## first_page.JPG

## VLSI testing Exercises



## LAB 1: ATPG and fault simulation (1)

## Exercise 1: Deterministic ATPG (path sensitization) for fan-out free circuit

* 1. Determine the number of all potential fault sites for the circuit shown in Figure 1.
  2. Use path sensitization to run ATPG for f/1 and h/1 stuck-at faults for the circuit shown in Figure 1.
  3. Given resultant test vectors, determine the expected value on the output (line z).
  4. Save test patterns in a format provided by the description file for combinational designs [Appendix 1].
  5. Run fault simulation [Procedure 3] using the test patterns obtained earlier and analyze the performance of the test (including fault coverage) [Procedure 4]. Next, analyze the report regarding detected faults and indicate the target fault.
  6. Generate test patterns using *Tessent* *FastScan* [Procedure 2] and analyze the performance of the test [Procedure 4].
  7. Compare the obtained results.

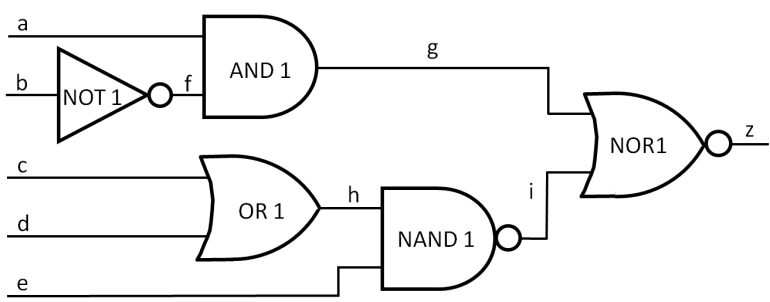


Figure 1: Circuit for exercise 1

## Exercise 2: Deductive fault simulation

1. Given the circuit shown in Figure 2, use deductive fault simulation for test patterns 01010, 11011, 00110 to determine a list of faults detected on its primary output z.
2. Run fault simulation by using *Tessent* *FastScan* [Procedure 3].
3. Analyze the fault report [Procedure 4] and indicate detected faults.
4. Compare the obtained results.

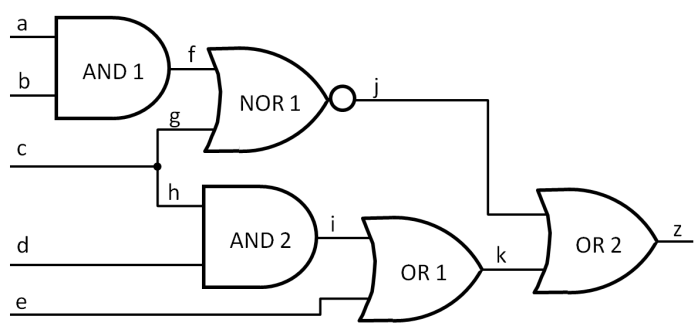


Figure 2: Circuit for Exercise 2

**LAB 2: ATPG and fault simulation (2)**

## Exercise 1: Test set minimization

1. Create a fault table for all stuck-at faults in the circuit of Figure 3.
2. Determine equivalent fault classes.
3. Create a reduced fault table by removing all faults but the first member of each class and undetectable faults.
4. Select a minimal set of tests to cover all detectable faults.
5. Run fault simulation for the selected set of test set by using *Tessent* *FastScan* [Procedure 3].
6. Generate test patterns using *Tessent* *FastScan* [Procedure 2] and compare the performance (the number of test patterns and test coverage) of both test sets [Procedure 4].

