Turbo Tester Reference Manual

Version 02.10

Department of Computer Engineering

Tallinn Technical University

Estonia

October 2002

Restrictions

License Terms

If you publish any results obtained with the Software, you should acknowledge the following paper in the list of references.

"G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. A CAD System for Teaching Digital Test. Proc. of the 2nd European Workshop on Microelectronics Education, Kluwer Academic Publishers, pp. 287-290, Noordwijkerhout, the Netherlands, May 14-15, 1998."

Termination

Table of Contents

1 Introduction

Audience

We assume that the reader of this manual has the basic knowledge in the field of digital test and design. The document provides for a reference of the Turbo Tester tools. Theoretical aspects are covered only for the features that are specific to our system.

Conventions

Throughout the document following conventions are used:

❑ `monospace`	Indicates strings entered at the command prompt or displayed by the system. Also file names.
❑ *italics*	Indicates variables, e.g. numeric parameters, working directories, design names.
❑ <*chevrons*>	Same as previous.
❑ [square brackets]	Denotes optional constructs.
❑ **!**	Emphasizes important notes.

Getting Help

If you encounter problems while installing or using the Turbo Tester system, please contact the Turbo Tester support group by sending an e-mail to `tt@pld.ttu.ee`. Any suggestions and remarks will be appreciated.

2 Installing Turbo Tester

Win NT

In order to install Turbo Tester under Windows NT operating system, copy the file `ttv0210.zip` to the directory you want to set up the system (refered to as *installation_directory*) and unzip the file.

Add the following line to your `autoexec.bat` file:

```
SET PATH=%PATH%;installation_directory\TT\bin
```

Subsequently restart the computer.

Linux and Solaris 2.X

In order to install Turbo Tester under Solaris 2.X operating system, copy the file `ttv0210.tar.gz` to the directory you want to set up the system (refered to as *installation_directory*) and enter the following commands:

```
gunzip ttv0210.tar.gz
```

```
tar xvf ttv0210.tar
```

Add the following line to your `.cshrc` file:

```
setenv    PATH installation_directory/TT/bin:$PATH
```

Subsequently type

```
source ~/.cshrc
```

or log out and log in again.

Directory Structure

Turbo Tester installation contains the following directories:

❑ `doc`	user documentation
❑ `lib`	technology library files for the EDIF interface
❑ `examples`	sample SSBDD models of ISCAS85 benchmarks
❑ `bin`	executables

3 System Overview

3.1 Data Flow of Test Tools

Figure 1 presents the data flow of the Turbo Tester system. The system consists of tools for Automatic Test Pattern Generation (ATPG), Built-In Self-Test (BIST) emulation, fault simulation, test set optimization and multi-valued simulation. Different methods for test pattern generation, simulation and BIST emulation can be applied. Test pattern generators and fault simulators are available for both, sequential and combinational circuits. Combinational circuits can be tested by deterministic, random and genetic algorithm based methods, while for sequential designs a random ATPG is available. In addition to fault simulators, the simulation tools include multi-valued simulation for hazard analysis in combinational circuits. For BIST emulation, BILBO (Built-In Logic Block Observer) and CSTP (Circular Self-Test Path) approaches can be chosen. Test set optimization tool implementing static compaction of generated tests and a report generator displaying statistics about the results of different tasks are also included to the installation.

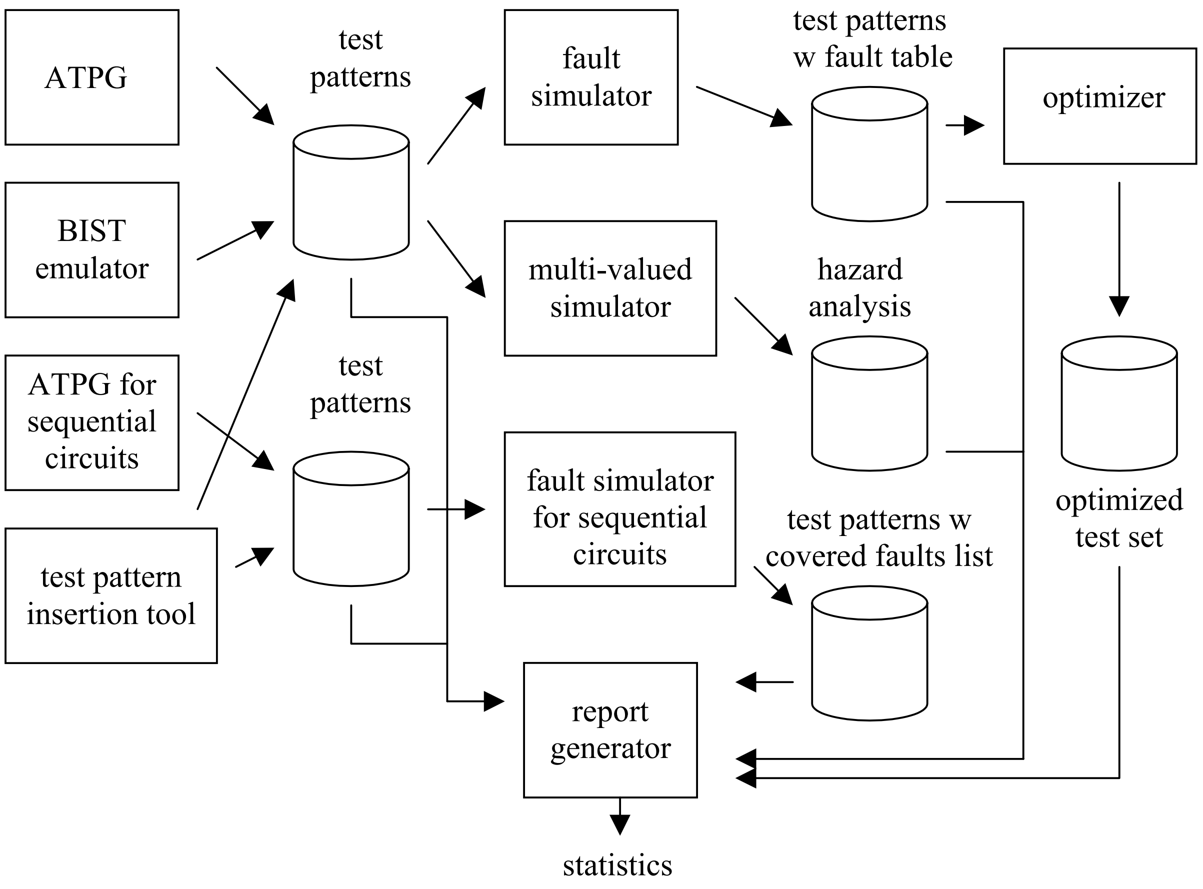
ATPG
BIST emulator
ATPG for sequential circuits
test pattern insertion tool
test patterns
test patterns
fault simulator
multi-valued simulator
fault simulator for sequential circuits
report generator
statistics
test patterns w fault table
hazard analysis
test patterns w covered faults list
optimizer
optimized test set

Figure 1 System Flow of Turbo Tester

3.2 Tools for Fault Diagnosis

In addition to the test generation and fault simulation tools, Turbo Tester package includes tools for design error localization and diagnosis. A subset of these tools can be used for solving real-world diagnostics tasks, while others are dedicated to a special laboratory course on design error diagnosis. More detailed information about the diagnostics tools is available in Chapter 5.

