Behavioral-Level Considerations for Design for Testability and Test Generation

APPROVED:

Jacob A. Abraham

Stephen A. Szygenda

To all the Luddites out there. You know who you are.

Behavioral-Level Considerations for Design for Testability and Test Generation

by

SCOTT LEE CRUTCHFIELD, B.S.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1995

Acknowledgments

It is of some wonder to me that work on this thesis is coming to an end. As I look back on the past two years, it all seems a blur. There were never-ending climbs up learning curves; a whole slew of classes; and countless hours of "tube time" programming, writing, running experiments, and plenty of staring into (cyber)space. The amazing thing is that there will be something to show for it all. It is a prize that I cannot by any right take credit for alone.

Many thanks to Professor Joydeep Ghosh, who took me under his wing and gave me opportunities to seek my true interests.

I am deeply indebted to my supervisor, Professor Jacob Abraham, for his guidance and encouragement. He provided me with much of the insight for this work, and also gave me the freedom to pursue it in my own way.

Professor Stephen Szygenda told me how to finish my degree efficiently, advice which I will always remember but which I followed not at all. I am most indebted to him for the insight he has shared in classes, especially regarding that American game called business.

Thanks also to Dan Saab for his help in using the CRIS test generator.

I must also share this accomplishment — all but the mistakes, which I selfishly claim for my very own — with my peers. Dinos and Yatin took great pains to explain to me about the formal methods. Ashok shared insight on test generation and on writing in general. Without Rathish's help, there would have been no Compass synthesis tool and consequently no results! There are many others – were it not for the fellowship of Alok, Aman, Ganapathy, Ghani, Helena, Jin, Jing, Krishnan, Kurt, Linda, Mallu, Nandan, Naveena, Prakash, Rick, Rick, Rob, Sandy, Shirley, Shyla, Tony, Tushar, and all the rest I doubt I would have survived so long.

My father, an avowed modern Luddite, has shown more than a passing interest in my studies, especially with the use of psychological terms to discuss electronic circuits (i.e., behavior modification). I told him it was a coincidence, but now I'm not so sure it is. Mom has always been more loving and supportive than words can say. Thanks to both of you for everything.

And finally to my beautiful Alexandra, I can only say:

No woman but my darling wife
E'er withstood such arduous strife
Now she's got what she needs:
A husband indeed,
Whose work is no longer his life!

Well, at least for a month or two. Look out, everyone! It looks like I'll be back!

Scott Lee Crutchfield

The University of Texas at Austin
May, 1995

Abstract

Behavioral considerations for testability are an important part of the VLSI design cycle. Large designs that cannot be considered completely at the gate level for testability issues often can be handled at the behavioral level. The techniques presented can be used for designing large VLSI circuits that need to be easy to test. Nondeterministic test generation is used because of its speed improvement over deterministic test generation. Two techniques are presented that assist with nondeterministic test generation: formally-generated seed populations and instruction templates for encoding behavioral information to be used by the test generator. Next, a design methodology is presented wherein the behavioral description, rather than the gate-level description, of a circuit is modified to improve the testability of the design. This involves evaluation of the testability from the behavioral description and a mechanism for suggesting changes to improve testability. Modifications are made to the circuit by hand or with the assistance of an interactive software tool. This technique is shown to be useful for increasing the testability of an example microprocessor design. The addition of test instructions to perform useful operations in a processor-like design is discussed. A program-level self test technique using these instructions is presented. Finally, combinations of and extensions to these techniques are also pointed out.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The design, validation and verification of VLSI circuits are all increasing in difficulty as the size of these circuits increases. When these circuits are manufactured, it is important to have simple methods of testing them to ensure that they are performing properly. There are two sides to the issue of testability: test generation and design for testability. This thesis considers both areas with respect to microprocessor-like circuits. A variety of techniques for enhancing testability are presented.

Automatic test pattern generation (ATPG) is the process of determining a sequence of input vectors for a circuit that will allow the detection of faults that might be present. Chapter 2 presents relevant previous work and provides the foundation for the subsequent discussion.

Standard ATPG programs such as HITEC [10] consider only low-level circuit information. There are advantages to providing behavioral information to a test generator to assist in searching for tests. Chapter 3 investigates using descriptions of useful behavior as the seed for CRIS, a nondeterministic test generator. Results of using this technique on several circuits are presented. Chapter 4 discusses performing test generation in terms of instructions or groups of instructions supported by the design. This technique was used on an example circuit.

Assisting test generation with behavioral information is just one aspect of the testability issue. The other aspect is using a design process that yields a highly testable circuit. Such design for testability (DFT) techniques have been used extensively at the gate level. Unfortunately, large circuits cannot be handled effectively at the gate level for DFT. Performing DFT at the behavioral level can make the process useful for very large designs. Chapter 5 discusses some extensions to Behavioral Modifications for Testability, a technique previously introduced [16]. A simple design methodology has been developed for the selection and application of appropriate behavioral modifications. Two techniques for analyzing the testability of a design are presented. These are used to make suggestions for behavioral modifications. The process of adding testability features to a design has been automated to a certain extent. Several experiments using these techniques were performed on the Viper microprocessor.

One extension to behavioral modification is to add test instructions to a design according to the testability requirements. Chapter 6 discusses adding these instructions and using them to build a program that performs built-in self test. Possible combinations of these techniques and other extensions to this work are briefly presented in Chapter 7. Finally, conclusions are drawn in Chapter 8.

Chapter 2

Background

2.1 Levels of Circuit Description

Digital circuits can be described on a number of levels. It is important to have these different levels of description for various stages of design. The following is a list of some important levels of description [15]:

Behavioral Level — A semantic description of high-level operations.

Functional Level — A description of the design in terms of boolean logic equations.

Register Transfer Level (RTL) — A description of behavior in terms of data transferred between registers.

Gate Level — A description of the structural interconnection of logic gates.

Transistor Level — A description of the structural interconnection of transistors and other elements (also called *circuit level*).

The most abstract level listed here is the behavioral level, and the most detailed level listed is the transistor level. At the behavioral level, a design is described in terms of the operations that it performs on its inputs. Ideally there is no structural information included in the behavioral description. In practice, however, it is impossible to eliminate all structural information from a behavioral description. Circuits described as behavioral in this thesis will in general be composed of behaviorally-described subcircuits that are interconnected structurally. For example, the Viper microprocessor in Figure 5.3 consists of three behavioral modules (ALU, Registers, and Control). High level circuit description languages (HDLs) such as VHDL and Verilog can be used to describe circuits at the behavioral level. They can also be used to describe circuits structurally, and these levels of description can be mixed [11, 14].

At the transistor level, a design is described in terms of the interconnections of transistors and other devices. This level of description is important for the analysis of timing and electrical characteristics, but it too detailed for the purposes of large circuit design and test generation that are the present concerns.

An important intermediate level, between the behavioral and transistor levels, is the gate-level description. Designs are described at the gate level as interconnections of standard gate primitives (which themselves can be thought of as modules containing transistors). Gates such as AND, OR, XOR, and their complements are usually used in the gate level description. The ISCAS89 benchmark circuits are described at the gate level in what is called the *bench* format. Gate-level descriptions are also called *netlists*. It is at this level that most of the work in testability has been done.

The other levels listed are not given direct consideration in this thesis. It is possible to design a large circuit using a behavioral description and then convert it directly to the gate level without working at any intermediate levels, by a process called synthesis [2].

2.2 Behavioral Synthesis

If a design is described at the behavioral level, it must ultimately be converted to a circuit-level description so it can be manufactured. A convenient way to do this is to convert the design first to the gate level, which itself has close correspondence to the transistor level. The automated process of converting from behavioral to gate-level is called synthesis.

The synthesis tool used in this work was AsicSyn, which is part of the Compass Design Automation package. It takes as input a VHDL description and synthesizes it to a proprietary Compass netlist format (the *nls* format) [2]. In order to use this with the test generation programs, it was necessary to convert from the *nls* format to the *bench* format. A program called *nls2bench* was written to perform this conversion.

2.3 Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) is the process of generating inputs for a design that will allow detection of faults in a particular circuit. A gate-level description of the design is usually used for this process. The basic process of ATPG, or test generation, is as follows:

1. Inject a fault into the circuit
2. Attempt to find an input vector or sequence of vectors that will
    (a) activate the fault, and
    (b) propagate its effect to the output
3. Repeat for the rest of the faults.

The end result of ATPG is a sequence of test vectors, called the test set, that can be applied to the circuit after manufacture to detect faults that might be present. Faults are detected by comparing the output of the circuit under test to the output of a known-good circuit (which can be obtained by simulating a good circuit); if the outputs ever differ then the circuit under test is known to be faulty. In general, it is not known which faults are present since each input vector might detect multiple faults.

The fault model usually used is the *single stuck-at* fault model, in which it is assumed that only one signal in the design is faulty at a time and that this fault is causing the signal to have a constant value, either 1 or 0. There are many other fault models, such as multiple stuck-at faults, functional transformation faults, intermittent faults and delay faults; most of these have the advantage that they are more realistic and the disadvantage that they are impractical to use because of the large number of fault combinations [1].

There have been many algorithms developed for test generation. The first successful algorithm for combinational circuits was the D Algorithm, which was improved upon in several other algorithms and also extended to sequential test generation [1]. The D Algorithm and its derivatives are all deterministic, in that for any given design and enough computation resources they will always find a test vector or sequence for each fault or determine the fault is untestable or redundant. Due to time constraints, test generation programs will give up on some of the faults because they are difficult to detect. These algorithms run for long periods of time; in the worst case they require exponential running time in the number of gates [1]. ATPG is especially difficult for sequential circuits, which require consideration of many different time frames of the circuit's operation.

One alternative to these deterministic test generation algorithms is to use nondeterministic methods of searching the input space such as genetic algorithms (GA) [8]. CRIS is a nondeterministic sequential test generator developed by Saab et. al. [13]. It uses a GA-based technique to search the input space of a given netlist to determine good input patterns to detect faults in the circuit. Input vectors that are successful at detecting faults are favored for slight alteration (mutation) in an attempt to detect other faults. CRIS is much faster than deterministic methods, but does not guarantee that it will eventually find tests for or prove redundant all faults in the circuit. It does provide certain features that can use behavioral information to improve performance; these are discussed in Chapters 3 and 4.

The metrics for comparing the results of test generation are as follows:

Fault Coverage The goal is to achieve a test set that covers 100% of the possible faults in a design.

Efficiency Efficiency refers to the fraction of all faults that are either detected by the test set or shown to be redundant.

Test Set Size Short test sets are preferred because they can be applied to the circuit quickly.

Test Generation Time For large designs, deterministic test generation can take weeks or months of CPU time. If ATPG is to be performed several times, a faster test generator is desirable.

Table 2.1: Comparison of Deterministic (HITEC) and nondeterministic (CRIS) ATPG

	Time (seconds)		Fault Coverage	
Circuit	CRIS	HITEC	CRIS	HITEC
s208	22.8	24	60.9	63.7
s526	48.4	168651	77.1	51.7
s832	105.1	6329	42.5	93.9
s838	216.8	168812	28.2	29.6
s1238	131.1	132	90.7	94.6
s1494	171.2	6874	90.1	96.4

An early version of a widely used deterministic sequential ATPG tool, HITEC, required up to 47 hours for some of the ISCAS89 circuits (s526, s838), and circuits requiring more than ten hours were not uncommon (See Table 2.1). If circuits with only a few hundred gates can require several days of CPU time for ATPG, it is easy to see that large ASIC designs cannot currently be handled all at once. Table 2.1 compares the fault coverage and CPU time for CRIS and HITEC. These results were reported in [10] and [13]. Times for these were from a Sun SLC.

The emphasis in comparison of test generation results in this thesis is on the fault coverage. HITEC usually reports an efficiency much higher than fault coverage because it seems to find many redundant faults in the circuit. The version of CRIS used for results in this thesis does not usually find many redundant faults, so the efficiency and fault coverage reported are usually very close.

2.4 Gate-Level Design for Testability

Test generators have trouble when signals within the circuit are not easy to set to a desired value from the primary inputs and also when values of internal signals are not easy to determine from the primary outputs. There are many techniques for designing circuits to assist test generation and to achieve high testability. Most of them concentrate on improving the controllability and observability of signals internal to the design [1]. The accessibility of a signal refers to these two qualities taken together.

There are two important variations of gate-level DFT: *ad hoc* and structured. *Ad hoc* changes are made to the design as the designer sees fit in order to improve testability. Important *ad hoc* test features include test points, initialization, and circuit partitioning [1]. Test points are signals in the circuit that are made specifically accessible at the primary inputs and/or outputs of the circuit. Initialization features ensure that the circuit can be easily reset to a known state, which is important for test generation. Circuit partitioning allows subcircuits to be considered independently during test generation. When the designer adds these features to the circuit he or she makes implicit use of knowledge of the circuit's behavioral operation.

Structured DFT methods are used to add testability features without requiring the designer to make all changes manually. Most of these techniques use scan-based techniques. This converts a sequential design into a combinational design, for which ATPG is substantially easier. Most techniques using scan consider only the gate-level description of the design [1].

2.5 Background Summary

This chapter has presented background information for the rest of this thesis. Circuits can be represented at various levels. A behavioral description indicates the high-level semantics of the operation of a design, and a gate-level description is concerned with the interconnection of logic primitives to realize this operation. Behavioral descriptions can be converted to gate-level descriptions by the process of synthesis. The gate level is useful for test generation, and there are various algorithms for this. Design for testability issues have usually been left for the gate level, without considering behavioral information that is often available.

Chapter 3

The Effect of Initial Sequences on Genetic-Algorithm Based Test Generation

3.1 Introduction

CRIS uses a technique based on genetic algorithms to generate tests, and then evaluates their fault coverage and efficiency by fault simulation. The genetic algorithm requires an initial sequence of seed vectors. CRIS can generate its own seeds if none are provided. This chapter investigates the effect of varying the initial sequence given to CRIS on its test generation performance. The goal is to assist the ATPG procedure by providing meaningful test vectors with which to begin its search.

