Functional Testing of ASICs Designed with Hardware Description Languages

by

Richard Davis

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

© Richard Davis, MCMXCV. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute copies
of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 1995

Certified by ....
G. Andrew Boughton
Research Associate, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 10 1995

Barker Eng

LIBRARIES

Functional Testing of ASICs Designed with Hardware Description Languages

by

Richard Davis

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1995, in partial fulfillment of the
requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Functional testing is a part of the VLSI design process for which there is no standard approach. Some research suggests that a method which integrates directed testing and random testing has the best chance of providing a bug-free design quickly. The functional testing system for the Arctic router chip uses this method and is designed with a very structured approach to shorten testing time further. This approach is comprised of the following three methods. Verilog is used to implement both the testing system and the Arctic chip itself. Signals are generated and recorded during test simulations with Verilog modules that connect to each functionally separate set of Arctic's pins. Finally, all tests have configuration, execution, and checking phases to force the users of the system to think about testing in a structured way. The result of this structured approach is a very fast and flexible testing system.

Thesis Supervisor: G. Andrew Boughton
Title: Research Associate, Department of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank John Kubiatowicz and David Chaiken for giving me helpful pointers to other research on functional testing. Thanks also go to Professor Arvind for supporting this work through the Arctic project. Many, many thanks go to the brave men and women of the Arctic team, including Tom Durgavich, Doug Faust, Jack Costanza, and Ralph Tiberio who were the main engineers on the project, and the undergraduates Thomas Deng, Wing Chi Leung, and Elth Ogston who helped me implement this testing system, and Yuval Koren who used it to test statistics. Most of all, I would like to thank the omniscient and omnipresent Andy Boughton who led the Arctic team, provided all the advice, support, and feedback I could ever have asked for, and taught me that a boss can be a friend, too.

Contents

1 Introduction **11**

1.1 The ASIC Design Process 12

1.2 The Shape of a Functional Testing System 13

1.3 The Approach to Testing Arctic 15

1.4 Overview of This Thesis 16

2 Background **19**

2.1 Abstract Testing Methods 19

2.2 Practical Testing Methods 21

3 Goals for the Arctic Testing Project **25**

3.1 General 27

3.2 Easy To Use 28

3.3 Fast 28

3.4 Repeatable 29

3.5 Randomizable 29

3.6 Low Level 29

4 Implementation of the Testing System **31**

4.1 Hardware 32

4.2 Software 34

4.2.1 Configure Phase 35

4.2.2 Execute Phase 35

4.2.3 Check Phase 38
4.3 Quick Mode 38
4.4 Running the System 40

5 The Randomized System 43
5.1 Hardware 44
5.2 Software 47

6 Evaluation of the Arctic Testing Project 51
6.1 General 51
6.2 Fast 52
6.3 Randomizable 52
6.4 Repeatable 53
6.5 Low Level 53
6.6 Easy to Use 54

7 Conclusions 57
7.1 Structure Is Important 57
7.2 Speed Is Necessary 58
7.3 Complexity Is The Enemy 59
7.4 Future Work 60

A Example Test Groups From Arctic's Functional Testing System 61
A.1 Routing 61
A.2 Statistics 66
A.3 Errors 68
A.4 JTAG 70
A.5 Randomly Inserted Packets 72
A.6 Packet Ordering 73

B The User's Manual for Arctic's Functional Testing System 77
B.1 Directories 78

B.2 General Structure 78
B.3 Running The System 84
B.4 Data Structures 85
B.5 Files 86
B.5.1 Packet Library 86
B.5.2 Packet List File 89
B.5.3 Packets Log File and Packets Check File 90
B.5.4 Stats Log File and Stats Check File 92
B.5.5 Errs&Control Log File and Errs&Control Check File 94
B.5.6 Configure File 96

List of Figures

3-1 The Arctic Router 26

4-1 The Arctic Functional Testing System's "Hardware" 32

4-2 Master Simulation File 41

A-1 `test003.config` file 62

A-2 `test003.code` file 63

A-3 `test003.a.pkts` file 64

A-4 `test003.chk.pkts` file 64

A-5 `test003.chk.stats` file 65

A-6 `test003.chk.errs` file 65

A-7 `test060.code` file 66

A-8 `test060.chk.stats` file 67

A-9 `test052.code` file 69

A-10 `test052.chk.errs` file 70

A-11 `test032.code` file 73

A-12 `test030.chk.pkts` file 75

Chapter 1

Introduction

The current trend in electronic systems design is to push as much functionality as possible into integrated circuits. Newer systems are generally much smaller and much faster than their predecessors, but they also have larger and more complex integrated circuits. These circuits are often called Application Specific Integrated Circuits (ASICs) because they are specific to their own systems. Over the past few years, the number of ASICs fabricated has greatly increased, and the size and complexity of those ASICs has also increased. In order for these chips to remain cost-effective, however, design time must remain fairly short, and this has proven to be a difficult problem.

How can ASIC designers go about shortening design time? Often, designers use Hardware Description Languages (HDLs) such as Verilog and VHDL to design their chips, and synthesis tools such as Synopsis to compile those designs down to gate descriptions, shortening design time. Scannable design methodologies together with Automatic Test Pattern Generators can shorten design time by automatically generating manufacturing tests for integrated circuits. Another part of the design process that could benefit from some automation is functional testing, sometimes called design verification. This part of the process ensures that a design performs its required functions before it is sent to be fabricated. This is at least as important as any other part of the design process, but it is an area that has received little attention. This thesis analyzes this part of the ASIC design process, and presents some experiences

with a testing system that may shed some light on the functional testing problem.

1.1 The ASIC Design Process

Let's begin by looking at the ASIC design process, and seeing how functional testing fits in. The term "ASIC" is a rather loose one, and generally refers to low-volume microchips, or chips that are designed for specific systems and are unlikely to find their way into other systems that the designers did not envision. Usually, designers will try to keep these chips simple, to reduce design time, but it is possible to make them as large and as complex as any integrated circuit on the market. Therefore, all three major methods used to design integrated circuits, full custom design, standard cell design, and gate array design, are used to design ASICs.

Full custom designs are the most low-level, requiring the designers to specify the exact location of every wire and transistor. Standard cell designs are a bit simpler; the designer is given a library of fairly simple logic elements and allowed to assemble them in any way. The gate array is the simplest design method. A Gate Array designer simply needs to supply a chip fabricator with a description of the chip in terms of logic gates and state elements, and the fabricator will handle the placement of those gates and the routing of all connecting wires. I will focus on the gate array design process because it is the simplest and because it is the method used to design the chip discussed in this thesis. However, the ideas presented here are general enough to be useful with any design method.

Since designing a Gate Array is as simple as coming up with a gate description of the chip, HDLs and synthesis tools are very popular among gate array designers. A Hardware Description Language provides an easy way to specify the behavior of the chip and provides an environment for simulating this behavioral model. The synthesis tools can turn this model into a gate-level description, and often provide ways to simulate that description as well, thus completing most of the design work.

After choosing a basic design methodology and gathering tools such as HDLs to help in the design process, one last major problem needs to be addressed; how should

the chip be tested to ensure that the final product performs the desired functions? Designers ensure that their design is correct with functional tests, many of which are performed before chip fabrication since re-fabrication is costly. Fabricators ensure that their chips are free from manufacturing defects by requiring that the designers come up with some kind of manufacturing tests.

Manufacturing Tests are usually generated in some automated fashion. If the chip has scannable state elements, then it is relatively straightforward to write some kind of Automatic Test Pattern Generator (ATPG) that can generate tests which will detect nearly all common manufacturing defects. Some ATPGs are even available commercially, removing the burden of writing this tool from the designer. If the chip does not have scannable state elements, then some kind of ATPG can still be used. Such generators often create functional tests that are capable of detecting a certain percentage of possible manufacturing defects.

This brings us finally to the issue of functional testing. Every one of the above steps in the ASIC design process has some clear methods or tools associated with it, but there is no clear functional testing method a designer can use to verify that his or her design is correct. With no tools or conventions to help designers along, they must solve this problem from the very beginning each time they begin a new design. This can be a considerable burden when a team is trying to finish a design quickly, because functional testing is very time consuming, and it is very hard to do right. It is very important, however, because every bug that is not discovered in the functional testing phase must be found after fabrication or once the chip is already in a system, at which time fixes are much, much more costly.

1.2 The Shape of a Functional Testing System

Unfortunately, functional testing can be a very daunting task because it needs to accomplish a very large number of things at once. At the highest level, we can say that functional tests should perform design verification, that is, verify that the design fits all specifications. This implies that functional testing should begin before

the design is completely correct, and functional testing is therefore, in many cases, debugging. Debugging a system is, in itself, a very haphazard and confusing task, usually accomplished by simulating the system with some set of inputs and monitoring the system's behavior to determine where a problem occurs. Let us assume that debugging *and* functional testing of ASICs are accomplished with such a system, and look at the form this system should take.

Because the system should perform design verification, it needs to work like a kind of virtual test fixture. A design for the chip is given, a number of tests are run, and the system should say, "Yes, it's working," or "No, it's not working." This implies two things. First, the testing system must be general; that is, it must be capable of exercising every function of the chip and putting the chip in nearly every possible state. Without this condition, it would be impossible to determine whether or not the design worked in every situation. Secondly, tests should be repeatable, so that it will be possible to determine when something that was broken begins to work.

There are two other requirements of a functional testing system that are less intuitive. One requirement is that the functional testing system should work at the lowest level of the design. In other words, the system should work with the actual design and not with some simulation for that design. This requirement is not so easily satisfied if a chip is being designed on paper of if no good simulation tools exist for the actual description of the chip. Some chip designers will write simulations for the behavior of their chips in some higher-level language like Lisp or C, but there is no obvious way to determine whether or not the simulation for the chip will behave exactly as the final design will, and it is therefore necessary to run functional tests on the lowest level description to ensure the proper behavior of the final design.

The other less intuitive requirement of functional testing is that it should be somewhat automated. Automating the functional testing system will give the system itself some of the responsibility for testing the chip. It is necessary to give some of this responsibility to the testing system since it is nearly impossible for a human to think of all the tests needed to determine whether or not a chip is functioning properly. Without some form of automation, the user is left trying to think of every possible

input pattern that could result in an erroneous output. The easiest way to automate test selection is to build some kind of random input pattern generators into a testing system. This method, which I will refer to as "randomization," can make the burden of functional testing significantly lighter.

Two final and very important requirements for this system are that it should be very fast and very easy to use. Since it is known that this system will be used as a debugger, it should obviously be fast enough to re-run tests quickly and simple enough that it will not hinder someone puzzling on a design problem. Nothing is more maddening than waiting hours for the results of a simulation that will tell whether or not a simple bug fix worked. Slow tools can be quite an annoyance, and can significantly increase the time needed to create a bug-free design. The same argument applies to the ease of use of those tools. Any design tools must be easy to use or they will be very little help to the design team. Unfortunately, however, building speed and ease of use into a system with the diverse requirements listed above is very difficult.

1.3 The Approach to Testing Arctic

As an example of the possible tradeoffs in a functional testing system, this thesis presents the approach used to test the Arctic router chip. Arctic is a 4x4 packet routing chip that has been designed to be a part of the *T multiprocessor, a parallel computing project in the Computation Structures Group of the MIT Lab for Computer Science [1]. Arctic has some very complex functions and a rather confusing control scheme. This complexity makes Arctic a good test bed for a number of interesting approaches to functional testing.

There are three main ideas in the approach used to design Arctic's functional testing system. The first is the use of Verilog as an implementation language. This was the natural choice of language for the system since Arctic is a gate array and is being designed entirely in Verilog, but we shall see later that this choice gives a number of other desirable traits to the system. The use of Verilog, for example,

ensures that the lowest-level specification for the design is being simulated at all times. Also, and perhaps most importantly, it makes it easy for the testing system to change state inside the chip during simulation which can sometimes allow the system to run many times faster.

The second main idea is the use of virtual testing modules to simulate components of the system outside the chip during tests. The term *module*, here, refers to the chief method of abstraction in Verilog, and Arctic is designed as one large module. In the testing system, all components that are connected to Arctic are also modeled as different modules, one for each logically distinct entity. For example, each of Arctic's input ports and output ports is connected to a module that knows how to communicate with that port. All signals related to a port are grouped together, giving the designer of the system a clear structure to work with, and giving the user of the system a clear, simple picture of the testing system.

The last major idea used in this functional testing system relates to the organization of the tests themselves. At this point, the reader may be wondering, "Just what is a 'test' anyway?" In Arctic's functional testing system, a test consists of three parts: a specification for a beginning configuration of Arctic, a set of inputs, and a specification for the state of Arctic after all the inputs have been sent. Together, these three parts form a "test group," the basic testing unit around which the entire system is built. This may seem like common sense, indeed all three of these ideas may seem like common sense, but since there is no conventional way to design a functional testing system, we must state them clearly. This will give us a starting point for discussion about this system, and serve as a reference against which other possible designs can be judged.

1.4 Overview of This Thesis

In the following chapters, I will detail the design and implementation of Arctic's functional testing system. The experiences contained here should be useful to anyone designing a similar system, because they show the strengths and weaknesses of an

interesting set of design choices. I will begin in Chapter 2 by describing some other work that has been done in functional testing; this will show the motivations for the design of this system. Chapter 3 will give a careful description of the goals of the Arctic testing project. Chapter 4 will give an overview of the original implementation of Arctic's functional testing system, while Chapter 5 will look at the newer system that has a great deal of randomization built in. Finally, Chapter 6 will discuss how well the testing system met our original goals, and Chapter 7 will present what wisdom was gained from experience with this system. Those readers desiring further details may be interested in Appendix A, which gives many examples of tests designed for the first functional testing system, and Appendix B, which contains the user's manual for that system.

Chapter 2

Background

Before diving into all the details of Arctic's functional testing system, it will be helpful to look at some of the recent thoughts and experiences of others who have addressed this problem. The ideas presented in this chapter are relevant to any VLSI testing effort, and they will help to explain why I made some of the choices that will be presented later in this thesis. First, I will look at some abstract functional testing methods, and then I will focus on more practical methods.