3.3 Design Representation

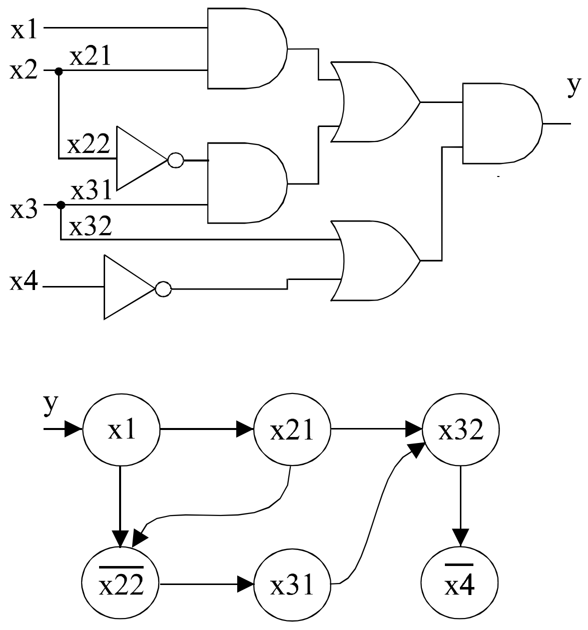

Figure 3 Combinational Circuit and Its SSBDD

All the tools in Turbo Tester system operate on the design model of Structurally Synthesized Binary Decision Diagrams (SSBDD). SSBDDs are a special case of Binary Decision Diagrams (BDD). However, unlike BDDs, they are capable of representing gate-level structural faults.

SSBDD models for combinational circuits can be synthesized by a simple superposition procedure. We generate an SSBDD for a circuit output by starting from the output, substituting recursively all the gates by their respective elementary BDDs until primary inputs are reached. In order to avoid repetitive occurences of subgraphs in the model, the recursion is terminated in fanout stems and SSBDDs can be synthesized for each primary output and fanout point separately. In that case the circuit will be described as a system where for each fanout-free region an SSBDD corresponds.

As an example, Figure 3 shows an SSBDD representation for a combinational circuit. For simplicity, values of variables on branches are omitted (by convention, the rightward branch corresponds to 1 and the downward branch to 0). In addition, terminal nodes with constants 0 and 1 are omitted (exiting the SSBDD rightwards corresponds to y = 1, and downwards to y = 0). The diagram contains 6 nodes whereas each of them represents a signal path in the circuit. The node labels in the SSBDD correspond to input branches of the circuit shown in Figure 3.

The worst case complexity of SSBDD synthesis by the superposition procedure is linear in respect to the number of logic gates in the model and is therefore fast even for very large circuits.

In Turbo Tester, two possible options for generating SSBDDs exist:

1. Macro-level SSBDDs
2. Gate-level SSBDDs

In the case of gate-level SSBDDs, each gate is represented by corresponding BDD and the circuit model does not differ from ordinary gate-level model.

! *All the Turbo Tester tools run slower on the gate-level model than on the macro-level. Therefore, in order to achieve higher performance, macro-level model should be chosen. However, if non-collapsed gate-level fault coverage is required, gate-level model can be selected.*

3.4 Design Interface

In Figure 4 the design interface of Turbo Tester system is presented. It is an EDIF 2.0.0 netlist interface, which links the system to most of the commercial VLSI CAD tools: Synopsys, Cadence, Mentor Graphics, ViewLogic, Compass, ASYL+, OrCAD, etc. Since the EDIF format does not include information of the behavior of the library cells, the interface requires corresponding target technology library file. This can be generated from an easy-to-describe library source format by Turbo Tester library generator. A set of pre-generated technology libraries are also provided with the installation.

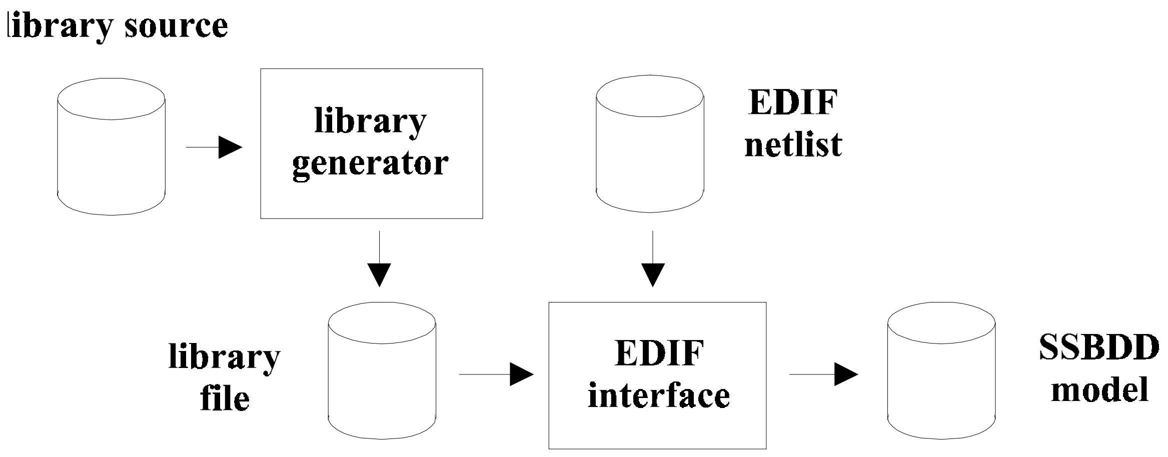

Figure 4 Turbo Tester Design Interface

In addition to EDIF 2.0.0 input, the interface supports the ISCAS'89 benchmark format. EDIF and ISCAS netlists can be converted into following optional file formats:

- SSBDD model
- Turbo Tester technology library
- ISCAS'85
- ISCAS'89

4 Turbo Tester Commands

4.1 Design Interface

Netlist Interface

command: `import`

Converts EDIF 2.0.0 or ISCAS'89 netlists into the following file formats:

- SSBDD model
- Turbo Tester technology library
- ISCAS'85
- ISCAS'89

input: EDIF or ISCAS'89 netlist file

output: SSBDD model file (.agm), technology library file or ISCAS'85 model file (.cir) or ISCAS'89 model file (.bench + file `LineNames`).

syntax: `import` *[options] <EDIF file> <library file*>*

options:

`-read_iscas89`	Read ISCAS'89 format.
`-gate_level`	Generate gate-level SSBDD model. Default output is macro-level SSBDD.
`-lib_cell`	Generate TT technology library format. (Do not use with `-read_iscas89` option)
`-iscas89`	Generate ISCAS'89 format. (Do not use with `-read_iscas89` option)
`-iscas85`	Generate ISCAS'85 format. (Do not use with `-read_iscas89` option)
`-tool` *<application>*	Options for *application* are `orcad` and `cadence`.
`-gnd` *<gnd name>*	*gnd name* is the name of the GND net.
`-vdd` *<vdd name>*	*vdd name* is the name of the VDD net.

! * - library_file *must be omitted when selecting `-read_iscas89` option.*

Error Messages and Warnings

The design interface displays error messages and warnings about possible failures while parsing the EDIF netlist file. Line numbers of the input netlist file where errors occurred are provided. Pay attention to ALL of the displayed errors and warnings if the program terminates abnormally!