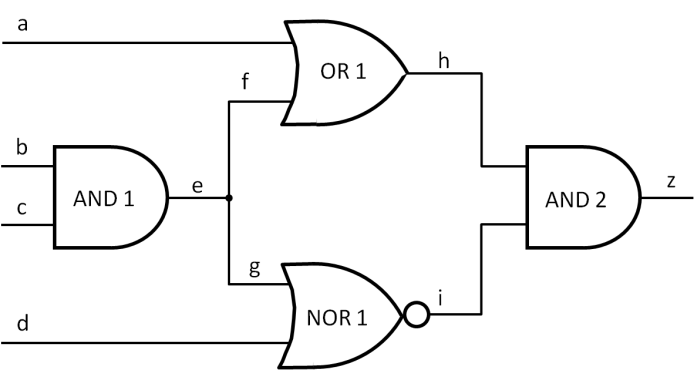


Figure 3: Circuit for exercise 2

**LAB 3: ATPG and fault simulation (3)**

## Exercise 1: Removing redundancy from a circuit with undetectable faults

1. Run ATPG for the circuit of Figure 4 using *Tessent* *FastScan* [Procedure 2].
2. Simulate the circuit using *ModelSim* [Procedure 5].
3. Analyze the performance of the test [Procedure 4] and indicate the list of undetected faults.
4. Use the redundant fault class report as guidance and modify the circuit netlist to delete any redundancy in the circuit.
5. To assure that your modification did not change the circuit functionality, simulate the modified circuit using *ModelSim* [Procedure 5] and compare the results.
6. Compare complexity (the number of gates and connections) of circuits and test results for the circuit with redundancy and after modifications.

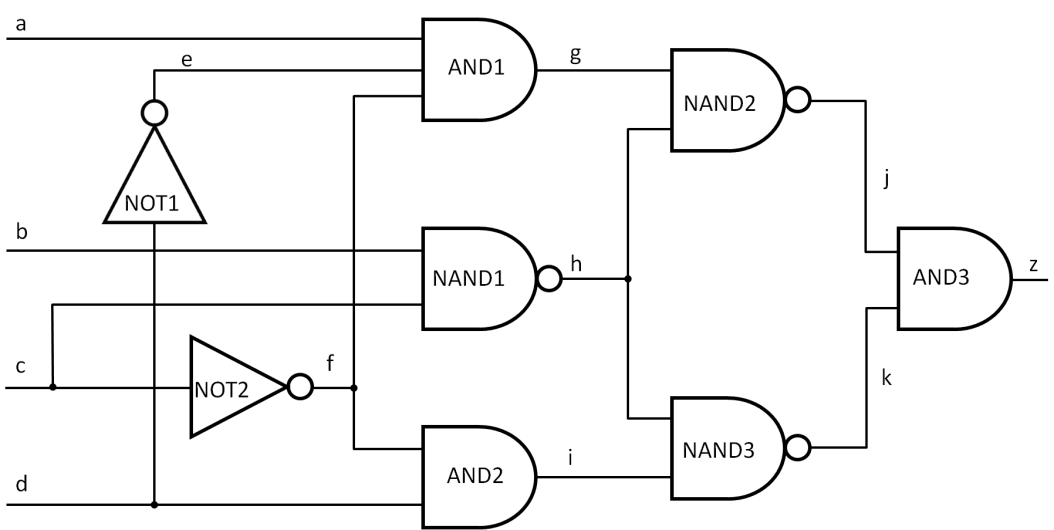


Figure 4. Circuit for exercise 1

## Exercise 2: Test point insertion

1. For the circuit shown in Figure 4, insert control and observation points to increase test effectiveness without removing redundancy.
2. Analyze the performance of the test [Procedure 3] and indicate the list of undetected faults [Procedure 4].
3. To assure that your modification did not change the functionality, simulate the modified circuit using *ModelSim* [Procedure 5] and compare the results.
4. Compare complexity (the number of gates and connections) of the circuits and test results for the circuit with redundancy and after modification.

## LAB 4: Pseudorandom testing

## bok_carbon.jpgExercise 1: LFSR as pseudorandom test pattern generator

1. Draw Galois and Fibonacci LFSRs for the following characteristic polynomial: x4 + x1 + 1.
2. For these LFSRs generate the maximum length sequence with the initial state (seed) 0001.
3. Save the obtained states from Galois LFSR as test sets with the increasing number of test patterns (*1.patt* – contains state # 1; *2.patt* - contains states # 1 and 2, *4.patt* – contains states from # 1 to 4, *8.patt* - contains states from # 1 to 8, *16.patt* contains states from # 1 to 16) for circuit C1 (*c1.v*) according to the test vector description file [Appendix 1].
4. Run fault simulation (*c1.run* *–* invocation, *c1.do –* set of commands) [Procedure 3] for circuit C1 using the prepared test sets and present the resultant fault coverage (*c1.log*).