The genetic algorithm is used specifically to search the input space of the circuit under test (CUT). First, fault simulation is performed to identify the faults detected by the initial sequence. Several other factors are also determined at this time, including the node activity and the locations in the circuit where the propagation of fault effects was blocked. This information is used to identify good input vectors and sub-sequences much more quickly and effectively than could be done using fault simulation alone. The candidates are transformed into even better potential tests using standard genetic techniques. CRIS duplicates them and then performs either mutation or splicing on them. Mutation changes some of the bits in the new copy, and splicing combines two separate vectors or sequences to produce a new one. CRIS then evaluates the performance of the new sequence, and repeats the process many times to build the final test sequence [13].

The search that CRIS performs thus depends heavily on the initial sequence. It has been suggested that the functional test vectors for a circuit are good candidates for the seed. CRIS can often do well with a random initial sequence, but a sequence of input vectors that exercises the circuit in a meaningful way might do even better. If it provides a good exercise of the functional characteristics of the circuit, then CRIS can find sequences that add additional fault coverage to that.

In the absence of functional test sequences, the initial sequences for CRIS need to be determined automatically. Since an important part of ATPG is moving a sequential circuit from state to state [1], this should also be the goal of the initial test sequences. The sequences themselves are not intended to achieve high fault coverage, although this would be helpful. An important consideration is the time to generate the sequences; it needs to be small compared to the ATPG time or the overhead will be too high.

3.2 Using SCFV for state-space traversal

SCFV is a tool developed by Dinos Moundanos and Yatin Hoskote intended for comparing two different circuits and generating a distinguishing input sequence [9]. It can also be used as needed here to generate an input sequence to take the circuit from a given initial state to a given target state. If several target states are specified, it will find the sequences from the initial state to all of the target states. A version of the tool planned for the near future will be able to generate sequences to arrive at all of the most-distant states (i.e. those states requiring the most state transitions from the initial state). As described below, this capability may prove to be the most useful for the purposes of generating initial sequences for CRIS. SCFV uses techniques based on implicit state enumeration and reachability analysis to traverse the state space of the circuit [9, 6].

Since large finite-state-machines (FSMs) have many states, it is necessary to limit consideration to certain important states. The circuit is assumed to have a reset state, an initial state from which the test generation will begin. This is a requirement of the SCFV tool. Currently the target states must also be specified. These target states can be determined by hand or generated randomly. Once SCFV is able to generate sequences for the most distant states, the software framework is in place to evaluate the use of these as initial sequences.

A consequence of the current requirement that the target states be listed explicitly is that circuits with a large number of memory elements cannot be considered. The number of possible states is too large for random targeting to be effective, and generating the sequences would take too long. Important states can be identified manually only if the behavior of the circuit is evident, which is not always the case for the ISCAS benchmarks, for example. Target states can be generated randomly but many of these will be unreachable or forbidden. For very small circuits all the states can be listed as targets.

3.3 Experimental Procedure

The circuit under test is specified in the ISCAS *bench* format. SCFV is used to generate the initial sequences. These include *don't-care* bits whose values do not affect the transitions specified by the input sequence. These raw sequences are then modified in various ways (described below) to arrive at several different types of input sequences. For reference, sequences of all zeros and all ones and sequences of random bits are generated as well. The fault coverage of each initial sequence by itself is determined by fault simulation using CRIS. CRIS is run for each initial sequence generated. Finally the results are collected as presented in Section 3.5. The list below summarizes the procedure.

1. Prepare bench-format circuit.
2. Use SCFV to generate raw initial sequences.
3. Change the raw sequences appropriately (e.g. remove don't cares).
4. Generate constant-0, constant-1, and random sequences for comparison.
5. Fault simulate to determine the fault coverage of the sequences themselves.
6. Run CRIS for test generation.
7. Compare the results in terms of fault coverage, efficiency, and CPU time.

Since the input sequences were all generated with respect to a single reset state, a reset sequence is prepended to all of the raw sequences if one is available. This ensures that the sequences are used in the proper context, at least in the initial sequence for CRIS. For some circuits a reset is not available so nothing is added to the raw sequences. Since these are only used as a starting point, it is assumed that the genetic algorithm can make use of them even if they are taken out of context. Experience has shown that the performance will usually be better if a reset is available.

Once the reset sequence is added to each raw sequence for each target state, the raw sequences are concatenated. CRIS expects a single seed sequence. It is able to extract these sub-sequences if it determines that they are effective at detecting faults.

There are four different conversions made on the raw initial sequences to create the seeds for CRIS. The first three are trivial replacements of the *don't-care* bits by zeros, ones, and random bits, respectively. The fourth transformation removes the final vector of each sequence. This shortens the initial sequences but the resulting vectors still give CRIS a start in moving from one state to another; again presumably it will be able to reach the original target state using a simple mutation or combination of the previous sequence. The *don't-care* bits are still replaced by random bits in these shortened sequences. This makes the most sense when the sequences move to the most distant states from the reset state. In that case there are many states along the paths to the most-distant state, so the sequences will exercise a good subset of the state space.

It is important to note that the initial sequences are not intended to be a method of testing by themselves. They do not need to guarantee that all states are visited. In practice it turns out that there are usually many *don't-care* inputs for each vector, so there are many possible sequences that can achieve the same state-traversal results and it is likely that a genetic algorithm can find similar transitions.

3.4 Software Framework

The software developed consists of scripts to run the tools and to perform appropriate transformations of the raw sequences. The important scripts are briefly described here. Only two files are required to begin an experiment: the *bench* circuit file and a parameter file describing the inputs to the circuit and its state variables. The parameter file also specifies the initial state of the circuit and the reset sequence. The process of the experiment is automated by shell scripts and Perl scripts.

A shell script (*make-sequences*) creates the appropriate input files for SCFV and runs it to generate the raw sequences. It calls a Perl script, *seq-extract*, to extract the sequences from the output of SCFV. It then calls several Perl scripts to prepend the reset sequence and then to make each kind of transformed initial sequence. Constant and random initial sequences are generated as well.

Another script (*runcris*) is used to run CRIS. It first runs CRIS with no initial sequence. Then for each initial sequence it does fault simulation using the sequence as the input to determine the fault coverage of the initial sequence. It then uses CRIS for test generation using the sequence as the seed.

3.5 Results

The experiment was repeated for six different circuits. Four of them are sequential circuits from the ISCAS89 benchmark suite (s208, s832, s1238, s1494). The last two are the control module of the Viper microprocessor and a complete 2-bit version of the Viper itself [7]. Table 3.1 shows the parameters of the circuits. The columns have the total number of gates, the number of D-flip

Table 3.1: Comparison of circuits used in experiments

circuit	gates	DFFs	PIs	targets	initial state
s208_a	104	8	11	11	11100101
s208_b	104	8	11	–	00000000
s832	292	5	18	9	00000
s1238	526	18	14	6	0...0
s1494	653	6	8	9	000000
vcontrol	101	7	13	4	0000000
viper_2bit	460	33	13	3	0...0

flops, the number of primary inputs, the number of target states for which a sequence was successfully generated and the initial state used by SCFV.

In the following sections for each circuit, results are presented for each different type of initial sequence for the test generation. The names for the initial sequence types are as follows:

- No seed: CRIS results without a seed sequence (its own random sequence)
- C-0, C-1, Random: constant-0, constant-1, random sequence
- 0, 1, r: *don't-cares* replaced by 0, 1, or random bits, respectively
- short: the final vector for each target state was removed.

The tables showing the results list the length of each sequence, the "inherent fault coverage" of the sequence (fault coverage of the sequence with no test generation), fault coverage after test generation, efficiency of the generated test, the number of vectors in the test, the number of redundancies detected and the total CPU time for test generation. The time to generate the initial sequences for each circuit was negligible compared to the time for test generation.

Table 3.2: Results for s208

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	60.93	71.16	250	0	337.80
C-0	-	0.93	60.93	63.26	452	5	174.57
C-1	-	3.26	56.28	57.21	303	2	136.24
Random	-	27.44	53.95	54.88	390	2	83.98
0	-	3.72	60.47	61.40	481	2	127.19
1	-	3.72	60.00	73.95	458	30	240.39
r	-	46.05	60.47	70.23	29	21	150.39
short	-	0.93	55.81	55.81	504	0	35.39

(Sequence lengths are not available.)

3.5.1 Results for ISCAS Circuits

Table 3.2 shows the results for the s208 circuit from the ISCAS benchmark suite. Notice that the CPU time starting with no seed is about twice as long as the test generation time for all the other cases. The fault coverage obtained without an initial sequence is about the same as for all other cases, but it detected 22 redundant faults, more than most of the other sequences. The best performance is seen in the "1" case (*don't-cares* were replaced by ones). Fault coverage is about as good as the no seed case, but even more redundancies are detected (30 compared to 22).

Table 3.3 shows the results for the s832 circuit. In this case the no-seed case has the best fault coverage at 53.8%, and all rest are about the same at around 49%. The case with the random initial sequence took the least amount of CPU time and had the best inherent fault coverage – a result that does not support the use of formally-generated initial sequences.

Tables 3.4 and 3.5 show the results for the s1238 and s1494 circuits, respectively. For both of these the best coverage is shown for the no-seed case.

Table 3.3: Results for s832

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	53.79	53.91	394	1	1579.80
C-0	46	1.03	49.31	49.31	561	0	1767.99
C-1	46	6.32	48.16	48.16	586	0	1703.09
Random	46	25.29	49.08	49.08	329	0	1295.59
0	46	1.15	49.20	49.20	516	0	1721.24
1	46	0.92	48.97	48.97	515	0	1727.35
r	46	1.38	48.85	48.85	509	0	1686.70
short	37	0.69	49.43	49.43	606	0	1717.95

Table 3.4: Results for s1238

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	90.70	92.47	1784	24	84.14
C-0	11	5.98	87.90	89.15	1685	17	89.32
C-1	11	2.66	90.48	92.32	1744	25	82.45
Random	11	22.73	89.08	90.18	1726	15	77.26
0	11	29.59	89.59	91.44	1661	25	70.59
1	11	21.62	89.08	90.55	1757	20	76.74
r	11	27.60	89.37	91.73	1695	32	73.95
short	5	6.20	89.82	91.44	1751	22	81.31

Table 3.5: Results for s1494

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	88.98	89.11	677	2	1199.11
C-0	126	9.03	78.75	78.75	1447	0	2191.08
C-1	126	0.07	86.79	86.85	781	1	1354.21
Random	126	48.94	84.00	84.00	580	0	508.36
0	126	0.46	84.13	84.20	889	1	1055.66
1	126	0.27	86.72	86.79	1176	1	1396.37
r	126	0.46	78.49	78.55	704	1	2102.64
short	117	0.27	86.65	86.72	872	1	1424.14

Table 3.6: Results for vcontrol

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	83.28	84.32	354	3	11.64
C-0	23	0.70	85.02	85.71	599	2	16.82
C-1	23	10.80	81.18	81.88	544	2	15.43
Random	23	38.33	83.62	83.97	628	1	15.94
0	23	52.96	86.06	86.06	633	0	25.26
1	23	56.79	83.62	84.32	610	2	21.66
r	23	59.93	82.58	82.93	420	1	20.50
short	19	41.12	86.06	86.76	700	2	33.80

For s1238 the no-seed case uses the most test vectors.

3.5.2 Results for the Viper Microprocessor

The results for the *vcontrol* circuit are shown in Table 3.6. This circuit is the control machine for a version of the Viper microprocessor. Its description was written in VHDL, synthesized using Compass AsicSyn and converted to the bench format for use with CRIS. In this case as well the no-seed case is better than the rest, with the exception of the constant-0 case which has a

Table 3.7: Results for 2-bit Viper

Sequence	length	i.f.c.	cov.	eff.	ntest	nred	cpu
No seed	-	0.00	80.34	80.60	2304	3	333.22
C-0	29	0.00	85.89	86.24	2683	4	505.86
C-1	29	1.41	81.39	81.57	2244	2	448.98
Random	29	13.58	87.48	87.48	3847	0	752.95
0	29	43.83	86.68	86.77	3831	1	777.34
1	29	19.22	88.10	88.18	3854	1	815.00
r	29	28.04	85.98	85.98	3769	0	550.07
short	26	31.04	85.01	85.01	3748	0	534.79

higher fault coverage.

Table 3.7 shows the results for a version of the Viper microprocessor with a two-bit wide datapath. Normally the processor has a datapath of 32 bits. The two-bit version was used in order to remove most of the memory elements related to the datapath from consideration. Most of the registers in the normal Viper are related to the general-purpose registers, which do not affect the state of the controller.

The 2-bit Viper has the most flip-flops of the designs considered (it is the second-largest in terms of gates). For this circuit alone, the results look promising. The highest fault coverage was achieved by the initial sequence with *don't-cares* replaced by ones, followed by the other versions of the formally-generated sequences. Test efficiency also followed this pattern. The no-seed test had fewer test vectors than those with the formally-generated seeds, and it also detected more redundant faults. Test generation time was also shortest for the no-seed case.

For this circuit, HITEC was able to achieve 98% efficiency and 95%

fault coverage using 2680 vectors. The test generation time was about 67000 seconds of CPU time (over 18 hours). In comparison, HITEC requires about 150,000 seconds to achieve 83% efficiency and 82% coverage on the complete 32-bit Viper design. An important experiment yet to be performed is to attempt to transform the sequences generated for the 2-bit Viper to apply them to the 32-bit version. The initial sequences could be obtained by simply expanding the datapath with *don't-care* values.

3.6 Summary

The results from these experiments do not point in a clear direction, but they are encouraging. The effects of the initial sequences are almost always to increase the test generation time, without changing the final fault coverage or test efficiency too much. The number of tests generated can vary. For the experiments performed here the no-seed case usually had the smallest number of tests. It also seemed more capable of discovering the redundant faults. For the larger two-bit Viper, the effect of the initial sequences was more pronounced, and showed more promise.