Note that some of the warnings should be ignored, however. For example, one of the most common warnings during reading a hierarchical design is as follows:

```
Design: COMPONENT_1
Parsing macro

Warning at line 1917: Gate not in library
```

This warning shows that there exists a cell named *COMPONENT_1* in the EDIF description. It either means that *COMPONENT_1* is a block in the hierarchy, or it is a cell, which is not specified in the technology library. In the first case, the warning should be ignored. In the latter case, EDIF parsing fails due to an incompletely specified technology library and the cell *COMPONENT_1* should be included to the library.

Library Generator

command: `libgen`

Generates user-specified technology libraries for the EDIF interface.

input: library source file (See ‘Library Source Format’).

output: technology library file (.lib)

syntax: `libgen` *<library source> <library file>*

4.2 Test Pattern Generation

Common Features

All of the Turbo Tester Automated Test Pattern Generators (ATPG) and the sequential fault simulator are capable of reading information about already detected faults from a test pattern file. This means that the ATPGs of the system can be run in an arbitrary sequence, where one ATPG generates test patterns covering a set of faults, and another test pattern generator continues to target the faults not detected by the previous one. In addition, this feature makes it possible to select a set of manually inserted, or functional, test patterns, fault simulate them, and feed the data about detected faults to an ATPG.

The ATPGs of Turbo Tester include the `–infile` <*file*> option, where *file* is the name of the test pattern file specified in TT test pattern format.

Note that, in order to obtain the information about covered faults, test patterns must be fault simulated prior to feeding them to an ATPG using the –infile option.

Deterministic Test Pattern Generator

command: `generate`

Generates deterministic tests for combinational circuits implementing the PODEM algorithm.

input: SSBDD model file (.agm)

output: test pattern file (.tst), list of redundant faults (.red)

syntax: `generate` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-backtracks` *<number>*	Maximal number of backtracks. Default is 10.
`-test_per_fault`	Generates test for every fault in the circuit. Preserves don't care values.
`-vector_limit` *<limit>*	Maximal number of generated patterns. Default is 1000.
`-fault_table`	Perform fault simulation for the final patterns.
`-infile` *<file>*	Read data about covered faults from test patterns file *file*. (See '4.2 Common Features').

Genetic Algorithm for Test Pattern Generation

1) Representation

In a genetic framework, one possible solution to the problem is called an *individual*. Similarly as we have different persons in society, we also have different solutions to a problem (one is more optimal than the others). All individuals together form a *population*. In the context of test generation, test vector will be the individual and the set of test vectors will correspond to population.

2) Initialization

Initially, a random set of test vectors is generated. Subsequently, this set is given to a simulator tool for evaluation. Following steps of algorithm are carried out repeatedly.

3) Evaluation of test vectors

Evaluation is used to measure the *fitness* of the individuals, i.e. the quality of solutions, in a population. Better solutions will get a higher score. Evaluation function is supposed to direct the population towards progress because "good" solutions (with high score) will be selected during selection process and "poor" solutions will be rejected.

We use fault simulation in order to evaluate the test vectors. Test vectors fitness value will be equal to the number of previously undetected faults that the vector covers. The best vector in the population is determined and added to the selected test vector depository. The depository consists of test vectors that will form the final test set.

4) Fitness scaling

As a population converges on a definitive solution, the difference between fitness values may become very small. Best solutions can not have significant advantage in reproductive selection. We use square values for test vector's fitness values in order to differentiate "good" and "bad" test vectors.

5) Selection of candidate vectors

Selection is needed for finding two candidates for *crossover*. Based on quality measures (fitness values), better test vectors in a test set are selected. *Roulette wheel* selection mechanism is used here. Number of slots on the roulette wheel will be equal to population size. Size of the roulette wheel slots is proportional to the fitness value of the test vectors. That means that better test vectors have a greater possibility to be selected. Assuming that our population size is N, and N is an even number, we have N/2 pairs for reproduction. Candidates in pair will be determined by running roulette wheel twice. One run will determine one candidate. With such a selection scheme it can happen that same candidate is selected two times, i.e. reproduction with itself is possible. This means the selected vector is a good test vector and it carries its good genetic potential into new generation.

6) Crossover

Exchanging corresponding genetic material from two parents allow useful genes on different parents to be combined in their offspring. Crossover is the key to genetic algorithm's power. Most successful parents reproduce more often. Beneficial properties of two parents combine.

From pair of candidate vectors selected by roulette wheel mechanism, two new test vectors are produced by one-point crossover as follows:

1) we determine a random position *m* in a test vector by generating a random number between *1* and *L*, where L is the number of bits in the test vector

2) first *m* bits from the first candidate vector are copied to the first child vector

3) first *m* bits from the second candidate vector are copied to the second child vector

4) bits $m + 1 \ldots L$ from the first candidate vector are copied to the second child vector (into bits $m + 1 \ldots L$)

5) bits $m + 1 \ldots L$ from the second candidate vector are copied to the first child vector (into bits $m + 1 \ldots L$)

7) Mutation

In order to encourage genetic algorithm to explore new regions in space of all possible test vectors, we apply *mutation* operator to the test vectors produced by crossover. In all the test vectors, every bit is inverted with a certain probability *p*. Random mutation provides background variation and occasionally introduces beneficial individuals. Without the mutation all the individuals in population will sooner or later be the same. We will be stuck in a local maximum and there will be no progress anymore.

Steps 2 –5 are repeated until all the faults from the fault list are detected or a predefined limit of evolutionary generations is exceeded. Test generation terminates also when the number of noncontributing populations exceeds a certain value that can be set by the user.

In current implementation, the test generation works in two stages, with different mutation rates.

1) In the first stage, when there are many undetected faults and fitness of vectors is mostly greater than zero (in each evolutionary generation many faults are detected), a smaller mutation rate is used (Default $p = 0.1$).

2) In the second stage, when there are only few undetected faults and none of the vectors in population detects these faults, the fitness values of the vectors will all be zeros. We can not say which vector is actually better than others. Now the mutation rate is increased (Default $p = 0.5$) to bring more diversity into population, and permit to explore new areas of the search space.

Genetic ATPG Command-Line

command: `genetic`

Generates tests for combinational circuits basing on fault simulation and string manipulation (genetic operators).

input: SSBDD model file (.agm)

output: test pattern file (.tst)

syntax: `genetic` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-mutation_rate1` *<float>*	Initial mutation rate. Default is 0.1. Recommended range 0.01 … 0.1.
`-popul_size` *<number>*	Number of vectors in a population. Must be an even value! Default is 32.
`-timeout` *<number>*	Timeout value. The greater the value the more thoroughly we search. Default timeout is 10. Recommended range 10 … 100.
`-mutation_rate2` *<float>*	Mutation rate applied when fitness value becomes zero. Default is 0.5. Recommended range 0.1 … 0.5.
`-max_generations` *<number>*	Maximum number of generations. Default is 1000. Suggested range 1 … 10,000.
`-infile` *<file>*	Read data about covered faults from test patterns file *file*. (See '4.2 Common Features').