2.1 Abstract Testing Methods

The purpose of testing a system is to prove that the system is "correct," and it is natural, therefore, that abstract thought on testing is concerned mostly with methods that prove the correctness of a system. Some theoreticians have created logical languages for specifying hardware and software systems with which it is possible, given a specification and a set of axioms, to prove certain properties about each hardware module or block of code. This might be thought of as a way of proving "correctness," and such methods have been used successfully with hardware systems. It is arguable, however, that these proving languages are not as useful with very large and complex systems, and even their supporters agree that this "testing" method cannot *replace* other testing methods [6]. All of the interesting theoretical work that relates to functional testing of integrated circuits focuses on randomization.

Randomized testing methods seek to automatically generate and test a subset of a hardware system's possible states with the intent of finding all design bugs with a certain *probability*. This descends from the tried and true "brute force" testing method. In "brute force," every possible state of a system is entered, and in each state it is determined whether or not the system is functioning properly. This assumes, of course, that there exists some method for determining whether or not the system is functioning properly in every state, but such methods do exist for many simple systems. The difficult problem is how to enter every possible state of a hardware system. As hardware systems continue to get more complex, the number of possible states in a typical is growing exponentially. For many modern systems, it is not possible to enter each possible state for as little as a nanosecond and expect to have entered all possible states in any one person's lifetime. This makes the brute force method impractical for most VLSI systems.

It hardly seems necessary, though, to enter *every* possible state in a hardware system. Logic is frequently replicated, and from a design standpoint, if one piece of a datapath is working, it is frequently the case that all the other pieces are working, too. In fact, the only states that *need* to be entered are those where design bugs can be detected, and, together, these states form a small subset of the possible states. Randomized testing methods generate a small set of states with a pseudo-random number generator. If the random number generator is implemented carefully, the designer can be sure that every bug in the system can be detected in at least one of the generated states. The trick, then, is to run the random number generator long enough to generate a subset of states so large that it includes every needed state. Unfortunately, it is generally impossible to determine how long is long enough. It is comforting to know, though, that the probability of detecting every bug gets higher if more tests are run. Also, since many bugs can propagate their effects over large portions of a chip, they can be detected in many different states. Chances are high, then, that one of these states will be entered early in testing if the random signal generators routinely exercise all parts of the chip. This can make random testing a very practical method, as well.

This probabilistic argument for proving correctness may seem a bit suspicious, but this method has been successfully used to test some very large and complicated chips. Wood, Gibson, and Katz generated random memory accesses to test a multiprocessor cache controller, and with it found every logical bug ever detected in the system except for two that were missed because of oversights in their testing system [3]. Clark tested a VAX implementation successfully by executing random instruction sequences on a simulation of the new implementation and cross-checking the results by executing the same instructions on the VAX that was running the simulation [2]. These are some of the examples that have proven randomization to be a powerful technique in testing hardware systems.

2.2 Practical Testing Methods

Hardware systems were being built long before any formal methods were developed to prove their correctness, and as we have noted, formal methods are not always practical for the largest and most complicated chips. How, then, is testing done in the real world? A designer that is presented with the task of performing functional testing on an chip usually has to figure this out for him or herself, but there are a few general methods that have been successful, historically. Most designers who wish to test their chips have the ability to simulate their design in some way. Let us assume that all testing projects have this in common and look at how the designers might choose the tests that are run on this simulation to show its correctness.

One way to choose tests is to attempt to list all the functions of the chip that need to be tested and try them out, one by one. This is a very satisfying method because there is a clear point when it can be said that testing is complete [2]. It is difficult to make this method work in modern designs, however. The number of cases will be huge, and it is likely that some tests that should be run will be overlooked. This method remains, however, as the naive approach to functional testing.

A slightly different approach to testing is to carry out a focused *search for bugs*. This differs from the previous method in that testers are not merely trying out every

function of a chip, but rather they are experimenting with it to see how it might break. There are several advantages to this method. Often, the tests run on the system using this testing method will be very similar to the actual inputs the system will be given when it is in regular use, giving the testers greater confidence that the system will work for its intended use. Also, the testers will be consciously trying to make the system break, and a conscious effort to find a problem with a system will probably uncover more bugs since many bugs tend to be related. Another big advantage of this method is that it has no set completion time. This is an annoying trait from the manager's point of view, but from the designer's, it emphasizes the fact that testing should stop only when no new bugs have been found for a long time, and not when some possibly incomplete list of tests has been finished. This testing method is fairly common, and is makes up at least part of the testing strategy of several successful chip projects, such as MIT's Alewife project [7]. It can be very time consuming and exhausting, however, and may not be appropriate for smaller design teams or teams with a tight schedule.

The above methods can be categorized as "directed testing," where a human is responsible for choosing the tests run on the system. We have seen before, however, that the computer can take responsibility for this task, which brings us back to the randomized testing ideas from the previous section. Most VLSI systems will not be as regular as a cache controller or a microprocessor, however, and they may not be so easily tested with randomization. With these devices, it sufficed to randomize memory accesses or instruction sequences, and this gave a set of inputs that was sufficiently random to check nearly all the functions. Other systems may have many disjoint sets of state, all of which can have a significant effect on the behavior of a system. For example, imagine a DSP chip that can process four signals at once, each one with a different function, and imagine that the four signals can even be combined to make new signals. The way a testing system should go about randomizing inputs becomes less clear.

The answer, again, is to make the system capable of randomizing every piece of state in the system, but now the programmer must be very careful that the testing sys-

tem does not inadvertently introduce some kind of pattern into the input sequences. Since all inputs are *pseudo*-random, there is always a danger of introducing a pattern and missing some important group of inputs, and the more disjoint groups of state are, the more likely that the programmer will slip up and miss something. If the four signal processors in our DSP example above were each given the same function at the same time, for example, the test would hardly be as complete as a test where the function of each processor was independent of the other three.

There is another difficulty a designer encounters when building a random testing system. How does the system determine "correct" behavior of the chip? In the directed testing methods, the user defines what the outputs of the chip should be for each test, and this problem does not arise. The random tester, however, must be able to define "correct" outputs by itself. This was simple for the cache controller and VAX implementation mentioned above, because models of correct behavior were easy to come by. In most cases, though, such a model is not so easy to find, and the only solution is to limit the range of behaviors that the system can simulate, so that the model can be made simpler. As a result, the random testing system loses the ability to detect certain kinds of bugs.

It seems, then, that each method has its own advantages and problems. A designer facing the task of functionally testing a chip design might be disappointed with the options, but some combination of the above ideas can lead to a fairly reasonable approach to testing. When designing Arctic's testing system, for example, we chose to combine a random testing strategy with a focused search for bugs. The resulting system is capable of both random and directed testing, where each approach makes up for the failings of the other. Since our design team is small, the random tester can find the subtle bugs we do not have the time to search for. Since the random tester cannot find *all* the bugs, we can search for the ones it cannot find with the directed tester. This is an approach to testing that fits well with the constraints of an ASIC design team, and it is this combination of ideas that has actually been used in the Alewife cache controller and VAX examples mentioned above [2, 7]. In the chapters that follow, we will see how this approach is used to build Arctic's testing system.

Chapter 3

Goals for the Arctic Testing Project

In the remaining parts of this thesis, I will explore the problem of functional testing by describing the functional testing system of the Arctic router chip. Arctic chips will form the fat tree network that will allow the processors in the *T multiprocessor to communicate with each other. In this chapter, I will go over the goals the design team had in mind when designing the testing system for Arctic, but before I begin, it may be helpful to give an overview of Arctic itself.

Figure 3-1 shows a block diagram of Arctic [1]. Arctic consists of four input ports connected to four output ports by a crossbar, and maintenance interface section through which Arctic is controlled. Message packets enter an input port and can exit out of any output port. Since all "links" are bidirectional in an Arctic network, input ports are paired with output ports that are connected to the same device. Packets vary in size and can be buffered upon arrival in each input port. Flow control in an Arctic network is accomplished with a sliding window protocol similar to the one used by TCP/IP. A transmitter (output port) and a receiver (input port) are both initialized with an initial number of buffers, and a receiver notifies the transmitter when a buffer is freed so that the transmitter knows to send packets only when buffers are available.

So that the system can tolerate any clock skew for incoming signals, each input

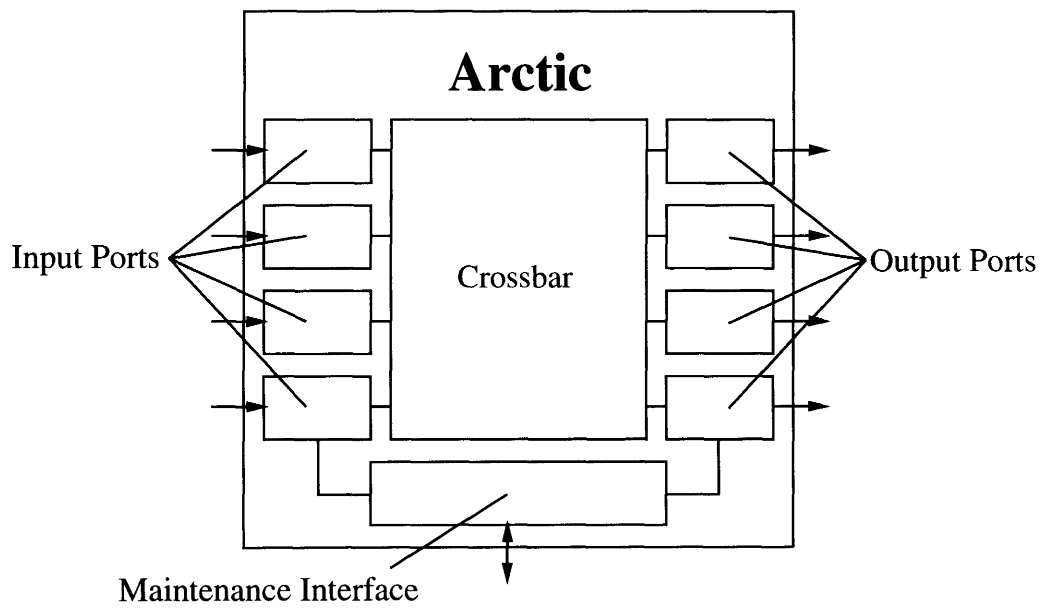

Figure 3-1: The Arctic Router

port runs on a different clock which is transmitted with the data. Data has to be synchronized into a local clock domain before it can be sent out. Also, data on the links is transmitted at 100 MHz, though the chip itself operates at 50 MHz, which causes a little more complexity. More sources of complexity are an extensive set of error checking and statistics counting functions, two levels of priority for packets, flow control functions such as block-port and flush-port, a "mostly-compliant" JTAG test interface, and manufacturing test rings that are accessible in system (not just during manufacturing tests).

Most of these details about Arctic can be ignored unless the reader wishes to dive into the examples given in Appendix A or the user's manual in Appendix B. The details are mentioned here only to give the reader an impression of Arctic's complexity. Arctic falls into that large category of ASICs that have many complex functions and for which there is no obvious way to design a functional testing system. We chose to begin by implementing a directed testing system, and approached the problem by first drawing up a set of goals as guidelines to help us with our implementation. The sections that follow list each goal we had for our system and explain why we found that goal important.

3.1 General

Because this system was to be the only system completely dedicated to testing Arctic, it seemed necessary to require that the system be general enough to test any of Arctic's functions. This meant that the testing system needed to be capable of putting Arctic in any state, or, stated another way, the testing system needed to be able to send any input signal to Arctic. With this ability, the system was guaranteed to be able to put Arctic in any state. Also, we decided that every output signal of Arctic should be recorded or monitored, so that no important behavior could be missed.

3.2 Easy To Use

Since the original time frame for the testing project was only three months, and since there were only 5 engineers and 4 students working on Arctic during that period, we decided that the testing system needed to be very easy to use, or testing would never get done in time. Three students were to implement the system, and it was to be available to anyone in the design team who had the time or the need to look for bugs. This meant that the intended users were more than just the designers of the testing system, and it would therefore have to be simple and well documented. Also, since the chip was still being developed, we knew that this system might be used as a debugging tool, and as mentioned in Section 1.2, any debugging tool has to be easy to use for it to be effective. When debugging, users need to be able to create intricate tests even if they lack experience with the system.

3.3 Fast

As with ease of use, the system had to be very fast because of the lack of time and human resources. Because Arctic was so complex, behavioral simulations were only running at about two cycles per second. We knew that the testing system would have to play some interesting tricks to boost speed or we would not be able to finish testing in time. Our hope was to make the system capable of running a short test in no more than 5 to 10 minutes, so that the user would not have to wait terribly long for the results of a simulation after a new bug fix.

An additional reason to speed up the system was the group's lack of computing resources. The members of the design team were sharing a fairly small number of workstations. We hoped to keep the load on these machines to a minimum by making simulations take as little time as possible.

3.4 Repeatable

It was also mentioned in Section 1.2 that all tests needed to be repeatable. We hoped to be able to save all the tests we generated so that we would be able to run any of them again as regression tests. Also, it was necessary for any input to be repeatable if the system was to be useful as a debugging tool. This meant that there could be no unpredictable behavior in the system. If any parts were random, the seeds for the random number generators needed to be saved so that the simulation could be run again in exactly the same way.

3.5 Randomizable

Our hope was to build random test generators into this system, but the immediate need was for a general tester and debugging tool. We knew that the random input generators might not be general enough to test any function of the chip, and we knew that debugging is impossible when all inputs are generated randomly, without any user control. We decided to build a directed testing system with the intent of adding some kind of random testing later on, since randomization seemed to be too complex a task for the first pass.

3.6 Low Level

We also saw in Section 1.2 that it is a good idea for a simulation to work with the lowest level specification of a chip design. This idea is presented well in a paper by Douglas Clark [2]. In this paper, Clark argues that all serious implementation and simulation work should be done at the gate level, and designers should not waste time designing and testing high-level models of their chips. His argument is that the latter method requires more design time since designers need to design and test separate high-level simulations in addition to low level designs. He also argues that tests on high level simulations are less accurate, lacking the subtle interactions between gates.

Arctic was being designed with Verilog and compiled to gates with Synopsis, so,

in a sense, all simulations were at a very low level. The Verilog model described every interaction between the sub-modules in full detail, and the gate description was generated automatically. We decided to follow Clark's wisdom to the letter and chose to make our system capable of simulating both the pre-compiled Verilog description *and* the compiled, gate-level description, which could be represented as Verilog code. This, we felt, would be a more rigorous test, and since we hoped to have a working chip in only three months, such a rigorous test was necessary.