Unfortunately it is not always apparent which kind of initial sequence to use. Even so, it is possible to perform several test generation runs to find the best sequence; even more important than CPU time for test generation is the test application time for the manufactured circuit. CRIS can often run several times in the same amount of time as HITEC, making it possible to evaluate different seed sequences as has been done here.

One important factor is the length of the initial sequences. It is likely that longer initial sequences that go to more target states will give a more useful initial population for the genetic algorithm. This would improve the performance of test generation and increase the usefulness of this technique.

The technique of removing the last vector from each sequence did not make much difference. It is likely that once SCFV is able to generate the sequences for each of the most-distant states, the sequences will be more useful.

Much more experimentation could be done. The effects of sequences to move from one initial state to several target states have been investigated. It could be extended to moving from the reset state to each of the most distant states. Several different initial states could be used. Random sequences could also be included along with the formally-generated ones. This project has shown that the techniques of providing an initial sequence to a nondeterministic test generator can have a positive effect on its results.

Chapter 4

Template-based Test Generation

Test generators usually build test sets in terms of single vectors that are applied in sequence to detect faults. At the gate level, this is all the information that is available to the test generator. The instruction-level behavior of the circuit can be encoded into templates that can assist the test generator. For a microprocessor, each instruction in the instruction set can be described by its input sequence.

The most recent version of the CRIS test generator has the ability to perform test generation in terms of behavioral-level instructions described by a set of templates [12]. It generates sequences of instructions based on these templates to detect faults.

4.1 Test Generation with Instruction Templates

An instruction template specifies a sequence of inputs to apply to the circuit. Each input vector is associated with a clock cycle. During each cycle, inputs can be specified directly or left unspecified. The test generator is allowed to choose the value of the inputs left unspecified every time that it chooses to apply the specific template. Figure 4.1 shows an example template for a processor-like circuit with a reset input and an 8-bit memory input. During the first clock cycle, the reset input is asserted and the memory input is left unspecified. In the second clock cycle four bits of the memory input are set

Cycle	Reset	Memory In
0	1	xxxxxxxx
1	0	1001xxxx
2	0	xxxxxxxx

Figure 4.1: Example Instruction Template

to 1001, for example, to specify the part of the opcode. The reset input is held low, and the bottom four bits of the memory input are left unspecified. Finally in the third clock cycle, the reset line is still kept low and the memory input is left unspecified. This template would be useful, for example, if all arithmetic opcodes began with the sequence 1001 and performed an operation on the following value from memory. In this case, the template would describe an arithmetic operation performed on a value that is determined by the test generator. CRIS chooses the exact operation and the operand.

There can be a template for each instruction supported by the design. Instructions can also be grouped into useful sequences; there is no requirement that the templates describe the operation of the machine in the same terms as its instruction set.

4.2 Creating Instruction Templates

There are many possible ways to create a set of templates for use with a given design. Templates can be generated randomly, described manually, or generated using formal methods. Each of these methods will be described in the following sections.

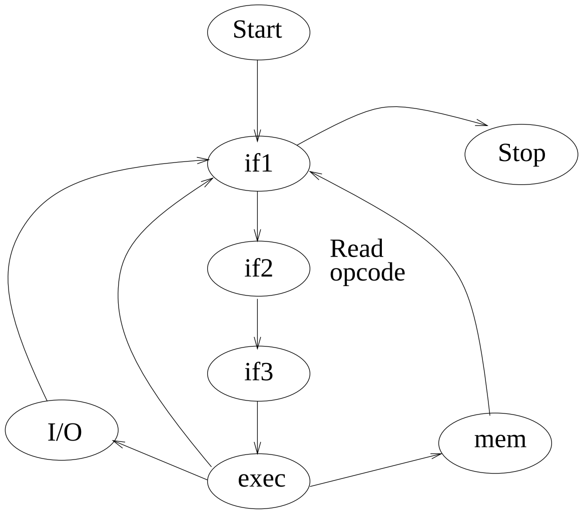

Figure 4.2: Viper Microprocessor State Machine

4.2.1 Randomly-generated templates

If the behavior of the circuit is not well known, a simple technique of randomly selecting from all possible templates can be used. This is not the preferred way of making the instruction templates but it can be used to get test generation results quickly from the design. If the test generation performance is acceptable using random templates, there is no need to spend the time to develop templates manually or formally. If, on the other hand, test generation performance is not acceptable, new templates can be developed using one of these other methods. The random technique used here was suggested by Saab [12].

The first step in randomly generating the templates is to determine a good structure for the instruction templates. This can be done using knowledge of the controlling finite state machine. For example, the Viper has a total of eight possible control states, and of these most instructions are described by four or five states (see Figure 4.2). Most instructions, then, can be described by templates that specify the behavior of the circuit for four or five cycles and present the desired opcode in the proper cycle (during if2). Given these constraints, useful templates can be generated.

It is also important to identify a template to reset the circuit. Experience with CRIS has shown that many of the templates should include a reset sequence. This ensures that CRIS will reset the circuit early in the process of generating tests, which is necessary before many faults can be detected. A template that leaves all inputs unspecified can also be included, so that the test generator can fall back to standard gate-level test generation. The result is a set of templates that may or may not give a good description of the behavior of the design.

Test generation performance may vary a great deal with the use of a random set of templates. There are many factors that can be varied: the length of the set, the number of cycles specified in each template, the number of inputs left unspecified for test generation, etc. If one set created using this technique does not lead to good test generation performance, it is always possible to create a new set and start again. It is also important to investigate the use of more structured methods, such as manually describing the templates or generating them formally.

4.2.2 Manually-generated templates

The real power of the use of instruction templates is their ability to describe the high-level behavior of the circuit. The random method of determining these does not exploit this capability. Instead of generating templates randomly, a designer can create them explicitly. Individual instructions and groups of instructions can be used as templates.

The most straightforward way to make a set of templates is to create a single template for each instruction in the instruction set of the processor. Certain fields in the opcode can be left unspecified, such as those that determine the location of the data to be used or the type of operation to perform. This leads to templates that describe groups of instructions. It is also possible to have multiple templates for the same instruction, for example using a different addressing mode in each one. If possible the set should also include templates for resetting the machine and also a completely unspecified template to allow the test generator to perform gate-level ATPG.

4.2.3 Formally-generated templates

Finally it also may be possible to generate a set of templates using formal methods. Implicit enumeration methods can be used to determine templates for certain high-level operations, such as applying inputs to a subcircuit and then taking its outputs to the primary outputs [9]. This allows the test generator access to the inputs and outputs of each module in the circuit, and effectively reduces the test generation problem to considering each module in turn, rather than the circuit as a whole.

4.3 Results

Template-based test generation was performed on the Viper microprocessor. Table 4.1 shows the results for not using templates (by a previous version of CRIS that did not support them) followed by the best results for several experiments using templates. For all cases, not enough redundant faults

Table 4.1: Instruction Template Results

Templates	# Templates	Test Vectors	coverage
Without Templates	—	3211	78.8
None	0	98	42.4
Hand-specified	15	3949	40.4
Random	21	5411	78.6
Combined	36	4964	68.0

were found to make the efficiency differ by more than 0.1% from the fault coverage. The second line shows the performance of the newer template-enabled version of CRIS when no instruction templates are given. It is evident that this version of CRIS has a great deal of trouble when it is not given any instruction-level information. The third line shows the results for a set of 15 hand-specified templates. There were seven basic templates, each with a version that asserted RESET and a version that did not. The 15th template is a RESET template. CRIS generated many tests, but this set of templates did not actually help: the fault coverage is actually lower than the case of no templates specified.

The fourth line in Table 4.1 gives results for using 20 random templates plus a reset template. These templates were generated by a program that first chose some of the memory input signals at random and then randomly generated a sequence of inputs for them. A reset vector was prepended to each template with a 50% probability. This was done because CRIS had trouble if it did not reset the circuit early in the test generation process. Of the 21 given templates, only 11 were actually chosen for execution, including the reset template. The use of random templates proved to be the best for this circuit with fault coverage of 78.6%. This is very close to the 78.7% fault coverage obtained by the previous version of CRIS shown in the first line of the table. Using more randomly-generated templates did not improve the results of the test generation. Trials were made with 30, 40, 50 and 100 random templates, but none of these had better results. The final line in the table shows the results of giving all the random and hand-coded templates to CRIS at the same time. Surprisingly, fault coverage is again diminished. This shows that CRIS is very sensitive to the templates it is given. Simply adding to a set of templates that has performed well does not guarantee that performance will be improved. Since there are more templates to consider, CRIS may not determine again how to use each of them optimally. It is evident that care should be taken in specifying these templates.

Randomly-generated templates fill the role of giving CRIS an initial population of sequences of vectors to use for searching the circuit's input space. It may be possible to improve the fault coverage with carefully written hand-coded templates. It may be necessary to run CRIS several times with different versions of the templates to determine the best set to use.

Chapter 5

Behavioral DFT Techniques to Improve Testability

The methods presented in the previous chapters for improving circuit testing may not provide acceptable testability in every case. Sometimes it is necessary to make changes in the circuit itself to improve testability. This chapter investigates the use of testability features applied to the behavioral-level description of a circuit. This technique has been called Behavioral Modifications for Testability (BMFT) [18]. Making certain changes to the behavioral description of a circuit has an effect on the corresponding gate-level implementation. If changes are made appropriately, the normal operation of the circuit will be augmented by behavior that is accommodating to test generation.

The original BMFT techniques have been extended in two areas. First, the choice of modifications to apply is assisted by targeting areas of the circuit with low testability. The process of applying the changes has been made easier with software that interactively modifies the design.

Figure 5.1 shows the different approaches discussed in this chapter. Figure 5.1a shows gate-level modifications for testability (GMFT) being applied to the netlist. Figure 5.1b shows BMFT with synthesis at each step for the purpose of testability evaluation. Figure 5.1c shows the case in which testability analysis is possible without the use of the gate-level description.

There are advantages to BMFT in addition to improved gate-level testability. In order for verification and simulation work done using the behav-

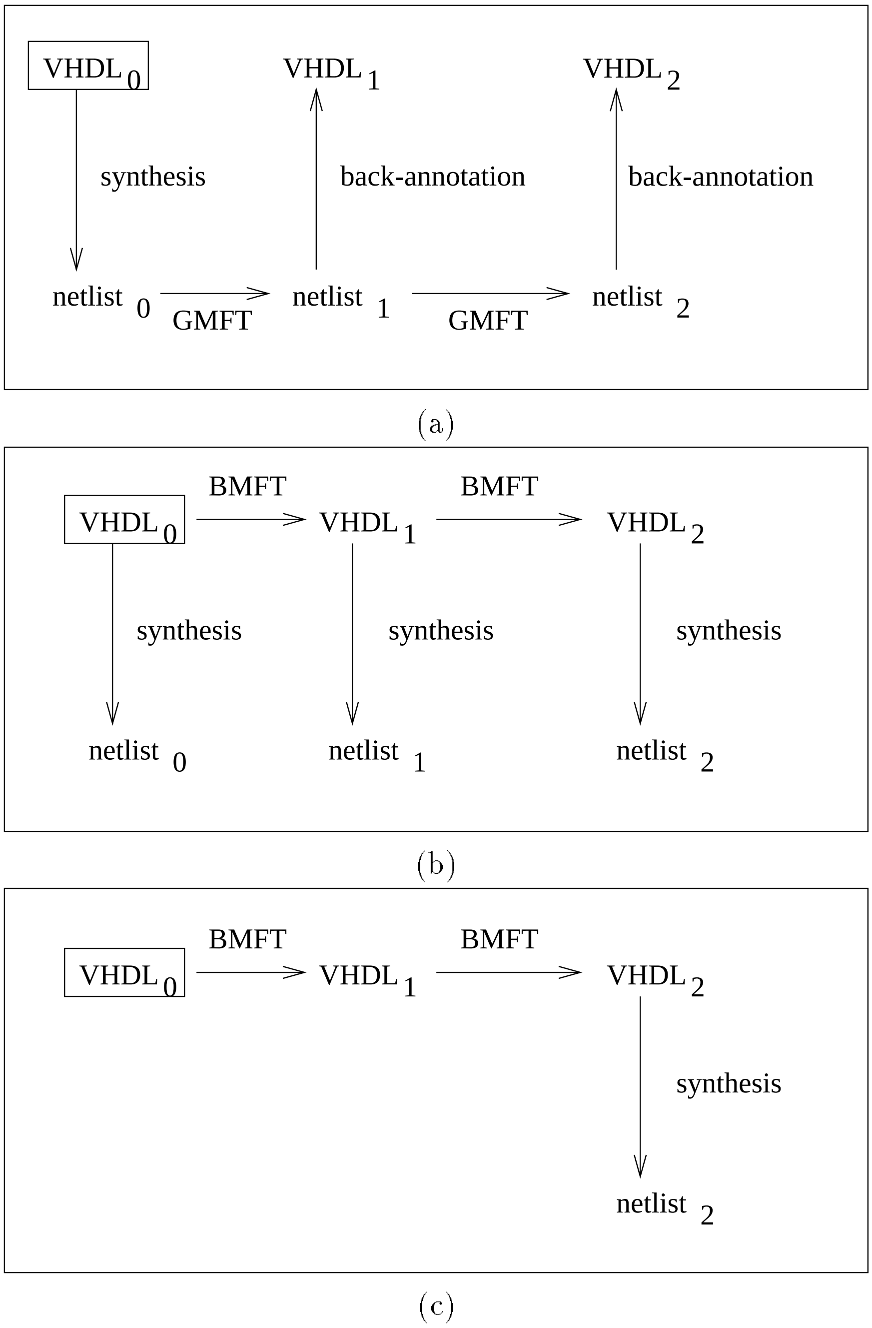

Figure 5.1: DFT Maintaining Behavioral and Gate-Level Equivalence

ioral description to apply to the gate-level description, the behavioral and gate descriptions must remain as consistent as possible. Traditional DFT methods (Figure 5.1a) often modify the gate-level description without regard for any behavioral description. These include inserting *ad hoc* testability features such as test points and reset lines, and structured techniques generally involving scan latches [1]. In order to keep the descriptions consistent, a back-annotation step would be needed in the direction opposite to that of synthesis (See Figure 5.1a). It is not clear if this back-annotation is even possible, since not all changes in a netlist can be modeled at a higher level. The proposed technique makes the changes at the behavioral level and does not require a back-annotation step.

