 \times C) + (2^{w-1} \times C) \bmod 2^w \\
&= ((n'_1 \times C) \bmod 2^w + (2^{w-1} \times C) \bmod 2^w) \bmod 2^w \\
&= (n_1 + (2^{w-1} \times C) \bmod 2^w) \bmod 2^w \\
&= (n_1 + (2^{w-1} \times 2 \times C') \bmod 2^w) \bmod 2^w \\
&= (n_1 + 0) \bmod 2^w \\
&= n_1.
\end{aligned}
$$

Thus, both values of N' map to the same value of N, and for even values of C, the multiply operator is not one-to-one. By extension, the multiply operator is not SVP for even values of C.

Case 2: C is an odd number. Rewrite C as $2 \times C' + 1$. Consider two different instances of the symbolic variable N' and call them n'_1 and n'_2. Without loss of

generality, assume that $n'_1 < n'_2$ and that $n'_2 = n'_1 + d$. When the value of N' is n'_1, the output of the multiplier is $n_1 = (n' \times C) \bmod 2^w$. When the value of N' is n'_2, the output of the multiplier is

$$\begin{aligned} n_2 &= (n'_2 \times C) \bmod 2^w \\ &= ((n'_1 + d) \times C) \bmod 2^w \\ &= ((n'_1 \times C) + (d \times C)) \bmod 2^w \\ &= ((n'_1 \times C) \bmod 2^w + (d \times C) \bmod 2^w) \bmod 2^w \\ &= (n_1 + (d \times (2 \times C' + 1)) \bmod 2^w) \bmod 2^w \\ &\neq n_1 \end{aligned}$$

Since $(d \times (2 \times C' + 1)) \bmod 2^w$ cannot equal zero, the two values n'_1 and n'_2 cannot map to the same value. Thus, when C is odd the multiply operator is one-to-one. Since the cardinality of the input and output is the same, the multiply operator is also bijective. Therefore, when the fixed input is odd, the multiply operator is SVP.

The fact that a multiplier is SVP only in cases where C is odd introduces some difficulties to the multiplier traversal routine. Obviously, an arbitrary symbolic constant will not (always) satisfy the requirements that C be odd. A new variable type could be introduced which is an “arbitrary odd symbolic constant” but this would only complicate the traversal routines for other operators. Instead, specific (odd) constants are attempted, repeating if necessary (up to a specified backtrack limit). As with the adder, the symbolic variable is attempted first and the input ordering is by CTM.

The algorithm to justify a symbolic value to the output of the multiply operator is as follows:

Goal: Justify N to the output of an multipler.

Step 1: Order the inputs by the CTM measure.

Step 2: Attempt to achieve N′ on the input with the highest CTM measure by achieving N′ on the output of the operator feeding the input. If the input is a primary input, store the input name and the value N′ in the test generation knowledge base.

Step 3: If Step 2 fails, attempt to achieve N′ on the input with the lowest CTM measure by achieving N′ on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Attempt to achieve a specific constant S with a value of 1 on the other input. If this fails, attempt with $S = -1$. If this again fails, attempt $S = 3$. Repeat this process until either the attempted constant is achieved of the backtrack limit reached. If the backtrack limit is reached, save the input line in the DFT knowledge base and quit.

Step 6: Replace N′ by N / S in the knowledge base.

This algorithm assumes that there is wraparound in the state space in the case of overflow (or underflow). The specific implementation of the multiplier will most likely dictate slight changes to this process. To propagate the symbolic value N from one input to the output, simply achieve an odd specific constant on the other input, repeating as necessary up to a specified backtrack limit. To justify an arbitrary symbolic constant C to the output of an adder, achieve arbitrary symbolic constants C_1 and C_2 on the two inputs. Then set the arbitrary symbolic constant C to $C_1 \times C_2$. To achieve a specified constant S, iteratively factor S into

S_1 and S_2 and attempt to achieve these values on the inputs (trying both input orderings). Repeat this procedure if necessary, up to a (user) specified backtrack limit.

8.1.3 Negate operator

The negate operator simply inverts the sign of the single input. It would appear that no information is lost through the function and the operator is symbolic variable preserving. This is true for most values but in some number systems (namely, the two's complement number system) the values are not symmetric about zero. Two's complement values range from -2^w to $2^w - 1$. This means that the negation of -2^w does not map to another valid number in this representation scheme. The specific implementation of the negation operator will effect the mapping and there may be a simple solution to the problem. In any case, there is a problem with only one of the 2^w values. Therefore, the missing output value will be ignored for symbolic variables and will be handled as a special case (when necessary).

To justify a symbolic variable N to the output of a negate operator, simply achieve the symbolic variable N′ on the input where $N' = -N$. The propagation of a symbolic variable from the input to output is automatic and needs no assistance. To justify an arbitrary symbolic constant to the output, simply achieve an arbitrary symbolic constant on the input. To propagate a specific constant to the output, achieve a specific constant whose value is the negation justified value. If

there is a conflict due to any asymmetry in the number system, give up and return to the routine attempting to achieve the specific constant.

8.1.4 Multiplex operator

Since data values are transferred directly from one input to the output, it is trivial to show that a multiplexor is symbolic variable preserving. In trying to justify a symbolic variable N to the output, the only problem that is encountered is in trying to decide which of the two input data-paths to select via the control input. If both inputs contained symbolic variables, the control input could be an arbitrary symbolic constant. If only one of the inputs contained a symbolic variable, the control input would have to be set to the appropriate specified constant value.

As will be shown later, the extraction of type 1 knowledge is done in level-order for the entire circuit. In this way previously extracted type 1 knowledge can be used to create additional type 1 knowledge as it flows through the circuit. If type 1 knowledge exists for each of the multiplexor inputs, the control input is simply set to an arbitrary symbolic constant. The symbolic variable is simply the input specified by the value of the constant. If type 1 knowledge exists for only one of the inputs, the control is set to the specific constant representing the input with the existing type 1 knowledge. The propagation of a symbolic constant from an input to the output is handled in a similar manner. The control input is set to the appropriate specific constant which selects the input to be propagated.

To justify an arbitrary symbolic constant to the output of a multiplexor, first achieve an arbitrary symbolic constant on the control input. Then achieve an arbitrary symbolic constant on the input selected by the value on the control input.

The justification of a specific constant to the output of the multiplexor is slightly more complicated. The question as to which input should be search first, either control or data, needs to be addressed. The solution that will be presented here searches for the specific constant on one of the multiplexor inputs and then attempts to achieve the required control input by then searching for the appropriate value on the control input. The algorithm is as follows:

Goal: Justify the specified constant S to the output of an multiplexor.

Step 1: Order the inputs by CTM measure.

Step 2: Attempt to achieve S on the input with the highest CTM measure by achieving S on the output of the operator feeding the input. If the input is a primary input, store the input name and the value of S in the test generation knowledge base.

Step 3: If Step 2 fails, attempt to achieve S on the input with the lowest CTM measure by achieving S on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Attempt to achieve a specific constant on the control input which will propagate the data input satisfied in Steps 2 and/or 3. If this fails and Step 3 has not been attempted, go to Step 3. If this fails and Step 3 has been attempted, save the input line in the DFT knowledge base and quit.

8.1.5 AND operator

The AND operator considered in this section is a bitwise operation, with each pair of input bits influencing one output bit. Traditionally, the propagation and justification of test values through gates has been the bread and butter of the test generation process.

Theorem 4: If F is an AND operator, the restriction F_C is symbolic variable preserving if and only if every bit in C is a 1 (i.e., $C = 2^w - 1$).