In the next chapters we will see that it is nearly impossible to reach all of these goals simultaneously. The desire to make the system general, for example, is almost diametrically opposed to the desire to make it easy to use, because the addition of functions always complicates a system. After taking a close look at the implementation of Arctic's testing system, we will return to this set of goals and evaluate the system's performance with respect to each of them.

Chapter 4

Implementation of the Testing System

In this chapter, I will discuss the implementation of Arctic's functional testing system. There are two conceptually separate parts to Arctic's testing system, the "hardware" that connects to the Arctic chip and the "software" used to control that hardware. However, since the entire system is being simulated in the Verilog HDL, this distinction is blurred. I will begin by discussing the hardware side of the design, which consists of a number of signal generators and monitors connected to the Arctic module, and then describe the software side, which manages the tests themselves.

Since a great deal of attention was paid to the speed of this system, I will also describe one of the system's special operating modes, quick mode. In this mode, operations that read or modify configuration and control information inside the chip can be completed almost instantaneously, speeding up most tests by an order of magnitude. I will explain how this mode is entered and show how the addition of such a feature is made easy because Verilog is used as the implementation language.

Ideally, this system would be entirely contained within a Verilog simulation, but unfortunately, Verilog is not powerful enough to perform all the functions this testing system needs to perform. For example, Verilog has very primitive file input capabilities, and it completely lacks the ability to allocate memory dynamically. This necessitates an additional pre-simulation step which can be thought of as completely

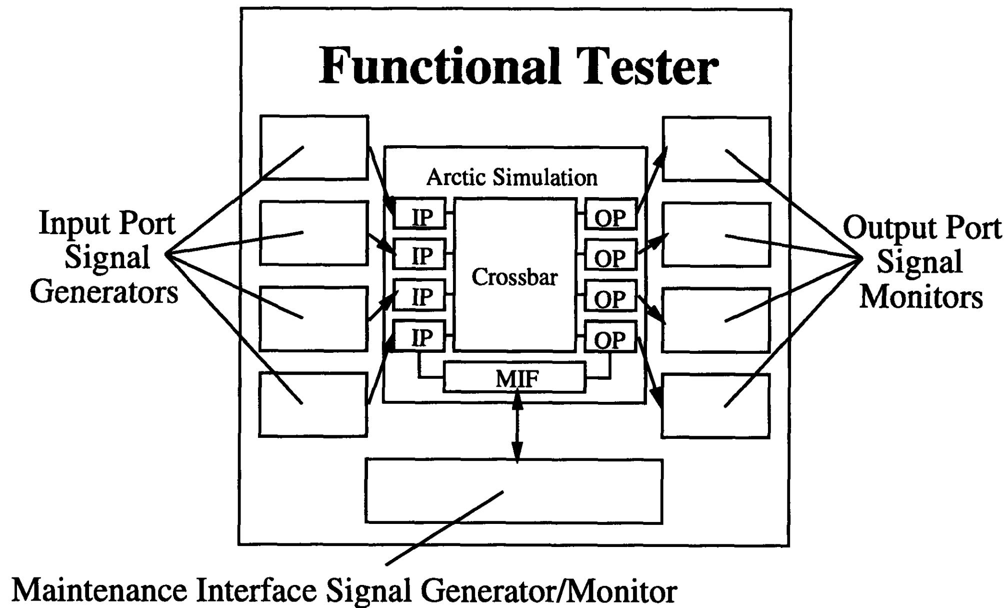

Figure 4-1: The Arctic Functional Testing System's "Hardware"

separate from the other parts, and I will therefore present it at the end of this chapter.

4.1 Hardware

The "hardware" in Arctic's functional testing system is presented in Figure 4-1. The Arctic module is inserted into a virtual testing fixture surrounded by smaller stub modules that communicate with each port and the maintenance interface, sending proper input patterns and recording or monitoring outputs. The Arctic design is represented as a Verilog module, and the structure for the testing system mimics the modular structure inside the chip.

This organization keeps functionally distinct units separate. In an actual system, each of Arctic's input ports would be connected to an output port (either on another Arctic chip or on some other network interface unit) and vice versa, and the maintenance interface would be connected to a JTAG controller. In this system, each smaller module imitates each separate functional unit. This gives the system a clear

structure, making it easier to expand and modify it.

It is also important to note that this structure is very flexible. Some testing systems have kept signal generators and monitors in separate C or Lisp processes that communicate with the hardware simulation through Unix sockets [7]. By keeping each unit in a Verilog module, we are able to modify the interface between the module and Arctic easily, should Arctic change, and we can easily modify the interaction between these modules.

Each of these smaller modules is really more like a stub than an actual functional unit. Whereas, in some of our early experiments, we hooked up a model for an entire output port to communicate with each input port, in this system we tried to keep each of these "stubs" as simple as possible. All contain logic that generates encoded signals (such as clocks and Manchester encoded signals) and records output patterns or signals and error if an incorrect pattern is received. Some contain functions that allow the user of the testing system to utilize the specific functions of the "stub." This gave more flexibility than using fully functional modules, which would have been easier to implement, but would not have allowed us to generate unusual (or erroneous) input patterns or record outputs in any way we saw fit. Following this reasoning, let us refer to a module connected to an input port the input port's stub, and the module connected to an output port the output port's stub. The module connected to the maintenance interface will likewise be called the maintenance interface's stub.

The recording machinery mentioned above is actually some of the most complex and varied logic in the entire system. Each stub receives Arctic's rather complicated output patterns and reduces it to a small piece of usable information, such as "packet X was received at time Y from port Z," or "signal B has just generated a Manchester encoding error at time T." Information that needs to be recorded, such as the received packet above, is stored as a record in some appropriate log file, to be scanned later when the system checks that all packets were received (this step will be discussed in the next section). Information that does not need to be recorded, such as the Manchester encoding error above, is error information and can stop the simulation immediately.

Most of the logic that transmits input signals to Arctic, on the other hand, is very simple. Most are simply registers or simple clock generators. The real complexity here lies in the control for these transmitters, which I consider software that is bound to the particular module in which it is used. This will be discussed further in the next section.

I believe this design offered the best flexibility and modularity of any organization. The different modules could be easily divided among the workers for implementation and maintenance, and the complexity of the system could be managed by observing the module boundaries. Another subtle advantage of this organization is that the Arctic module itself can be removed and replaced with any other model. This makes it easier to update the system when new releases of Arctic's design are made available, even if the new design is a gate-level model. This allows the testing system to be useful at every point in the design process.

4.2 Software

The software for this system is organized as a number of functions that are defined within the modules they relate to, with some of them kept at the top level. When a simulation begins, a top-level program that consists of many calls to these functions begins. These function calls in the top-level program can be thought of as actions taken during a session with this chip. The first actions reset and configure the chip, and these are followed by other actions that send in packets, extract internal state, or whatever is desired. The user is presented this interface to the functional testing system, but in order for it to be useful, there needs to be some logical organization for a test.

We decided to organize all tests in small groups. These groups were to be the smallest independent testing units in the system. In other words, the smallest piece of code that can accomplish a useful test is a "test group." This "test group" (called a "test sequence" in the user's manual in Appendix B) consists of a configuration phase, where Arctic is placed in a known state, an execute phase, where a number of

actions are carried out on the chip, and a check phase, where the state of Arctic is checked against a specification of what the state should be. This must be the smallest unit of a test, because it is the smallest unit that can allow the user to attempt to put the chip in a certain state and check whether it worked or not.

This organization mimics several other functional testing systems [3, 2, 7]. Most of these systems have some kind of testing unit that consists of an initial state for the system, a set of actions to be performed, and a specification of the final state of the system. It does seem to be the the one point that designers of such systems agree on, because it forces the users of the system to think about testing in a structured way, and it gives a convenient way to organize groups of related tests, any number of which can be run in one simulation, if desired.

In our system, test groups are specified with a number of files, each named `testxxx.type`, where "`xxx`" is the number of the test group and "`type`" is an extension such as "`code`" or "`config`." In the following sections, I will describe each phase of a test group and describe the file types that are relevant to it.

4.2.1 Configure Phase

To begin a test group, Arctic needs to be put in a known state. This phase begins by reading the file `testxxx.config` and using the specification there to store appropriate values in Arctic's configuration registers. After that, a known value is loaded into Arctic's control register, and the chip is ready to run a test.

Before the test actually begins, however, a little bookkeeping needs to be done. Several files that will be used to record the outputs and state of the chip during the execute phase need to be opened for writing. These files will be closed in the check phase or if the simulation dies suddenly due to an error.

4.2.2 Execute Phase

In this phase, all the actions that need to be taken on the chip are carried out. These actions are listed in the file `testxxx.code`, which contains actual Verilog code (pre-

dominantly function calls) that is spliced into the main program before the simulation begins. The first action in this file is a function call that carries out the configure phase of the test group, and the last action is a function call that carries out the check phase. All the code in between is part of the execute phase.

These remaining actions are predominantly calls to other functions that perform tasks such as writing Arctic's control register, sending a set of packets through the system, or reading and storing Arctic's statistics information. The system is very versatile, though, and allows the user to write any block of sequential Verilog code in this file, making it possible to specify any desired input pattern or check for any error that might not be detected in the check phase. This kind of versatility was necessary if the system was to meet our generality goal. Arctic is very complex, and it would be difficult to build special functions to generate all possible input patterns or detect every possible error. With the ability to use actual Verilog code, the user can test all those functions that cannot be tested with the existing input/output machinery.

Most of the actions performed during a test group do, however, conform to a relatively small number of patterns. This indicated that a set of functions that facilitated the generation of these patterns was a good idea. Many of these actions, such as writing a certain value to the control register or resetting the chip, can be implemented simply, but one action that deserves some explanation is the action that sends a set of packets into the chip.

Sending packets into Arctic is the most common and most complex input pattern. Transmitting packets is the basic function of the chip, but it requires the coordination of many encoded signals for a long period of time. For this reason, there is considerable machinery in this testing system that is devoted to sending packets.

First of all, all packets in the universe of the testing system have a unique number called a packet identifier. This identifier is actually part of the payload of the packet, which reduces some of the generality of the system by fixing those bits of the payload to be a certain value. This was necessary, however, in order to track the packets as they go through Arctic. A library of packets is read in at the beginning of a simulation.

These packets are used by the function **send_packets(file_name)**, where "**file_name**" is a file specifying a list of packet send commands. Each of these commands specifies a packet to be sent, the time it should be sent, and the input port it should be sent to. The name of this file is **testxxx.y.pkts**, where "**pkts**" indicates that this is a list of packet send commands, and the "**y**" can actually be any letter. This extra letter is useful because it is often necessary in a test group to use this function to send several sets of packets, and the files used by the different calls need to be distinguished in some way. As one might imagine, this is the most frequently used function, and it greatly simplifies the problem of sending packets though Arctic. Whether or not it helps enough is debatable, as we shall see in Chapter 6.

Up to this point, I have detailed many ways that the user can specify input patterns to Arctic, but I have not discussed how Arctic records outputs. Recall that during the configure phase of the test group, several files were opened for output. Each of these files has a name of the form **testxxx.log.logtype**, where "**logtype**" is either "**pkts**," "**stats**," or "**errs**." These files are used to store a specific type of output from the chip so that, in the check phase, this output may be checked against another file, **testxxx.chk.logtype**, which contains a specification for how the log *should* appear. The information in these log files is gathered either automatically, or when the user specifies.

The only information that is gathered automatically is a set of records specifying packets that are received at each output port's stub. Whenever a stub receives a packet, the stub checks that the packet matches the packet that was transmitted, and then a record specifying the packet's identifier, time of receipt, and output port is placed in the file **testxxx.log.pkts**. This file will be checked against **testxxx.chk.pkts** in the check phase to make sure that the correct packets emerged from the correct ports.

The other two kinds of log files gather information only when the user specifies. The function **write_stats** reads all the statistics counters in Arctic and stores their values in the file **testxxx.log.stats**. This information can be gathered any number of times during a test group, so that the user can monitor statistics before and

after some action is taken. The function write_errs_contr is a similar function that records all the bits of state not accounted for in any other check and writes to testxxx.log.errs. It reads the Arctic control register and the error counters in addition to some other small bits of information. As with write_stats, this information can be gathered any number of times, and at the end of the execute phase, these two files are checked against the files testxxx.chk.stats and testxxx.chk.errs.

4.2.3 Check Phase

In this final phase of the test group, the log files are first closed, and then each log file is loaded and checked against its corresponding check file. Since Verilog can only read in numerical data (and no strings), all of these log and check files must be encoded as numerical data, a relatively easy task since most of the data that can be collected from the simulation (such as statistics and errors) is already numeric. This gives a very simple check algorithm; read in the check file and the log file and make sure the two are the same for each value. For values in the log file that will not matter, an "x" can be placed in the location corresponding to that value in the check file. This will cause the check to work for any value in that location in the log file.

With this carefully defined structure of a test group, we have completed a clear definition of the hardware and software portions of the testing system. This defined, it is much easier to decide which new functions are possible, and how they should be added to the system.

4.3 Quick Mode

From the very beginning, we could tell that these tests were going to run very slowly. The design was so complex that it required nearly 50 megabytes of memory to simulate, and when running on an unloaded sparc10, the simulation averaged about two cycles per second. To make matters worse, all of Arctic's internal state was accessed through a JTAG test access port. This interface, which we called the maintenance interface, required all non-packet-related communication with Arctic to be done one

bit at a time through a scan path running at a clock speed of one-fifth of the system clock. Since each configuration register was 160 bits long, this meant that an absolute minimum of four minutes was needed just to scan in a single configuration register. To make matters worse, the maintenance interface had a very large overhead to manage the different types of data that could be scanned in, increasing the time needed to configure Arctic to about 45 minutes, on average. This was clearly an unacceptably long time to wait just to finish the first step of a test group.

The truly aggravating thing about these long maintenance interface operations is that they did not seem to be accomplishing any "real" work. The ability of the chip to scan in control and configuration information needed to be tested, but most of the time, these functions were only being used to set up the chip for another test and weren't performing any "interesting" actions. If there were a way for us to bypass this long configuration step, we thought, we could significantly shorten the time needed to run a test.