## Exercise 2: Pseudorandom testing of combinational circuit

1. Write a function (for example in *C++*: */lfsr\_cpp/lfsr.cpp*), which allows generating a test pattern description file [Appendix 1] by using an LFSR with a given characteristic polynomial, a specific initial state and the length of a generated sequence.
2. Apply the above function (*/lfsr\_cpp/lfsr*) to generate pseudorandom test pattern sets (# test patterns according to Table 1) for circuit c6288 (c6288.v). Assume that each LFSR stage is connected with a designated input of CUT. Select the proper primitive polynomial from Appendix 3.
3. Run fault simulation (*c6288.run* – invocation, *c6288.do* – set of commands) [Procedure 3] for the obtained test sets and circuit c6288 to analyze the performance of the test (fault coverage) [Procedure 3].
4. Complete the Table 1 and present the resultant fault coverage as a function of the applied pseudorandom test patterns (see Figure 5).
5. Draw conclusions about the effectiveness of pseudorandom testing.

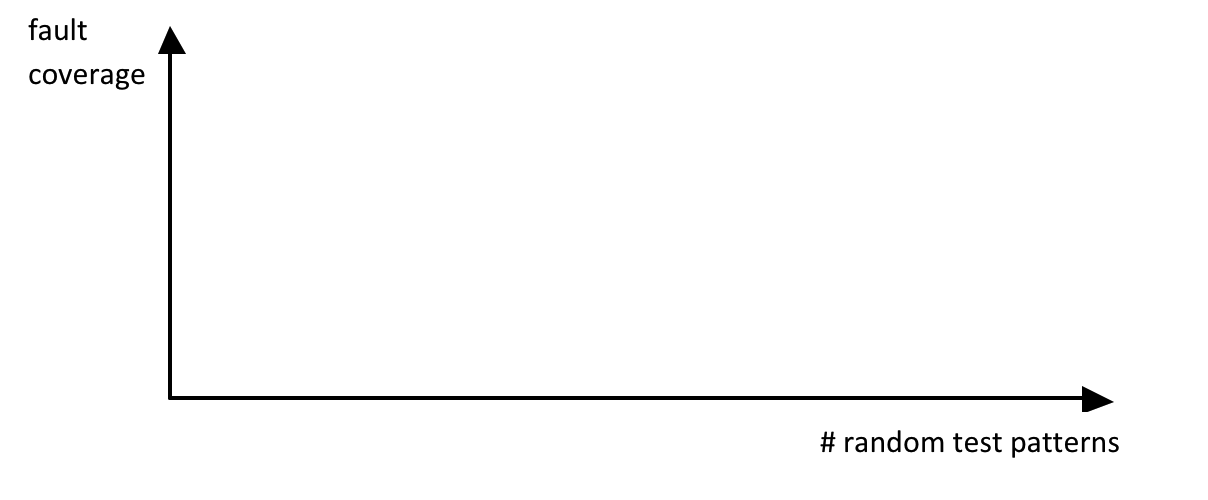
  
  
**LAB 5: Design for testability (DFT)**

Figure 5: The effectiveness of generated patterns

Table 1: Test patterns statistics for exercise 2

## bok_carbon.jpgExercise 1: Basic scan insertion

Go to directory *lab\_5/ex\_1*. Notice that each step of this exercise has its own subdirectory and common directories are used for netlists, libraries, and reports.

1. In *0\_lspec\_synth* synthesize design from RTL HDL description to gate level Verilog netlist using *Leonardo Spectrum* [Procedure 7].
2. In directory *1\_scan* invoke the *Tessent Scan* and insert 100 scan chains into synthesized netlist [Procedure 8].
3. Report and analyze the statistics.
4. Determine the number of simulation gates, the number of sequential elements in this circuit, and the number of scan cells.

## Exercise 2: Scan and ATPG flow

Go to directory *lab\_5/ex\_2*. Notice that each step of this exercise has its own subdirectory and common directories are used for netlists and libraries.

1. In a manner similar to that of exercise 1, insert 4 scan chains into netlist cpu.v.
2. In directory *2\_atpg* invoke *Tessent* *FastScan* on the scan-inserted netlist (4 scan chains) and create test patterns (also in a parallel form).
3. Analyze ATPG messages and test coverage statistics.

**LAB 6: Built-In Self-Test (BIST)**

## Exercise 1: BIST

Consider a BIST scheme shown in Figure 6 for combinational design C1(CUT). An LFSR with the characteristic polynomial x3 + x2 + 1 and the initial state 111 has been deployed on the input side of C1 as a pseudorandom test pattern generator. Also, a MISR with the characteristic polynomial x3 + x2 + 1 and the initial state 000 has been employed as a test response compactor on the output side.