Random Test Pattern Generator

command: `random`

A random test pattern generator for combinational circuits. Generates random patterns in packages of 32 vectors. Best vectors (covering bigger number of previously undetected faults) are included to the final test set.

input: SSBDD model file (.agm)

output: test pattern file (.tst)

syntax: `random` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-failure_limit` *<limit>*	Maximal number of consecutive failed packages. Default is 64.
`-pack_size` *<size>*	The number of vectors in a package is *size* multiplied by 32. Default for *size* is 1.
`-criterion` *<faults>*	Vectors detecting more previously undetected faults than specified by *faults* will be selected. Default for *faults* is 1.
`-packages` *<packages>*	Maximal number of packages to be simulated. Default is 1000.
`-select_max` *<vectors>*	Maximal number of vectors selected from a package. Default is 32.
`-fault_table`	Perform fault simulation for the final patterns.
`-infile` *<file>*	Read data about covered faults from test patterns file *file*. (See ‘4.2 Common Features’).

Random ATPG for Sequential Circuits

command: `sbgen`

Random test pattern generator for sequential circuits. Generates and fault simulates random test sequences. The user can specify the number of test vectors in the sequences (`-sequence_length` option). Test generation will terminate automatically when certain number of subsequent test sequences fail to detect any new faults. However, minimum and maximum limits for the number of simulated sequences can be set by the user (`-min_simulations` and `-max_simulations` options, respectively).

input: SSBDD model file (.agm)

output: test pattern file (.tst)

syntax: `sbgen` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-reset_index` <*index*>	Index of the primary input variable corresponding to global reset in the SSBDD model.
`-sequence_length` <*seq*>	*seq* is the number of vectors in each test sequence. Default is 128.
`-max_simulations` <*max*>	Maximum number of simulated sequences. Default is 1000.
`-min_simulations` <*max*>	Minimum number of simulated sequences. Default is 0.
`-fault_table`	Perform fault simulation for the final patterns.
`-infile` <*file*>	Read data about covered faults from test patterns file *file*. (See '4.2 Common Features').

4.3 Built-In Self-Test Emulation

Built-In Self Test Architectures

Turbo Tester supports two types of BIST (Built-In Self-Test) architectures: BILBO (Built-In Logic Block Observer) and CSTP (Circular Self-Test Path). In BILBO the vector generator and signature analyzer are separate LFSRs (Linear Feedback Shift Registers), while CSTP uses a single shift register. The advantage of the BILBO approach lies in the fact that the generated pseudo-random vectors do not depend on the circuit output values. On the other hand, CSTP requires less overhead in terms of silicon area, since only one LFSR is required. The two types of BIST architectures are shown in Figure 5.

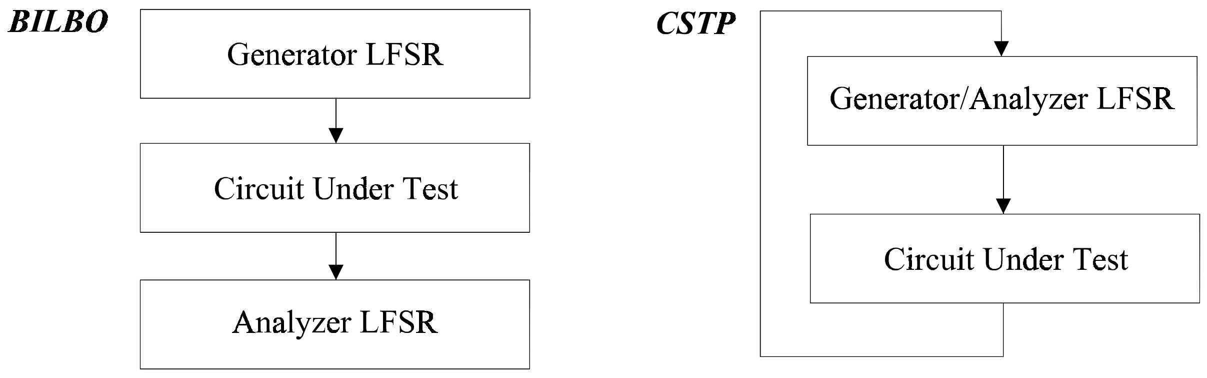

Figure 5 Built-In Self-Test Architectures

Turbo Tester BIST emulators allow the user to specify the feedback polynomials, initial states and bitwidths of the LFSRs. Figure 6 explains these concepts on a generator LFSR example. The generator has an initial value of 011011 in the register and its feedback polynomial is 101101.

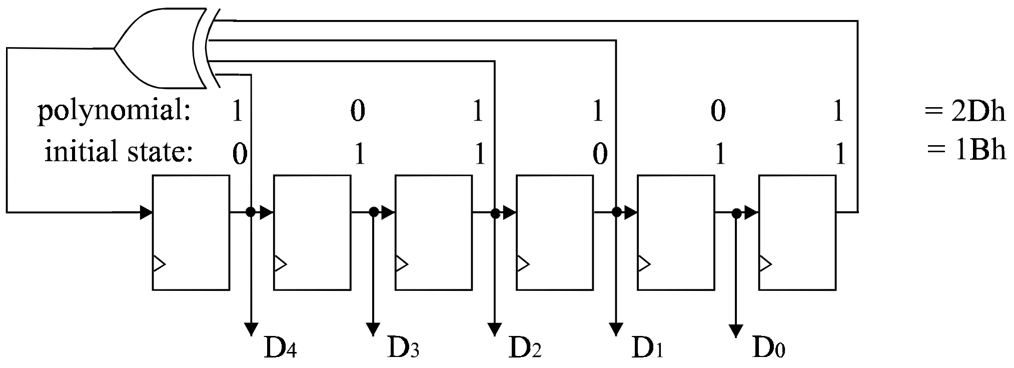
polynomial: 1 0 1 1 0 1 = 2Dh
initial state: 0 1 1 0 1 1 = 1Bh
D4
D3
D2
D1
D0

Figure 6 Generator LFSR

! *In the example, the LFSR bitwidth (6 bits) is greater than the number of circuit inputs (5 inputs). In that case the circuit inputs are connected to the side of more significant bits of the generator LFSR (See Figure 6). Generator bitwidth has to be greater or equal to the number of circuit inputs.*

Figure 7 presents an example of the analyzer LFSR with an initial state of 10110 and a feedback polynomial of 11001.