This observation resulted in the creation of "quick mode." In this mode, any "uninteresting" actions can be abstracted away, i.e. they can take place without being simulated. Operations performed through the maintenance interface, for example, can be done instantly, without using the maintenance interface at all. These operations can be given an extra flag as an argument. If this flag is 1, then the function will bypass its maintenance interface operation. By "bypassing," here, we mean that the operation will directly read from or write to the registers holding the state information that needs to be manipulated. With Verilog, any register buried deep inside a module can be accessed directly with a hierarchical name, which makes it easy for each function to read or modify any of Arctic's registers. Therefore, any function that manipulates state through the maintenance interface can be told which registers to access, giving it the ability to run in quick mode. The user must only keep in mind that, after executing an operation in quick mode, the system will not necessarily be in *exactly* the same state it would have been if the operation were performed normally. In most cases, though, the similarity is close enough to make a simple test run smoothly.

Since quick mode is implemented by accessing state inside Arctic, it might stop

working whenever the model of Arctic is changed significantly. Since this was likely to happen often, and we did not want the entire testing system to stop functioning when a change was made, we needed a way to turn off quick mode for the entire simulation. We defined a global variable, `SPEED`, that would be set at the beginning of a simulation to determine whether it should be run in quick mode or not. Even if the simulation were running in quick mode, however, we knew it was possible that the user might not want to run every possible function quickly. For this reason, every function that can run quickly is passed the extra argument `SPEED` that determines whether or not the function should run quickly for each separate call. In this system, then, it is possible to turn quick mode on and off globally, and it is possible to control it locally, at each function call.

This quick mode seems like a simple idea, but it is a very powerful one because it reduces the time needed to run a simple test from 45 minutes down to about 5 minutes, an order of magnitude improvement! For other tests that deal almost exclusively in maintenance interface operations, such as the the ones described in Appendix Section A.2, the improvement can be as much as 50 to 1.

4.4 Running the System

With the information presented above and the user's manual in Appendix B, a user can begin to experiment with this testing system. However, running the system needs to be discussed before the user can start up a simulation, and the user might want to know how to choose the tests that he or she wants to run. These problems, as well as a few implementation problems that cannot be solved using Verilog, are all solved with a pre-simulation step that is used to run the system.

The user actually runs the system by typing `run_test filename` where "`filename`" is the name of a master simulation file such as the one in Figure 4-2 that contains a list of packet libraries followed by a list of tests that need to be run. The libraries contain all the packets that the system can use, and must be generated before the simulation, either by hand or with some generation tool. This organization gives the

```
../packets/lib1
../packets/lib2
../packets/lib3
../packets/lib5

sequences

test003
test009
test010
test051
test082
```

Figure 4-2: Master Simulation File

user precise control over the packets generated, and easily accommodates new groups of packets. A list of tests follows this list of libraries. This list specifies which tests will be run and what order they will be run in.

The pre-simulation step sets up the Verilog simulation by instructing it to begin by loading in the specified libraries of packets. It then scans each of the tests listed and instructs the Verilog simulation to run each test. This is done by splicing the `testxxx.code` file for each test into the top level simulation. After this step completes, the Verilog simulation begins and performs all the tests that were specified in the master simulation file.

This pre-simulation step solves another rather difficult problem. Verilog does not have the ability to allocate memory dynamically, and the size of every data structure needs to be specified before the simulation can begin. This is difficult to deal with in this system, because the size of the buffers needed to hold log files or lists of packet send commands will vary widely from test to test, and unless we used some gross overestimate of the maximum space needed, the simulation would risk running out of memory. To avoid this problem, and to avoid wasting memory, the pre-simulation step scans through the tests to be run and sets up some important data structures in the simulation to have just the right size. This makes the system a bit more difficult to understand, but it does make it much easier to use when the user can abstract

away the details of this step.

This completes a full picture of Arctic's functional testing system. The system has other operating modes which the user has access to, but the information given above is sufficient for a user to begin writing useful tests. The amount of detail presented here may seem excessive, but I believe it is useful as an example of the problems that arise when designing such a system. For those readers desiring even more details about how this system is used, Appendix A presents many example test groups. The first example presented there is particularly helpful, demonstrating how the different parts of the system interact in a simple case. Appendix B contains the user's manual for this system, and may also be of interest for readers desiring greater detail.

Chapter 5

The Randomized System

In earlier chapters I discussed at length the importance of randomization in a testing system, but Arctic's testing system, as I have described it, has few random testing features. To remedy this situation, I designed a second system devoted entirely to random testing. This system is based on the basic structure and modules of the older system, but it is kept separate, because it, unlike the tester-debugger of the previous section, is designed to be run with almost no human intervention.

We have already seen a number of reasons why some kind of random tester is necessary. Perhaps the most important is that it alleviates the burden of generating tests. With a random tester, a design team can let the testing system look for obscure bugs while the team members concern themselves with debugging known trouble spots. Also, randomized testing is desirable for theoretical reasons. The number of possible states in a large chip like Arctic is huge, and it is nearly impossible for a human to pick out a set of tests that will uncover *every* design bug. A carefully built randomized tester can find this set with a fairly good probability. Douglas Clark expresses this idea well in his paper on hardware verification strategies.

> The method I advocate for dealing with the problem of a huge space of behaviors is to sample randomly (really pseudo-randomly) from one or more sets of possible tests. A random selection can pick any test with some probability, and the longer a random test-selector runs, the more

likely it is that any particular test will be found. [2]

In this chapter I will present the design and implementation of Arctic's random testing system. This part of the testing project is much more recent, and as a result, important parts of the system have not yet been implemented. Therefore, I will give only a brief overview of the design and where I cannot give examples of problems encountered with the system, I will try to predict the problems that would have been encountered if the system had been finished. To remain consistent with Chapter 4, I will begin by discussing changes to the system's "hardware" and then discuss changes to the "software."

5.1 Hardware

In the new system, each stub has been modified to hold some kind of randomizer. Each of the stubs connected to the input ports, for example, has a random packet generator that is capable of creating a completely new packet whenever it is needed. The user is given the ability to specify the distribution of these random packets by controlling certain constants for each port, such as the probability that a packet is a high-priority packet, the mean length of a packet (modeled as an exponential random variable), and the mean wait time between receiving a packet and sending a free buffer signal (both of which are also modeled as exponential random variables). This is a very loose control mechanism, allowing the user to have only a general influence on the traffic through the chip. For the most part, the simulations execute without any outside influence at all.

Because all packets could be generated randomly in this system, I chose to create completely new packet management mechanisms that would be contained in the input and output port's stubs. In this system, packets were generated when needed and did not need to be kept in files and loaded at the beginning of a simulation. Also, records of received packets were not written to any file. Instead, each input port's stub maintained records of the packets which it sent. An output port's stub that received a packet would immediately check the sender's record to see if the packet was sent

out the correct port, and if it was, the sender's record would be deleted. Managed correctly, this system could continue to send packets forever using a constant amount of storage.

Let's take a closer look at how each sender manages the transmission of packets. When a new packet is created it is given a unique identifier in the form of a timestamp generated with the Verilog command `$time`. This timestamp is not completely unique, since other input port stubs might be generating packets at exactly the same time, but packets can also be distinguished by their sender, so this does not really matter. Note, however, that I have sacrificed a bit of generality here by requiring each packet to have correct timestamp and point of origin information in its payload.

After the packet has been generated, the input port's stub stores its timestamp, length, and checksum in a queue corresponding to the output port of Arctic from which the packet should leave. When an output port's stub receives a packet, it looks in the queue that should hold a record for the packet. Since packets are sent in FIFO order between any input port and output port, the output port's stub is guaranteed to find the needed record at the front of the queue, as long as the system is working. The stub checks the timestamp, length, and checksum of the packet with the values in the queue, and if they are not the same, it signals an error. Otherwise, it pops the element off the queue and waits for the next packet.

Since there is a limited amount of storage inside Arctic, this system knows the maximum number of outstanding packets there can be at any given time. Therefore, the size of these queues is rather small and constant, a considerable improvement over the old system, which had to hold every packet in a buffer for the entire length of a simulation.

The random packet management system described above is almost all the randomization "hardware" that is required for this system. This part of the system has been completed; it generates and receives packets continually, modeling "correct behavior" for Arctic by simply keeping track of the packets sent. However, there are two other parts of the system that have not been implemented yet. One of these is, of course, the randomizer connected to Arctic's maintenance interface. This seems, at

first glance, to be simple to implement; the transaction functions from the previous system can be used to send randomized configuration and control words to Arctic. The other randomizer that is needed is some kind of error generator to create link errors, bad packets, incorrect maintenance interface operations, and other detectable errors. This seems a bit more complicated, but still manageable since there is a clear definition of what kinds of errors Arctic can detect. These will not be simple to implement, however, because the system must not only generate these inputs, but also predict how those inputs will affect Arctic.

Both of these randomizers can have a dramatic effect on Arctic. The configuration and control words can affect the flow of packets in ways that are very difficult to deal with. Randomly changing the routing information in a port's configuration register, for example, can extremely complicate the problem of tracking randomly generated packets. Also, some detectable errors can create a confusing avalanche of errors. Some link errors can cause this by fooling an input port into thinking that a packet is being transmitted when the port is really idle, thereby creating a flood of errors that are detected on a packet that does not actually exist.

There are a few tricks that can be played to simplify this problem. Maintenance interface operations can be managed by structuring the testing "software" so that the effects of each of the operations has a more limited scope of effects. This will leave a larger space of tests untried, but some of these untried tests may not be needed, and those that are can be tested with the older, more flexible system. If we restrict configuration changes to ones that do not change routing control bits, for instance, we can keep packet management simple and still avoid routing bugs by conducting extensive tests on the routing bits with the older testing system. More complex behavior can be avoided by restricting the moments when Arctic's control register can be changed. Changes in the control register that take effect when there is still packet traffic in the system are very hard to model. These situations can be overlooked in the testing system on the grounds that, in *most* cases, changes in control information take place when packets are not flowing.

Random errors present a slightly different problem. Ideally, we would not like

to limit the kinds of errors that can be generated, because error counter tests are hard to create with the old system (as seen in Appendix Section A.3), and we cannot easily claim that the test cases we will miss are unlikely to occur in normal situations. Perhaps an avalanche of errors can be tolerated, though. If, when an error is generated, the testing system stops worrying about all state in Arctic *except for* the error counter that should reflect the detection of that error, then any avalanche of errors can be tolerated. Once the system generates the error, the system will determine if the error was detected, and then reset Arctic and the testing system to begin from a known state. This is a valid approach because it actually models how Arctic will be used. Arctic is designed to *detect* and not *recover from* errors.

Designing these randomizers is not altogether trivial, then. The structure borrowed from the earlier system makes it easier to decide where certain parts can be placed and does provide some functions for managing maintenance interface transactions, but predicting how these random generators affect the state of Arctic can be hard. The benefits are considerable, however. This system is capable of generating a wide variety of tests that a human would never be able to create.

5.2 Software

The "software" in this system coordinates the activities of the randomizers. The randomizers are capable of running without any central controller, but the system still needs to occasionally stop sending inputs and determine if Arctic is in a correct state. For this reason, the concept of a configure-execute-check loop is still useful. The system should put Arctic in a known state, send random inputs for a certain amount of time, and then pull state information out of Arctic to determine if Arctic's state matches the predicted state. Also, if a bug is detected, it is convenient to have a checkpoint from which the simulation can be restarted so that the conditions under which the bug appeared can be repeated without re-running the entire simulation. Tests are therefore organized in test groups, just as in the earlier system. Again, the group is the smallest unit of a test, and any group can be run in isolation.

The difference with this system is, of course, that very little human intervention is needed before the simulation can begin. Some constants need to be set to define the average length of packets, the frequency of packets, the probability of generating certain errors, the number of cycles in a test group, etc., but these are provided mostly for the purpose of tweaking the system so that a nice balance of inputs can be found. The only piece of information in the system that need change for each simulation is a single number that serves as the random seed for the entire simulation. All random numbers in the entire system are derived from this seed, and it need change by only one bit to generate a completely new random simulation. The register containing the value for this seed is modified after every random number is generated, and the seed is output at the beginning of every test group. A test group can be re-started simply by using the given seed as the initial seed for the simulation.

Let's look at each of the phases of a test group, individually. At the beginning of a new group, the configuration phase is entered and a new configuration is chosen. As mentioned in the previous section, this configuration is not completely random, because the bits corresponding to routing rules are always the same. All other configuration data that affects packet flow or statistics is randomizable. A random initial value for the control register is also chosen in this phase. When all this is complete, the execute phase begins.

The execute phase is very simple in this system. A counter is set to a user-defined initial value, and begins to count down to the moment when the system will begin to enter the check phase. While the system is running, each input port's stub generates packets, and each output port's stub processes the packets it receives, signalling errors when it detects them. The errors and maintenance interface randomizers cause various errors and non-destructive control transactions as well. Thus, the system can imitate the widely varied interactions that would happen during the normal operation of Arctic.

The check phase begins when the aforementioned counter reaches 0. First, the system waits long enough for all packets that are in transit to emerge from Arctic and for all outstanding maintenance interface operations to complete. Then, the system

begins to check the internal state of Arctic. During the execute phase, the system maintained what it believed to be appropriate values for the statistics counters, error counters, control register, and any other registers that could be read out of Arctic. During the check phase, each of these values is read out of the chip, and if any discrepancies are noted, then an error is signalled. At the end of the check phase, some statistics reporting the actions taken on the chip during the test group are printed in the simulation log file, so that the test system itself can be monitored, and the system re-enters the configure phase.

This system is rather pretty, as described above, but there are a few ugly details that have not quite been worked out yet. This biggest of these problems is deciding exactly how to manage the changes in packet flow that are caused by some changes to Arctic's control register. Arctic has some functions, manipulated through the control register, that can block, re-route, or flush packets. The behavior of each of these functions can depend on whether or not any one of the other functions preceded it and whether or not packets were flowing at the time the function went into effect. Even if we decide to simplify the problem by not allowing packets to be flowing when a function goes into effect, there are still problems deciding how to stop the flow of packets to make the change in the control register and how to embed knowledge into the system about how one function should transition into another function. At present, I see no solution other than brute force – building into the system all the information it needs to know to modify the control register safely and to predict the resulting changes in packet flow.