1. Follow the steps below to complete Table 2:
   * Determine the LFSR maximum length sequence.
   * Determine the fault-free signature.
   * Determine signatures for faults A/0 and A/1.
2. Check if the above faults can be detected and explain why.

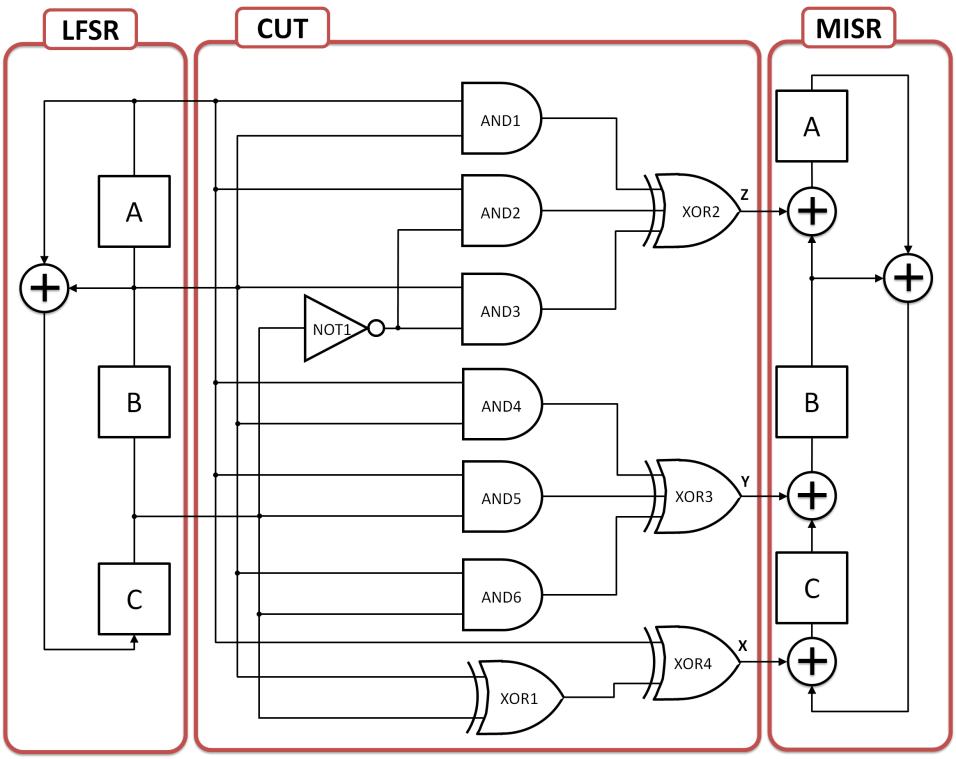


Figure 6. BIST scheme for exercise 1

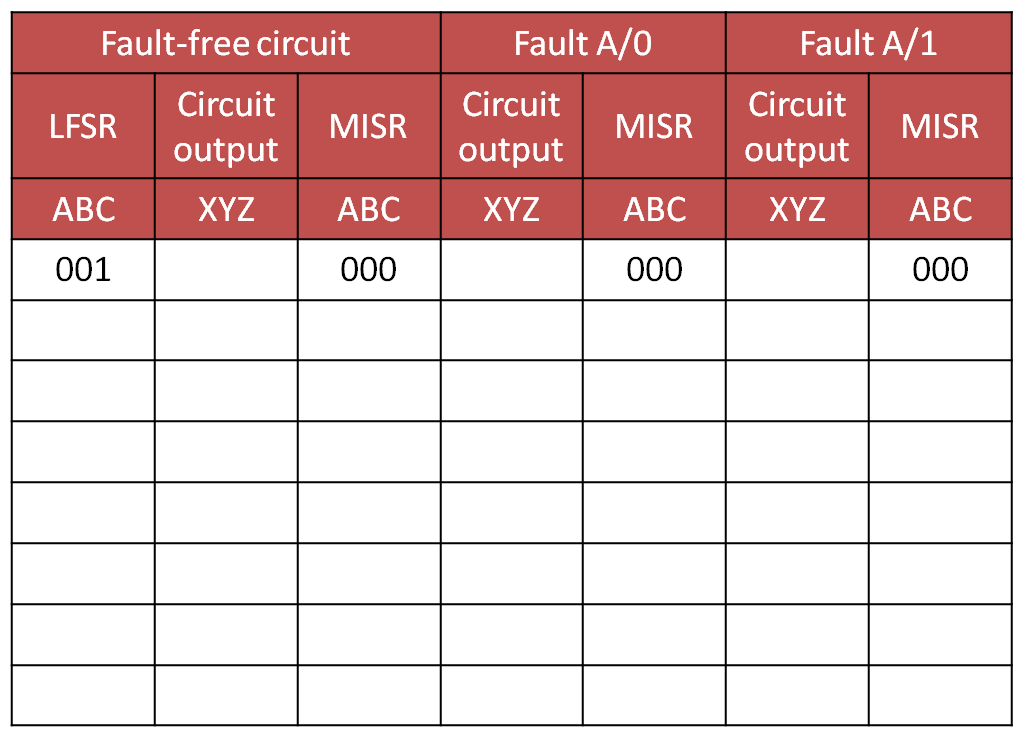


Table 2. Signature report for exercise 1

## LAB 7: Test compression



We will use a non-scan RTL-level Verilog implementation of a simple 8-bit microcontroller. It is binary code compatible with the Microchip 16C57 microcontroller. This source code has been developed by Thomas Coonan and is publicly available at www.mindspring.com/~tcoonan (2011). According to the copyrights, it can be redistributed and/or modified in a source code form under the terms of the General Public License (GNU).

## Exercise 1: Test pattern compression flow

Go to directory *lab\_7/ex\_1*. Notice that each step of the exercise has its own subdirectory, and common directories are used for netlists, libraries, and logfile reports.

1. In the *0\_lspec\_synth* directory synthesize the model of the given microcontroller [Procedure 7]. Hint: use the ami05\_worst.syn library and attach source files in the following order: *alu.v, dram.v, regs.v, idec.v, cpu.v*.
2. In directory *1 \_scan* invoke *Tessent Scan* on the gate-level netlist and insert 20 scan chains [Procedure 8].
3. In directory *2\_fs\_pg* run *FastScan* on the scan-inserted netlist (20 scan chains) and generate test patterns [Procedure 9]. Next, analyze the *TestKompress* report and answer these questions [Appendix 6].