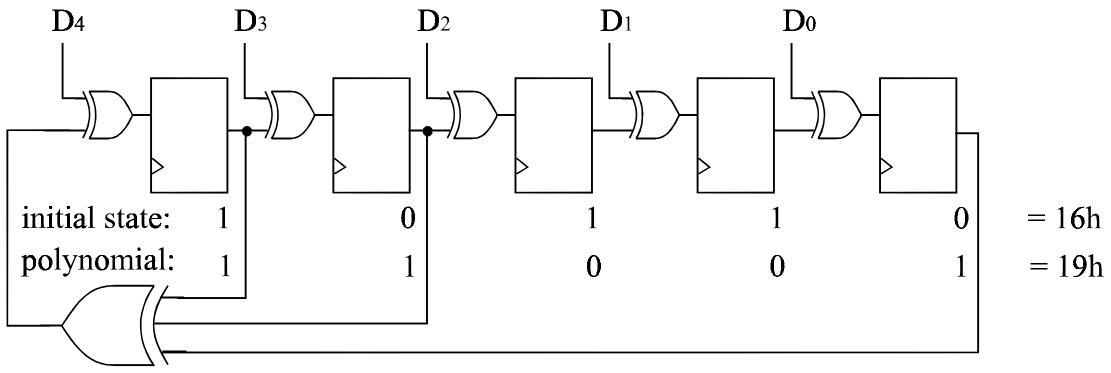

Figure 7 Analyzer LFSR

! *In the case of analyzer, if the LFSR bitwidth is greater than the number of circuit outputs then the outputs are by default connected to the side of more significant bits of the LFSR. However, this can be overridden by the –lsb option. Analyzer bitwidth must be greater or equal to the number of circuit outputs.*

BIST Emulator

command: `bist`

Emulation tool for Built-In Logic Block Observer (BILBO) and Circular Self-Test Path (CSTP) architectures.

input: SSBDD model file (.agm)

output: test pattern file (.tst)

syntax: `bist -rand -glen` <*generator_length*> `[-alen` <*analyzer_length*>`]` [*options*] <*design*>

or

`bist -gpoly` <*generator_poly*> `-ginit` <*generator_init*>

`[-apoly` <*analyzer_poly*> `-ainit` <*analyzer_init*>`]` [*options*] <*design*>

generator_length:	Length of the generator LFSR in bits. (Use only with `-rand` option!).
analyzer_length:	Length of the analyzer LFSR in bits. (Use only with `-rand` and `-simul bilbo` options!).
generator_poly:	Feedback polynomial of the generator LFSR in binary digits. (Do not use with `-rand` option!).
generator_init:	Initial value of the generator LFSR in binary digits. (Do not use with `-rand` option!).
analyzer_poly:	Feedback polynomial of the analyzr LFSR in binary digits. (Do not use with `-rand` and `-simul cstp` option!).
analyzer_init:	Initial value of the analyzer LFSR in binary digits. (Do not use with `-rand` and `-simul cstp` option!).
design:	Name of the design file without .agm extension.

options:

`-rand`	Generate random LFSR feedback polynomials and initial states.
`-aliasing`	With this option selected, exact fault coverage values will be reported. BIST emulation will be slower but it will take into account possible fault aliasing in the analyzer LFSR.
`-simul < bilbo | cstp >`	Choses between BILBO and CSTP architectures. (Default is BILBO).
`-count` <*cycles*>	The length of the test in clock cycles. Default is 1000.
`-optimize`	Dismiss test patterns at the end of the test sequence that do not detect any additional faults.
`-lsb`	Design outputs are connected to the side of less significant bits of the analyzer LFSR. (Default is the side of more significant bits).

! *LFSR bitwidth is determined by the number of binary digits in the specified initial state and polynomial. The number of digits in initial state must be equal to the number of digits in polynomial.*

4.4 Fault Simulation

Fault Simulator for Combinational Circuits

command: `analyze`

Stuck-at fault simulator for combinational circuits.

input: SSBDD model file (.agm), test pattern file

output: test patterns with fault table (.tst)

syntax: `analyze` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-extension` *<extension>*	*extension* is the file name extension of the test pattern file. Default extension is `tst`.

Fault Simulator for Sequential Circuits

commands: `sequential`, `sequential_fast`

Stuck-at fault simulators for sequential circuits. `sequential` includes the `-nodrop` option but runs slower than `sequential_fast`.

input: SSBDD model file (.agm), test pattern file (.tst)

output: test pattern file (.tst)

syntax: `sequential` *[options] <design>* |

`sequential_fast` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-fault_free`	Perform fault-free simulation.
`-nodrop`	Don't perform fault dropping. (`sequential` only!).
`-extension` *<extension>*	*extension* is the file name extension of the test pattern file. Default extension is `tst`.
`-infile` *<file>*	Read data about covered faults from test patterns file *file*. (See '4.2 Common Features').

4.5 Multi-Valued Simulation

Hazard Analysis Basing on 5-Valued and 8-Valued Alphabets

In Turbo Tester, multi-valued simulation is applied to model the possible hazards that can occur in logic circuits. In this approach each waveform type has a corresponding symbol of the given alphabet. Turbo Tester's multi-valued simulator implements 5-valued and 8-valued alphabets:

$A_5 = \{0,1,E,H,x\}$,

$A_8 = \{0,1,E,H,o,i,e,h\}$.

The meaning of these symbols is explained in the following:

0 - constant zero

1 - constant one

E - rising transition

H - falling transition

o - static zero hazard

i - static one hazard

e - rising transition hazard

h - falling transition hazard

x - hazard of unspecified

The dynamic behavior of a logic network during one single transition period is the corresponding representative waveform of the output or simply its corresponding logic value. Every logic gate in a network can be regarded as an operator, which computes the output value of the gate if the values of the input variables taken from the set *A* are given. The operators for logic OR, logic AND and INVERSION in the case of five-valued simulation are presented in the following table.

OR	0	1	E	H	x
0	0	1	E	H	x
1	1	1	1	1	1
E	E	1	E	x	x
H	H	1	x	H	x
x	x	1	x	x	x

AND	0	1	E	H	x
0	0	0	0	0	0
1	0	1	E	H	x
E	0	E	E	x	x
H	0	H	x	H	x
x	0	x	x	x	x

NOT	0
0	1
1	0
E	H
H	E
x	x

From this table and its transivity we can compute the logic value of any line in the network.

Multi-Valued Simulator

command: `multival`

Performs multi-valued simulation on 5-valued and 8-valued alphabets.

input: SSBDD model file (.agm), test pattern file
output: hazard analysis (.mvl)

syntax: `multival` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-values` <*alphabet*>	if *alphabet* is `5` then 5-valued simulation is performed (default). `8` selects 8-valued simulation.
`-extension` <*extension*>	*extension* is the file name extension of the test pattern file. Default extension is `tst`.

4.6 Test Set Optimization

command: `optimize`

Minimizes the number of test patterns in the test set by means of static compaction.

input: SSBDD model file (.agm), test pattern file

output: test pattern file of the minimized test set (.tst)

syntax: `optimize` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-extension` *<extension>*	*extension* is the file name extension of the test pattern file. Default extension is `flt`.

4.7 Utilities

Report Generator

command: `report`

Displays various statistics.

input: SSBDD model file (.agm), test pattern file

output: Report file or display

syntax: `report` *[options] <design>*

design:	Name of the design file without .agm extension.
options:	
`-file` *<ReportFile>*	Report is written in a file *ReportFile*. (By default the report is displayed on the screen).
`-coverage`	Shows the fault coverage data.
`-patterns`	Shows the number of test patterns.
`-tested`	Shows the list of tested faults.
`-not_tested`	Shows the list of not tested faults.
`-untestable`	Shows the list of untestable faults. (For deterministic ATPG only).
`-signals`	Shows model variable names and their values at every clock cycle.
`-ports`	Same as previous but includes I/O ports only.
`-table`	Shows the fault table information.
`-cycle` *<t>*	Displays data for clock cycle *t*. Use with *-table*, *-ports* or *-signals* options only.
`-progress`	Shows achieved stuck-at fault coverages after each pattern. (Useful for BIST emulation tools for determining optimal test length in clock cycles).
`-block` *<instance>*	Processes only nodes associated with subinstance *instance* in a hierarchical design.
`-extension` *<extension>*	*extension* is the file name extension of the test pattern file. Default extension is `.tst`.