Proof: Under the restriction, $N = F_C[N'] = N' \ C$. Rewrite N' and C as their binary expansions $N'^1 N'^2, \ldots, N'^{w-1} N'^w$ and $C^1 C^2, \ldots, C^{w-1} C^w$. Let n'_1 and n'_2 be two instances of N' such that they differ in only bit position i. Now choose a value for C such that $C^1 = 0$. Applying these two values of N' to the restriction produces the values $n_1 = n'_1 \ C$ and $n_2 = n'_2 \ C$. Since all bits other than bit i in n'_1 and n'_2 are the same, n_1 and n_2 must also have the same values for these same bits. Since the value of C^i is a 0, $n_1^i = n_2^i = 0$. This implies $n_1 = n_2$. Therefore, any bit in C that is set to 0 will allow two different input values to map to the same value. This violates the requirement that an SVP function be bijective; thus, the AND operator is not SVP for any C with zero valued bits.

Now consider the value of C which has every bit set to 1. The resulting restriction F_C is simply the identity function and is thus bijective and symbolic variable preserving.

The algorithm to justify a symbolic value to the output of an AND operator is as follows:

Goal: Justify N to the output of an AND operator.

Step 1: Order the inputs by CTM measure.

Step 2: Attempt to achieve N′ on the input with the highest CTM measure by achieving N′ on the output of the operator feeding the input. If the input is a primary input, store the input name and the value N′ in the test generation knowledge base.

Step 3: If Step 3 fails, attempt to achieve N′ on the input with the lowest CTM measure by achieving N′ on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Attempt to achieve a specific constant S with a value of $2^w - 1$ on the other input. If this fails, save the input line in the DFT knowledge base and quit.

Step 6: Replace N′ by N in the knowledge base.

To propagate a symbolic constant from one input to the output, a value of $2^w - 1$ must be achieved on the other input. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. In justifying a specific constant to the output of an AND gate, we rely on techniques developed by previous researchers (i.e., Kunda et al. [43]) to achieve the desired goal.

8.1.5 OR operator

As with the AND operator, the OR operator is a bitwise operation, with each pair of input bits influencing one output bit.

Theorem 4: If F is an OR operator, the restriction F_C is symbolic variable preserving if and only if every bit in C is a 0 (i.e., C = 0).

Proof: Under the restriction, $N = F_C[N'] = N' \mid C$. Rewrite N' and C as their binary expansions $N'^1 N'^2, \ldots, N'^{w-1} N'^w$ and $C^1 C^2, \ldots, C^{w-1} C^w$. Let n'_1 and n'_2 be two instances of N' such that they differ in only bit position i. Now choose a value for C such that $C^1 = 1$. Applying these two values of N' to the restriction produces the values $n_1 = n'_1 \mid C$ and $n_2 = n'_2 \mid C$. Since all bits other than bit i in n'_1 and n'_2 are the same, n_1 and n_2 must also have the same values for these same bits. Since the value of C^1 is a 1, $n_1^i = n_2^i = 1$. This implies $n_1 = n_2$. Therefore, any bit in C that is set to 1 will allow two different input values to map to the same value. This violates the requirement that an SVP function be bijective; thus, the and operator is not SVP for any C with nonzero bits.

Now consider value of C which has every bit set to 0. The resulting restriction F_C is simply the identity function and is thus bijective and symbolic variable preserving.

The algorithm to justify a symbolic value to the output of an OR operator is as follows:

Goal: Justify N to the output of an OR operator.

Step 1: Order the inputs by CTM measure.

Step 2: Attempt to achieve N' on the input with the highest CTM measure by achieving N' on the output of the operator feeding the input. If the input is a primary input, store the input name and the value N' in the test generation knowledge base.

Step 3: If Step 2 fails, attempt to achieve N' on the input with the lowest CTM measure by achieving N' on the output of the operator feeding the input.

Step 4: If both Steps 2 and 3 fail, save the input line in the DFT knowledge base and quit.

Step 5: Attempt to achieve a specific constant S with a value of 0 on the other input. If this fails, save the input line in the DFT knowledge base and quit.

Step 6: Replace N′ by N in the knowledge base.

To propagate a symbolic constant from one input to the output, a value of 0 must be achieved on the other input. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. To justify a specific constant, previously developed techniques [43] are used.

8.1.6 NOT operator

The NOT operator is very similar to the negate operator with the exception that symmetry about zero does not influence the mapping. Since each bit of the input is simply inverted, there is no information loss between the input and output and the operator is symbolic variable preserving.

To justify a symbolic variable N to the output of a negate operator, simply achieve the symbolic variable N′ on the input where $N' = \neg N$. The propagation of a symbolic variable from the input to output is automatic and needs no assistance. To justify an arbitrary symbolic constant to the output, simply achieve an arbitrary symbolic constant on the input. To propagate a specific constant to the output, achieve a specific constant whose value is the bitwise negation of the justified value.

8.1.7 Shift left operator

The shift left operator takes the input value and produces a copy of the input shifted one bit to the left (i.e., it multiplies by two). The high-order bit is lost. Obviously, this loss of information indicates that the shift left operator is not symbolic variable preserving and cannot be used to justify or propagate a symbolic variable. Instead, DFT knowledge is collected when a shift left operator is traversed.

To justify an arbitrary symbolic constant to the output of a shift left operator, simply achieve an arbitrary symbolic constant on the input. To justify a specified constant to the output of a shift left operator, the constant cannot have a low-order digit of 1. If if does, give up and return to the routine attempting to achieve the specific constant. If the low order digit is 0, simply achieve a constant on the input whose value is one-half the value to be justified to the output.

8.1.8 Shift right operator

The shift right operator takes the input value and produces a copy of the input shifted one bit to the right (i.e., it divides by two). The low-order bit is lost. Obviously, this loss of information indicates that the shift right operator is not symbolic variable preserving and cannot be used to justify or propagate a symbolic variable. Instead, DFT knowledge is collected when a shift right operator is traversed.

To justify an arbitrary symbolic constant to the output of a shift right operator, simply achieve an arbitrary symbolic constant on the input. To justify a specified

constant to the output of a shift right operator, the constant cannot have a high order digit of 1. If if does, give up and return to the routine attempting to achieve the specific constant. If the high order digit is 0, simply achieve a constant on the input whose value is twice the value to be justified to the output.

8.1.9 Comparison operators

The comparison operators differ from the other type of operators considered earlier since their entire output space contains only two elements, 0 and 1. A symbolic variable on the output need only take two values for the 2^w possible different inputs. To justify a symbolic variable to the output, a symbolic variable on the input is varied relative to the constant on the other input. This variation on the input will provide the necessary variation on the output to produce a symbolic variable.

Consider the EQUAL comparator. To justify a symbolic variable to the output of the comparator, simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output, simply set the symbolic variable value to the value of the arbitrary symbolic constant. To produce a 0 on the output, simply set the symbolic variable to a different value. To propagate a symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output of an EQUAL

comparator, simply achieve arbitrary symbolic constants on both inputs. To achieve a specified constant of 0 on the output, achieve an arbitrary symbolic value on one input and a specified constant with a different value on the other input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 1 on the output, achieve an arbitrary symbolic value on one input and a specified constant with the same value on the other input (this process may be repeated as necessary up to a specified backtrack limit).

A description of the algorithms for the other comparator operators is listed below.

NOT_EQUAL: To justify a symbolic variable to the output of the comparator simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output, simply set the symbolic variable value to the value different than the arbitrary symbolic constant. To produce a 0 on the output, simply set the value of symbolic variable to the value of the arbitrary symbolic constant. To propagate a symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. To achieve a specified constant of 1 on the output, achieve an arbitrary symbolic value on one input and a specified constant with a different value on the other

input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 0 on the output, achieve an arbitrary symbolic value on one input and a specified constant with the same value on the other input (this process may be repeated as necessary up to a specified backtrack limit).