4. In the *3\_tk\_edt* directory invoke *TestKompress* on the scan-inserted netlist (20 scan chains) and create *TestKompress* logic with 4 scan channels [Procedure 10]. Next, analyze the *TestKompress* report and answer the following questions [Appendix 6].
   * How many initial shifts are there?
   * What is the size of the decompressor?
   * What is compression per pattern?
5. In the 4\_lspec\_synth\_edtrm directory synthesize EDT logic using Leonardo Spectrum [Procedure 7]. Hint: use the ami05\_worst.syn library and analyze source files in the following order: *design\_scan.v, design\_edt.v, design\_edt\_top.v*.
6. In the *5\_tk\_pg* directory run *TestKompress* on the synthesized top-level of the design and generate compressed test patterns [Procedure 9]. Next, analyze the *TestKompress* report and answer the following questions [Appendices 4 and 7].
   * + What is the fault coverage and the test coverage?
     + How many test patterns are there?
     + What is the total data compression?

## LAB 8: Fault diagnosis

## bok_carbon.jpgExercise 1: Logic level diagnosis

Go to directory *lab\_8/ex\_1*.

1. Run ATPG for the circuit of Figure 7 using *FastScan* and save test patterns in ASCII format [Procedure 2].
2. Analyze the fault report and determine fault equivalence classes [Procedure 4].
3. Divide the test pattern set into isolated patterns.
4. Analyze fault reports for particular test patterns and partition the faults into classes.
5. Determine a diagnostic tree.
6. Analyze the test responses for two case studies, complete Table 3 and determine failures.
7. Improve the fault resolution by the generation of tests to distinguish faults: e0 f0 g0.
8. Enhance the diagnostic tree and explain why additional patterns can distinguish targeting fault sites.