Test Pattern Insertion Tool

command: `vec_insert`

A utility that allows the user to manually insert test stimuli.

input: test vectors in test pattern insertion format. (File or screen input).
output: test pattern file (.tst).

syntax: `vec_insert` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-file` *<ReportFile>*	Test patterns are read from file *ReportFile*. (By default the patterns are read interactively from the screen).

Test Pattern Insertion Format

Test patterns must be specified as follows. For each primary input variable a row corresponds, where the name of the variable is followed by ':' and logic values (0 or 1) at each time step (pattern) are separated by spaces.

Example

Consider the test pattern insertion file for a circuit with five primary inputs (i_1, i_2, i_3, i_4, i_5) and six test patterns specified.

```
i_1: 0 1 0 1 0 1
i_2: 0 1 1 0 1 0
i_3: 0 1 0 1 1 0
i_4: 0 1 1 0 0 1
i_5: 0 1 0 1 0 1
```

5 Tools for Laboratory Course on Diagnosis

5.1 Introduction

The diagnostic tools were primarily worked out for teaching diagnostics using the design error diagnosis problem as an example. The basic tool in this set is `prediag`. It uses test patterns to compare the specification and the implementation and defines some rough suspected erroneous area within the circuit. The diagnosis refinement is left for students' practicing. They use verification tool (`verify`) which shows the difference in output responses between the specification and the implementation.

The implementation and the specification can be obtained from the initial design using the `xtimport` tool which is an extended version of the design interface (`import`). As the specification should be generated from the same original design, we do it by insertion of an error into the design. The type of the error and its location are saved into the report file and encrypted in order to provide the possibility for the teacher to check the student's work but to prevent the student to obtain this information easily. The teacher can use the decryption tool (`decrypt`) to get the information.

The test patterns for the diagnostic tools can be generated by an ATPG or inserted manually with the test pattern insertion tool (`vecmanager`).

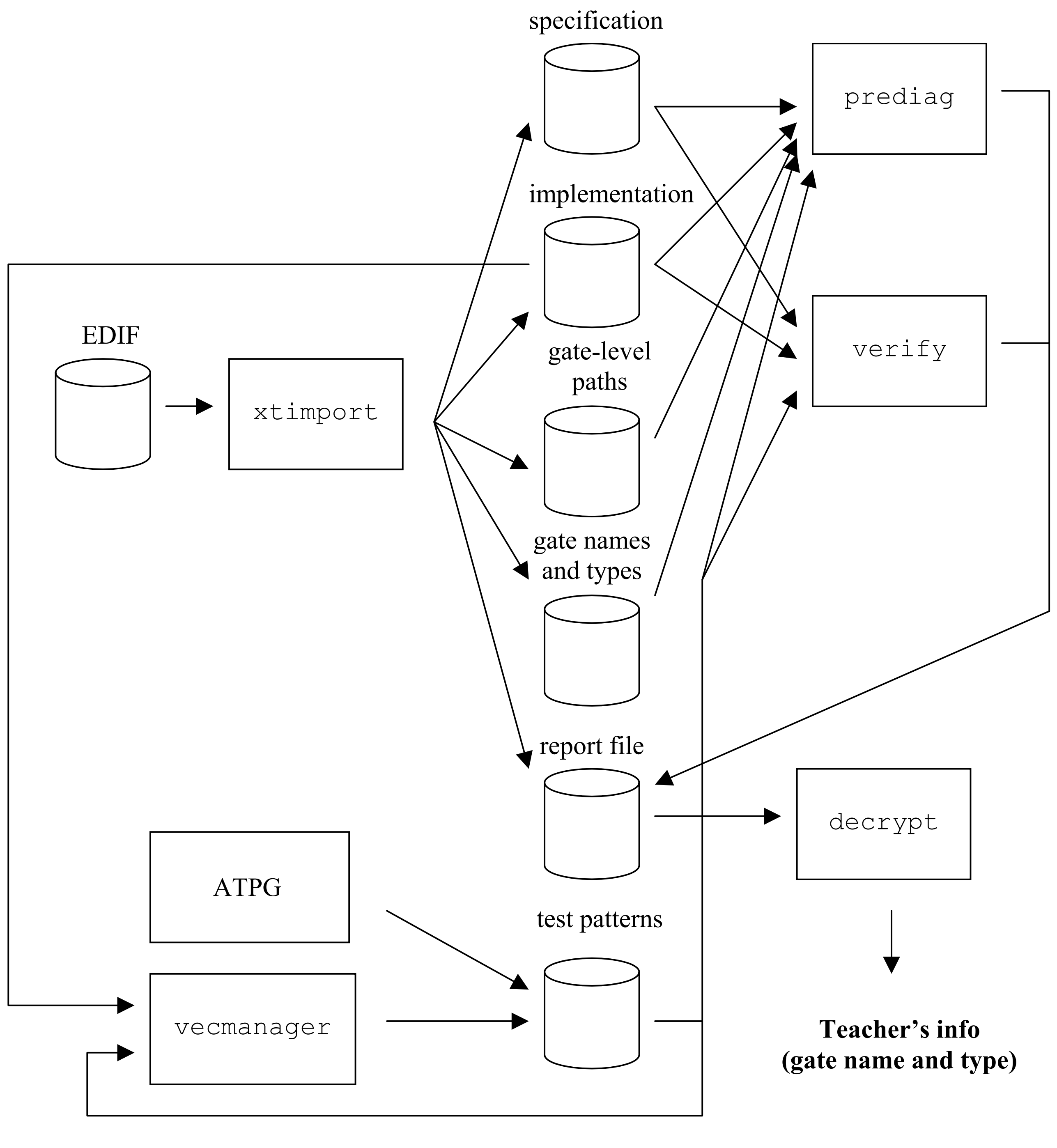

Figure 2 Data Flow of Laboratory Course on Diagnosis

5.2 Fault Diagnosis Commands

xtimport – Extended Netlist Interface

command: `xtimport`

`xtimport` is an extended version of the `import` tool. It has all the functionality of `import` but includes some additional features. It has been designed specially for "Design Error Diagnosis" practical work. It allows to create an implementation and a specification. Where the specification is the same design as implementation but with randomly changed function of a randomly chosen gate. Specification can be created using `-spec` option. Specification is always a gate-level SSBDD model. When creating the specification, the `xtimport` tool creates also a report file (<design>`.rep`), where stores the information about the gate, whose function had been changed as encrypted data. Thus, students do not know which gate is erroneous but the teacher can finally decrypt the information and check the student's work. Some data about gate-level signal paths should be reserved during generation of the implementation in order to allow the prediagnostic (see below) tool work. The data is saved using the flag `-paths`.

input: EDIF or ISCAS’89 netlist file

output: specification (.spec), implementation (.agm), gate-level paths (.gat), gate names and types (.pat), test patterns (.tst).

syntax: `xtimport` *[options] <EDIF file> <library file*>*

options:

`-paths`	Preserve information about gate-level signal paths.
`-spec`	Create a specification (insert a design error).
`-read_iscas89`	Read ISCAS'89 format.
`-gate_level`	Generate gate-level SSBDD model. Default output is macro-level SSBDD.
`-tool` <*application*>	Options for *application* are `orcad` and `cadence`.
`-gnd` <*gnd name*>	*gnd name* is the name of the GND net.
`-vdd` <*vdd name*>	*vdd name* is the name of the VDD net.