GREATER_THAN: To justify a symbolic variable to the output of the comparator, simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output simply set the symbolic variable value to the value larger than the arbitrary symbolic constant. To produce a 0 on the output simply set the symbolic constant to a value less than or equal to the value of the symbolic variable. To propagate a symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. To achieve a specified constant of 1 on the output, produce an arbitrary symbolic value on one input and a specified constant with a larger (or smaller, depending on which input has which value) value on the other input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 0 on the output, invert the reasoning required to achieve a 1.

LESS_THAN: To justify a symbolic variable to the output of the comparator, simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output, simply set the symbolic variable value to the value smaller than the arbitrary symbolic constant. To produce a 0 on the output simply set the symbolic constant to a value greater than or equal to the value of the symbolic variable. To propagate a symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. To achieve a specified constant of 1 on the output, achieve an arbitrary symbolic value on one input and a specified constant with a smaller (or larger, depending on which input has which value) value on the other input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 0 on the output, invert the reasoning required to achieve a 1.

GREATER_THAN_OR_EQUAL: To justify a symbolic variable to the output of the comparator, simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output, simply set the symbolic variable value to the value larger than the arbitrary symbolic constant. To produce a 0 on the output simply set the symbolic constant to a value less than or equal to the value of the symbolic variable. To propagate a

symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output simply achieve arbitrary symbolic constants on both inputs. To produce a specified constant of 1 on the output, achieve an arbitrary symbolic value on one input and a specified constant with a larger (or smaller, depending on which input has which value) value on the other input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 0 on the output, invert the reasoning required to achieve a 1.

LESS_THAN_OR_EQUAL: To justify a symbolic variable to the output of the comparator, simply achieve a symbolic variable on one of the inputs and an arbitrary symbolic constant on the other input. To produce a 1 on the output, simply set the symbolic variable value to the value smaller than the arbitrary symbolic constant. To produce a 0 on the output, simply set the symbolic constant to a value greater than or equal to the value of symbolic variable. To propagate a symbolic variable from an input to the output, the other input must also be justified with a symbolic variable. The second symbolic variable is varied as necessary to propagate the result of a test to the output of the comparator. To justify an arbitrary symbolic constant to the output, simply achieve arbitrary symbolic constants on both inputs. To achieve a specified constant of 1 on the output, achieve an arbitrary symbolic value on one input and a specified constant with a

smaller (or larger, depending on which input has which value) value on the other input (this process may be repeated as necessary up to a specified backtrack limit). To achieve a specified constant of 0 on the output, invert the reasoning required to achieve a 1.

8.1.10 Delay operator

The delay operator simply passes a value through without modifying it; to produce some value on the output simply achieve that same value on the input. The only effect of a delay operator is to shift time. Thus if the knowledge extraction process were searching backward and passes through a delay operator, the time reference will be shifted one time step toward the past. This will be further discussed in Section 8.3

8.2 General Structure Traversal for Nonsequential Circuits

All knowledge extracted from nonsequential circuits refers to a single time step; therefore, no timing information is included in the test knowledge base that is generated. Note that the latency of pipelined circuits will delay the propagation of a test result to a primary output. This latency is not included in the test generation knowledge. The general traversal routines for the four types of knowledge are fairly simple. Figure 25 illustrates the process to extract type 1 knowledge for an arbitrary net in the circuit. When type 1 knowledge is extracted, it is done in level order so that previously generated type 1 knowledge can be combined to

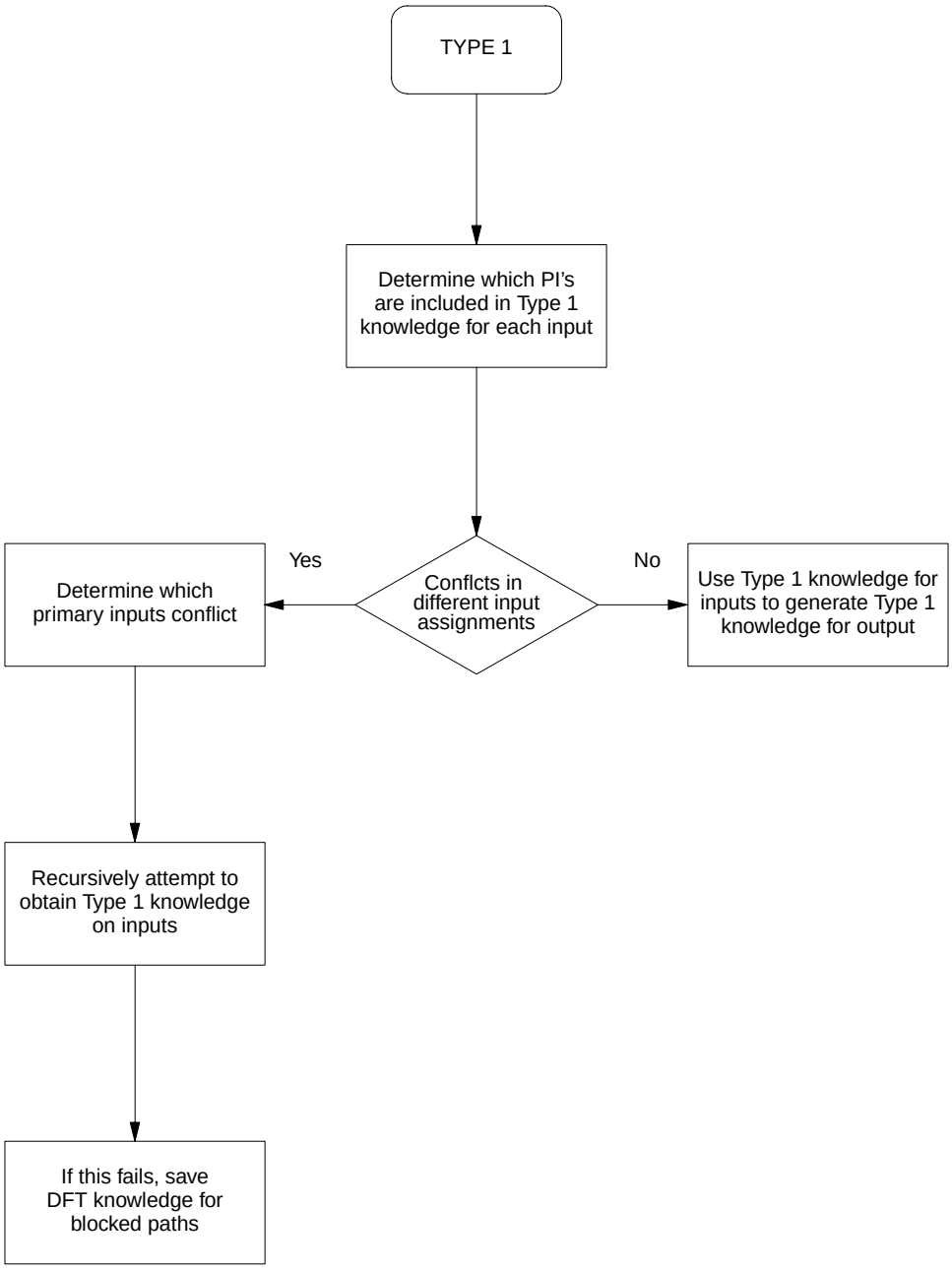

Figure 25: Algorithm to Extract Individual Piece of Type 1 Knowledge

create new type 1 knowledge. In circuits with no reconvergent fanout, this combination of type 1 knowledge can be used to create all of the type 1 knowledge for a circuit.