One important advantage of making behavioral changes instead of gate-level changes to improve testability is that these can be done early in the design cycle. It is easier to start the design of a very large VLSI circuit at the behavioral level than the gate level, and in fact most designs are first simulated at the behavioral level. By taking testability into consideration at the behavioral level, it can be applied more effectively than if it is relegated to the gate level alone and the end of the design cycle.

Another advantage to behavioral modifications as opposed to gate-level modifications is that the changes made in the behavioral description are easier to make correctly. The exact impact of a change at the gate level can be difficult to characterize. In contrast, the effects of a behavioral-level change can be easily identified. If possible, the changed circuit should be taken through the same simulation steps that were used to validate the original description. An even better alternative is formal verification of the changed description against the original description or against the initial specifications. If nothing else is available, it is much easier for a designer to understand the changes made to a VHDL description than those made to a large netlist.

1. Evaluate the testability of the design in its current form,
2. Make suggestions regarding modifications that can improve the testability, and
3. If testability is not acceptable, apply one or more suggested modifications and repeat from Step 1.

Figure 5.2: Proposed Design Cycle

5.1 Proposed Design Cycle

The proposed design cycle is a simple one. The process begins after a behavioral description of the circuit is available. The steps are shown in Figure 5.2. These steps are repeated until the testability of the design is satisfactory.

Testability evaluation (in Step 1) can be done at either the behavioral level or the gate level. In the ideal case, this analysis is done at the behavioral level so that the cycle will not involve the use of a gate-level description at all (Figure 5.1c). If behavioral testability analysis is not available, the designer must rely on gate-level testability analysis (Figure 5.1b), which involves synthesizing the behavioral description into a gate-level description and performing test generation. Both of these operations are computationally intensive and if they cannot be avoided then they must be done as quickly as possible. This is the situation for the analysis performed in the present work.

Areas of the circuit must be targeted for consideration in Step 2. Ideally, the targeting process would also take place completely at the behavioral level of the design, and in fact there are promising methods for doing this. If behavioral-level techniques are not available, techniques that rely on the netlist can be substituted. This is done by synthesizing, running gate-level ATPG and mapping the undetected faults back to the behavioral level. Two different mapping techniques are considered in Section 5.2.

Once areas of the design are targeted for modification, an appropriate testability feature must be selected and applied to the design in Step 3. With some work, the designer can make changes to the design that will improve the testability. It is also possible to apply standard modifications from a library of templates interactively. Section 5.3 discusses both of these methods.

5.2 Testability Evaluation

In order to determine where to apply the modifications, it is necessary to evaluate the testability of the design at the behavioral level. There are techniques for this reported in the literature. Specifically, ATKET ("Automatic Test Knowledge Extraction Tool") analyzes behavioral VHDL designs to indicate testability information such as justification and propagation modes [19]. This information has been successfully used for improving testability [20, 18].

Another method of testability measurement, called BETA (BEhavioral Testability Analysis) has been developed [4]. This measures the testability of a design based on a *Control Flow Graph* (CFG) extracted from the behavioral description. This information has been used to perform *Test Statement Insertion* in the CFG description for improving testability [3]. Changes are not made to the behavioral description itself, however.

In the absence of such tools that work at the behavioral level, meaner tactics are required. In this case, a bridge must be built between the gate level testability and that of the behavioral level. Two techniques were developed for this purpose and analyzed. They involve mapping gate-level faults to the behavioral level in terms of behavioral modules and behavioral signals.

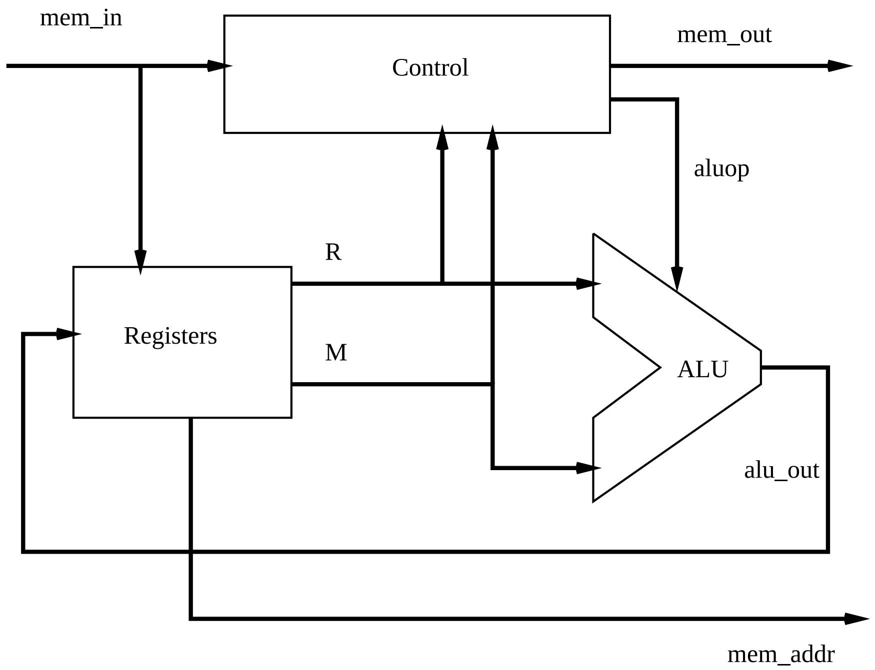

Figure 5.3: The Hierarchical Design of the Viper Microprocessor

Figure 5.3 shows the design of the Viper microprocessor, which was originally used for formal verification purposes [7]. It will serve to illustrate the techniques described here. The original design was modified to have a two-level hierarchy, in which behaviorally-described modules are interconnected structurally at the top level. The design is contained in a single VHDL file that is written in a style that can be synthesized by the Compass AsicSyn tool. The modules ALU, Registers, and Control are behavioral VHDL entities containing concurrent VHDL processes. There are no structural components beneath any of these entities. The ALU is described by a single process and, when synthesized, contains no state elements. The Registers component has three concurrent processes: one to assign the general-purpose registers themselves, one to assign the R and M registers, and one to assign the value of the address lines. The Control component has a clocked process for the state machine and an unclocked process for assigning the outputs.

5.2.1 Mapping faults to behavioral modules

The Compass AsicSyn synthesizer preserves information about behavioral circuit modules in its netlist description (see Section 2.2). The *nls2bench* conversion utility maintains a mapping of gate-level signals to the high level modules. After test generation, this allows the undetected faults to be mapped to the behavioral level modules that contain them. As mentioned above, test generation is not desirable early in the design cycle. This technique should only be used when behavioral-level techniques are not available to do testability analysis. Fast methods for synthesis and test generation are also preferred; this is why nondeterministic rather than deterministic test generation is being used.

A program (*trace_faults*) was written to perform this mapping and to present a list of the modules and the number of undetected faults each one contains. For a hierarchical design, the desired resolution of the mapping can be specified down to the lowest level of the hierarchy. If the design has many levels of structural hierarchy, the faults can be traced to very specific behavioral modules. If, at the other extreme, there is just a single behavioral module in the design, the results will not be useful since there is only one module to which to map them. This illustrates a tradeoff between the amount of structural and behavioral description used in a design. A completely structural design (with many interconnected VHDL entities) will allow for good localization of undetected faults but gives no advantages of behavioral design. A completely behavioral description (a single VHDL entity) may be more useful for describing a large circuit, but in the case of this simple fault mapping procedure it will give no information about the specific location of undetected faults.

For the purpose of this mapping, the modules PI and PO are added to the hierarchy to include those signals that are primary inputs and primary outputs.

This technique has been applied to the Viper. Table 5.1 shows that the ALU contains about 40% of all the possible faults in the design, but Table 5.2 shows that after ATPG, it contains about 68% of the undetected faults. These tables also show the number of nets associated with each module, to give an indication of the relative sizes. The total number of nets is over 100% because those nets shared by modules have been counted twice. In particular, the primary inputs are counted as being within module PI and also in the modules to which they are connected.

The fault coverage per module can be determined using the information provided by this technique. First, the test generation program makes a list of all of the stuck-at faults in the circuit. Using the *trace_faults* program, the total number of faults for each module is determined. The coverage for each module is computed as the number of *detected* faults in each module divided

Table 5.1: Example of mapping all faults to modules

module	nets	%total	faults	%total
PI	34	0.82	66	0.66
PO	0	0.00	66	0.66
ALU	1536	36.98	3969	39.63
REGS	2364	56.91	5319	53.11
CONTROL	256	6.16	595	5.94
(totals)	4190	100.87	10015	100.00

Table 5.2: Example of mapping undetected faults to modules

module	nets	%total	undetected	%total
PI	34	0.82	0	0.00
PO	0	0.00	0	0.00
ALU	1536	36.98	1277	68.25
REGS	2364	56.91	547	29.24
CONTROL	256	6.16	47	2.51
(totals)	4190	100.87	1871	100.00

Table 5.3: Example of in-circuit fault coverage by module

Module	In-Circuit coverage
PI	100.00 %
PO	100.00 %
ALU	67.83 %
REGS	89.72 %
CONTROL	92.10 %

by the total number of faults in each module:

$$\text{in-circuit coverage} = \frac{\text{faults detected}}{\text{total faults}} = \frac{\text{total faults} - \text{undetected faults}}{\text{total faults}}$$

This is called the *in-circuit* coverage because it gives the fault coverage of each module for tests that were generated for the circuit as a whole. Table 5.3 gives an example of in-circuit fault coverages for the Viper microprocessor.

Test generation can be performed on a module itself (to determine the *standalone* testability) and the results compared to the in-circuit results. Constraints on the module imposed by its position in the larger circuit mean that it will often not be possible to apply every possible test to the module. Thus the standalone testability is the upper limit of the in-circuit testability for the module. A comparison of the standalone fault coverage to the in-circuit coverage can indicate if a module is being tested well. This can be used to indicate possible locations for applying modifications to the circuit. For example, if the ALU module has a standalone fault coverage of 100% but an in-circuit coverage of 68% as indicated in Table 5.3, the designer should consider making the inputs and outputs of the ALU more accessible to improve test generation.

Modular fault mapping is a useful tool for determining where in a circuit the test generator is having difficulty and for guiding decisions about circuit modifications to make a more easily testable circuit. Section 5.3 describes modifications that can be used according to these suggestions. Experimental results using this technique are presented in Section 5.4.1.

5.2.2 Mapping faults to behavioral signals

An important problem with mapping undetected faults to behavioral-level modules is that it has limited resolution. If the modules are large then the technique does not indicate problem areas very precisely: resolution is limited to the size of the smallest modules. Instead of mapping faults to large behavioral-level modules, it is possible to map them to behavioral signals themselves. This technique gives better resolution than the modular mapping technique, since there are many more behavioral signals than modules.

Undetected faults are listed in terms of gate-level signals on which they occur, so mapping them to behavioral signals requires mapping gate-level signals to corresponding behavioral-level signals. Synthesis creates many signals at the gate level, so not all of the gate-level signals will be mapped directly to behavioral signals. Many faults will occur on these signals that do not map directly to behavioral signals. The mapping technique must therefore include a means of associating these signals with behaviorally-mapped signals that are in some sense nearby. The proposed technique traverses a signal dependency graph to find *upstream* behavioral signals that are likely to affect the fault and *downstream* behavioral signals that are likely to be affected by the fault. For each behavioral signal, a count of the nearby faults is kept, along with the average distance from the behavioral signal to the undetected faults. Those behavioral signals with high counts are considered candidates for behavioral modification. The steps involved in signal fault mapping are shown in Figure 5.4. Figure 5.5 depicts the interactions of the tools used for this process.

Each step will be described in turn.

1. Create a mapping between VHDL and gate-level signals
2. Determine gate-level signal dependencies
3. For each fault, increment the counters of "nearby" behavioral signals.
4. List the behavioral signals close to the most undetected faults.

Figure 5.4: Steps in Signal Fault Mapping

Gate-to-Behavioral Signal Mapping The Compass AsicSyn synthesis tool preserves behavioral names in the synthesized netlist. During conversion from the *nls* format to the *bench* format, *nls2bench* makes a list of all of the behavioral names and corresponding gate-level signal names. During the signal mapping process, the program obtains directly from the VHDL a list of behavioral signals along with the position of each one in the module hierarchy. It then scans the signal list from *nls2bench* and identifies the correspondences to build the data structure for mapping from gate level signals to behavioral signals. Many gate-level signals can be mapped to the same behavioral signal, for example in the case of a signal vector (bus). It is also possible for many behavioral signals to be mapped to the same gate-level signal (e.g. when two signals at different behavioral levels are directly connected). Thus the mapping can be many-to-

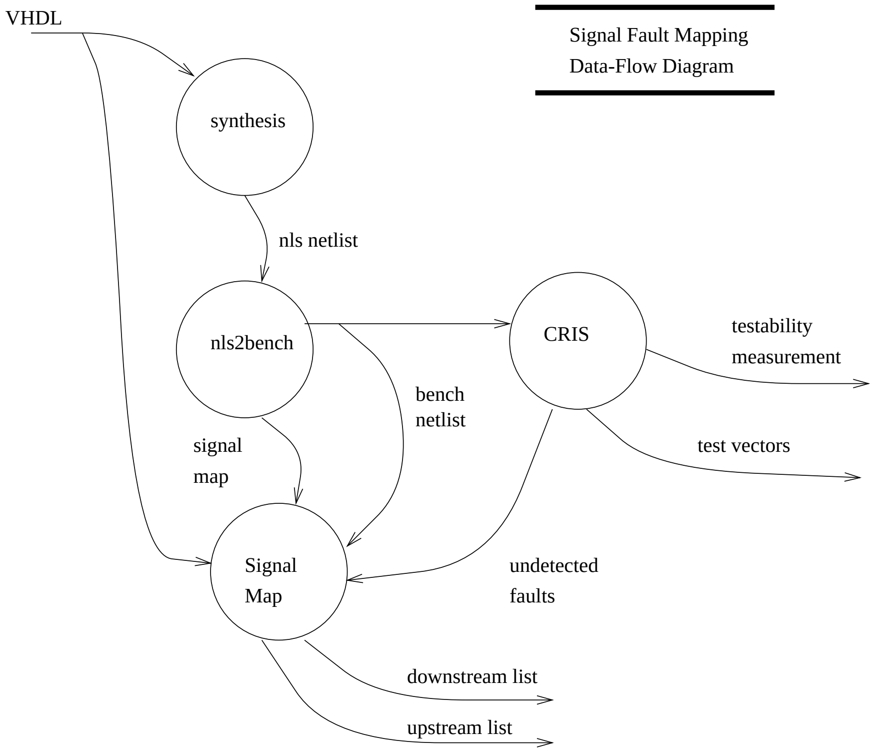

Figure 5.5: Mapping Gate-Level Faults to Behavioral Signals

many.