! * - library_file *must be omitted when selecting* `-read_iscas89` *option.*

xtimport - Error Messages and Warnings

The design interface displays error messages and warnings about possible failures while parsing the EDIF netlist file. Line numbers of the input netlist file where errors occurred are provided. Pay attention to ALL of the displayed errors and warnings if the program terminates abnormally!

Note that some of the warnings should be ignored, however. For example, one of the most common warnings during reading a hierarchical design is as follows:

```
Design: COMPONENT_1
Parsing macro


Warning at line 1917: Gate not in library
```

This warning shows that there exists a cell named *COMPONENT_1* in the EDIF description. It either means that *COMPONENT_1* is a block in the hierarchy, or it is a cell, which is not specified in the technology library. In the first case, the warning should be ignored. In the latter case, EDIF parsing fails due to an incompletely specified technology library and the cell *COMPONENT_1* should be included to the library.

prediag – Tool for Preliminary Diagnosis

command: `prediag`

The tool for obtaining the intermediate diagnosis. The diagnosis is a set of several gates of the implementation, which are suspected to be faulty. It applies test patterns generated by an ATPG, and detects the differences in output responses between the implementation and the specification. Using this information and the fault table it produces the prediagnostic results. By default it shows the information concerning suspected gates and suspected SSBDD nodes. It is also capable of showing the data about the outputs where the errors were observed. By default it updates the report file (<design>`.rep`) with the chosen information.

input: specification (.spec), implementation (.agm), gate-level paths (.gat), gate names and types (.pat), test patterns (.tst).

output: report file (.rep).

syntax: `prediag` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-f`	Show failing outputs.
`-n`	Do not show suspected nodes.
`-g`	Do not show suspected gates.
`-v` *<extension>*	*extension* is the file name extension of the input test pattern file. Default extension is `tst`.
`-s` *<extension>*	*extension* is the file name extension of the specification model file. Default extension is `spec`.
`-o` *<extension>*	*extension* is the file name extension of the output file. Default extension is `rep`.
`-scr`	Print everything on the screen only.

verify – Tool for Verification

command: `verify`

`verify` compares the outputs of the implementation and the specification and gives the information about the outputs where errors were observed. It applies some test vectors in order to obtain this information. By default it also updates the report file (<design>`.rep`) with this data.

input: specification (.spec), implementation (.agm), test patterns (.tst).

output: report file (.rep).

syntax: `verify` *[options] <design>*

design:	Name of the design file without .agm extension.

options:

`-v` *<extension>*	*extension* is the file name extension of the input test pattern file. Default extension is `tst`.
`-s` *<extension>*	*extension* is the file name extension of the specification model file. Default extension is `spec`.
`-o` *<extension>*	*extension* is the file name extension of the output file. Default extension is `rep`.
`-scr`	Print everything on the screen only.

decrypt – Tool for Checking the Student's Work

command: `decrypt`

The teachers tool is used to check the results of students' work. It decrypts the name of the gate, which was replaced by the `xtimport` tool. It works in several following modes. In the interactive mode (simply type `decrypt`) the encrypted message is read from the screen. The message is a single word of ASCII characters without spaces. In the command line mode (usage: `decrypt -m` <message>) the message in the same format is read from the command line. In these modes the decrypted message is printed to the screen. In the third mode (usage: `decrypt -f` <filename>) the message is read from a text file. The file can contain an arbitrary number of words consisting of ASCII characters. The output in this mode is written into a file.

input: encrypted name and type of the erroneous gate. (Can be read from file <*design*>.rep).

output: decrypted name and type of the erroneous gate.

syntax: `decrypt` [*options*] [`-f` <*InFile*>| `-m` <*message*>]

InFile:	File containing the encrypted message.
message:	String of printable ASCII characters (encrypted message).

options:

`-o` <*file*>	*file* is the output file.
`-s`	Print output also on the screen. (default: if `-f` specified - don't, otherwise - yes).

vecmanager – Tool for Test Pattern Insertion

command: `vecmanager`

`vecmanager` is an interactive tool for manual insertion, deletion or update of input patterns in a test pattern file(s). It can work in an interactive as well as in a command line modes. In interactive mode (usage: vecmanager) the program asks for the name of the design, the name of the test pattern file, and finally gives the following menu:

S. Show existing test patterns
N. Insert completely New test
A. Add vectors
D. Remove some vectors*
R. Automatically geneRate some random sequence
E. Automatically genErate the exhaustive test*
F. PerForm fault simulation
X. Save patterns and eXit

In the command line mode you can add only a single test vector.

input: SSBDD file (.agm), test patterns file (.tst).

output: test patterns file (.tst).

syntax: `vecmanager` *[options] <design>*

design:	Name of the design file without .agm extension.
options:	
`-new`	Create completely new test set.
`-add` *<vector>*	Add test pattern *vector*.
`-i` *<extension>*	*extension* is the file name extension of the input file. Default extension is `tst`.
`-o` *<extension>*	*extension* is the file name extension of the output file. Default extension is `tst`.
`-ftable`	Perform fault simulation.

* - not implemented in current version

Appendixes

Appendix A SSBDD Model Format (.agm)

SSBDD format is a line-based format. The Maximum length of a line is 255 characters. Lines starting with ';' character are considered to be comments.

`'STAT#` <*nodecount*> `Nods,` <*variablecount*> `Vars,` <*graphcount*> `Grps,` <*inputcount*> `Inps,` <*outputcount*> `Outs '`

Reflects the number of nodes, variables, graphs, primary inputs (PI) and primary outputs (PO) of the circuit, respectively.

`'MODE# '` <*mode*>
- Where mode can be one of the following:

`STRUCTURAL` – SSBDD model.
`FUNCTIONAL` – Traditional BDD model. (Not supported in present implementation).

The graph model is described as follows:

{Lines of variables}
{<Line of a variable>
<Line of a graph>
{Lines of nodes}}

Line of a variable in AG model:

`'VAR# '` <*index*> `':' ['VAL =' '0'|'1']-` Where *index* is the index of the variable in the SSBDD model. SSBDD variables have the following ordering:
1. Primary inputs
2. Constants
3. Internal lines (fanouts) or flipflops
4. Primary outputs

`'('` *<flags>* `')'` – Flags. When a flag is not set, an underscore '_' will be placed to its position. At present, six flags are implemented:

`'i'`	variable is an input variable
`'o'`	variable is an output variable
`'c'`	variable is a constant input. In the case of constants, the logic value is determined by the following construct: `'VAL ='  '0'|'1'`
`'d'`	one clock cycle delay at the variable (e.g. flipflops)

`'"'` *<string>* `'"'` – Variable name

Line of a graph:

`'GRP# '` *<index>* `':'` – The index of a graph in the SSBDD model.

`'BEG =` *<index>*`, LEN =` *<length>* `-----'`
- *index* is the global index of the first node in the graph, and *length* is the number of nodes in the graph, respectively.