Type 2 knowledge is extracted using the previously extracted type 1 knowledge. If there is no conflict between the individual pieces of type 1 knowledge feeding a particular operator, type 2 knowledge can be created using simple symbolic manipulation (i.e., N is replaced by N_i in the type 1 knowledge for the i'th input). When there is a conflict, new (and unconflicting) type 1 knowledge must be extracted for the inputs.

Type 3 knowledge is also related to type 1 knowledge. Instead of starting at a primary input, type 3 knowledge starts with a symbolic variable at the output of a particular component. Using the type 1 knowledge on the other inputs along the path between the initial symbolic variable and a primary output, the values necessary for propagation may be achieved. If there is conflict, backtracking to achieve unconflicting type 1 knowledge is performed.

Type 4 knowledge simply combines type 2 and type 3 knowledge. When conflicts arise, backtracking to remove the conflict is performed.

Thus the techniques to extract the four types of test knowledge are conceptually very simple. In many circuits, little backtracking will be necessary; thus, little or no pruning of the search space is performed (outside of the custom traversal routines for each operator). Some heuristics are used to help avoid conflicts but they are only to order the backtracking. One heuristic simply avoids

attempting to achieve symbolic variables on primary inputs shared by conflicting pieces of type 1 knowledge. Converserly, arbitrary symbolic variables are specifically directed toward shared inputs. Each component has a listing of primary inputs which are shared by its inputs, and this information is used during the traversal routines. Some operators can be used to block the paths from a primary input by appropriately setting the other input. This effect allows for other heuristics to be used to remove the influence of shared inputs. Achieving a zero on one input of a multiplier removes the influence of the other input. Similar heuristics are used for AND gates, OR gates, and multiplexors.

As stated earlier, the recursive searching that is performed does very little pruning of the search space. If there were no pruning of the search space, it would be trivial to show that the test knowledge extraction system is complete (i.e., all possible input combinations are considered), assuming that any backtrack limits are set to infinity. The one type of pruning that is performed is done then attempting to generate type 1 knowledge by propagating a symbolic variable through an operator. Figure 26 illustrates this point.

The goal is to propagate a symbolic variable from primary input D through operator C4. Assume that there is a setting for inputs A, B, C, and D such that a symbolic variable is propagated through C2 and C3. The problem occurs when trying to propagate through C4. The other input cannot be justified with an appropriate constant value which will allow the symbolic value to propagate. At this point the search space is pruned and no attempt to propagate a symbolic

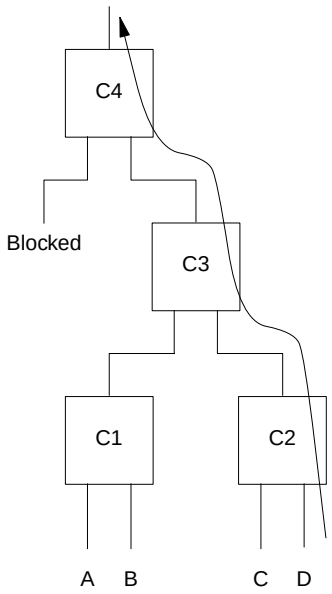

Figure 26: Pruning the Search Space at a Blocked Path

variable from inputs A, B, or C is attempted, since there would still be a blocked path at C4. Another consequence of the blocked path is that the blocking input is also not searched for a symbolic variable. Since the desired constant cannot be achieved on that input, it would be impossible to achieve an arbitrary value (e.g., a symbolic variable) on that line. At this point the search retreats and attempts to achieve the symbolic variable via another path.

8.3 General Structure Traversal for Sequential Circuits

The traversal of the structure of a sequential circuit is similar in most respects to the traversal of nonsequential circuits. As stated earlier, a time reference is added to test knowledge to indicate relative time indices. When delay operators are traversed, the time index is incremented (or decremented, depending on the traversal direction). Conceptually each individual time frame replicates the circuit; thus, each time a delay operator is traversed the search space increases. To mitigate this problem, the search is ordered such that delay paths are searched last.

The major difference between sequential and nonsequential occurs when traversing backwards through a delay operator. In cases in which the justification of a symbolic variable is the goal of a backwards traversal, this fact is stored at the delay operator when it is encountered. If during the same search an attempt is made to again traverse that same delay operator, the search is stopped and another (different) search to achieve the desired goal is attempted. This prevents a simple variable transformation from putting the knowledge extraction routine into a never-ending loop. A very simple example illustrating this point is shown in Figure 27. When the search reaches operator C4, a choice is made to attempt to achieve a symbolic variable on the second input (which is fed by the delay operator). After the search traverses the delay operator, it is back in the same situation, albeit at a different (relative) point in time. If the same traversal back through the delay operator is attempted, the loop is detected and the search is aborted. At this point the symbolic variable is search for on the first input (i.e.,

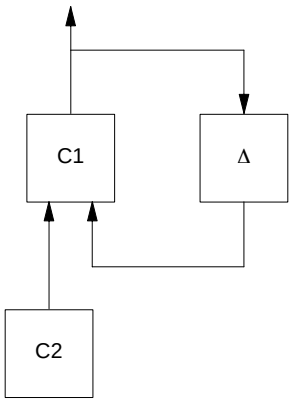

Figure 27: Avoiding Never-Ending Loops in Sequential Circuit Search

the input fed by operator C2).

Other than with the delay operators, the structure of a sequential circuit is searched in the same way as a nonsequential circuit. The next chapter presents the results of applying the test knowledge extraction techniques described here.

9. KNOWLEDGE VERIFICATION

As stated earlier, it is not always the case that the users of DELPHI are of no help in the test generation process. The user may be able to provide useful knowledge and we want DELPHI to be flexible and to make the best use of this additional knowledge. The most helpful knowledge that a user can provide is information about resetting sequential data paths. This reset information may be difficult to extract from the circuit structure, but it may be a simple series of steps that the user has defined in the design. In addition, the use of user provided knowledge may produce simpler symbolic test sets since a user understands "the big picture." Using a forward chaining [67] approach to symbolic simulation, verification of test knowledge is done efficiently in DELPHI. Knowledge to be verified is given in a format similar to that for extracted knowledge. The only additional information required by the test knowledge verification procedure is the circuit component (either net or operator) where the knowledge is directed. The algorithm used to verify test knowledge is as follows:

Step 1: Assign user specified input values to appropriate circuit nets.

Step 2: Mark all functional operators with at least one input net assigned and whose output net is not assigned (all for the same time step). For all marked operators, check to see that the input assignments are sufficient to produce an output assignment. If the input assignments are not sufficient, remove the mark. In addition, if the output of a marked

operator does not influence the knowledge being verified, remove the mark.

Step 3: For all marked operators, compute the output net value using the assigned input net values (the result may be symbolic). Assign the computed value to the output net.

Step 4: Iterate Steps 2 and 3 until there is no change in the assignments or an assignment occurs for the desired piece of test knowledge.

Step 5: Compare simulated value from Step 4 (if it exists) with the user supplied value. If they agree, add user provided test knowledge to knowledge base. If they do not agree (or there is no value for the component of interest), inform user that supplied knowledge is not correct.

Since there is no search performed, verification is guaranteed to take time less than or equal to the time required to extract that same knowledge. In situations where the search space is large, a great deal of time savings can be achieved by avoiding the backtracking which may occur during extraction. In the next section it will be shown that test knowledge can be verified by DELPHI faster than it was extracted.