It seems then, that building this randomized testing system is no simple task. The structure of the earlier system did, however, give me a clear starting place for this work and was flexible enough to accommodate all of my changes. Though this system has not been completed, I believe it could be extremely helpful in eliminating subtle design bugs. Remember, the advantage of this system is that it has some probability of catching almost any bug. The bugs that it is not capable of catching can still be found by focusing some directed tests in the areas that the system cannot reach. The benefit of the random tester is not that it takes care of all testing, but that it narrows

the amount of extra testing that needs to be done to something more tractable. Thus, this random tester does not *replace* the older system, but merely adds to it, because the older system is still necessary as a debugging tool and tester for those functions not testable with the randomized system.

Chapter 6

Evaluation of the Arctic Testing Project

Before we can begin to consider what wisdom Arctic's testing system has taught about the functional testing problem, we need to evaluate how well it has met each of the goals laid out in Chapter 3. This functional testing system has been up and running in some form since the summer of 1993, and in that time I have had many chances to evaluate its performance relative to those stated goals. In this chapter, I will list those goals and comment on how well the system meets them.

6.1 General

One promise this system does indeed deliver is generality. It was never the case that the system could not generate an input pattern that was desired, nor was it the case that it could not detect some erroneous output pattern.

Some prime examples of the value of this generality, as seen in the Appendix Section A.3, are the error detection test groups. These test groups needed to create some very unusual input patterns in order to determine if the error detection functions were working correctly. Without the ability to put any block of Verilog code in a test group, these tests could never have been integrated seamlessly with the rest of the system. Many subtle bugs were uncovered by the test groups that focused on errors.

Other good examples of tests that rely on the generality of this system are the JTAG tests, one of which is described in Appendix Section A.4. These tests did not integrate quite as seamlessly into the system as did the errors test groups, since special functions needed to be written for them, but they were relatively easy to write because of the structured framework they fit into. The JTAG tests uncovered several problems related to the stitching of the scan rings.

The random tester appears to be very general as well. Considerable effort has gone in to making it capable of generating a wide variety of inputs. Also, it is possible to determine types of bugs that the random tester has no chance of detecting, and focus directed testing on cases that have a good chance of detecting those bugs. This seems to be a very general, comprehensive approach to random testing.

6.2 Fast

Because of our efforts to give many procedures the ability to run in quick mode, this system did end up being fast enough to use as a debugging tool. By speeding up only two functions, configuration and writes to the Arctic control register, the running time of a simple test could be reduced from 45 minutes to about five minutes. We came to rely on this quick mode very heavily, but it was always easy to turn off when it broke because of the `SPEED` variable mentioned in Section 4.3.

I used this quick mode quite heavily whenever I found myself debugging portions of the chip that I did not design. By placing signal probes and re-simulating I was able to track down several typographical errors without any help from the implementors themselves. The time to re-simulate after changing probes was short enough to make this kind of debugging possible.

6.3 Randomizable

Our intention was to extend the original system to run random tests and search for bugs without our intervention. In actuality, the second system is not connected to

the first, but since the basic structure of the first system was borrowed to build the randomized system, I can claim that this goal was met in some way. The true goal, of course, was to build a random testing system that could be relied on to search for subtle bugs. Since the system has not been completed, I cannot claim that the system met that goal. It should certainly be clear from Chapter 5, however, that the system should, in its final form, be able to generate quite a wide variety of tests.

6.4 Repeatable

The original functional testing system was obviously repeatable, and that made it very useful as a debugging tool. The fact that the randomized functional tester was able to restart any generated test group without re-running all previous groups was also very handy. In its half-completed form, the random tester only found one bug, but that bug would have been very troublesome to track down without this ability. The error was caused by a pattern that was generated after several hours of simulating, but because we were able to restart the test group, it took us only about 10 minutes to reproduce the error.

6.5 Low Level

One of the more obscure requirements we made of this system was that it should be able to simulate Arctic at the lowest possible level. Most of the time, functional testing was performed on the original Verilog model of Arctic. However, the system is capable of running on a Verilog gate-level description of Arctic that is generated by Synopsis. This model can also be back-annotated with timing data, so that the simulation can come very close to the actual behavior of the fabricated chip. Unfortunately, this part of the system is not working either, due to some problems which we believe are related to the crossing of clock boundaries, but this may be corrected soon.

6.6 Easy to Use

At the end of Chapter 3, I stated that some of these goals were almost in direct opposition to each other, but in all of the preceding sections, Arctic's testing system seems to have met its goals well. Unfortunately, many of these goals are met at the expense of this final goal, ease of use. In nearly all places where a tradeoff was made between ease of use and some other quality, ease of use was sacrificed. This is especially true of generality. The confusing file structure and packet management system of the original system is all done in the name of generality.

This does not mean that the system is impossible to use. Great effort was put into providing simple ways to specify common inputs, and generating a specific input pattern and checking for a specific output pattern was generally very easy to do, no matter how unusual the pattern was. For example, it would take about 30 minutes for someone familiar with the system to write a test group that disables output port 2 and sends, 3 cycles after disabling, a packet destined for port 2 with the value `057b3aff` in the 14th word of its payload into port 1. This could be very useful if such an input pattern was believed to reveal a bug. Our hope was to make specification of such tests easy, and the system succeeds in this goal.

The main problem with this system is that normal cases are not much easier to specify than unusual cases, despite our efforts to simplify these cases. If the user wanted a test group that did something as simple as sending 100 different packets through the system each 10 cycles apart, this could take hours to create because the user has to specify every bit of every packet and define exactly what time each of these packets should be sent into the system, as well as exactly which output port each should emerge from. In most test groups, the user needs to send a large number of packets, but the system gives the user such precise control over these packets that it is impossible to send a packet unless it is described in every detail. Some solution to this problem was needed. We created some packet generation programs, but these were rather awkward and often generated an unmanageable number of packets. Perhaps this problem could have been solved by putting random packet generators in each

input port's stub. Test groups could specify a few parameters for these random packets and let the system worry about generating them. Tests of this kind were called "bashers" by the team working on the Alewife cache controller, and apparently uncovered many bugs [7].

Another reason designing test groups was so difficult was that the structure of the files used in these test groups was very unintuitive. Remember that Verilog can only read in files with binary or hexadecimal data. That meant that every file that specified some detail of a test had to be encoded, adding an extra layer of confusion to the testing process. Imagine, for example, that the user wants to send packet 5 into port 1 at time 0. To do this, the user has to create a packet insertion file that looks like this.

```
0000000000000001
0000000000000052
```

Since it would take an inhuman memory to recall the exact position of each of these bits, the user is forced to call up the testing system user's manual when this file is created, or whenever it needs to be modified. This slows the user down, and makes life very confusing. If the file instead contained the line "Insert packet 5 into port 1 at time 0," the system might have been much easier to use. It would not have been easy to add this ability to the system, but it is possible it could have been supported as a pre-simulation step that would translate more easily understood descriptions like the one above into hexadecimal numbers before the simulation. A simpler and perhaps just as effective approach would be to make all test files a kind of table that the user would fill out. The pre-simulation step could then remove everything but the numbers, which would be input by the Verilog simulation.

It seems, then, that the only big problem with this functional testing system is its complexity. Indeed, many members of the team never learned to use the system at all because it takes so long to learn. Even those who do understand it avoid using it regularly.

Still, this system has been successfully used to carry out extensive testing of the

Arctic chip. Countless bugs have been uncovered through directed testing with the original system. Many examples of these tests can be found in Appendix A. The random tester has found only one bug, but has the potential to find many more if it is completed. Also, beyond these practical concerns, this system has been a wonderful test bed for many ideas related to functional testing.

Chapter 7

Conclusions

In this thesis, I have presented many details about the implementation of Arctic's functional testing system to give the reader an idea of the many subtle benefits and complications that arise from each design decision. To conclude, I will discuss what the Arctic Testing Project has taught me about functional testing and what I hope the reader will take away from this thesis.

Since Arctic's testing system has grown out of the idea that there are unique advantages to implementing a chip's functional testing system in the same language used to implement the chip itself, these ideas may seem applicable only to gate arrays and other ASICs designed entirely with HDLs like Verilog or VHDL. However, these languages are being used more and more frequently to generate significant amounts of logic in standard cell and full custom chips. We may be approaching the days when entire full custom chips will be designed with derivatives of these hardware description languages that integrate the specification of actual transistors with higher-level specifications. If this were to happen, the ideas in this thesis would be as applicable to full custom designs as they are to gate array designs.

7.1 Structure Is Important

One of the main focuses of this thesis has been how the careful organization of testing "hardware" and "software" has simplified Arctic's testing system. Grouping signal

generation and monitoring hardware into functionally distinct Verilog modules greatly simplified extensions and maintenance to the system. Grouping testing software into test groups with configuration, execution, and checking phases forced the users of the system to think about testing in a structured way. Any testing system can benefit from such a clear and well defined structure. The structure helps by reducing the time needed to design and use a testing system, which in turn shortens design time.

Another key idea in this thesis has been the co-existence of a directed testing system and a randomized testing system. Neither makes a complete testing system by itself, but together the two can give the designer great confidence that a relatively bug-free design is being produced. Since the designs of these systems are related, they may be part of the same system or at least share a large number of modules.

The paramount idea here is that a clear structure is important. It is my belief that without this structure or one very similar to it, we could not have tested Arctic in so many varied ways so quickly, and we could not have weeded out so many design bugs before the first place and route.

7.2 Speed Is Necessary

Speed is a problem for any hardware simulation system, and our design team knew that this testing system would also be disappointingly slow if we could not give it some kind of fast operating mode. Over the course of this project it has become clear, however, that Arctic's functional testing system would have been completely useless as a debugger without quick mode. I believe that this is one of the most important ideas to be taken from Arctic's functional testing system. It is likely that many current chip designs simulate as slowly as Arctic. For any such design it is absolutely necessary to simulate in some faster mode or no debugging can take place before the chip is fabricated.

In Arctic's functional testing system, quick mode was implemented by abstracting away "uninteresting" behavior. Certain functions that read or modified state inside Arctic did so by manipulating the necessary registers directly, rather than going

through Arctic's maintenance interface. Thus, an action that changed the value in the configuration registers could be done instantly rather than taking the normal 3000 or more simulated cycles of operation. This is obviously advantageous if actions that use quick mode are common in testing simulations, which, in fact, they are. In an actual Arctic network, the percentage of time spent doing maintenance interface operations would be almost negligible, but in order to *test* Arctic, functions that configure and control Arctic through this interface are needed very frequently.

This idea can easily be extended to other testing systems. In any testing system, debugging is very important, and in any individual test, it is likely that certain parts can be thought of as "the test," and other parts can be thought of as "setting up for the test." Those parts that are setting up are generally uninteresting, and if they take a very long time to simulate, having a mode where they can run quickly (without much loss of detail) can be a great help.

7.3 Complexity Is The Enemy

This system had only three regular users. Though it was designed to be useful to every member of the design team, the learning curve was so steep that most members of the team used their own simulation tools. For this reason, the danger of complexity is an important lesson to be learned from Arctic's functional testing system.

In fact, both the directed and the randomized testing system suffered from complexity. The directed tester and debugger had confusing file formats that became unmanageable when tests became too large. The random system did not need much of a user interface, but it's complexity has made it very difficult to determine whether or not it generates the wide variety of tests that we expect. These types of problems are fairly common for large software projects. It is important to remember that problems such as these can easily arise in any system that tests a very large and complex chip.

Fortunately, this complexity did not ruin the testing project, because those of us that taught ourselves to live with the complexity of the system were able to perform

extensive testing and get valuable experience from the testing system. This experience has taught us a great deal about the functional testing problem and has given us reason to believe that, one day, the ideas in this thesis may be used to make functional testing as tractable as every other part of the ASIC design process.

7.4 Future Work

It has been mentioned before that Arctic has not yet had its initial place and route. This gives us some time to add the remaining functions to the randomized testing system, and perhaps to get the system working with the compiled, gate-level model of Arctic. The ultimate test will be to see if only timing-related bugs are detected after the first place and route. Beyond that, the system may be extended to generate functional tests that will be used as manufacturing test vectors to supplement those generated with the Mustang Automatic Test Pattern Generator.

Appendix A

Example Test Groups From Arctic's Functional Testing System

This appendix has been included for those readers wishing to see examples of how Arctic's functional testing system is used as a debugger. It will clarify a number of details already presented, and present a few more to show how the system can be extended even further. I assume that the reader fully understands the description of Arctic given at the beginning of Chapter 3. Other details are given as they are needed. I will begin with a simple routing test, follow it with some tests of Arctic's errors and statistics counting functions, describe a test of one of the JTAG test functions, and finally give some examples of some tests that did not work very well, namely a random packet insertion test and a high priority test.

A.1 Routing

One of the most simple examples of the use of this system is a test that checks if the routing logic is working correctly. Arctic uses a two-comparator routing scheme to determine which output port each packet is destined for [5]. As described in the

```
00000001
0ffbdff8
040f7bfe
00000000
00000000
000300ff
```

Figure A-1: `test003.config` file

Arctic user's manual, this scheme basically has the form

$$if\ B_1\ then\ R_1\ else\ if\ B_2\ then\ R_2\ else\ R_3$$

where B_1 and B_2 are boolean expressions which are the comparators in this scheme. R_1, R_2, and R_3 are expressions which evaluate to 2-bit values that determine which of the four output ports a packet is sent out. The comparators are defined as follows.

$$B_1 = (DOWNROUTE\ and\ RULE1_MASK)\ equal?\ RULE1_REF$$

$$B_2 = (DOWNROUTE\ and\ RULE2_MASK)\ equal?\ RULE2_REF$$

Where RULE1_MASK, RULE2_MASK, RULE1_REF, and RULE2_REF are all defined in Arctic's configuration registers. B_1 and B_2 are used to determine which of R_1, R_2, or R_3 is used to determine the output port.

The reader may note that this routing scheme is rather complicated, and we have not even discussed the definitions of R_1, R_2, and R_3! However, we can already see an opportunity to try a simple test. By setting both RULE1_MASK and RULE1_REF to 0, all packets sent into the system should emerge from the port determined by R_1. This is the simple test performed by `test003` in our system.

