The Automatic Self Test Analysis (ASTA) tools are an integrated part of ICL's DA-X system. ASTA analyses a design for self-testability, automatically generates chip test data, calculates signatures and outputs tester format data. Tests for primitive logic will be exhaustive or quasi-exhaustive, but for larger cells, specific TestMethods can be defined using an ASTA language. The system, which is Esprit-funded, has been used successfully for two complex ASIC chips.

**ABSTRACT**

The Automatic Self Test Analysis (ASTA) tools are an integrated part of ICL's DA-X system. ASTA analyses a design for self-testability, automatically generates chip test data, calculates signatures and outputs tester format data. Tests for primitive logic will be exhaustive or quasi-exhaustive, but for larger cells, specific TestMethods can be defined using an ASTA language. The system, which is Esprit-funded, has been used successfully for two complex ASIC chips.

**INTRODUCTION**

BIST as a test methodology is becoming a more widespread solution to the testing issue for complex ASIC chips [1]. However, there are few tools available today to support the automatic generation of self-test data.

This paper describes a set of integrated, automatic BIST generation tools used in the Mainframe Systems division of ICL. The system, ASTA, has been used successfully to assist the production of manufacturing test data.

ASTA is part of the in-house design automation system, DA-X [3] [6], which supports capture, design rule checks, timing analysis tools, a multilevel simulator, ATPG, automatic layout tools, and links to manufacturing - all encompassed within a comprehensive data management environment. The software runs on ICL mainframes and Sun workstations. DA-X is used for many technologies of gate array and VLSI chips, and for PCB and system design.

The DA-X tools are largely data-driven and so are not limited to any one technology or design methodology.

The ASTA system uses two basic techniques to automatically generate chip test data. Areas of primitive logic are grouped together by the use of a trace algorithm, which tests them with exhaustive or quasi-exhaustive patterns. Larger, or complex cells are treated independently by the analysis of specific test methods.

The early development of ASTA was supported by the UK Alvey programme. Over the last four years it has been part of the Esprit AIDA project.

**THE BIST ARCHITECTURE**

The ICL BIST chip architecture consists of an on-chip diagnostic unit which provides a simple tester interface, and controls, via that interface, up to 31 loops constructed of ICL test registers [2]. The test registers have four basic modes of operation:

User mode - the normal functional mode.

Diagnostic Hold mode - providing fixed contents.

Diagnostic Shift mode - providing access via the scan loop mechanism.

Test mode - which can perform one of four functions: generate pseudo-random patterns, collect a signature, generate circular shift patterns, hold a pattern.

To improve the testability of a circuit the designer can insert out-of-line registers to create 'testable units'. This is achieved by the use of 'fences' (see fig 1), a fence being a combination of a multiplexor and a test register arranged so that the register can, when in self-test mode, be used either to provide patterns or collect a signature.

The feedback connections for the registers can be used to form single or multiple register LFSR configurations [2].

**THE CELL LIBRARY**

To support the DA-X design system there are a number of cell libraries, one for each technology. The definition of a
cell, and in fact every unit of design, is held in a set of objects. The objects of interest to test are:

**SYMBOL** - this defines the drawing shape, properties and interfaces of the unit of design.

**NETWORK** - If the unit has a lower level structure then this describes the structure in terms of a network of Symbol instances.

**FUNCTION** - This describes those functions of the unit that are relevant to test, eg: transparent paths.

**TESTMETHOD** - This describes one or more methods of testing the unit.

**BEHAVIOUR** - The behavioural description used for simulation.

In addition, Technology objects and a TestSpec object are used to define certain rules and properties common to the particular technology and test style.

### DATA

**Schematic Data (Symbols & Networks)**

The design of a chip is captured in Symbols and Networks where Symbols describe the visible interface of a unit of design and Networks describe the connections between different lower level units of design. This hierarchical structure is flattened for self-test by using a network compiler which reads the top level network and expands all the symbols and networks at each level of hierarchy below it.

**Technology**

A technology object is used to contain data which is dependent only on the technology of a chip. Self-test uses a technology object to identify the registers and clocked combinatorial logic (to differentiate it from sequential logic). The power logic levels and the maximum length of the loops containing the registers are also defined here.