As an example of verification, consider the circuit shown in Figure 3. Let us modify the operation of the circuit slightly such that the R input is tied to a constant value of .5. A user could provide the test knowledge necessary to apply symbolic tests to the adder unit feeding output O2. Let us say that that user provides the following piece of knowledge to verify:

{{K,0,-1},{Z,N_2 – .5,-1},{S,1,0},{X3,.5,0},{Y3,.5,0},{T,0,0},{K,1,0},{Z,N_1,0}}

The inputs that are assigned (for various time frames) are S, T, K, X3, Y3, and Z. Using symbolic simulation from K (at time T_{-1}), the output of the AND gate will be zero (and thus the input of multiplexor that will be selected is R). Thus the output adder will have inputs $N_2 - .5$ and .5 (producing a value of N_2). This is then fed to the delay operator which shifts time forward one step to T_0. When all values settle down, the inputs to the adder have the symbolic values N_1 and N_2, which is the desired goal.

Now suppose that the user forgot to specify the input assignment {K,1,0}, which is necessary to get the proper setting on the multiplexor control input. Eventually all the simulated values will settle down but the second input to the adder will not be assigned (since the multiplexor input feeding that input has not been selected). Thus, the desired knowledge has not been achieved and the user provided test knowledge is not added to the test knowledge base.

An evaluation of the performance of the verification procedure is discussed in the next chapter.

10. RESULTS

The test knowledge extraction and verification techniques described in the previous two sections were implemented in a SUN workstation environment. The resulting system is named DELPHI and has been applied to a number of compiled circuits. The system was implemented using the Mathematica symbolic mathematics program [68]. This greatly simplified the handling of the necessary symbolic mathematics but undoubtably reduced performance due to the overhead of unnecessary (to DELPHI) features provided by Mathematica. The data structures used are lists of the same form as described in Chapter 7. Each operator has custom traversal routines which operate in an object-oriented programming manner. Circuits are specified in a net-list type format with each component representing a single object. A component is either an operator or net, with primary inputs and outputs tagged as a special instance of a net. The timing results presented in this chapter were obtained using the Mathematica “Timing” function.

The goal was to evaluate DELPHI using circuits actually generated in industry rather than the toy circuits typically used to evaluate new techniques. To this end, the circuits described in this section were developed by persons in industry using either the bit-serial silicon compiler (BSSC) or the flexible architecture compilation environment (FACE).

10.1 Nonsequential Circuits

One of the more complicated compiled circuits given to DELPHI was the 102K transistor video processor known as the CMC chip shown in Figure 5. This is a pipelined circuit, but since there is no feedback, all inputs are applied simultaneously. The basic structure of this circuit is illustrated in the dataflow diagram shown in Figure 28. There are 50,860 faults in the gate level representation of

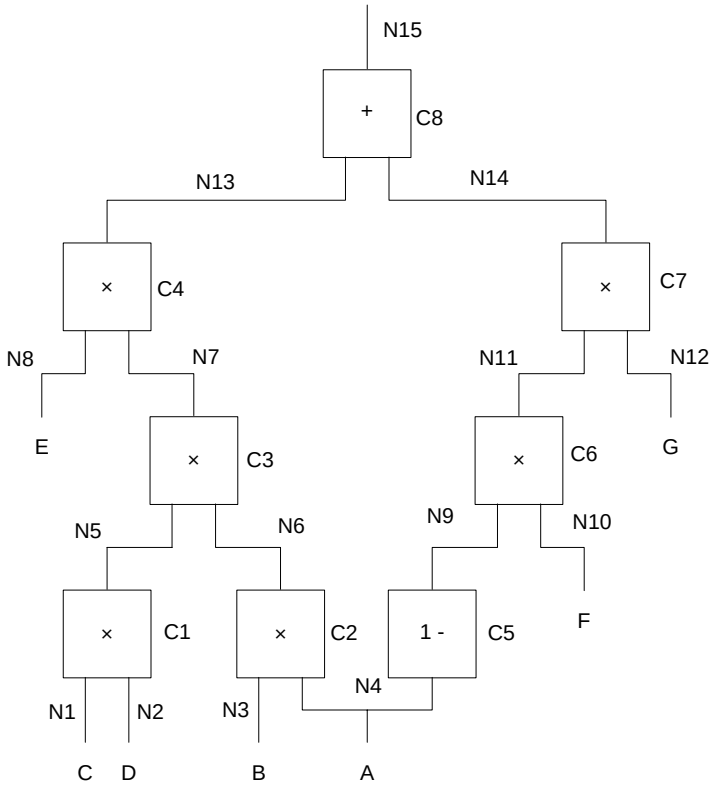

Figure 28: Basic Structure of CMC Circuit

the circuit. Previous attempts at generating tests for this circuit are described by Kunda et al. in [43]. Their work is done also at the functional level with propagation and justification done for all of the specific test values. Tests for the individual operators were done hierarchically and took a total 6,993 seconds. The functional level test generation for the individual operator tests took an additional 17,630 seconds. The fault efficiency that was achieved was 98.46%. When Kunda attempted to do gate level test generation (to compare with their results), the circuit was too large to be accommodated.

When DELPHI was given the structure of the CMC chip, test knowledge was extracted for each of the components and internal circuit nets. The knowledge that was extracted is listed in Tables 7 through 10. The total time required to extract this knowledge (on a Sun 4/280) was 0.8, 0.9, 0.7, and 1.35 seconds (respectively, for each type of knowledge). As can be seen in Table 10, the tests for components C1 through C7 are obtained symbolically and all that needs to be done is to replace the symbolic variables in the knowledge base by specific test values. On the other hand, the test knowledge for component C8 is not complete. This is due to the effect of the reconvergent paths from the A input. If the circuit is examined carefully, it can be seen that it is impossible to have both inputs to adder C8 with MSBs of 1 (the circuit is implemented using a finite precision fraction number representation). To alleviate this problem, DELPHI collects DFT knowledge during the test knowledge extraction. The DFT knowledge extracted for functional unit C8 is

Table 7: Type 1 Knowledge for Circuit in Figure 28

Net	Type 1 Knowledge
N5	{{C,1},{D,N}}
N6	{{A,1},{D,N}}
N7	{{A,1},{B,1},{C,N},{D,1}}
N9	{{A,1 – N}}
N11	{{A,0},{F,N}}
N13	{{A,1},{B,1},{C,1},{D,1},{E,N}}
N14	{{A,0},{F,1},{G,N}}
N15	{{A,1},{B,1},{C,1},{D,1},{E,N},{G,0}}

Table 8: Type 2 Knowledge for Circuit in Figure 28

Module	Type 2 Knowledge
C1	{{C,N_1},{D,N_2}}
C2	{{A,N_1},{D,N_2}}
C3	{{A,1},{B,N_1},{C,N_2},{D,1}}
C4	{{A,1},{B,1},{C,N_1},{D,1},{E,N_2}}
C5	{{A,1 – N}}
C6	{{A,1-N_1},{F,N_2}}
C7	{{A,0},{F,N_1},{G,N_2}}
C8	Unable to Generate

Table 9: Type 3 Knowledge for Circuit in Figure 28

Net	Type 3 Knowledge
N5	{{A,1},{B,1},{E,1}}
N6	{{B,1},{C,1},{E,1},{G,0}}
N7	{{E,1},{G,0}}
N9	{{E,0},{F,1},{G,1}}
N11	{{E,0},{G,1}}
N13	{{G,0}}
N14	{{E,0}}

Table 10: Type 4 Knowledge for Circuit in Figure 28

Module	Type 4 Knowledge
C1	{{A,1},{B,1},{C,N_1},{D,N_2},{E,1}}
C2	{{A,N_1},{B,N_2},{C,1},{D,1},{E,1},{G,0}}
C3	{{A,1},{B,N_1},{C,1},{D,N_2},{E,1}}
C4	{{A,1},{B,1},{C,N_1},{D,1},{E,N_2}}
C5	{{A,N},{E,0},{F,1},{G,1}}
C6	{{A,1 – N_1},{E,0},{F,N_2},{G,1}}
C7	{{A,0},{F,N_1},{G,N_2}}
C8	Unable to Generate