Determining Gate-Level Signal Dependencies The second data structure built by the signal mapping software is the dependency graph for the gate-level circuit itself. For every signal in the circuit, the immediate predecessors and successors are determined. This facilitates the searching done in the next step.

Mapping Faults to Nearby Behavioral Signals Once the gate-to-behavioral mapping and the gate-level signal dependencies are determined, the mapping of faults to behavioral signals can begin. For each undetected fault, the signal dependency graph is searched for signals that map directly to behavioral signals. The graph is traversed in the forward direction to find signals in the output cone of the fault that should be made observable. It is traversed in the backward direction to find signals in the input cone that should be made controllable. Figure 5.6 shows an example of this search method.

When a behavioral signal is found, a counter associated with that signal and the search direction is incremented. The search does not stop when the first behavioral signal is found in each direction; rather, it is limited by the depth of the search. This means that all behavioral signals within a certain number of levels before and after the fault in the circuit will have their counters incremented. This strategy was adopted to ensure that the final values of the counters for each behavioral signal corresponds to the number of faults in the vicinity of the signal. The maximum search depth can be set to any desired value. In practice it is useful to keep it at a small value, to keep the search in a localized area. If faults are deep within blocks of circuitry that do not have

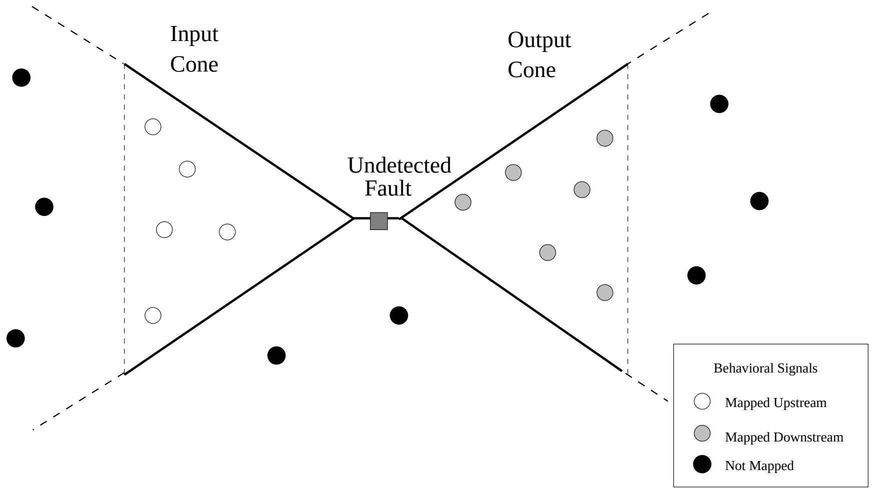

Figure 5.6: Mapping Faults to Nearby Behavioral Signals

behavioral signals, then it is possible that they will not be mapped at all.

Suggesting Behavioral Modifications Once the gate level faults are mapped to nearby behavioral signals, this information can be used to suggest appropriate behavioral modifications. If a signal is downstream from many faults it is a candidate for observation. Similarly, if a signal is upstream from many faults it is a candidate for adding controllability. The number of faults that are near the signal and their distance to the signal serve to indicate how much improving that signal's accessibility will affect the testability of the circuit.

Table 5.4 shows an example output of the software that performs signal fault mapping. In the downstream direction, the ALUOUT signal has 286 faults nearby at an average distance of 3.7 gates away. In the upstream direction, the M signal has 1391 faults nearby at an average distance of 2.8 gates away. This indicates that ALUOUT is a candidate for observation and M is a candidate for controlling. For this example, the search depth limit

Table 5.4: Signal Fault Mapping Example

Cone	Signal	Fault Count	Average Distance
Output	ALUOUT	286	3.70
Output	REG3	133	3.70
Output	REG2	104	3.30
Output	M	96	3.30
Output	REG1	34	3.90
Input	M	1391	2.80
Input	IR	594	3.00
Input	REG_SEL	219	3.30
Input	REG3	218	2.80
Input	FETCH	115	3.60
The search depth was 5, a total of 556 faults were mapped downstream and 1633 were mapped upstream; 1700 total faults were mapped and 171 were not mapped.			

Table 5.5: Signal Fault Mapping Example

Cone	Signal	Fault Count	Average Distance
Output	REG3	2213	7.70
Output	ALUOUT	2199	7.00
Output	REG2	1845	7.50
Output	REG1	1343	7.80
Output	REG0	1275	8.10
Output	M	542	6.60
Input	STATE	7818	8.20
Input	M	7608	6.60
Input	IR	6277	7.30
Input	ASSIGN_R	4221	7.90
Input	FETCH	2493	8.10
Input	REG3	2215	7.30
The search depth was 10, a total of 1274 faults were mapped downstream and 1834 were mapped upstream; all 1871 undetected faults were mapped.			

was 5 gates. A total of 1700 of the 1871 undetected faults were mapped to behavioral signals. Of these, 556 were mapped to downstream signals and 1633 were mapped to upstream signals. Table 5.5 shows the effect of increasing the maximum search depth. In this case, all of the undetected faults were mapped to behavioral-level signals. Some of the signals listed indicate more faults than were actually being mapped; the reason for this is twofold. First, the faults are mapped to all of the nearby behavioral signals, so each fault can be mapped many times. Second, most of the behavioral signals listed are several bits wide, so they actually correspond to many gate-level signals.

5.3 Behavioral Modification techniques

Once the testability of a circuit has been determined and suggestions made about behavioral changes to improve testability, the final step is to decide which changes to make and to apply them. The choice of modifications is guided by the suggestions made by the testability evaluation techniques, but there can be other factors to consider, such as the area overhead of the modifications and pin requirements. This section describes several testability features that can be used to meet these various criteria. The actual modification of the circuit can be done manually or with the assistance of an interactive computer program. In general, manual modifications are more time-consuming and error prone than the automated technique, but they allow for better optimization.

Often, the behavioral changes must only be active when the circuit is not in normal operation. This is particularly true of testability features that improve controllability by assigning values to certain signals. It is also true of some observability features, for example when normal output pins are used to show the value of internal signals. Only when dedicated outputs are added is this not the case. The behavioral modifications described here are divided into those applied manually by the designer and those that are applied interactively with the help of an interactive computer program. The modifications presented here are shown applied to the Viper microprocessor. Figure 5.3 shows the unmodified circuit at the behavioral level.

5.3.1 Manual *Ad Hoc* BMFT

Any change that increases the accessibility of an area of low testability in the circuit will improve the testability of the circuit as a whole. If the suggested target for modification is a module, then the goal is to provide better controllability of the inputs to that module, better observability of the outputs, or both. If the target is a signal in the behavioral description, then the controllability of the signal should be improved if it is upstream of the faults or the observability should be improved if it is downstream of the faults. The following testability features are described:

- Reset and Partial Reset
- Module Transparency
- Signal Controllability and Observability
- Observation Encoding
- State History Compression
- Test Instructions

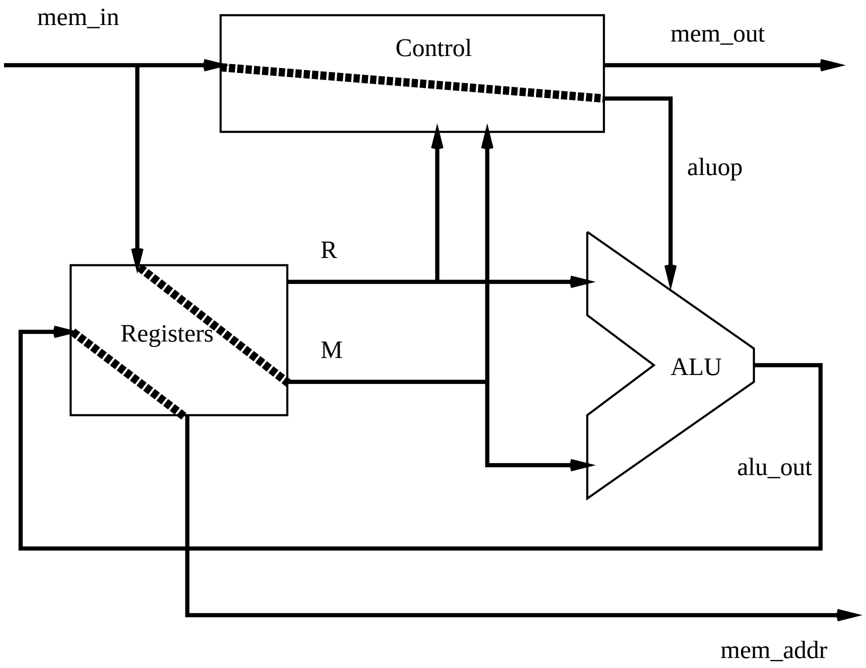

Figure 5.7: Module Transparency in the Viper

Reset and Partial Reset If the original circuit does not have a mechanism to reset some or all of its state elements, test generation will be difficult. In fact, both CRIS and HITEC had great difficulty generating tests for the hierarchical Viper without a reset; adding a reset made it possible to generate tests. Providing a reset line is important for simulation and verification as well. Most circuits are designed with support for reset.

A refinement of the reset testability feature is *Partial Reset*. Different groups of state elements are reset independently. This allows the test generator to explore the state space of the circuit from different starting points.

Module Transparency Allowing data to travel through a module unmodified can be useful for applying tests to other modules. For example, Figure 5.7

```
IF Reset THEN
        M ⇐ Mem_In;                              – in Regs module
        Mem_Addr ⇐ ALUOUT(19 downto 0);          – in Regs module
        IR ⇐ Mem_In;                             – in Control module
ELSE
        ...
END IF
```

Figure 5.8: Module Transparency Pseudocode

shows the use of module transparency to increase the observability and controllability of the ALU. In this example, transparency is added to the Control and Registers modules. The dashed lines in this and similar diagrams below indicate the signal paths and other testability features added by the BMFT techniques. The ALUOP and M signals are derived from the MEM_IN input and the output of the ALU (ALU_OUT) is propagated to the MEM_ADDR output. This must not occur during normal operation, so in this case it occurs only during reset. Another way to achieve this is to add a single input pin to control whether the design is in a test mode. There are two important caveats regarding this example. First, the ALUOP and M signals are not independent, since they are derived from the same input signal. Second, the complete output is not available, since the ALU_OUT signal is 32 bits wide and the MEM_ADDR signal is only 20 bits wide. Note also that in the Control module, the IR signal is assigned; this is because the ALUOP signal is derived directly from it.

Figure 5.8 gives an indication of the VHDL code required for this modification. Note that the actual description would have to be modified in

```
IF Test THEN
        R(15 : 0) ⇐ Mem_In(15 : 0);            – in Registers module
        M(15 : 0) ⇐ Mem_In(31 : 16);           – in Registers module
        Mem_Out ⇐ ALU_out;                     – top level
ELSE
        ...
END IF
```

Figure 5.9: Bus Controllability/Observability

Test Mode	Assignments
00	normal operation
01	R $\Leftarrow$ MEM_IN; MEM_OUT $\Leftarrow$ ALUOUT
10	M $\Leftarrow$ MEM_IN; MEM_OUT $\Leftarrow$ ALUOUT
11	R,M $\Leftarrow$ MEM_IN; MEM_OUT $\Leftarrow$ ALUOUT

Figure 5.10: Multiple test modes

several places, since these assignments are in different behavioral modules.

Signal Controllability and Observability Another important testability feature is *Signal Observability and Controllability* [17]. Using this technique, VHDL signals (which can be any VHDL type, including a single bit or multiple bits) are directly controlled or observed during testing (or reset) conditions. Figure 5.9 shows one possible application of this technique to the Viper. Notice that the signals are only partially controlled due to the small number of inputs. It is important to be able to control the signals independently. It is also possible to have these modifications performed automatically (see Section 5.3.2).

In some cases, it may be desirable to have separate control of the signals under different testing conditions. This provides the test generator with different types of access to the same signals. Figure 5.10 shows the use of two test control pins. When both pins are held low, the circuit is in normal operation. Other combinations of the pins perform different assignments to improve the accessibility of the ALU. Note that R and M are assigned by themselves in two of the modes, making the assignments independent. In the last mode, both of them are assigned. In all three test modes, the output of the ALU is taken directly to the primary outputs. Figure 5.11 shows the application of both of these kinds of bus accessibility. The dashed lines indicate signal paths added to achieve this control during test mode.