The first step in defining this test is to create the `test003.config` file, which is shown in Figure A-1. The file consists of only numerical data, as Verilog requires, and the numbers shown correspond to the actual values loaded into the configuration registers. Each of arctic's input ports has five 32-bit configuration registers, and one

```
start_sequence("../sequences/test003/test003.config",
               "../sequences/test003/test003.log.pkts",
               "../sequences/test003/test003.log.stats",
               "../sequences/test003/test003.log.errs",`SPEED,`TIMED);
in.send_packets("../sequences/test003/test003.a.pkts");
lclk_delay_cycles(100);
end_sequence;
```

Figure A-2: `test003.code` file

32-bit datum is given for each of these registers. The 00000001 that appears at the top indicates that these values should be copied into the configuration registers of every arctic input port. If this value were a 00000000, then five 32-bit values could be listed for each of the four input ports. I will not detail the meaning of each bit in the configuration information. Suffice it to say that RULE1_MASK, RULE2_MASK, RULE1_REF, and RULE2_REF are all set to 0.

Now that the initial configuration has been specified, the list of actions needs to be specified in `test003.code`, which appears in Figure A-2. The only action taken in this test group is the sending of a set of packets, and, as a result, this file is very simple. The first line carries out the configure phase of the test group. (The function is called `start_sequence` because test groups were originally called test sequences.) The arguments to `start_sequence` are the file names of the log files that will be output at the end of the test, plus the `SPEED` argument mentioned in Section 4.3, plus another flag that we shall discuss in Section A.5. Following this action is a `send_packets` command and an `lclk_delay_cycles` command that makes the system wait for 100 local clock cycles before entering the check phase. The final command carries out the check phase. This is all that is needed for a very simple test such as this, and this file is about as simple as a `.code` file gets.

Next we need to specify the packets that are going to be inserted into the system. Commands that will direct the system to insert a packet will be kept in a file named `test003.a.pkts` (shown in Figure A-3), which is the argument to the procedure `send_packets`. For this test, `test003.a.pkts` contains only eight commands, one for

```
0000000000000008
00000000040081080
00000000040081082
00000000040081084
00000000040081086
00000005040081f80
00000005040081f82
00000005040081f84
00000005040081f86
```

Figure A-3: `test003.a.pkts` file

```
0400810810000000000
0400810810000000000
0400810810000000000
0400810810000000000
040081f81000000000
040081f81000000000
040081f81000000000
040081f81000000000
1f0000000000000000
```

Figure A-4: `test003.chk.pkts` file

each packet inserted into Arctic. The value at the beginning of the file is the number of packet send commands contained in the file, and it is followed by the commands themselves. Each of these commands consists of three fields. The left most field is a 32-bit number specifying the number of cycles after calling send_packets that the transmission of the packet should begin. The middle field is the 32-bit packet identifier, and the last four bits specify which input port the packet should be sent to (shifted left by one bit). In this packet insertion file, the first half of the commands insert packet `04008108`, and the last half insert packet `040081f8`. The first four are sent to each of the four input ports at cycle 0, and the next four are sent to each input port at cycle 5.

With all the inputs to the system specified, all that remains is to create the check files which will determine what the generated log files should look like. The

```
000000000
```

Figure A-5: `test003.chk.stats` file

```
0000000000000000
```

Figure A-6: `test003.chk.errs` file

file `test003.chk.pkts` appears in Figure A-4 and consists of eight lines specifying the packets that should come out of Arctic during the test followed by a unique value, `1f00000000000000`, that marks the end of the file. The lines that specify how packets should be received are made up of four fields. The left most field is the packet identifier, and it is followed by a four-bit field specifying the output port the packet was sent from, and another four bit field which is 1 if every bit in the received packet should match every bit of the sent packet. The right most 32-bit field corresponds to the time that the packet is sent out, but we will ignore this field until Section A.6. In this file, we have one entry for each packet, each of which should be received from output port 1, and none of which should match the transmitted packet bit-for-bit, because a new checksum word should be inserted at the end.

The files `test003.chk.stats` and `test003.chk.errs` appear in Figures A-5 and A-6 respectively. Both consist of nothing but trailer words of all zeros that signal the end of the files, because during this test, neither the statistics information nor the errors and control information was ever read from Arctic.

We have completed the last step in specifying this test. The test exercises a specific part of the routing logic, and at the same time gives a nice, basic test to see if the packet transmission mechanisms are working correctly. The reader may note that specifying this test was a very involved and time-consuming process, because of both the complexity of Arctic and the cryptic format of the files. The time needed to create a test can be reduced by duplicating and modifying old tests to make new ones, rather than creating new ones from scratch, but this is still rather complicated because the test is spread over so many files. In Section 6.6, I explore just how much

```
start_sequence("../sequences/test060/test060.config",
               "../sequences/test060/test060.log.pkts",
               "../sequences/test060/test060.log.stats",
               "../sequences/test060/test060.log.errs",'SPEED,'TIMED);
mn.clear_stats('SPEED); //clear statistics
in.send_packets("../sequences/test060/test060.a.pkts");
lclk_delay_cycles(100);
write_stats(0,'SPEED);
write_stats(1,'SPEED);
end_sequence;
```

Figure A-7: `test060.code` file

of a burden this complexity is.

A.2 Statistics

Let's take a briefer look at `test060`, which runs some basic tests of Arctic's statistic's counters. The file `test060.code` is shown in Figure A-7. After the initial `start_sequence` command, the statistics are cleared with the command `clear_stats`. This needs to be done first because resetting Arctic does not clear the statistics counters. After this is done, a set of eight packets is sent into Arctic, and the system waits 100 cycles before starting to read out statistics information. The statistics are then read out twice, first non-destructively, and then destructively (this is determined based on the value of the first argument to `write_stats`). A destructive read clears a statistics counter when it is read, so in this test, the first set of statistics should be nearly identical to the second.

The file `test060.chk.stats` appears (in part) in Figure A-8. Since statistics information is read out twice, this file contains two reports of the statistics information (though only one is shown because the second is exactly the same as the first). Each report contains a header word, `000000001`, followed by each of the 24 statistics values, arranged in order of increasing address, as specified in the Arctic User's Manual [5]. The statistics specified as "`x`" are all idle statistics that increment once for every three cycles that an output port is not busy or waiting. We put an "`x`" in these locations

```
000000001
000000000
000000000
000000000
000000000
000000xxx
000000000
000000000
000000000
000000000
000000000
000000xxx
000000000
000000004
000000000
000000000
000000004
000000xxx
000000000
000000004
000000000
000000000
000000004
000000xxx
000000000

<All the above text is repeated here>

000000000
```

Figure A-8: `test060.chk.stats` file

because we are not interested in these statistics during this test. At the end of the file is a trailer word of all zeros, as in the previous test group.

Each of the statistics locations must be read independently through Arctic's maintenance interface. Even ignoring configuration time, this test group can take over an hour to run, even on the fastest computers this group has available. Fortunately, if someone is trying to debug the statistics counters, it is possible to run the test in quick mode, in which case the test will run in about five minutes. For this test, quick mode is even more helpful than usual.

A.3 Errors

Now we will look at a test group that demonstrates the benefits of using actual Verilog code to specify inputs to Arctic. `test052` sends a number of packets into Arctic and then forces values onto the wires that are transmitting these packets to generate link errors (errors that are signalled when a link is not functioning or is transmitting an incorrectly encoded signal). The file `test052.code` appears in Figure A-9. Notice that after the packets are sent, specific wires are forced to have specific values at specific times. This precise control is needed because a link error can be caused by an incredibly wide variety of patterns, and the user needs to be aware of exactly what kind of input pattern causes certain errors to be detected.

The first error is generated when a bad value is forced on input port 3's data transmission line at a moment when no packet is being transmitted. This should cause Arctic to increment port 3's idle error count. Next, a value is forced on input port 2's data line at the moment a packet's length field is being transmitted. This should cause Arctic to increment the length error counter for port 2. Finally, the frame wire for input port 0 is given a bad bit, so that the frame error counter for port 0 will increment.

By the end of this test, four errors should have been detected (the above three plus a checksum error because one of the packets transmitted had a bad checksum word). The file `test052.chk.errs` in Figure A-10 reflects these errors. This file consists of

```
start_sequence("../sequences/test052/test052.config",
               "../sequences/test052/test052.log.pkts",
               "../sequences/test052/test052.log.stats",
               "../sequences/test052/test052.log.errs",`SPEED,`TIMED);
in.send_packets("../sequences/test052/test052.a.pkts");

begin
#12 force data_ni3[0] =1;    // this should cause an idle error
#10 release data_ni3[0];     // and a freeze before either of two
end                          // packets arrive on input 3

begin
#352 force data_ni2[0] = 1; // This should cause a length error on
#10  release data_ni2[0];   // input 2 (but no checksum error
end                         // because the checksum of this packet
                            // assumes this force).  A freeze should
                            // occur before input 2 receives a
                            // second packet.  The second packet is
                            // addressed to output 2.

            // Packet with bad checksum is being inserted on input 1.
            // It should cause input 1 to get a checksum error and
            // freeze (before getting a second packet and the second
            // packet is addressed to output 2).

begin
#382 force frame_ni0 = 1;  // This should cause a 11 pattern where a
#10 release frame_ni0;     // 10 is expected.  That in turn should
end                        // cause a frame error on input 0.  A
                           // freeze should occur before input 0
                           // receives a second packet and the
                           // second packet is addressed for output 2

join
#2000 write_errs_contr(0,`SPEED);
end_sequence;
```

Figure A-9: test052.code file

```
00000000000000001
00000000000010000
00000000000000001
00000000001000000
00000000100000000
11000000000000000
00000000000000000
xxxxxxxxxxxxxxxxx

00000000000000000
```

Figure A-10: `test052.chk.errs` file

one report of the errors, control, and free buffer information, which is generated by the line `write_errs_contr(0,'SPEED)` in the file `test052.code`. The "0" argument indicates that the error counters should be read non-destructively, and "`'SPEED`" is the familiar quick mode flag. Each report contains 8 binary words. The first is simply a header. The next four are the error counters for each port, in order, with the bits arranged as specified in the Arctic User's Manual [5]. These are followed by the first half and then the last half of the value stored in Arctic's control register. The last line contains the free buffer counters, also encoded as specified in the Arctic User's Manual. In this file, the final line consists only of "`x`" entries, because free buffer information is not important to the test. As before, the last line of the file consists entirely of zeros.

Tests like this one are very precise, but they do take a very long time to generate. This is one of the prime motivations for the creation of a randomized tester presented in Chapter 5.

A.4 JTAG

The JTAG test features in Arctic are some of the most difficult to test because they are so unlike any other function of the chip. They are included in Arctic because of the JTAG standard which was formed by the Joint Test Action Group in hopes that

VLSI designers would give their chips a common set of connectivity test functions. These functions, defined in IEEE standard 1149.1 [4], test whether or not integrated circuits are connected properly by their system boards. Each chip has a boundary scan ring with one register for each of the chip's pins. These registers can force values onto output wires, capture values from input wires, and have their values scanned in and out via a Test Access Port. Arctic has a boundary scan ring and is capable of performing most of these functions through its Test Access Port, which we have been calling its maintenance interface.

These functions are important, but they are not commonly used during normal operation of the chip. Building complex functions into the functional testing system to test these functions, therefore, seems a bit wasteful. Unfortunately, they are hard to test by brute force using Verilog commands in a `.code` file, because they require coordination between at least 175 of Arctic's pins. For this reason, we compromised our testing standards for JTAG functions, and wrote special procedures that execute them, but do not verify that they complete correctly. This way, the user can let the system worry about executing the JTAG functions, but still needs to watch the output pins of Arctic to verify that the functions are working properly.

As an example, the `extest(vector)` procedure can be placed in a `.code` file, and cause the system to carry out an `Extest` function with the given vector. The vector has one bit for each register in the boundary scan ring and is split up, at the beginning of the function, into bits corresponding to input pins and bits corresponding to output pins. The next step is to force the input values onto Arctic's input wires and begin the `Extest` function by capturing input values. Then, a vector that contains the appropriate output bits is scanned into Arctic, and at the end of the test, the user must verify that the correct input bits were captured and scanned out of Arctic, and the correct output bits are being output from the appropriate output pins. This seems like quite a task for the user, but it can be done fairly easily for some simple input patterns, such as all 0's, 1's, or checkerboard patterns.

A.5 Randomly Inserted Packets

In the system I have described, all packets are sent at the specific time that is listed in the file `testxxx.a.pkts`, but as we have noted, this can be a very cumbersome way to list packets. Sometimes, a set of packets needs to be sent into Arctic, but the user does not care what the packets are and would rather not have to think about it. One possibility would be to use a set of packets that was created for another test. This could result in the same packets being used for every test, however, which would not be wise since subtle bugs that are not triggered by that set of packets could creep into the system.

Some kind of randomized generation of packets is needed to take this burden off the user and guarantee that a wide variety of packets are being sent through the system. Our first attempt to solve this problem was to create programs that generated packet libraries and packet insertion files randomly. This works well up to a point, but since it is very hard to determine exactly when packets will leave Arctic without running a simulation, it is very hard to predict when buffers will free so that new packets can be transmitted to Arctic. Remember that each packet insertion command must specify exactly when each packet is transmitted.

To solve this problem, we introduced another mode of operation to the testing system, randomized insertion mode. In this mode, each stub keeps a count of the number of free buffers in the input port it is connected to, just as an Arctic output port would. With this ability, the stubs can handle flow control themselves and send packets as fast as possible. This mode is chosen with the last argument to `start_sequence`, which is `TIMED` if the original insertion method is used, and `PACKED` if this newer method is used.

`test032` is the only test we created which uses this mode, and the file `test032.code` appears in Figure A-11. A library of random packets and a random packet insertion file were generated with utility programs. This was a useful test because it sent a large number of widely varied packets through Arctic, but even before it was completed, we realized that it was still very cumbersome and did not greatly simplify test

```
start_sequence("../sequences/test032/test032.config",
               "../sequences/test032/test032.log.pkts",
               "../sequences/test032/test032.log.stats",
               "../sequences/test032/test032.log.errs",'SPEED,'PACKED);
in.send_packets("../sequences/test032/test032.a.pkts");
lclk_delay_cycles(2450);
end_sequence;
```

Figure A-11: `test032.code` file

generation in general. The use of packet library and insertion file generation programs was awkward and did not allow us to generate an unlimited number of random tests. This was another motivation for building the random tester described in Chapter 5.