{{N13,C8},{{Scan,N_1},{A,0},{F,1},{G,N_2}}}

{{N7,C4},{{Scan,N_1},{E,1},{A,0},{F,1},{G,N_2}}}

{{N6,C3},{{Scan,N_1},{C,1},{D,1},{E,1},{A,0},{F,1},{G,N_2}}}

{{N4,C2},{{Scan,N_1},{B,1},{C,1},{D,1},{E,1},{A,0},{F,1},{G,N_2}}}

{{N14,C8},{{Scan,N_2},{A,1},{B,1},{C,1},{D,1},{E,N_1}}}

{{N11,C7},{{Scan,N_2},{G,1},{A,1},{B,1},{C,1},{D,1},{E,N_1}}}

{{N9,C6},{{Scan,N_2},{F,1},{G,1},{A,1},{B,1},{C,1},{D,1},{E,N_1}}}

{{N4,C5},{{Scan,1 – N_2},{F,1},{G,1},{A,1},{B,1},{C,1},{D,1},{E,N_1}}}

Using this information, a single scan latch can be inserted to allow 100% testability. In comparing the results using test knowledge extraction with the functional level test generation of Kunda and Abraham, it is easy to see that test knowledge extraction can be performed much more quickly than for the propagation and justification of individual test vectors. In addition to the 1.35 seconds required to extract the test knowledge, an additional 112 seconds were required

to replace the symbolic variables in the test knowledge base with the tests for the individual operators (using the pessimistic assumption that there was a single test for each fault). This creates a speedup factor of $\frac{17630}{1.35 + 112} = 155$ for the propagation and justification of test vectors. If the time required to generate each of the individual test vectors is included, the speedup factor becomes $\frac{17630 + 6993}{1.35 + 112 + 6993} = 3.46$. It is easy to see that time required to generate the individual test vectors is a dominant factor in the test generation time when test knowledge extraction is used. If precomputed tests were used, the cost of generating the individual test vectors could be born only during the operator design phase.

10.2 Sequential Circuits

Now we will turn our attention to sequential circuits. One of the many applications of the BSSC is in the area of digital signal processing. DELPHI was given the structure of an N'th order nonrecursive digital filter, for various values of N. The results for complete test knowledge extraction (i.e., symbolic tests are available for all operators in the circuit) are listed in Table 11. The times listed are for a SUN 4/280. Since each of the individual BSSC operators is 100% testable, the complete filter circuit is 100% testable using the extracted test knowledge.

Table 11: Test Extraction Results for Digital Filter

N	Number of Functional Operators	Test Extraction Time (Seconds)
5	11	1.33
10	26	2.51
15	41	4.01
20	56	5.18

Another sequential circuit that was analyzed by DELPHI was the solid-state circuit breaker shown in Figure 3. This 50,000 transistor was designed using the BSSC. Since we have already discussed the extraction of test knowledge from nonsequential circuits, we will concentrate on extracting symbolic tests (i.e., type 4 knowledge) for the sequential portion of the circuit. Figure 29 shows the portion of the circuit that will be described here.

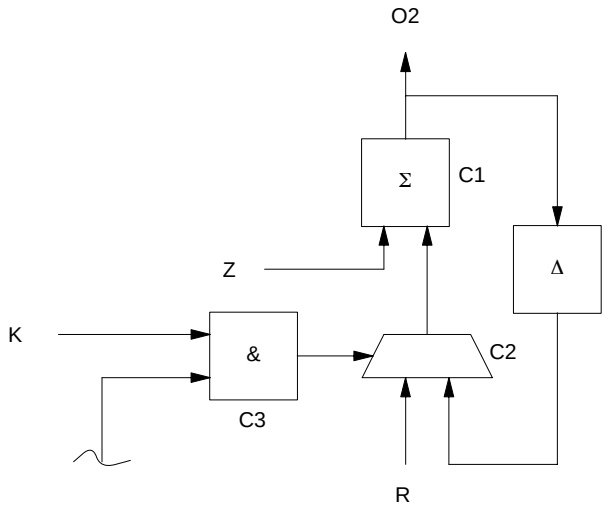

Figure 29: Sequential Section of Solid-State Circuit Breaker

The combinational portion of this circuit is handled in the same way as the circuit in the previous section. From this test knowledge, it is easy to see that any functional unit in the circuit can be tested using symbolic test values without the addition of any extra DFT circuitry. As in the previous section, each of the functional operators (adders, multipliers, comparators, multiplexors) are individually 100% testable. Thus, without any DFT modifications this circuit is 100% testable.

The symbolic tests for functional operators C1, C2, and C3 were extracted in 1.2 seconds on a Sun 4/280 and are listed in Table 12.

Table 12: Symbolic Tests for Circuit in Figure 29

Module	Time	Type 4 Knowledge
C1	T_0	{{0,K,0},{0,R,N_2},{0,Z,N_1}}
C2	T_{-1}	{{-1,K,0},{-1,R,0},{-1,Z,N_3}}
	T_0	{{0,K,N_1},{0,T,0},{0,S,1},{0,X3,1},{0,Y3,1},{0,R,N_2},{0,Z,0}}
C3	T_{-1}	{{-1,K,0},{-1,R,0},{-1,Z,N_1}}
	T_0	{{0,K,1},{0,T,0},{0,S,1},{0,X3,1},{0,Y3,1},{0,R,N_2},{0,Z,0}}

10.3 Knowledge Verification

To evaluate the performance of the test knowledge verification facility in DELPHI, the test knowledge previously extracted was verified. Since very little "bookkeeping" is necessary to verify test knowledge, verification was quicker than extraction in all cases. The relative speedup for verification versus extraction for various circuits is listed in Table 13 (all times are in seconds).

Table 13: Performance of Verification Versus Extraction

Circuit	Extraction Time	Verification Time	Speedup
CMC	1.35	1.12	20.5%
5-Stage Digital Filter	1.33	0.99	34.3%
10-Stage Digital Filter	2.51	1.93	30.0%
15-Stage Digital Filter	4.01	3.07	30.6%
20-Stage Digital filter	5.18	3.98	30.2%
Circuit Breaker	1.2	0.85	41.1%

As stated in Chapter 9, knowledge verification is most useful when there are datapaths that require complicated input patterns to reset the datapath. One

circuit that was evaluated required the difference between values for a weighted sum of an input to be greater than a threshold value. The extraction of test knowledge for the reset signal path was somewhat complicated and required several backtracks during the extraction process. By understanding this reset mechanism, a designer could provide DELPHI with a simple sequence of inputs to achieve the same goal. When verified by DELPHI, it was found that an 850% increase in speed was achieved relative to extraction.

11. CONCLUSIONS

11.1 Summary of Thesis Work

Various researchers have previously shown that high-level test knowledge can be used to greatly accelerate the test generation process. This thesis describes solutions to the problem of extracting this high-level knowledge from the structure of a compiled circuit. Since users of a silicon compiler are not design and test experts, it is necessary to automate the process of extracting test knowledge from a circuit.

Two different types of knowledge were considered. The first type of test knowledge is a testability measure. This thesis presents solutions to the problem estimating the testability for circuits defined at a functional level. By using an information theoretic testability measure, the concepts of controllability and observability are captured. Instead of requiring exhaustive enumeration of the input space to compute the measure (as was previously required), we have presented two different methods for efficiently and accurately estimating the measure. In addition, we have presented various applications of the measure, including automatic circuit partitioning and test point insertion.