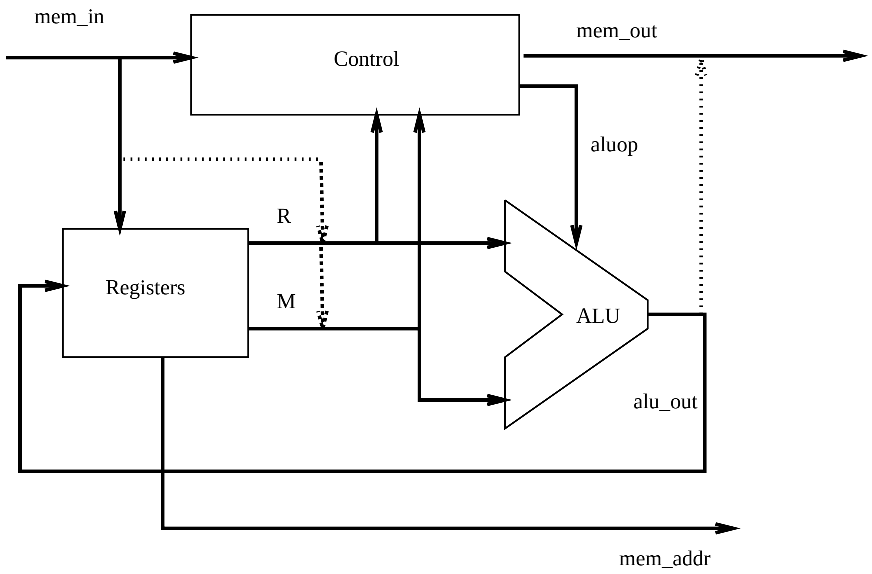

Figure 5.11: Signal Accessibility

Observation Encoding Sometimes it is necessary to add dedicated outputs for observing internal signals. In this case, it is important to keep the number of extra outputs as low as possible. One method of doing this is to encode the

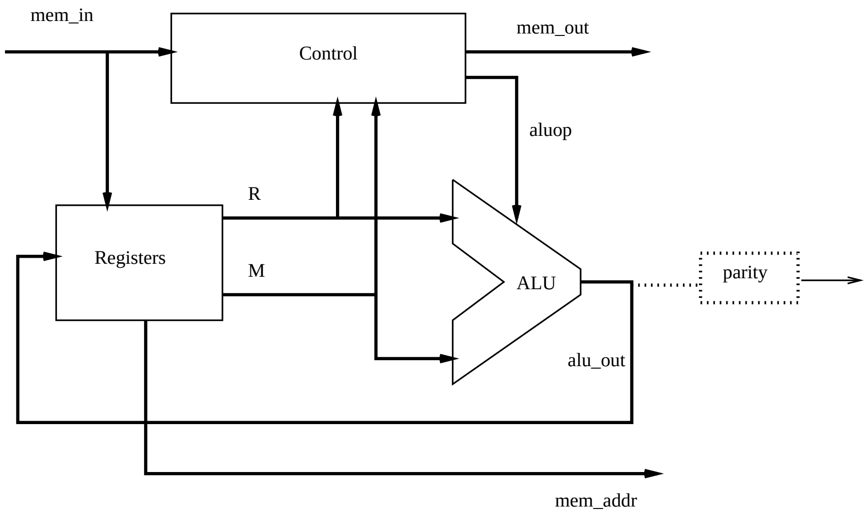

Figure 5.12: Observation Encoding

observations into fewer bits than are being observed, for example using a parity function (this is another modification that can also be performed automatically; see section 5.3.2). Figure 5.12 shows the addition of a parity encoder to *aluout*. This technique is similar to the popular gate-level XOR-tree technique [5], applied at the behavioral level.

If more output signals are available, priority encoding can be used. Each single bit of the vector to be observed is assigned a number. The outputs are set to the value of the highest logically active bit in the vector. If the vector is n bits wide, the required number of outputs is $\log n$. This would be useful for observing a vector of observation points that are not closely related in the circuit and are that are usually inactive. Since only one point is observed at a time, much information is lost if many of them are active at once. This is nonetheless a useful way to cut down on the number of observation pins needed.

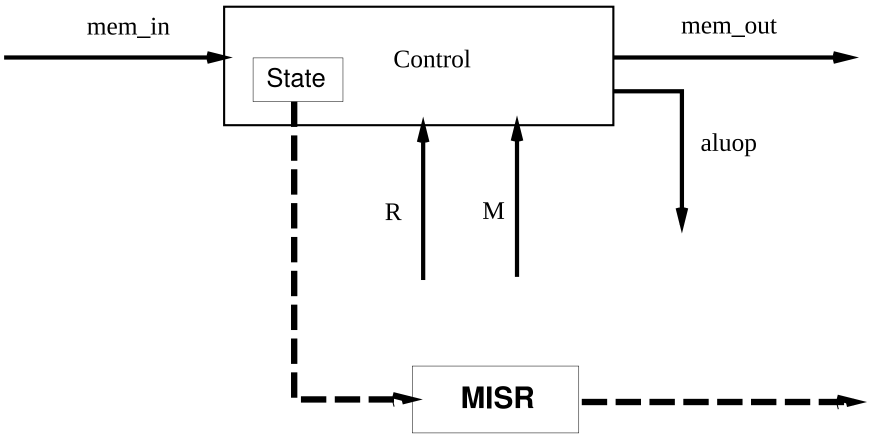

Figure 5.13: Example of State History Compression

State History Compression Another method of minimizing the number of output pins required for observation is to encode the history of the values of a bus using a multiple-input signal register (MISR). MISRs are often used for as output response analyzers (ORAs) for built-in self test [1]. Using a MISR, the history of values on any particular bus in the design can be compressed and shifted out one bit at a time on a single output pin. As test generation or test application progresses, fault effects can be caught in the MISR and seen at the output after a finite number of clock cycles. Figure 5.13 shows the application of a MISR to encode the history of the Viper state machine.

This technique can also be used to reduce the number of extra observation outputs added to a design. The number of output pins added in order to observe interior signals can be reduced by the addition of one or more MISRs. Test generation can be applied to the circuit after the outputs are added but before the MISR is in place. After test generation, an appropriate MISR can be selected to encode the faults detected on those outputs, and in the final circuit the extra outputs will be replaced by the single output of the MISR. Depend-

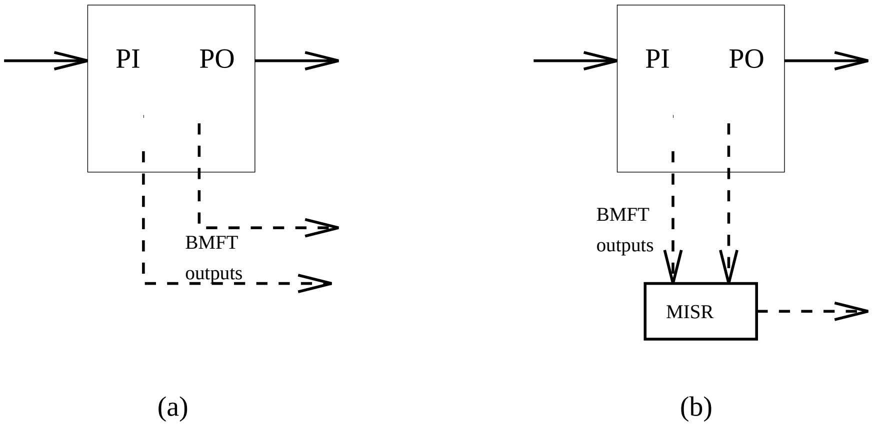

Figure 5.14: Using a MISR to reduce output overhead

ing on the structure of the selected MISR, it is possible that some aliasing of multiple fault effects would occur; fault simulation on the final circuit using the generated test set would detect any aliasing. Figure 5.14 shows an example of this technique. Circuit (a) is used for test generation, and after test generation the MISR is added to the final design (b).

Test Instructions In a similar vein, instructions can be added to the instruction set of a microprocessor-like circuit to assist in test generation and application. This is especially useful if there are unused instructions in the instruction set. During normal operation these instructions should be forbidden, but in test mode they can be made to do something useful. Chapter 6 describes this technique in more detail.

5.3.2 Interactive BMFT

A set of standard templates has been developed for interactively modifying the design at the behavioral level. Currently there are restrictions on language (only VHDL is supported) and coding style. However, the techniques are applicable to other behavioral-level languages.

When a template is applied, the template program directly modifies the VHDL program. This is in contrast with the *Test Statement Insertion* (TSI) technique [3] previously discussed. TSI makes changes to the *Control Flow Graph* derived from the high-level description, whereas the present technique modifies the high-level description itself. The important difference is that BMFT is a design technique, whereas TSI and similar techniques are intended for strictly automated testability improvements as extensions of behavioral synthesis. BMFT is intended to assist the designer in making appropriate choices about testability features.

It is important to note that only signals local to a behavioral module (those that are neither inputs nor outputs) need to be considered for modification. Consider first the top level, for which the primary inputs and primary outputs do not need to be themselves changed. The inputs are completely controllable and the outputs are completely observable. Thus it is only the internal signals, those interconnecting submodules, that need to be considered. This argument applies recursively to the submodules themselves, since their inputs and outputs are connected to signals from the level above. This also removes duplicate signals from consideration, since these usually are signals at different levels connected to each other.

A suite of programs written in the Perl language has been developed to make changes to circuits described VHDL. It currently supports designs composed of behavioral components interconnected structurally. The three kinds of testability templates that are currently supported are *observe*, *XOR-*

observe, and *control*. The *observe* template makes any given VHDL signal directly observable. The *XOR-observe* template adds a single-bit output for the parity output of any signal. The *control* template adds a test pin to determine whether a given signal is assigned normally or from another signal in the design.

Observe In order to observe an internal signal of a VHDL entity, another signal is needed to take it to the outputs (for simplicity, bidirectional signals are not currently supported). The *observe* template applies such a modification to a circuit, with the following steps:

1. Add a new primary output of the same type as the signal to be observed.
2. Add outputs to each component in the structural hierarchy containing the signal.
3. Insert a VHDL assignment statement to connect the signal to this new output.

XOR-Observe The *xor-observe* template is similar to the *observe* template but it adds only a single bit output to the design. It currently can only be used with vector signals (e.g. signals of type *STD_LOGIC_VECTOR*). It adds a single new output in the same fashion as *observe*. Instead of a direct assignment, however, it adds an assignment that computes the parity of the given vector using exclusive-or gates.

Control Signals can be made directly controllable using the *control* template. This modification is more involved than the addition of observability, since signal assignments and evaluations must be modified in the design. In the following discussion, S is the signal being controlled, C is the signal containing the desired value, and T is the control signal that selects the normal or controlled value for S. The assignment $S \Leftarrow \text{mux}(T, C, S')$ indicates a multiplexer that takes the value of C or S' depending on T. The procedure for making the modifications is as follows:

1. Add the input signal C to the design if it is not already present
2. Add the test pin T to the design if it is not already present.
3. Add local signal S' to the given module.
4. Replace S with S' when it is referenced (not where it is assigned).
5. Add to the module the assignment $S \Leftarrow \text{mux}(T, C, S')$

Figure 5.15 shows an example of adding controllability to a behavioral-level signal. S is a signal local to the behavioral module, meaning that it is neither an input nor an output of the module. To make it controllable, references to S are replaced by S'. Assignments of S remain unchanged. S' is then assigned the output of the multiplexer so that T can switch between normal operation ($S' \Leftarrow S$) and test operation ($S \Leftarrow C$).

Note that there is considerable latitude in choosing the conditions under which the signal is controlled and how the controlling value is derived. The designer can specify a test pin that is already present in the design, or one can be added automatically. Similarly, the controlling value C can be supplied by another signal already present in the design or an input for it will be added

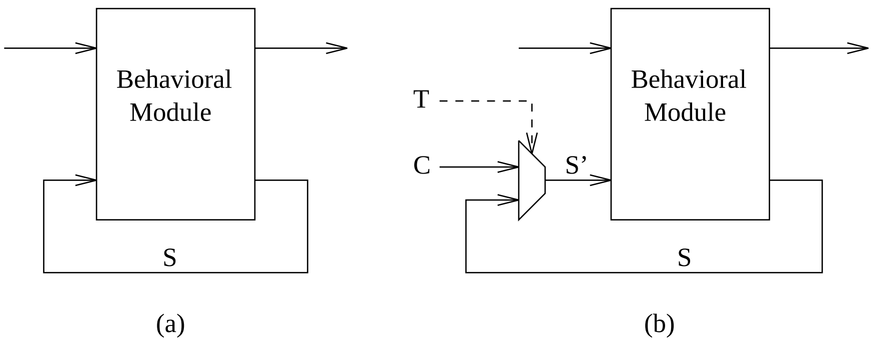

Figure 5.15: Adding controllability to a behavioral signal

Table 5.6: Summary of Modular Fault Mapping Experiments

viper h	Original hierarchical Viper
viper hr	Hierarchical Viper with reset
viper hr 1	Viper with Module Transparency for ALU accessibility
viper hr 2	Signal control/observe (See Figures 5.11 and 5.9.
viper hr 3	Observation encoding: ALUOUT parity
viper hr 4	Signal control/observe with multiple test modes

automatically. The software requires that the controlling value be the same type as the signal to control.

5.4 BMFT Results

5.4.1 Modular Fault Mapping Results

Several experiments were run using the modular fault tracing technique to suggest areas to apply modifications. The test features were applied manually. The experiments are summarized in Table 5.6.