A.6 Packet Ordering

Arctic guarantees that all packets entering port A and leaving port B will be sent out in the order received, for any pair of ports A and B. This was not the case in the original design, however. Early designs of Arctic sent out packets "mostly in order," a specification that allowed for simpler scheduling algorithms but still guaranteed no packet was trapped in Arctic for an "unreasonable" amount of time. Such a vague specification is very difficult to test, but we attempted to address the problem in earlier versions of Arctic by adding some unusual abilities for checking the order of packets.

Up to now, we have not discussed how this functional testing system checks the timestamp for each received packet recorded in the `testxxx.log.pkts` files. This is because there were no facilities for checking this field until we addressed this problem. We decided that each received packet should be given a rank in the check file that would determine when the packet should arrive. This rank would be stored in the part of the received packet check file that corresponded to the time of receipt in the log file. Instead of checking the exact time of receipt, the system would check that all packets of rank 1 were received before any packet of rank 2 and so on. This would

make it possible to determine the order of packets, either individually or for very large groups. In addition, a rank of 0 would indicate that a packet could be received at any time, giving the user the ability to mark some packets "don't care" if the order of these packets was unimportant.

These ranks were used in `test030` to test the ordering of packets. In this test, 80 packets were sent into Arctic in 10 groups of 8. The first group was given rank 1, the second group rank 2, and so on, as can be seen in the abbreviated version of the `test030.chk.pkts` file in Figure A-12. While it was not the case that all packets had to emerge exactly in order, each packet of one group had to be received before any packet of another group. If the test did not generate an error, then it could be said that Arctic was working well. If some packets did leave Arctic out of order and cause an error, however, the send times and receive times of each of these packets would have to be checked to make sure that they did not stay in Arctic for an "unreasonable" amount of time.

This was imprecise solution to an imprecise problem. The test was not by any means guaranteed to find every bug in the output ports' scheduling algorithm, but it did give some idea of how well the system was working. This was a very troublesome test, however, because every packet that was a potential problem had to be tracked down. Fortunately, the specifications for Arctic were later changed, and packets delivered from any point to any other point were guaranteed to be kept in first-in-first-out order, which is a much easier ordering to test. We decided to implement routine checks for this ordering in our randomized tester.

This long list of tests has been included to give some impression of the huge amount of effort it takes to test a chip as large and complex as Arctic. The wide variety of features necessitates a very flexible system, which should help to explain why the user is given such precise control over this system. For those readers desiring even more detail about Arctic's testing system, Appendix B contains the system's user's manual, which presents every detail about the system that the user could possibly need.

```
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000001
048111e80100000002
048111e80100000002
048111e80100000002
048111e80100000002
048111e80100000002
048111e80100000002
048111e80100000002
048111e80100000002
.
.
.

e48111e87100000008
e48111e87100000008
e48111e87100000008
e48111e87100000008
e48111e87100000008
e48111e87100000008
e48111e87100000008
e48111e87100000008
e40111e87100000009
e40111e87100000009
e40111e87100000009
e40111e87100000009
e40111e87100000009
e40111e87100000009
e40111e87100000009
e40111e87100000009
1f0000000000000000
```

Figure A-12: `test030.chk.pkts` file

Appendix B

The User's Manual for Arctic's Functional Testing System

This manual was first prepared before Arctic's functional testing system was implemented, and it has been updated over the life of the project. Its purpose is to define all the details of the original testing system that a user might ever need to know. The first few sections give an overview of the system and describe exactly how to run it. The later sections explain every detail of how to manage data structures and how to format the test files. Note that this manual does not cover the randomized testing system at all. Also note that "test groups" were originally called "test sequences," which is why the word "sequence" is used so often in this manual.

The original document begins with a short introduction and the following list of sections. From this point on, very little has changed from the original document.

1. Directories

2. General Structure

3. Running the System

4. Data Structures

5. Files

- Packet Library
- Packet List File
- Packet Log File and Packet Check File
- Stats Log File and Stats Check File
- Errs&Control Log File and Errs&Control Check File
- Configure File

B.1 Directories

The following directories are needed by this testing system.

`run_test` will contain all the top level descriptions of tests (each in a file), and will contain the program `run_test` that will run the test specified in a file when given that file as an argument.

`test_drv` will contain the files for our current version of the testing system.

`arctic` will contain any files from the current release of Arctic that have been modified to make the system work.

`sequences` will contain a directory for every test sequence.

`specs` will contain this file and any other documentation for the system.

`packets` will contain the Packet Libraries, which will hold all the packets we need for the test. This directory will also hold some programs used to generate those packets.

B.2 General Structure

In this testing system, the Arctic module will be instantiated once, and several modules will be created to connect to it. One module will be connected to each input port

and will be capable of doing anything necessary to insert packets, cause link errors, and generate an independent clock. One module will be connected to each output port and will be capable of detecting link errors, receiving packets, and storing information about received packets in a log file. It will also check received packets against the packet sent to make sure the packet was not changed (if it was changed, the packet will be stored in a log file). Another module will be connected to the maintenance interface which will facilitate communication with Arctic's configuration/control bits and test rings. The top level module will contain Arctic and connections to all these modules. It will also contain a list of test sequences.

Testing will be accomplished by running a number of test sequences. Each sequence will test a few functions of the chip, and, hopefully, nearly all functions of the chip will be tested thoroughly by at least one of these test sequences. At the end of every simulation (i.e. every set of sequences) the errors detected during that simulation will be written to a file, `Simulation_Errs_File` (specified in the `test_template.h` file).

Each sequence will be divided into three parts:

- Start Sequence
- Run Sequence
- End Sequence

Starting the test sequence is done with the function

```
start_sequence("configuration_file_name", "packets_log_file_name",
               "stats_log_file_name", "errs_contr_log_file_name",
               <quick>, <fast_ins>)
```

which will do two things. First, it will put Arctic in configuration mode and set it up according to what is specified in the file `"configuration_file_name"`, or it will skip configuration altogether if `"configuration_file_name"` has a value of zero. The `<quick>` argument is a flag that will "cheat," i.e. configure Arctic without using the maintenance interface, if set to 1. Secondly, the task opens all the files to which

data will be written during the test sequence. These files will be accessed with three Verilog "Multi Channel Descriptors."

`packets_log_mcd` – the stream packets log information will be stored to, under the name `"packets_log_file_name"`

`stats_log_mcd` – the stream to which stats log information will be stored, under the name `"stats_log_file_name"`

`errs_log_mcd` – the stream to which errors, control, and buffer free log information will be stored, under `"errs_contr_log_file_name"`

There will actually be three other MCDs.

`packet_err_mcd` – the stream changed packets will be written to. This has the file name `Packet_Err_File`, specified in `test_template.h`. There is one of these per simulation (not per sequence).

`sim_err_mcd` – the stream error messages are written to. This has the file name `Simulation_Errs_file`, also specified in `test_template.h`. There is one of these per simulation (not per sequence).

`time_data_mcd` – the stream time_data is written to. If Arctic is inserting packets in `PACKED` rather than `TIMED` mode, information about when each packet is inserted into the system is kept here. There is one of these per test sequence, stored in the sequence's directory under the name `time_data`.

The `<fast_ins>` bit configures the system to insert as fast as it can, or based on timing information given in the packet insertion file. Each packet insertion command begins with a number specifying when the packet is to be inserted into the system. If `<fast_ins>` is set to 1, the system will ignore these times and insert packets as fast as it can, in the order they appear in the insertion file.

Running the test will involve alternating run and store procedures. A number of actions will take place, and then the current state of the system will be stored. After

that, new actions can take place and the current state will be appended to the files created before.

"Actions" refer to any data or information sent into the chip after configuration. Each action in this system is performed with a Verilog function call. Some of these actions have prefixes such as "`mn`" or "`in`" because they are defined within a Verilog module by that name. At this time, there are seven different kinds of actions:

`in.send_packets("packet_list_file_name")` will insert all the packets listed in `"packet_list_file_name"` into the specified Arctic port at the specified time.

`lclk_delay_cycles(<cycles>)` will cause the system to do nothing for `<cycles>` clock cycles.

`mn.write_control_reg(<value>,<time>,<quick>)` will change the Arctic control register to contain the value `<value>`. The `<time>` is the number of cycles after being called that the changing process will begin, and `<quick>` again is set to 1 if the maintenance interface will not be used.

`mn.write_reset_reg(<value>,<time>,<quick>)` will write `<value>` to the reset register `<time>` cycles after being called. If `<quick>` is 1, the maintenance interface will not be used.

`mn.change_mode(<value>,<time>,<quick>)` will change Arctic's mode to `<value>`. The change will start `<time>` cycles after being called and if `<quick>` is 1 the maintenance interface will not be used.

`mn.clear_stats(<quick>)` will write to the location that clears Arctic's statistics. If `<quick>` is 1, the maintenance interface will not be used.

`mn.extest(<vector>)` will carry out the JTAG `Extest` function with the given vector. Those parts of the vector corresponding to input pins will be applied at the inputs. After the input is captured, those values should be at the input pin registers. After the vector has been scanned in, the outputs of the chip should have the values in those registers connected to the outputs.

`mn.intest_start(<cycles>)` will setup the test system to have all inputs and outputs of the chip run through the Arctic's version of the JTAG `Intest` function. This should be slow as molasses. The `<cycles>` argument tells the system how long after an `Intest` starts that it should be stopped. During an `Intest`, packet sending and receiving will work as before but until the `Intest` stops, no other instructions can be given to the system without majorly weird things happening except for the following instruction.

`mn.intest_control(<value>,<cycles>)` This function, designed to be used with `intest_start`, will put `<value>` in Arctic's control register `<cycles>` cycles after the beginning of the `Intest`. To execute this function, it must appear after an `intest_start` instruction, and both it and the `intest_start` instruction must appear between a `fork/join` pair.

`mn.bypass(<vector>,<time>)` scans the given 32 bit vector through Arctic's bypass register. The instruction will start `<time>` cycles after being called.

`mn.simple_llm_test()` This function puts the chip in low level test mode, scans out the current state, scans it back in, and resumes chip functions. If everything works correctly, the chip should not even notice that anything has happened.

Storing state will be done with one of two functions; one stores statistics information, and the other stores errors, control and any other information. The received packets must also be stored, but that is done without making an explicit call to a function. A log entry is is also stored for every buffer free signal received from every input port. This is also done automatically. Each output port module continually stores a Packets Log File entry for every packet received. Also, any packet received is checked against the packet that was sent into the system to make sure it emerged unadulterated. If the packet was modified, the packet itself is stored in another log file. Each packet will have a text header giving a little information about when the packet was received. There will be no further specification of this file, because it will not be read again by the system.

`write_stats(<clear>,<quick>)` will store all current statistics information in the file `stats_log_file_name` (given above as an argument to `start_sequence`). If `<clear>` is set to 1, all statistics will be cleared when they are read. Again, `<quick>` allows the stats to be read without using the maintenance interface.

`write_errs_contr(<clear>,<quick>)` will store all current errors and control information and the buffer free counters in the file `errs_contr_log_file_name` (given above). If `<clear>` is set to 1, the errors are cleared when they are read. Again, `<quick>` allows the errors and control info to be read without using the maintenance interface.

To end the current sequence, call the function `end_sequence`. This function will close the necessary log files (packets, stats, and errs), and will compare all these logs with corresponding "check" files that are created before the test is run. There is a check file corresponding to each log file. Each piece of information in the log files must have a corresponding piece in the check file or at least a "don't care" field must appear in an appropriate place.

If any log file does not correspond exactly with the appropriate check file (ignoring don't care fields), error messages explaining the error will be printed to the screen and to a file, `Simulation_Errs_File` (which is specified in the `test_template.h` header). When errors are detected, the simulation may halt. If `HALT_ON_ERR` (also defined in `test_template.h`) is set to 1, execution of the simulation will halt at the end of a sequence when errors are detected. If it is 0, the simulation will continue.

Also note that this `HALT_ON_ERR` bit determines whether or not the system will halt when link errors are detected. These errors can occur at any time, not just at the end of a sequence. Error messages caused by link errors will also be written to `Simulation_Errs_File`.

`end_sequence()` will close the log files (Packets Log File, Stats Log File, and Errs&Control Log File) and check them against all the check files (assumed to have the same name as the logs, but with a `.chk` instead of a `.log` extension). If any check finds that the two files do not correlate, execution will halt.

One additional function remains. `Stop_sim` ends the simulation.

`stop_sim()` must be called after an `end_sequence` and before another `start_sequence`. It will close any other unopened files and stop the simulation, taking it to interactive mode.

Hopefully, this structure will be sufficiently general to test most of the chip. It will continue to be updated. One thing not mentioned is the function that will load all the packets in the Packets Library and how information in that file will be accessed. This is addressed in Section 4 of this document.

B.3 Running The System

Executing the command `run_test <filename>` (where `run_test` is found in the `run_test` directory and `<filename>` is the name of a test description file) will start the system running whatever tests are specified in `<filename>`. The system will stop when all tests are finished or an error is found. The test description files will have the following format.

There are two parts to the file. The first part is a list of Packet Library Files. Each Packet Library that is to be used in the test needs to appear in this file. These libraries must appear as filenames given relative to the `test_drv` directory. After all packet libraries are given, the keyword `sequences` must appear. This word must be at the beginning of a line, and it must be on the line by itself. An arbitrarily long list of test sequences will follow. Please note, while the system allows sequences to be run without specifying Configure Files (if the user wants to run a sequence without re-configuring), the system will not be able to run unless the first sequence in the list configures Arctic.

Each test sequence is specified by number as in `test001` or `test999` or any other number. We have specified how the sequences will appear on disk. The program `run_test` will enter each test sequence directory and splice the appropriate code into the right location in `test_drv.v`, and then run the simulation. The code for each test sequence is assumed to be in the file `test***.code` (i.e. for the test sequence

test004, the code should be in /sequences/test004/test004.code). The format for a test sequence is given above.

Just to add a little sugar, all white space will be treated equally in the test description files, so the actual text format does not matter. Just list the packet libraries, then write the word sequences, and then list all the sequences. Also, on every line, all text following the symbol "#" will be treated as a comment and will be ignored.

There is a single command line option. The -b switch tells the system to use a compiled and back annotated description of the chip rather than a Verilog behavioral description.