**TestSpec**

The TestMethods and Functions used in self-test are written in terms of constants, functions and procedures specific to a certain test style or project. It is therefore important that constants, functions and procedures used for one testmethod have the same meaning when used in other testmethods and functions.

To ensure this all constants, functions and procedures are defined in a central location, the TestSpec object (see fig 2).

Constants define integers, patterns or strings which will be substituted for the constant name wherever it may be used.

Functions define processes which will be applied to ports.

Procedures define processes expected from the test registers.

The function and procedure definitions also state the number and types of parameters they can accept, ie: Pattern, Value or String.

**TestMethod**

It is necessary to describe how to test each cell used on a chip in terms of the pattern sets which can be produced by its LFSRs. This is written in a test method language and stored in a Test Method object. Any unit of design can have a TestMethod:

* The basic cells and paracells have a handwritten TestMethod containing one or more alternative ways of testing them.

* The top level chip network will have a TestMethod generated from the base level TestMethods (see TestMethod Analysis and Generation) and/or directly from the network (see Trace).

The body of the TestMethod consists of two optional blocks: Initialisation and TestMethod. Some cells, eg: RAMs, may require initialisation before testing can commence - the initialisation block contains the steps necessary to do this.

The TestMethod block may occur any number of times, each occurrence will describe an alternative way of testing a unit. It consists of any number of test blocks each of which will describe a test which must be applied together with any other tests in the current test method block. They will be applied in the stated order and if any of them fail then the complete test method is deemed to have failed.

The tests consist of a sequence of port assignments which specify the functions required to test the unit (see fig 3).

**Function**

All primitive units of design will have Functions. The body of the Function consists of Assertions, which describe how to propagate test patterns through a unit (see fig 4), and Definitions, which
describe how to make the LFSRs collect/generate the patterns (see fig 5). Definitions use three procedures: Load, Beats, and Unload, which describe how the LFSR can provide the patterns.

The Function also contains a trace block which has two sections TRACEFROM and TRACETHROUGH. These control the starting points and progress of the Trace command through the units.

**TOOLS**

**Trace**

This command generates a TestMethod directly from the Network by breaking it down into areas of logic which can be tested exhaustively or quasi-exhaustively. Patterns are applied to their inputs and signatures collected from their outputs.

It takes a list of signature collection points and, by tracing back through the logic, finds the pattern generating points. Signature collection and pattern generation may be at registers or fence multiplexors. The output is in the form of a report and a TestMethod object.

The command traces backwards from each TRACEFROM port found in the TRACE block of the Function for each fence multiplexor and register. It only passes through instances which are not fence multiplexors, not registers, and not sequential. The route it takes through an instance is determined by the TRACETHROUGH section found in the TRACE block of the Function for the instance.

**TestMethod Analysis**

This is the stage where a TestMethod is analysed against the Network to determine its testability. This involves searching through the network, examining alternative routes from the unit being tested, to the LFSRs to see if the requirements of the TestMethod can be satisfied.

Each unit of design in a Network has a TestMethod which describes how to test it in terms of what is required on its ports. In order to test these units, LFSRs have to be found which can drive these ports with the correct data via other units in the network. This is achieved by propagating the test requirements through the network from the unit to be tested through other units until the LFSRs are reached. As each unit is encountered a path through it is found by reference to the assertions in its Function.

Analysis can be used in one of two ways: as the next stage after Trace, or as an alternative to Trace.

**Analysis of a Trace TestMethod**

The trace stops at the pattern generating and signature collection points which are not necessarily the LFSRs, they may be the fence multiplexors, so the generated TestMethod must be analysed and the test requirements propagated back to the LFSRs.

**Analysis of an Instance TestMethod**

Alternative to Trace: Trace tests the network by using Random or Exhaustive patterns. To restrict the size of the tests to about one million beats [2] the number of inputs to a testable unit is limited to 20 inputs. Some testable units identified by Trace may break this limit and if the Network cannot be altered to solve this problem, the TestMethods for each instance in this area can be analysed instead.