The second type of knowledge is test generation knowledge. In this thesis we have described techniques to extract high-level test and DFT knowledge from

the structure of compiled circuits. Unlike previous approaches to knowledge-based test generation, the techniques described in this thesis can be used to automatically (and autonomously) extract the test knowledge from the structure of a circuit. This system has been implemented in a SUN workstation environment and is known as DELPHI. It operates on the high-level data flow representation of a compiled circuit and generates the test knowledge in the form of lists of primary input assignments. Achieving both high levels of fault coverage and fast performance, DELPHI can extract test knowledge from both nonsequential and sequential circuits. When test knowledge extraction is unsuccessful, additional DFT knowledge is obtained to efficiently represent design for testability options. These options can be evaluated in terms of a cost/benefit analysis (where the cost is additional area overhead and the benefit is reduced test generation complexity of increased fault coverage) to determine if they are appropriate. Finally, in cases where users are able to provide test knowledge, techniques to verify user provided knowledge were described. The verification of test knowledge has been shown to improve the test knowledge extraction time, and in some cases this improvement is quite drastic.

11.2 Some Proposals for Future Work

The merging of test generation and high-level synthesis is a fairly recent event. The work presented in this thesis is but a small step in that direction. There are numerous areas where additional work could be beneficial.

The work presented in this thesis does not attempt to manipulate the structure of a designer's circuit (with the exception of DFT modifications). An area of research with many possibilities to test generation and synthesis is in functional level structure transformation. Modifying the structure but keeping the functionality allow for further exploration of the design space. It would be expected that other points in the design space would have different testing requirements that may be used to improve fault coverage. As before, there would be a cost/benefit tradeoff that could be used to evaluate the effect of test coverage on circuit performance.

Another area that shows promise in the area of synthesis and testing is in fault modeling, which has traditionally been a complicated procedure with little detailed analysis being done for actual designs. Models such as bridging and stuck-open have been discussed theoretically but practical application is difficult in large designs. Since silicon compilers use standard cells to build designs, any in-depth analysis could be archived in the design database and used to improve fault models for entire circuits. Since the standard cells are designed only once but used many times, the cost of the fault analysis would have to be paid only once. The benefits, though, would be realized many times.

REFERENCES

[1] O. Ibarra and S. Sahni, “Polynomially complete fault detection problems,” *IEEE Transactions on Computers,* pp. 242-249, Mar. 1975.

[2] D. Brahme and J. Abraham, “Knowledge based test generation for VLSI circuits,” *Proceedings of the 1987 International Conference on Computer-Aided Design,* pp. 292-295, Nov. 1987.

[3] A. Parker and S. Hayati, “Automating the VLSI design process using expert systems and silicon compilation,” *IEEE Proceedings,* pp. 777-785, June 1987.

[4] A. Kessler and A. Ganesan, “Standard cell VLSI design: A tutorial,” *IEEE Circuits and Devices Magazine,* pp. 17-33, Jan. 1985.

[5] M. McFarland, A. Parker, and R. Camposano, “The high-level synthesis of digital systems,” *IEEE Proceedings,* pp. 301-318, Feb. 1990.

[6] J. Jasica, S. Noujaim, R. Hartley, and M. Hartman, “A bit-serial silicon compiler,” *Proceedings of the 1985 International Conference on Computer-Aided Design,* pp. 91-93, 1985.

[7] F. Yassa, J. Jasica, R. Hartley, and S. Noujaim, “A silicon compiler for digital signal processing: methodology, implementation, and applications,” *Proceedings of the IEEE,* pp. 1272-1282, Sept. 1987.

[8] R. Hartley and J. Jasica, “Behavioral to structural translation in a bit-serial silicon compiler,” *IEEE Transactions on Computer-Aided Design,* pp. 877-886, Aug. 1988.

[9] General Electric, *Bit-Serial Silicon Compiler User’s Manual*. Schenectady, NY: General Electric Corporate Research and Development Center, 1987.

[10] R. Hartley and P. Corbett, “A digit-serial silicon compiler,” *Proceedings of the 1988 Design Automation Conference,* pp. 646-649, June 1988.

[11] R. Hartley and P. Corbett, “A digit-serial compiler operator library,” *Proceedings of the 1989 International Symposium on Circuits and Systems,* pp. 641-646, June 1989.

[12] A. Casavant, M. d’Abreu, M. Dragomirecky, D. Duff, J. Jasica, K. Hwang, and W. Smith, “A synthesis environment for designing DSP systems,” *IEEE Design and Test of Computers,* pp. 35-43, April 1989.

[13] W. Smith, J. Jasica, M. Hartman, and M. d’Abreu, “Flexible module generation in an integrated design environment,” *Proceedings of the 1988 International Conference on Computer-Aided Design,* pp. 396-399, Nov. 1988.

[14] W. Smith, D. Duff, M. Dragomirecky, J. Caldwell, M. Hartman, J. Jasica, and M. d'Abreu, "FACE core environment: the model and its application in CAE/CAD tool development," *Proceedings of the 1989 Design Automation Conference,* pp. 466-471, June 1989.

[15] W. Smith and D. Duff, "SMALL: A system design language," GE Corporate Research and Development Center Internal Report, May 1989.

[16] M. Dragomirecky, E. Glinert, J. Jasica, D. Duff, W. Smith, and M. d'Abreu, "High-level graphical user interface in the FACE synthesis environment," *Proceedings of the 1989 Design Automation Conference,* pp. 331-336, June 1989.

[17] C. Chin, T. Chang, M. Hartman, C. Ho, J. Jasica, W. Smith, G. Buchner, and D. Orton, "10MHz IC's for graphics processing designed on a silicon compiler," *Proceedings of the 1988 International Solid-State Circuits Conference,* pp. 347-350, Feb. 1988.

[18] J. Abraham and W. Fuchs, "Fault and error models for VLSI," *IEEE Proceedings,* pp. 639-654, May 1986.

[19] J .Guzolek, W. Rogers, and J. Abraham, "WRAP: An algorithm for hierarchical compression of fault simulation primitives," *Proceedings of the 1986 International Conference on Computer-Aided Design,* pp. 338-341, Nov. 1986.

[20] A. Friedman, "Easily testable iterative systems," *IEEE Transactions on Computers,* pp. 1061-1064, Dec. 1973.

[21] M. Breuer and A. Friedman, "Functional level primitives in test generation," *IEEE Transactions on Computers,* pp. 223-235, March 1980.

[22] T. Williams and K. Parker, "Design for testability - A survey," *Proceedings of the IEEE,* pp. 98-112, Jan. 1983.

[23] E. McCluskey, "A survey of design for testability scan techniques," *VLSI Design,* pp. 38-61, Dec. 1984.

[24] T. Williams, "VLSI testing," *IEEE Computer,* pp. 126-136, Oct. 1984.

[25] L. Goldstein, "Controllability/observability analysis of digital circuits," *IEEE Transactions on Circuits and Systems,* pp. 685-693, Sept. 1979.

[26] L. Goldstein and E. Thigpen, "SCOAP: Sandia controllability/observability analysis program," *Proceedings of the 1980 Design Automation Conference,* pp. 190-196, June 1980.

[27] R. Bennetts, *Design of Testable Logic Circuits*. Reading, MA: Addison-Wesley, 1984.

[28] S. Tomas and J. Shen, "A survey of functional level testing and testability measures," Carnegie Mellon University Center for Computer-Aided

Design, Technical Report CMUCAD-83-18, Pittsburgh, PA, Nov. 1983

[29] J. Fong, “A generalized testability analysis algorithm for digital logic circuits,” *Proceedings of the 1982 International Symposium on Circuits and Systems,* pp. 1160-1163, June 1982.