B.4 Data Structures

As has been stated, the packets will be read into an array where they can be accessed by any part of the system. At the beginning of the simulation the function read_in_packets will read the packets contained in the packets files into an array called packet_arr[]. this is an array of 16 bit words. to find where the data for a particular packet begins, use the packet's 32-bit identifier and the function get_packet_adder(<packet_id>) in the following way.

```
integer      packet_word_addr, packet_id;
reg [15:0]   packet_word0, packet_word1;

//Assume packet_id has been set to some known value.
//Remember integers function as 32 bit registers in Verilog

packet_word_addr = get_packet_addr(packet_id);
packet_word0 = packet_arr[packet_word_addr];
packet_word1 = packet_arr[packet_word_addr+1];
```

In this way, all words in a packet can be accessed. It is up to the module or task calling **get_packet_addr** to keep track of how long the desired packet is. **get_packet_addr** will return 0 if the given **<packet_id>** is invalid.

B.5 Files

This lists all the files needed for each test sequence and the format each will be stored in. All numbers are written left to right, most significant bit to least significant bit. The symbol `[31:0]`, for example, refers to a 32 bit number where the left most bits are the most significant bits and the right most bits are the least significant bits.

At many places in this section the file names themselves will be specified. Usually, these are of the form **test*.<ext>** where "*" is an identifier unique to each test sequence and "**<ext>**" is some extension. Here, we specify that this unique identifier, "*" must be a three digit decimal number ranging from 000 to 999. This is required because the length of the filenames (in characters) must be kept constant.

B.5.1 Packet Library

These files will contain all the packets to be used in the system. Everything will be stored in textfiles as hexadecimal numbers. The file will begin with two 32-bit numbers, each split into two 16-bit parts followed by newlines. The first number indicates how many 16-bit words are contained in the file (including the header) and the second indicates how many packets are in the file.

After these two words come the packets themselves. Each packet begins with a newline followed by a 32-bit packet identifier (also split into two 16 bit parts followed by a newline) and then the 16 bit words of the packet are given one by one with a newline after each one. Each new packet follows immediately after the previous one. No restriction is placed on the what form the packet itself must take, except that the first 2 data words (the first 32-bit word in the payload) must be the 32-bit identifier given before. Finally, after the last packet, a 32-bit trailer will follow. This trailer is also split into 16 bit parts. It is guaranteed to be different from every possible packet

identifier. The format is further detailed below.

It must be stated that the specified fields in the packet identifier are used only to aid the programmer in differentiating between packets. The only requirement on the packet identifiers that is needed to make the program work is that all identifiers must be unique each time the program is compiled.

Following is the exact format of this file with each type of word defined exactly.

```
[31:16] header 0 \n
[15:0]  header 0 \n
[31:16] header 1 \n
[15:0]  header 1 \n
\n (added for readability)
[31:16] packet identifier  \n
[15:0]  packet identifier  \n
[15:0]  packet word 0 (uproute) \n
[15:0]  packet word 1 (downroute) \n
[15:0]  packet word 2 (partid & len) \n
[31:16] packet identifier (first data word) \n
[15:0]  packet identifier (second data word) \n
[15:0]  third data word \n
.
.
.
(next packet)
\n
[31:16] packet identifier  \n
[15:0]  packet identifier  \n
[15:0]  packet word 0 (uproute) \n
.
.
```

```
.
(after last packet)
[31:16] trailer \n
[15:0]  trailer \n


header0 [31:0]
        [31:0]  - Total number of 16-bit words in the file


header1 [31:0]
        [31:0]  - Total number of packets stored in the file


packet_identifier [31:0]
        [31]    - Unused.  Should be set to 0 unless needed for some
                        reason
        [30:29] - Output port number (0-3) out of which the packet is
                        to leave (or 0 if not needed)
        [28:24] - The size of the whole packet in # of 32 bit words.
                        Note that this is not the number of 16 bit
                        words.  The length field is kept this way to
                        be consistent with the length field in the
                        actual packet.
        [23]    - 1 iff the packet is a priority packet
        [22]    - 1 iff the packet should generate a crc error
        [21]    - 1 iff the packet should generate a length error
        [20:15] - Identifier saying which packets file this packet's in
        [14]    - Always 0
        [13:4]  - Identifier saying which configuration file was used
```

```
                to generate this packet (This will be the number of
                the first test sequence where this configuration
                appears).  Alternatively, if this is being used in a
                packet library where this field is not needed, it
                can be used simply as a generic field to
                differentiate packets in any way.
        [3]     - 0 if this packet was designed to go into a
                      particular input port.
                  1 otherwise
        [2]     - Unused.  Should be set to 0 unless needed for some
                      reason
        [1:0]   - The input port this packet is meant to be sent into
                      if one needs to be specified, or 0 otherwise.


trailer [31:0]
        [31:0]  - Always one value (1f00_0000)
```

B.5.2 Packet List File

These files contain the lists of packet identifiers telling the system when and where it inserts packets. There will probably be at least one for every test sequence. The name of this file will be `test*.#.pkts` where "*" is a number unique to each test sequence and "#" is a letter (a-z) separating this list from others in the same sequence.

This file is also stored as a sequence of hexadecimal numbers, starting with a 68-bit header, a newline, and a list of 68-bit words (each followed by a newline), one for every packet to be inserted into the system. Packet insertion commands *must* be listed in order of increasing time.

```
[67:0] header \n
[67:0] packet insertion command \n
[67:0] packet insertion command \n
[67:0] packet insertion command \n
.
.
.
```

```
header [67:0]

        [67:32] - Always zero
        [31:0]  - The number of packets to be inserted into the system
```

```
packet insertion command [67:0]

        [67:36] - Time (in # of cycles) after starting that the packet
                        should be sent in.
        [35:4]  - Packet identifier
        [3]     - Always zero
        [2:1]   - Input Port the packet is inserted into
        [0]     - Always zero
```

B.5.3 Packets Log File and Packets Check File

There will be one Packets Check File and one Packets Log File for each test sequence. Both of these files will also be stored as hexadecimal numbers. Their names will be `test*.log.pkts` and `test*.chk.pkts` where "*" is again a unique number identifying this test sequence. The Packets Log File will be a list of 72-bit words (one for each

received packet or buffer free signal received, each followed by a newline), followed by a 72-bit trailer (guaranteed to be different from every other word in the file).

```
[71:0] - received packet record or buffer free record
[71:0] - received packet record or buffer free record
.
.
.
[71:0] - received packet record or buffer free record
[71:0] - trailer

received packet record [71:0]

        [71:40] - Packet identifier
        [39:38] - Always 0
        [37:36] - Output Port packet was sent from
        [35:33] - Always 0
        [32]    - 1 iff received packet matches the transmitted packet
                        bit for bit
        [31:0]  - Time in # of cycles after the beginning of a sequence
                        that the packet was finished being transmitted
                        from ARCTIC

buffer free record [71:0]

        [71:35] - The number 1e0000000 (37 bits)
        [34]    - Always zero
        [33:32] - The input port the buffer free signal came from
        [31:0]  - Time in # of cycles after the beginning of a sequence
                        that the buffer free signal was given.
```

```
trailer [71:0]

        [71:0]  - The number 1f_0000_0000_0000_0000
```

The corresponding check file has nearly the same format. Major differences are as follows:

1. The buffer free records will not be in the check file. The system will make sure that there are the same number of buffer free records and received packet records, but will make no other use of these records.

2. Received packet records may be out of order and may contain `x`'s in certain fields to indicate that some values are not important. These `x`'s can appear anywhere except bits `[31:0]` which indicate the timing/order of the packet.

3. Bits `[31:0]` of a received packet record will not contain timing information. Instead, this field should hold a number indicating the order the packets are supposed to emerge from each output port. All packets labeled "1" for an output port must emerge before any packets labeled "2" and so on. This gives the ability to check that one group of packets emerges before another. There is one exception. When the number 0 appears, it is assumed the user does not care when the packet emerges (it can come out at any time). It is also important to note that, for each output port, all packets numbered "1" in this fashion should precede those numbered "2" and so forth, and any 0's should appear at the beginning.

The trailer must be in the same place and must be left unchanged.

B.5.4 Stats Log File and Stats Check File

There will be one Stats Check File and one Stats Log File for each test sequence. Their names will be `test*.log.stats` and `test*.chk.stats` where "*" is again

a unique number identifying this test sequence. The stats log file is stored as a textfile with hexadecimal numbers and contains any number of reports of all the statistics information in Arctic. Each report consists of a newline, followed by a 36 bit value set to 1, followed by 24 36-bit words (one corresponding to each piece of statistics information in Arctic, each followed by a newline). These 36-bit words will be arranged starting with output port 0 and moving through to output port 3. The 6 words for each output port will be arranged in the following order: packets, priority, up, down, idle wait. At the end of the last report, there will be a newline followed by a 36 bit 0.

```
\n
[35:0] - header (0_0000_0001) \n
[35:0] - Port 0 packets statistic \n
[35:0] - Port 0 priority statistic \n
[35:0] - Port 0 up statistic \n
.
.
.
[35:0] - Port 3 down statistic \n
[35:0] - Port 3 idle statistic \n
[35:0] - Port 3 wait statistic \n
(If a second report follows..... these lines are added)
\n
[35:0] - header (0_0000_0001) \n
[35:0] - Port 0 packets statistic \n
[35:0] - Port 0 priority statistic \n
.
.
.
\n
```

```
[35:0] - trailer (0_0000_0000) \n
```

Again, the check file is the same format, except that **x**'s may be inserted into fields that are not important.

B.5.5 Errs&Control Log File and Errs&Control Check File

There will be one Errs&Control Check File and one Errs&Control Log File for each test sequence. Their names will be `test*.log.errs` and `test*.chk.errs` where "*" is again a unique number identifying this test sequence. The Errs&Control Log File is stored in a textfile with *binary* numbers. It contains any number reports of Arctic's errors and control information and buffer free counters.

Each report consists of a newline, followed by a 17 bit value set to 1, followed by 4 17-bit words (each corresponding to the errors information for one port and each followed by a newline), followed by two 17-bit words giving the status of the control register (in a weird format), followed by a single 17-bit word holding the buffer free counters.

The 4 error words will be arranged in port order, starting with port 0 and continuing to port 3. The control word format is just plain strange because of the move from an 8-port to a 4-port arctic. The first 2 bits appear at the beginning of the first word and the last 16 appear at the end of the second. The format for the buffer free count word is given in the Arctic user's manual. Since this word is only 16 bits long, we add a 0 in bit location 17. After the last report there is a newline followed by a 17 bit 0.

```
\n
[16:0]   header (0_0000_0000_0000_0001) \n
[16:0]   error word for port 0 \n
[16:0]   error word for port 1 \n
[16:0]   error word for port 2 \n
```

```
[16:0]  error word for port 3 \n
[33:17] control status report \n
[16:0]  control status report \n
[16:0]  buffer free report\n
(If a second report follows..... these lines are added)
\n
[16:0]  header (0_0000_0000_0000_0001) \n
[16:0]  error word for port 0 \n
[16:0]  error word for port 1 \n
.
.
.
\n
[16:0]  trailer (0_0000_0000_0000_0000) \n
```

```
error word [16:0]

        [16:15] - Always set to 0 (unused)
        [14:13] - Buffer free error count
        [12]    - ICLK error
        [11:10] - Route error count
        [9:8]   - Idle error count
        [7:6]   - Length error count
        [5:4]   - Frame error count
        [3:2]   - Overflow error count
        [1:0]   - CRC error count
```

```
control status report [33:0]
```

```
[33:32] - Control Status word bits [17:16]
[31:16] - Always zero
[15:0]  - Control Status word bits [15:0]
```

The format of the control status word is given in the Arctic user's manual as it is too lengthy to detail here.

`buffer free report [16:0]`

```
[16]    - Always set to 0 (unused)
[15:0]  - Buffer Free Counter Word
```

The format of the Buffer Free Counter word is given in the Arctic user's manual as it is too lengthy to detail here.

Again, the check file is the same format, except that `x`'s may be inserted into fields that are not important.

B.5.6 Configure File

This optional file contains all the information necessary to configure an arctic. The file name will be written as `test*.config` where "*" is a number unique to each test sequence. It is stored in ASCII as hexadecimal numbers, and can have one of two formats. Both formats begin with a 32-bit header telling which format the file is in. If the 32-bit header number is 0, then format 0 is used. In this format, configuration information for each port is given in port order (from 0 to 3). For each port, the five configuration words are given in order (0 to 4).

```
[31:0] header (0000_0000) \n
[31:0] Port 0 configuration word 0 \n
[31:0] Port 0 configuration word 1 \n
```

```
[31:0] Port 0 configuration word 2 \n
[31:0] Port 0 configuration word 3 \n
[31:0] Port 0 configuration word 4 \n
[31:0] Port 1 configuration word 0 \n
[31:0] Port 1 configuration word 1 \n
.
.
.
```

A definition of the bits in each of these configuration registers is given in the Arctic user's manual and is too lengthy to detail here.

If the header contains the value 1, then the file contains only 5 configuration words. These five words will be broadcast to the entire chip.

```
[31:0] header (0000_0001) \n
[31:0] configuration word 0 \n
[31:0] configuration word 1 \n
[31:0] configuration word 2 \n
[31:0] configuration word 3 \n
[31:0] configuration word 4 \n
```

Bibliography

[1] Boughton, G.A. *Arctic Routing Chip* CSG Memo 373. MIT Laboratory for Computer Science. April. 1995. *In Proceedings of the 1994 University of Washington Parallel Computer Routing and Communication Workshop. May. 1994.*

[2] Clark, D.W. "Large-Scale Hardware Simulation: Modeling and Verification Strategies" *Proc. 25th Anniversary Symposium.* Computer Science Department, Carnegie-Mellon University, September. 1990.

[3] Wood, D.A. et al. "Verifying a Multiprocessor Cache Controller Using Random Test Generation" *IEEE Design and Test of Computers.* August. 1990.

[4] Test Technology Technical Committee of the IEEE Computer Society. *IEEE Standard Test Access Port and Boundary-Scan Architecture.* Std. 1149.1-1990. New York: IEEE, 1990.

[5] Boughton, G.A. et al. *Arctic User's Manual.* CSG Memo 353. MIT Laboratory for Computer Science. February. 1994.

[6] Hall, Anthony. "Seven Myths of Formal Methods" *IEEE Software.* September. 1990.

[7] Kubiatowicz, John. Personal Interview. May 10. 1995