Table 5.7 shows the overhead of each modification compared to the hierarchical Viper with reset. The first three columns are reported by Compass AsicSyn, and the last four are taken directly from the *bench* version of the

Table 5.7: Overhead of Modified Circuits

	Compass			Bench file			
	modules	eq. gates	size	inputs	outputs	gates	signals
basis	3	5727	44067	33	53	4087	4120
viper hr	1.00	1.00	1.00	1.00	1.00	1.00	1.00
viper hr 1	1.00	1.00	0.99	1.00	1.00	1.01	1.01
viper hr 2	22.33	1.04	1.03	1.03	1.00	1.06	1.06
viper hr 3	8.00	1.02	1.02	1.00	1.02	1.01	1.01
viper hr 4	33.33	1.06	1.05	1.06	1.00	1.09	1.09

Table 5.8: CRIS performance on modified Viper designs

Name	tries	tests	faults	det.	red.	cov.	eff.
viper h	*CRIS unsuccessful*						
viper hr	8000	3211	10045	7902	3	78.7	78.7
viper hr 1	8000	3725	10025	8136	10	81.2	81.3
viper hr 2	8000	3262	10495	8788	14	83.7	83.7
viper hr 3	8000	2798	10151	8472	30	83.5	83.8
viper hr 4	8000	3730	10757	9726	3	90.4	90.4

Table 5.9: In-circuit module coverages for modified Viper designs

Name	ALU	REGS	CONTROL	Overall
goal	90.70	92.4	92.6	90.5
viper hr	63.2	88.7	87.7	78.7
viper hr 1	67.8	90.1	87.9	81.2
viper hr 2	75.2	88.1	88.1	83.9
viper hr 3	75.7	88.4	88.9	83.5
viper hr 4	82.2	95.2	90.1	90.4

circuit (by the *bench_count* program). The column labeled "modules" shows how many submodules are included in the design. The three modules in all designs are ALU, Registers, and Control; those with more modules indicate that logic was added to the top level during synthesis. This does not correspond to the size of the circuit; it simply indicates how many modules there are to which faults can be mapped. The "equivalent gates" and "size" columns show the number of equivalent gates and the total width of the synthesized logic reported by Compass. The "inputs" and "outputs" columns show the total number of primary inputs and outputs in the bench file. Under "gates" is listed the total number of gates in the bench file; this is different than that reported by Compass because the bench format supports a different set of gate primitives than does Compass. In some cases, Compass primitives are replaced by several gates in the bench format. Finally the "signals" column in Table 5.7 indicates the total number of signals in the bench file, to give an idea of the number of possible fault locations (in general there will be many more because fanout lines are considered individually). Notice that the overhead of the testability features added was never over a few percent for the circuit size or for the numbers of inputs or outputs.

Table 5.8 gives the performance of CRIS on each of the trial circuits. Note that CRIS had a great deal of trouble with *viper h*. Once a reset line was added, CRIS could handle the design. The rest of the designs all include the reset line. See Section 2.3 for a description of each parameter. Notice that, in general, the fault coverage and efficiency increased with each experiment. The number of tests for each circuit is between 3000 and 4000. The identification of redundant faults is not improved by the addition of the testability features (CRIS generally has a difficult time identifying redundancies). The number of faults in each circuit is not changed significantly by the behavioral modifications. The more involved modifications introduced more fault locations to the circuit, but tracing the undetected faults to their respective modules has shown that the faults introduced by the modifications were all detected in each design.

All of the modifications were performed to make the ALU more accessible to test generation. When taken by itself, CRIS achieves 90.7% fault coverage on the ALU. Table 5.9 shows the best standalone fault coverages attained for each module in the row labeled "goal." These coverages are combined according to each module's fraction of the faults of the whole design to obtain a goal of 90.5% fault coverage for the overall Viper. The remaining rows indicate the actual in-circuit coverages of each module for the experiments. In general, the fault coverage for the entire circuit increased with each experiment in response to the increased accessibility of the ALU. Finally in the *viper hr 4* experiment, a fault coverage of 90.4% was achieved, very close to the indicated goal. Note that in this case, the coverage for the REGS module was actually higher than the ideal value shown; this is due to the fact that CRIS is a nondeterministic test generator, and the coverages given are not necessarily the best possible for each case. In general several runs of the test generator are required to obtain a good test set.

5.4.2 Interactive BMFT Application

Results in the previous section were obtained by modifying the design manually after performing testability analysis in the form of modular fault mapping. It is also important to investigate the effects of modifications made

Table 5.10: Interactive BMFT Experiments

Experiment	Description
Unmodified Viper	for reference
Observe REG3	REG3 made observable
Observe ALUOUT	ALUOUT made observable
Observe Top	Top-level (local) signals made observable
Observe All	All local signals made observable
C/O ALU	ALU made controllable and observable
ALU Parity	Observation of ALUOUT parity

Table 5.11: Overheads of Viper Interactive Modifications

Experiment	Width	Inputs	Outputs	Gates	Signals
basis	43385.60	33.00	53.00	4121.00	4154.00
unmodified	1.00	1.00	1.00	1.00	1.00
unmod – hitec	1.00	1.00	1.00	1.11	1.11
OBSERVE REG3	1.00	1.00	1.60	1.00	1.00
OBSERVE ALUOUT	1.00	1.00	1.60	1.00	1.00
OBSERVE top	1.00	1.00	3.74	1.00	1.00
OBSERVE all	1.00	1.00	7.83	1.00	1.00
COMBINED alu	1.04	3.21	1.64	1.07	1.09
XOR OBS alu1	1.00	1.00	1.02	0.98	0.98

interactively, as discussed in Section 5.3.2. The circuits used in this section were modified without strict testability evaluation; in the next section results are presented for interactive modifications made according to suggestions from signal-based fault mapping.

Interactive behavioral modifications were applied to the Viper microprocessor. Table 5.10 summarizes these experiments. In each of these experiments, primary inputs and/or outputs were added to the design to make the indicated signals controllable or observable. When REG3 was made observable,

Table 5.12: ATPG results on interactively-modified Viper

Experiment	F. Coverage	Eff.	# Red.	# Tests
NO MOD	81.29	81.32	3	4027
Observe Reg3	84.14	84.17	3	3361
Observe Aluout	84.33	84.36	3	3543
Observe top	88.78	88.79	1	3711
Observe all	89.66	89.67	1	3550
Combined ALU	89.40	89.43	3	2168
XOR OBS alu 1	86.17	86.19	2	3438

for example, 32 primary outputs were added to the design. In experiment "C/O ALU", both of the data inputs and all the control inputs to the ALU are made directly controllable from the primary inputs. This added 73 primary inputs to the Viper, which unmodified has only 33 inputs. This is quite obviously not the practical way to improve the testability of the design, but it serves to show the applicability of the interactive-modification software. Table 5.11 shows the overhead of each modification with respect to the unmodified Viper. All the circuits had 251 D-flip-flops. Notice that, apart from inputs and outputs, these modifications have only a very small impact on the actual area of the design. In practice, these modifications could be made without adding as many pins by making them available via multiplexers on the inputs and outputs or by test instructions. Other techniques, such as compression using XOR-trees or MISRs would also help keep the pin counts lower at small expenses in area.

Without exception, these modifications to the Viper improved the testability of the design. Table 5.12 shows the performance of CRIS on each of these modified circuits. The best performance was achieved when all local signals in the design were made observable. Targeting the ALU alone had similar

Table 5.13: CRIS performance on Modified Viper with Signal-Mapping Suggested Modifications

Experiment	Signals Observed	F. Coverage	Eff.	# Red.	# Tests
Sugg obs 1	aluout, reg2, reg3	84.71	84.74	3	3069
Sugg obs 2	aluout, reg1, reg0	82.75	82.77	3	2968
Sugg obs 3	reg1, reg0, m	81.97	82.17	21	3300

performance with a much smaller impact in the number of inputs and outputs. Each of the results in Table 5.12 shows the best of three trials with CRIS. Notice that these results are not as good as those reported in Section 5.4.1. This is because the test features were applied directly to individual signals without the use of testability evaluation.

5.4.3 Signal Fault Mapping Results

In a final experiment, the signal fault mapping technique was used to guide the selection of signals for observation in the Viper. The goal was to show the usefulness of the signal mapping technique. Since the interactive modifications available are also based on controlling and observing individual signals, this is the technique used to add testability features to the design.

The Viper was modified in three different ways. First, the suggestions obtained from signal fault mapping were used. Table 5.4 shows that the three signals downstream of the most faults are ALUOUT, REG3 and REG2. These three signals were made observable in experiment "Suggested Observe 1." For comparison, two other sets of three signals were chosen and made observable in "Suggested Observe 2" and "Suggested Observe 3." The overhead for all three cases is the addition of 96 output pins, corresponding to three 32-bit signals each being observed. Table 5.13 shows the performance of CRIS on each of the designs, in the best of three runs each. CRIS was able to achieve a slightly higher fault coverage and efficiency on the design for which the modifications were suggested by the signal fault mapping technique.

5.5 BMFT Summary

This chapter has presented a methodology for improving the testability of a design by making modifications at the behavioral level. The proposed design cycle involves evaluating the testability of the design and using this information to suggest the locations in the circuit to add test features to improve testability.

Testability evaluation is done ideally at the behavioral level, and there are tools such as BETA and ATKET that can provide this kind of information. If these tools are not available, one way to perform testability evaluation is to synthesize the design to the gate level and perform ATPG, then map the undetected faults back to the behavioral level. The fastest available methods should be used for this since it is done early in the design cycle and may be necessary several times. For this reason nondetermistic test generation was used for these results. Two methods for mapping undetected faults to the behavioral level, modular fault mapping and signal fault mapping, were presented. There are tradeoffs between the two methods: modular fault mapping has a resolution limited to the size of the modules in the design, and signal fault mapping does not guarantee that all faults will be mapped to the behavioral level. Nevertheless, both methods were shown to be useful in indicating the areas in the design where testability was low.

Once these areas are found, the design is modified to improve the testability. Several kinds of behavioral modifications were discussed. Some of these testability features can be applied with the assistance of interactive software developed for this purpose. Finally, the effectiveness of these modifications were shown as they were applied to an example circuit, the Viper microprocessor. Modular fault mapping coupled with manual design modifications achieved at best 90.4% coverage and efficiency with a slight overhead in area and output pins. The use of interactive modifications in the *OBSERVE all* experiment achieved 89.7% coverage and efficiency, with no area overhead but significant I/O pin overhead. The similarity of these results shows that the techniques are comparable.

Determining which techniques to use is an important step. Modular fault mapping is useful when there are several behavioral modules to which to map undetected faults. It is also useful for considering complete modules in making behavioral changes. Signal fault mapping is useful when there are many behavioral signals in the design for the purpose of mapping undetected faults. Modifications can be made manually in order to optimize the effectiveness of the testability features added. They can also be added interactively to speed the process and to help decrease the probability of introducing errors in the design.

The use of behavioral modifications, guided by the evaluation of the testability of the design, is a useful design tool. It can be used early in the design cycle so that testability can be considered from the behavioral level onward, rather than just at the gate level. In this way testable circuits can be designed more easily than with other methods. The behavioral description of the design remains faithful to the operation of the circuit in test mode as well as in normal operation, making it a useful description to keep even after the transition to the gate level. The addition of testability features at the behavioral level is a useful design technique.

Chapter 6

Test Instructions and Programmable Built-In Self-Test

Future work involving behavioral changes will build on the techniques described in the previous chapter. One way to improve the testability of a microprocessor or similar design is to add test support instructions to the instruction set. These instructions can perform operations that make testing the design easier, such as providing accessibility for important internal signals.

A related concept is to provide a means of built-in self-test (BIST) for the design. Instructions can be added to the instruction set to assist in generating pseudorandom tests for and analyzing responses from the functional units in the design. These instructions can then be combined into a test program to let the processor test itself. One of the advantages of BIST is that it requires no automatic test equipment (ATE): the design can indicate if it passes its test or not. This also means that the circuit can test itself even after being included in a larger system.

6.1 Adding Test Instructions

An instruction represents part of the functionality of a processor. For each operation specified by an instruction, there is a set of functional units in the design that are used to perform the operation. Thus the addition of extra instructions to the design for the purpose of testing can correspond to the addition of extra functionality in the design. Test instructions simply specify a method of accessing the various testability features added to the design.

One important advantage of using test instructions is that they eliminate the need to add extra inputs and/or outputs for each additional testability feature. The overhead of most of the behavioral modifications described in Chapter 5 was primarily in the addition of pins to the design. The tradeoff in using instructions rather than I/O pins is that all the test features will not be available simultaneously, which can increase testing time (this is similar to the tradeoff in using multiplexers to share pins for normal use and control and observation points in standard BIST techniques [1]).

Test instructions should be forbidden during normal operation. A test mode can be added to the processor that enables these instructions, similar to the use of supervisor mode for operating system functions. The test instructions can even be left undocumented, so that different versions of the same processor can have different test features. The importance of test instructions is that they provide for the testability of the current design by a specialized test program.

Given the design of a processor, it is important to be able to determine what kind of instructions will be useful for increasing testability. One technique is to perform testability analysis as described in Section 5.2 and add instructions that provide signal accessibility according to these suggestions. Another important function of test instructions is to assist in test pattern generation and signature analysis used in standard BIST techniques. These techniques are discussed in the following sections.

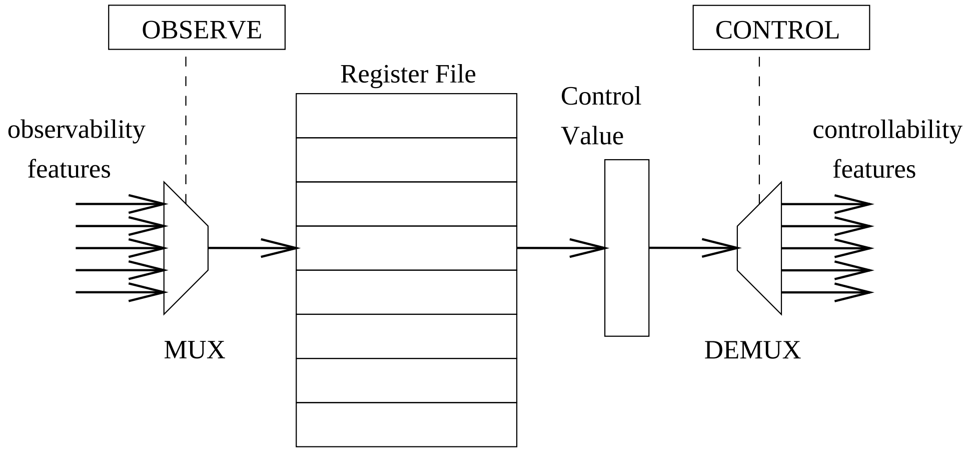

Figure 6.1: Design for signal accessibility by test instruction

6.2 Instructions for Signal Accessibility

Test instructions can be used to perform operations suggested by testability analysis, such as making internal signals easily accessible. For example, registers that are not directly accessible to the programmer can be made available during test mode. Signals that are targeted for the addition of testability features can be observed and controlled by means of instruction execution. In fact, any of the testability features discussed in Chapter 5 can be made accessible via test instructions rather than the primary inputs and outputs. This puts them under program control.