Trace is faster than Analysis as it requires less functional knowledge of the chip's units of design so it is usually used first to test as much of the chip as possible.

A Report command is available to document the success or failure of the analysis.

**Function Generation**

A Function for the chip is generated containing:

* LFSR data describing the construction of LFSRs from individual registers.
* A loop map describing the placement of the test registers within the loops.
* The information describing how LFSRs are used to generate and collect bit patterns (EXHAUSTIVE, RANDOM, SIGNATURE etc) at specific registers.

Before generating the Function, these LFSRs are identified and checked to ensure the on-chip registers have been correctly formed into LFSRs using the correct feedback connections. It is also checked that they are connected correctly to the loops and the loops themselves are correctly linked to the diagnostic unit.

**TestMethod Generation**

After successfully tracing and analysing the chip's network to ensure that all of the chip's logic is testable, a TestMethod for the chip is generated from the data produced by the full analysis. This TestMethod contains all the tests derived for each testable unit on the chip as well as a section describing the necessary initialisation of the chip.

**Test Merging**

The individual tests within the TestMethod set LFSRs to hold, generate or analyse
mode and preload with bit patterns (seeds), execute a number of clock beats and check the collected signature values against their expected values. As signatures may be produced by up to a million clock beats, their expected values can only be derived by a logic simulation of the chip. With most chip designs, a large number of these tests are produced, several of which may span upwards of a million clock beats. If these tests are performed serially then the simulation time required to generate a full set of verification signatures may not be feasible in terms of resource and cost. With this in mind a merge command is used to bring together tests which can be run simultaneously and thus reduce the overall number of tests. The priority of the algorithm is to first overlay tests which use a large number of clock beats.

Signature Generation

The generation of the set of test signatures is accomplished by simulating the full test sequence on our logic simulator [5] using a model of the chips network and the test information held in its TestMethod and Function. It is essential that the simulation model of the chip is initialised to a known state before any individual tests are simulated. Exact sequences are followed for each test. Loops are loaded with specified bit patterns which result in each LFSR register in the loop being set to the required mode. The diagnostic unit counter is set to the required number of clock beats. The simulator then executes these beats and the loops are unloaded to obtain the collected signatures. The TestMethod is then updated with the signature values.

Test Data Converters

A suite of bridging commands is available for translating the TestMethod into appropriate tester data formats. Parametric tests are added to the BIST tests to enable complete testing of the manufactured chip.

DESIGN ROUTE

This section of the paper outlines the development route used by the chip designer to check the testability of his design and to produce the chip test data.

First the loop and LFSR checks are performed, these ensure that the register loops and LFSRs have been correctly constructed. If these checks are successful, the chip level function is generated. The next step is to run the Analysis and Report commands on the chip input/output pad instances. The TestMethod for an input pad requires that a signature can be collected from its output. The TestMethod for an output pad requires that random patterns can be generated on its input. By running Analysis on these instances it is checked that input/output logic has been correctly isolated ('fenced') from the chip internal logic and its only tested in isolation from the rest of the chip.

Next, the Analysis and Report commands are run on all memory cell test instances. This will ensure that the memory cells can be individually tested. Our test methodology does not allow for a mixture of combinational and sequential logic to be tested together. All memory elements must be BIST registers or regular, structured memories such as RAMs, register files etc.

The combinational logic is then checked. The Trace command is run in order to identify testable units which can be tested by exhaustive or quasi-exhaustive tests.

The Trace command will reject any testable unit with more than 20 inputs and give an error report. For the remaining units it will write a TestMethod. The Analysis command can then be run on this level TestMethod to check that the testability rules have been obeyed. For instance, it is not permitted to generate patterns and collect a signature in the same LFSR, if exhaustive patterns are being used the generating LFSR must be no longer than 20 bits etc. Any fails will be detailed in the report.

The next stage is to run Analysis on individual symbol instances which are not covered by Trace. Such symbols may be for large paracells with predefined TestMethods. For example, parity trees, equality/magnitude comparators, arithmetic units may all be tested with a random subset of patterns even when they have too many inputs for exhaustive testing. TestMethods may also be available for large regular units of design which have been implemented by the individual designers.