[30] J. Fong, “On functional controllability and observability analysis,” *Proceedings of the 1982 International Test Conference,* pp. 170-175, Sept. 1982.

[31] S. Takasaki, M. Kawai, S. Funatsu, and A. Yamada, “A calculus of testability measure at the functional level,” *Proceedings of the 1981 International Test Conference,* pp. 95-101, Sept. 1981.

[32] J. Stephenson and J. Grason, “A testability measure for register transfer level digital circuits,” *Proceedings of the 1976 Fault-Tolerant Computing Symposium,* pp. 101-107, June 1976.

[33] V. Agrawal, “An information theoretic approach to digital fault testing,” *IEEE Transactions on Computers,* pp. 582-587, Aug. 1981.

[34] J. Dussault, “A testability measure,” *Proceedings of the 1978 Semiconductor Test Conference,* pp. 113-116, Oct. 1978.

[35] H. Fung, and J. Fong, “An information flow approach to functional testability measures,” *Proceedings of the 1982 International Conference on Circuits and Computers,* pp. 460-463, June 1982.

[36] H. Fung, S. Hirschhorn, and R. Kulkarni, “Design for testability in a silicon compilation environment,” *Proceedings of the 22nd Design Automation Conference,* pp. 190-196, June 1985.

[37] H. Fung, “A testable-by-construction strategy for the SILC silicon compiler,” *Proceedings of the 1985 International Conference on Computer Design,* pp. 554-557, Oct. 1987

[38] H. Fung and S. Hirschhorn, “An automatic DFT system for the SILC silicon compiler,” *IEEE Design and Test,* pp. 45-57, Feb. 1986.

[39] C. Gebotys and M. Elmasry, “VLSI design synthesis with testability,” *Proceedings of the 1988 Design Automation Conference,* pp. 16-21, June 1988.

[40] C. Gebotys and M. Elmasry, “Integrated design and test synthesis,” *Proceedings of the 1988 International Conference on Computer Design,* pp. 398-401, Oct. 1988.

[41] C. Gebotys and M. Elmasry, “Integration of algorithmic VLSI synthesis with testability incorporation,” *IEEE Journal of Solid-State Circuits,* pp. 409-416, April 1989.

[42] M. Abadir and M. Breuer, “A knowledge-based system for designing testable VLSI chips,” *IEEE Design and Test,* pp. 56-68, Aug. 1985.

[43] X. Zhu and M. Breuer, “Analysis of testable PLA designs,” *IEEE Design and Test,* pp. 14-28, Aug. 1988.

[44] A. Krasniewski, “Automatic designing of self-testing VLSI circuits,” University of Rochester, Department of Electrical Engineering Technical Report, July 1985.

[45] A. Krasniewski and A. Albicki, “Automatic design of exhaustive self-testing chips with BILBO modules,” *Proceedings of the 1985 International Test Conference,* pp. 362-371, Sept. 1985.

[46] J. Kalinowski and A. Albicki, “Computer-aided design of self-testable VLSI circuits,” *Proceedings of the 1988 CompEuro Conference,* pp. 324-328, April 1988.

[47] R. Kunda, P. Narain, J. Abraham, and B. Rathi, “Speed up of test generation using high-level primitives,” *Proceedings of the 1990 Design Automation Conference,* pp. 111-117, June 1990.

[48] B. Murray and J. Hayes, “Hierarchical test generation using precomputed tests for modules,” *Proceedings of the 1988 International Test Conference,* pp. 221-229, Aug. 1988.

[49] B. Robinson, “Artificial intelligence and testing,” *Proceedings of the 1984 International Test Conference,* pp. 200-203, Aug. 1984.

[50] C. Matthaus, B. Kruger-Sprengel, and H. Vierhaus, “EXTEST - A knowledge based system for the design of testable logic circuits,” *Proceedings of CompEuro 1989,* pp. 137-139, May 1989.

[51] C. Robach, D. Lutoff, and N. Garcia, “Knowledge-based functional specification of test and maintenance programs,” *IEEE Transactions on Computer-Aided Design,* pp. 1145-1156, Nov. 1989.

[52] B. Krishnamurthy, “Hierarchical test generation: can AI help?” *Proceedings of the 1987 International Test Conference,* pp. 694-700.

[53] N. Singh, *An Artificial Intelligence Approach to Test Generation*. Boston: Kluwer, 1987.

[54] M. Shirley “Generating circuit tests by exploiting designed behavior,” MIT Artificial Intelligence Laboratory Technical Report 1099, Dec. 1988.

[55] P. Anirudhan and P. Menon, “Symbolic test generation for hierarchically modeled digital systems,” *Proceedings of the 1989 International Test Conference,* pp. 461-469, Sept. 1989.

[56] K. Roy and J. Abraham, “High level test generation using data flow descriptions,” *Proceedings of the 1990 European Design Automation Conference,* pp. 313-317, March 1990.

[57] P. Wu, “Test generation guided design for testability,” MIT Artificial Intelligence Laboratory, Technical Report 1051, May 1988.

[58] R. Marlett, “Evolution of an effective DFT/ATG solution for sequential ASICs,” *Proceedings of the 1989 International Symposium on Circuits and Systems,* pp. 1950-1953, June 1989.

[59] R. McEliece, *The Theory of Information and Coding*. Cambridge, England: Cambridge University Press, 1977.

[60] G. Koob, “An abstract complexity theory for boolean functions,” Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1987.

[61] A. Barron, Personal Communication, Urbana, IL, Nov. 1988

[62] S. Akers, “Partitioning for testability,” *Design Automation and Fault-Tolerant Computing,* pp. 133-146, Feb. 1977.

[63] E. McCluskey and S. Bozorgui-Nesbat, “Design for autonomous test,” *IEEE Transactions on Computers,* pp. 866-874, Nov. 1981.

[64] T. Payne, “Automated partitioning of hierarchically specified digital systems,” Stanford University, Departments of Electrical Engineering and Computer Science, Technical Report No. 215, July 1981.

[65] J. Udell and E. McCluskey, “Efficient circuit segmentation for pseudo-exhaustive test,” *Proceedings of the 1987 International Conference on Computer-Aided Design,* pp. 148-151, Nov. 1987.

[66] E. Trischler, “Incomplete scan path with an automatic test generation methodology,” *Proceedings of the 1980 International Test Conference,* pp. 153-162, Aug. 1980.

[67] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*. Reading, MA: Addison-Wesley, 1985.

[68] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*. Reading, MA: Addison-Wesley, 1988.

VITA

Kurt Henry Thearling was born in Wyandotte, Michigan, in 1963. He received dual B.S. degrees in Electrical and Computer Engineering from the University of Michigan in 1985. From 1985 through 1990 he was a student at the University of Illinois, receiving his M.S. degree in Electrical Engineering in January 1988. At the University of Illinois, he was employed as a research assistant in the Coordinated Science Laboratory, working in the area of reliable and fault-tolerant computing. He was employed by various companies in industry during each of the nine summers from 1982 through 1989, including IBM, General Electric, and Bechtel.

In 1987, he was a member of the first-place team in the Project 2000 contest sponsored by Apple Computer. This contest challenged teams of students from twelve national universities to design the personal computer of the year 2000. The winning entry, TABLET, has received international attention in such publications as *Business Week*, *USA Today*, *MacWorld*, *EOS* (Holland), *SuperInteressante Especial* (Brazil), and the book *Future Stuff* edited by M. Abrams and H. Bernstein (Viking Press, 1989).

He is currently employed by Thinking Machines Corporation in Cambridge, MA, working in the area of complex systems, adaptive learning, and nonlinear dynamics. In addition, he is a founding member of Marc and Kurt's Excellent

Science Institute (MK/ESI), an interdisciplinary research organization.