Line of a node:

`'` *<index1>* *<index2>*`:'` – *index1* shows the global index of the node in the SSBDD model. *index2* is the relative index of the node inside the graph.

`'('` *<flags>* `')'` – Flags. When a flag is not set, an underscore '_' will be placed to its position. At present only one flag is implemented:

`'I'`	Denotes that the node is inverted

`'(`*<successor1>* *< successor2>*`)'` – Relative indexes of the nodes down and right from current node, respectively.

`'V =` *<index>*`'` – Index of the variable weighing the node.

`'"'` *<string>* `'"'` – Node name

Appendix B Test Vectors Format (.tst)

Test patterns file has a line-based format. The length of the lines is not limited. Comments start with `';'` character.

The format has the following syntax:

vectorCount
nodeCount
variableCount
[LFSR]
testPatterns
[faultTable]
[faultList]
[faultCoverage]

vectorCount := `.VECTORS` <*integer*>

integer shows the total number of all patterns.

testPatterns :=
`.PATTERNS`
<*patterns*>

patterns are rows containing the test patterns. Each character in a row represents a variable in the SSBDD model. The order of the variables is the same as in the SSBDD model file (.agm). The following notation is used:

For input variables: `'1'` - logical one, `'0'` - logical zero.
For internal variables: `'h'` - logical one, `'l'` - logical zero.
For output variables: `'H'` - logical one, `'L'` - logical zero.

Internal and output variables can be omitted.

faultTable :=
`.TABLE`
<*table*>

Each row in *table* corresponds to a test pattern and each column to a node in SSBDD model. The order of the nodes is the same as in SSBDD model file (.agm). The following notation is used:

For stuck-at fault model:

`'X'` - no faults detected at the node by the test pattern
`'0'` - stuck-at zero detected
`'1'` - stuck-at one detected

For delay fault model:

`'X'` - no faults detected at the node by the test pattern
`'/'` - rising edge fault detected at the node
`'\'` - falling edge fault detected at the node

faultList `:=`
`.FAULTS`
<faults>

faults is a row of characters. Each character corresponds to a node in SSBDD model. The order of the nodes is the same as in SSBDD model file (.agm). The following notation is used:

For stuck-at fault model:

`'X'` – no faults detected at the node by the test set
`'0'` – stuck-at zero detected
`'1'` – stuck-at one detected
`'&'` – both, sa-0 and sa-1 detected

For delay fault model:

`'X'` – no faults detected at the node by the test set
`'/'` – rising edge fault detected
`'\'` – falling edge fault detected
`'&'` – both, rising edge and falling edge delay faults detected

faultCoverage `:=`
`.COVERAGE`
<numberOfDetectedFaults> `/` *<numberOfFaults>* `=` *<Percentage>* `%`

numberOfDetectedFaults - is an integer indicating the number of faults under consideration.
numberOfFaults - indicates the number of detected faults.
Percentage - is a floating point number, showing the fault coverage.

LFSR :=
generator
[analyzer]

This section is used for describing linear feedback shift-registers in BIST structures.

Generator :=
`.GENERATOR`
initialState
feedbackPolynomial

Used to describe the LFSR in CSTP and generator LFSR in BILBO.

Analyzer :=
`.ANALYZER`
initialState
feedbackPolynomial

Used to describe the analyzer LFSR in BILBO.

initialState :=
`.INITIAL_STATE` *<bitVector>*

Shows the initial state of an LFSR.

feedbackPolynomial :=
`.POLYNOMIAL` *<bitVector>*

Shows the feedback polynomial of an LFSR.

signature :=
`.SIGNATURE` *<bitVector>*

Shows the signature of an LFSR.

Appendix C Library Source Format

The library source format is used to generate custom technology libraries for the EDIF interface. The syntax of the source is as follows.

`GATE` <*cellName*> <*BooleanExpression*> `;`

, where *cellName* is the name of library cell

and *BooleanExpression* is:

<*outputPin*> `=` <*BooleanFunction*>

, where *BooleanFunction* is Boolean function of input pins.

Operators used in Boolean functions in order of priority:

`()`	brackets (for priority),
`!`	inversion,
`*`	conjunction,
`+`	disjunction.

If *BooleanFunction* is `CONST0` or `CONST1`, the library cell will correspond to logical zero or logical one signal.

Example

The following is an example of a source description for a library cell `my_nand2`. It describes a two input nand gate with output pin `y` and input pins `a` and `b`:

```
GATE my_nand2 y=!(a*b);
```

Appendix D Technology Library Format

The technology library is required by the EDIF interface in order to get information about the functionality of current technology library cells. This information is not implicitly described in the EDIF netlist itself. The syntax of the technology library is as follows.

" cellName " numberOfRows

{<Row>}

, where *cellName* is the name of library cell;

numberOfRows is the number of rows in the cell;

and row consists of:

lineNumber name type numberOfOutputs numberOfInputs

[listOfInputs]

, where *lineNumber* is the index of row in the library cell, *name* is the name of the gate in the cell and *type* can be one of the following:

`inpt`	cell input
`from`	fanout branch
`and`	and gate
`nand`	nand gate
`or`	or gate
`nor`	nor gate
`not`	inverter
`buff`	cell output
`flipflop`	D-flipflop

numberOfOutputs is the number of fanout branches of the gate

numberOfInputs is the number of gate inputs

listOfInputs contains an array of indexes separated by spaces to indicate the line numbers of pins connected to corresponding gate inputs.

Appendix E Synopsys Design Interface

In order to generate EDIF compatible to the EDIF Interface, select *File/Save As…*from Synopsys Design Analyzer window. Set *File Format* field to `EDIF`.

Note that the following Synopsys variables must be set!

```
edifout_netlist_only = "true"
edifout_numerical_array_members = "true"
```

It is required to use similar names in Synopsys EDIFOUT ground and power variables as with the EDIF interface `-gnd` and `-vdd` options.

For example, if you set

```
edifout_ground_net_name = "gnd"
edifout_ground_net_pin = "gnd"
edifout_power_net_name = "vdd"
edifout_power_net_pin = "vdd"
```

then Turbo Tester EDIF interface must be run with options

```
-gnd gnd -vdd vdd
```

! *Please use only letters while naming the ground and power variables! For example, names like gnd! or vdd+ are not valid.*

Appendix F Cadence Design Interface

In order to generate an EDIF output compatible to the EDIF Interface select *File/Export/EDIF 200*.... Please note that:

1. *Design Name* must be specified!

The name can be arbitrary. It is only needed to force EDIFOUT to identify the cell containing the top-level design.

2. *External Libraries* field should be left blank.
3. *Ripper Library Name*, *Ripper Cell Name* and *Ripper View Name* should be set to `basic`, `patch` and `symbol`, respectively. These are also default values for EDIFOUT.
4. *Netlist Only* option should be turned on to exclude all graphic information and to reduce the size of output file.
5. *Generate Scalar EDIF* must be selected.

Both, hierarchical and flattened netlists are supported.

See the Cadence documentation for more information on how to fill the 'EDIF 200 Out' form window.

! *Cadence schematic editor is not case sensitive. However, the EDIF Interface is. Therefore do not use names such as e.g.* `A1` *and* `a1` *while describing ports and/or instances in the schematic!*