In order to fully test the chip, all symbol instances must be covered either by running Analysis directly on the symbol TestMethod, as for memories, large paracells etc, or by including the instances in a testable unit which is exhaustively or quasi-exhaustively tested using the Trace command.
The use of the above commands will identify errors in loops, LFSRs and untestable areas of logic. The chip designer can correct these modifications to the logic design and/or developing new TestMethods for particular symbols. This checking process can then be repeated until the design is error-free and testable.

When the design is correct, the commands are re-run using the GenerateTestMethod command instead of Report, this will produce the chip level TestMethod. The Merge command is then run on this TestMethod and signature generation performed to calculate the signatures for each of the tests. The resulting data is then converted to appropriate tester data formats.

**RESULTS**

The facilities described have been used on the Macrolan chip [4]. This is a 35K gate, 256 pin device designed using a paracell based semi-custom design methodology in a CMOS technology. The final test data required 80K test vectors (excluding clock bursts) and used a total of 4 million clock beats. Signature generation required 358 hours of computer time. It was possible to check the testability of the complete design and generate the test data in approximately 12 hours of computer time (excluding signature generation).

A second chip of lower complexity has also been designed. Self-test data has been automatically generated successfully.

**CONCLUSION**

The facilities described in this paper provide an integrated set of tools for verifying the design of chips with BIST and generating test data. The tools have been proven on the design of two complex ASICs.

**REFERENCES**


**ACKNOWLEDGEMENTS**

This work has been supported under Esprit Project 888: Advanced Integrated Design Automation (AIDA) and we would like to thank our partners in the project, Siemens and SGS-Thomson Microelectronics for their contribution.

This project has been supported by several groups within ICL, these were the Macrolan design team, the LSI Applications team and the Design Automation Self-test team. Particular thanks are due to Monica Brady, Steve Clarke, and Maureen Norris.
This TestMethod demonstrates the use of two alternative test method blocks and the use of two tests within a test method which must both work for the test method to be successful.

**TEST BLOCK example**

**TEST METHOD A, STUCK FAULTS = 100**

TEST part 1 BY

A1 = Hold(0) II
A2 = Exhaustive II
A3 = Exhaustive II
X = Signature

END TEST

TEST part 2 BY

A1 = Hold(1) II
A2 = Exhaustive II
A3 = Exhaustive II
X = Signature

END TEST

END TEST METHOD

END TEST BLOCK

**Fig 3 - TestMethod example**

**FUNCTIONAL DESCRIPTION MUX127**

TRADE

TRACE


END TRACE

ASSERTION O IS A IF

SL = Hold(0) [ Transparency ]

END ASSERTION

ASSERTION O IS B IF

SL = Hold(1) [ Transparency ]

END ASSERTION

END DESCRIPTION

**Fig 4 - Function example - Multiplexer**

**FUNCTIONAL DESCRIPTION REG18**

REGISTER [ Defines which ports are connected ]

3 E -> ES [ to which register bits ]
2 T
18 U -> D & S

END REGISTER

TRACE

TRACE FROM D

END TRACE

DEFINITION S = Random(VALUE: MinBeats) IF

Load(GenerateMode, NonZeroSeed) ;
Beats(SelfTestMode, >= MinBeats)

END DEFINITION

DEFINITION S = Hold(PATTERN: ThePattern) IF

Load(HoldMode, ThePattern) ;
Beats(SelfTestMode, >= 1)

END DEFINITION

DEFINITION D = Signature(VALUE: TheSignature) IF

Load(AnalyzeMode, Signature) ;
Beats(SelfTestMode, >= 1)

UnLoad(TheSignature)

END DEFINITION

END DESCRIPTION

**Fig 5 - Function example - Register**

If Port D bit 0 of Reg3 is traced back to Reg1 and Reg2 the test the trace will produce is :-

TEST Trace
Reg1: S[1:4] &
Reg2: S = Random(2:6) ||
Reg3: D[0] = Signature

END TEST

**Fig 6 - Trace example**