Observation of signals can be performed with the use of an instruction that specifies which signal to observe and, optionally, the destination register for the current value. Similarly, signals can be controlled by issuing an instruction that selects the control point and the register containing the value to use. Figure 6.1 shows the extra hardware required to support these operations. The test instructions are used to specify which test feature to use by setting the OBSERVE and CONTROL registers. For observability, the value of the

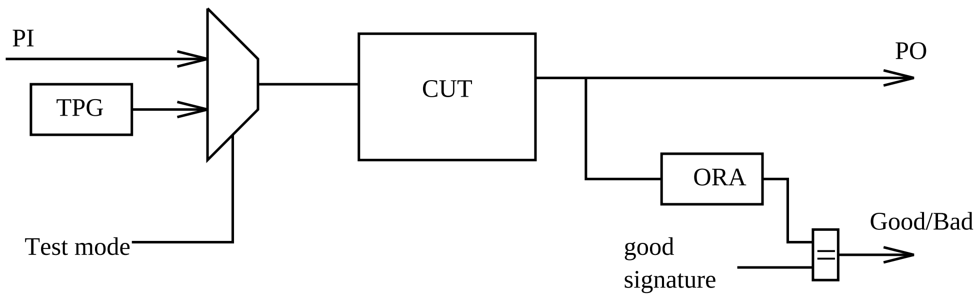

Figure 6.2: Example of Gate-Level BIST

specified signal is written into a register. For controllability, the value in one of the registers is written into the *Control Value* register, which drives the signal specified by the CONTROL register.

6.3 Instructions for Pattern Generation and Response Analysis

In general, built-in self-test involves adding specialized circuitry to the gate-level design of the circuit to apply tests and to evaluate the output response [1]. Figure 6.2 shows this type of BIST organization. The Test Pattern Generator (TPG) is a circuit that generates pseudorandom input vectors that are applied to the circuit. The response of the circuit is compressed by the Output Response Analyzer (ORA), and its output is used to indicate if the circuit contains any faults. The TPGs are often realized with linear feedback shift registers (LFSR), and ORAs with multiple input signature registers (MISRs).

Depending on the design of the processor, it may be possible to perform the TPG and ORA operations without extensively modifying the design. Rather than adding extra circuitry for test pattern generation and signature analysis, the functional units already present in the design can be used to

Enter Test Mode
Initialize each TPG and ORA register
For each functional unit FU_i
    For each operation OP_j of FU_i
        $ORA \Leftarrow OP_j(TPG)$
        Repeat for many different patterns
Compare ORA signatures to expected values

Figure 6.3: Example of a Self-Test Program using Test Instructions

perform these functions. For example, pseudorandom test patterns can be generated by an ALU in conjunction with a register in the register file. This same technique can also be used for signature analysis. If there are enough registers, it is possible to maintain separate signatures concurrently, for example to keep separate the signatures of tests applied to distinct modules.

The TPG and ORA circuits can also be directly added to a design and controlled by test instructions to make them available for use in a test program. This speeds the application of tests and the computation of signatures because these operations are performed automatically in the hardware rather than by sequences of instructions. During test mode, some of the general-purpose registers in the design can be logically replaced by TPGs and ORAs. Then functional units in the processor can be exercised using the TPG registers as operands and the ORA registers as destinations. Each time a TPG register is read, it is automatically updated to the next test pattern. When a value is written to an ORA register, the current signature is updated using that value. After executing many operations, the value stored in the ORA can be compared to the expected value. If the value stored is incorrect, the functional unit is then known to be faulty. Figure 6.3 shows an example of a program to test each functional unit using these instructions.

Another use of the TPG and ORA registers is to combine them with the control and observe test instructions. If a signal is being observed by a response analyzing register, the signature computed can give more information than a single instantaneous observation. Similarly, a TPG can provide more than instantaneous control of a given signal. Combining these two techniques provides more support for testing.

It is not always possible to take the output of a TPG and use it as an input to another arbitrary module in the circuit. The testing program will need to be able to present some of these values directly. For example, to test the control logic of the processor it is necessary to execute arbitrary instructions. If it is not possible to create a program that contains all the necessary combinations of instructions, it may still be possible to have the program generate them and proceed to execute them as instructions. In this way even the control circuitry of the design can be tested.

If the test program cannot apply all the necessary tests to the control circuitry, a combined technique may be necessary. Some behavioral DFT changes can be applied to the control logic, and the test program can be used to test the rest of the circuit. In this way the modifications to the design are localized and only made where absolutely necessary.

6.4 Programming for Self-Test

After the addition of test features and instructions to a design, a program can be written to test the design using these instructions. Conditions for optimally using each testability feature should be determined and exploited.

Depending on the set of test instructions added different programming methods should be used. If only instructions for TPG and ORA were added, then the algorithm in Figure 6.3 can be used. If no test instructions were added, the same algorithm can be used once it is determined what sort of TPG and ORA operations can be performed using the available instructions. The TPG and ORA operations are simply replaced by the appropriate sequences of instructions.

Another important consideration for self-testing is determining what seeds to use for TPGs and ORAs. This depends heavily on the design of the special registers or the algorithms with which they are implemented. TPGs need to generate sequences that are useful in detecting faults in the unit or units under test. ORAs need to compute signatures that will not alias fault effects that come to them. In order to avoid any problems with aliasing, it is possible to compare the signatures at several times during test application, rather than waiting until the testing is complete.

With the availability of test instructions for signal accessibility, ATPG algorithms might be able to determine good ways to use them. In fact, a modified ATPG algorithm might be able to suggest testability features to add to the design and also provide a sequence of instructions for using these features appropriately.

Once a test program has been written, testing the processor is a simple matter of running the program. Small test programs could be loaded into the processor's cache and executed directly from there. This can be accomplished by factory testing equipment (which could broadcast the program to several processors simultaneously). It also could be done once the processor is included in a computer system, either by dedicated hardware containing the code in ROM or by the operating system itself. True built-in self test could be achieved by including the test program in a small ROM on the processor chip itself. Testing could be performed without relying on any external components by running the test program directly from this ROM.

6.5 Summary

The addition of instructions to a processor design allows programs to be written to perform highly effective tests of the design. These test instructions can be used to observe and control important internal signals. They can also be used to control specialized hardware added to the design for the purpose of test pattern generation and creating response signatures for tests applied to various functional units. It is even possible to have the processor perform built-in self-test if the test program is small enough to fit on-chip or in the local cache. The programmed testing technique will be useful for testing processors just after manufacture and also once they are incorporated into a computer system.

Chapter 7

Future Work

7.1 Combining ATPG and DFT Techniques

One thing that is evident from the results presented in the preceding chapters is that the use of each technique by itself cannot guarantee the best possible improvements in testability. It is important to be able to combine the techniques so that they can work together. Assisting ATPG with formally-generated initial sequences or instruction templates can easily take advantage of any testability features added to the design: a new sequence or template can be made explicitly to use the added feature. Even the addition of test instructions to the design can be described to the test generator in this way.

Another important way of bridging the two aspects of testability is to allow the test generator to perform modifications directly to the design. The use of *partial scan*, a capability of CRIS, is one existing example of this kind of technique. If test generation proves difficult in the vicinity of a particular register in the design, that register is made directly accessible via a scan chain. If a test generator were endowed with a larger set of behavioral testability features to add, test generation might be improved over the use of partial scan.

7.2 Using formal methods to apply tests

Implicit enumeration methods can be used to find sequences to justify and propagate arbitrary tests for functional modules. These sequences can then be used to apply separately-generated test sets for subcircuits. For example, HITEC can achieve 100% fault coverage for the Viper ALU by itself (HITEC took 410 seconds to find 938 test vectors for this coverage). The sequences to apply the tests to the ALU and propagate the outputs can be determined by implicit enumeration. If all the tests can be applied in this manner, the ALU can be removed from consideration since the standalone and in-circuit coverages will then be identical. It is possible that the in-circuit constraints on the ALU will prevent certain of the tests from being realized in this manner. Still, the tests that can be applied will immediately eliminate many of the faults in the ALU from further consideration.

Chapter 8

Conclusions

Behavioral considerations for testability should be an important part of the VLSI design cycle. Much of the previous work on testability has considered the gate level description only or intermediate levels of description that are not easy for the designer to work with. Large designs that cannot be considered completely at the gate level for testability issues often can be handled at the behavioral level. The techniques presented in this thesis should be useful in designing large VLSI circuits that need to be easy to test.

Improving the test generation process is an important method of increasing the testability of a design. One way to do this is to encode behavioral information for the test generator to use in searching the input space for good tests. This is effectively what is done when initial seed sequences or sets of instruction templates are given to CRIS. Both of these abilities allow CRIS to generate tests in terms of input sequences that are meaningful at the behavioral level. Input sequences and instruction templates can be generated using the formal methods of implicit enumeration methods.

Another way to improve testability is to modify the design itself to have better controllability and observability of internal signals. Extensions to Behavioral Modifications for Testability were presented in Chapter 5. The addition of testability features was presented in the context of a design cycle that involves targeting areas of the design for modification. Behavioral modifications are done interactively using software developed for that purpose.

One of the testability features, the addition of test instructions, is useful for making designs that can be tested by running a special program. These test programs can take advantage of test features that have been added to the design. It is also possible to use this technique in a limited fashion for unmodified designs. Unexplored issues in this area are determining appropriate kinds of test instructions to support and the synthesis or compilation of test programs using them. Built-in self test may be possible if the program can be kept small enough to be stored on-chip or in the processor cache.

It is important to combine the techniques for assisting test generation and improving hardware testability. The increasing capabilities of formal methods can be used to perform much of the work of encoding behavioral information for use by test generators or in behavioral-DFT techniques.

VLSI test generation and design for testability are separate but related issues in making designs that can be tested easily and well. It is possible to use high-level design information, such as the behavioral description of a design, to improve test generation on the original circuit and to make design changes to simplify testing. Some techniques for both areas have been presented and discussed. There is much more work to be done in this vein to give good testability solutions for modern processors and similar circuits.

BIBLIOGRAPHY

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[2] Compass Design Automation. *VHDL for the ASIC Synthesizer*, 1994. online documentation.

[3] Chung-Hsing Chen, Tanay Karnik, and Daniel G. Saab. Structural and behavioral synthesis for testability techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(6):777–785, June 1994.

[4] Chung-Hsing Chen, Chienwen Wu, and Daniel G. Saab. BETA: Behavioral testability analysis. In *ICCAD*, pages 202–205, 1991.

[5] Vivek Chickermane, Elizabeth M. Rudnick, Prithviraj Banerjee, and Janak H. Patel. Non-scan design-for-testability techniques for sequential circuits. In *30th ACM/IEEE Design Automation Conference*, 1993.

[6] Hyunwoo Cho, Gary D. Hachtel, and Fabio Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, July 1993.

[7] W. J. Cullyer. *VLSI Specification, Verification and Synthesis*, chapter 1. Kluwer Academic Publishers, Boston, 1988.

[8] Kenneth A. De Jong. Genetic algorithms are not function optimizers. In *Foundations of Genetic Algorithms 2*. Morgan Kaufmann Publishers, 1993.

[9] Dinos Moundanos. Personal communication.

[10] T. M. Niermann and J. H. Patel. Hitec: A test generation package for sequential circuits. *Proc. Euro. Des. Auto. Conf.*, 1991.

[11] Douglas L. Perry. *VHDL*. McGraw-Hill, Inc., second edition, 1993.

[12] Daniel G. Saab. Personal communication.

[13] Daniel G. Saab, Youssef G. Saab, and Jacob A. Abraham. CRIS: A test cultivation program for sequential VLSI circuits. In *IEEE/ACM International Conference on Computer-Aided Design*, 1992.

[14] Eliezer Sternheim, Rajvir Singh, and Yatin Trivedi. *Digital Design with Verilog HDL*. Automata Publishing Company, 1990.

[15] Stephen A. Szygenda. Digital systems simulation. class notes, 1994.

[16] Thomas Thomas. Behavioral modifications for testability. Master's thesis, The University of Texas at Austin, 1993.

[17] Thomas Thomas and Jacob A. Abraham. Synthesis of testable systems. SRC Research Review presentation, May 1994.

[18] Thomas Thomas, Praveen Vishakantaiah, and Jacob A. Abraham. Impact of behavioral modifications for testability. In *12th IEEE VLSI Test Symposium*, April 1994.

[19] Praveen Vishakantaiah, Jacob Abraham, and Magdy Abadir. Automatic test knowledge extraction from VHDL (ATKET). *Proceedings of the 29th IEEE/ACM Design Automation Conference*, pages 273–278, June 1992.

[20] Praveen Vishakantaiah and Jacob A. Abraham. High level testability analysis using VHDL descriptions. *Proceedings of the European Design Automation Conference*, pages 170–174, February 1993.

VITA

Scott Lee Crutchfield was born in Ruidoso, New Mexico on May 29, 1971, the son of Charles N. Crutchfield and Karen Jenkins Crutchfield. He graduated from Permian High School in Odessa, Texas in 1989, and went on to attend Duke University in Durham, North Carolina. During the summer of 1990 he worked for Analysis and Technology in Crystal City, Virginia. During the summers of 1992 and 1993 he worked for Research Triangle Institute in Research Triangle Park, North Carolina. In May, 1993, he graduated from Duke with a Bachelor of Science in Engineering degree. In the fall of 1993 he entered The Graduate School at The University of Texas at Austin.

Permanent address: 3500 Greystone Dr.
Apartment 205
Austin, Texas 78731

This thesis was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.