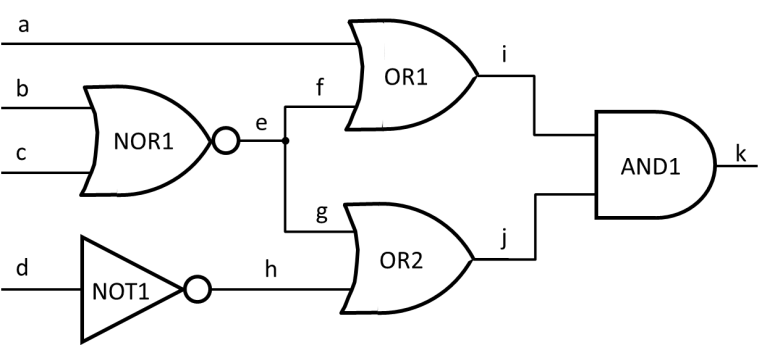


Figure 7. Circuit for exercise 1

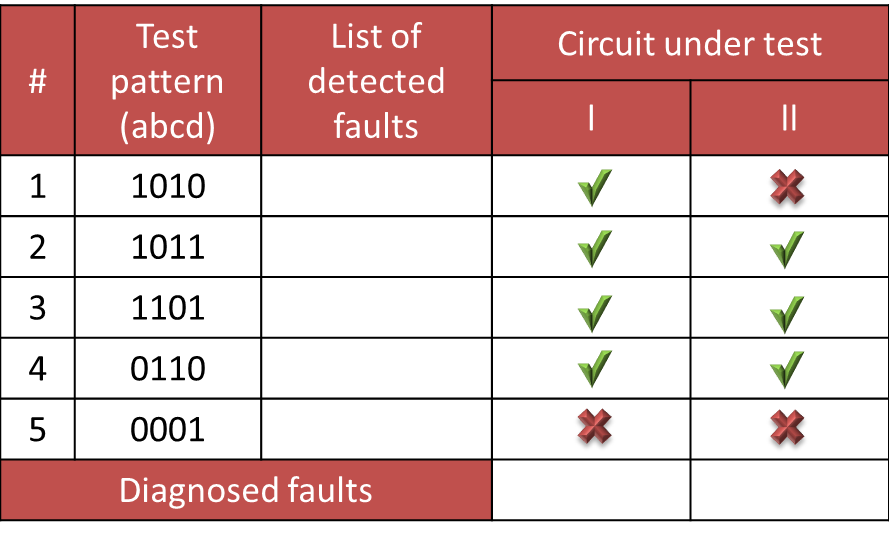


Table 3. Case study report for exercise 1

## LAB 9: Iterative Diagnosis

## bok_carbon.jpgExercise 1: Iterative pattern generation

Go to directory *lab\_9/ex1*.

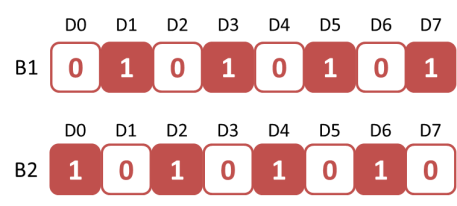
1. Run *Tessent Diagnosis* with the failure file (*fail1.log*) from one device [Procedure 11].
2. Analyze the diagnosis report and determine the number of symptoms, the number of suspects and the diagnosis score [Appendix 8].
3. Select a good candidate for the iterative diagnosis [Appendix 8].
4. Invoke *Tessent Diagnosis* with an emulation of the selected failure and create new iterative test patterns [Procedure 11].
5. Run *Tessent Diagnosis* for both created test pattern sets and failure files.
6. Analyze the diagnosis report and compare results [Appendix 8]. How many suspects were reported for the symptom number 2 after the iterative diagnosis?

**LAB 10: Memory Built-In Self-Test (BIST)**

## Exercise 1: Algorithm for programmable controller

Consider a memory built-in self-test scheme shown in Figure 8. *Tessent MemoryBIST* with programmable controller has been deployed to test a word-oriented memory array (128 x 8b). This architecture enables you to define customized test algorithms to target specific memory defects that are difficult to detect with the existing algorithms. The algorithm programing uses a specific syntax wrapper [Appendix 9]. For a complete description of the Algorithm wrapper syntax, refer to the *Reference for Algorithm Wrapper Contents appendix*.

Go to directory *lab\_10*.

1. Use the programmable controller to generate multiple data backgrounds to detect all coupling faults between neighboring cells of the same word.
2. Use the following data backgrounds:  
   
3. Propose a sequence of data background instructions (write, read, compare). Use the wrapper and its contents to develop your algorithm (file *1.alg)*.
4. Invoke *ModelSim* algorithm simulation (*makealg file*), which dumps simulation data around the area of the fault-injected memory and the memory BIST controller.
5. Analyze the simulation report to examine the number of compare failures and verify a correctness of the proposed algorithm.

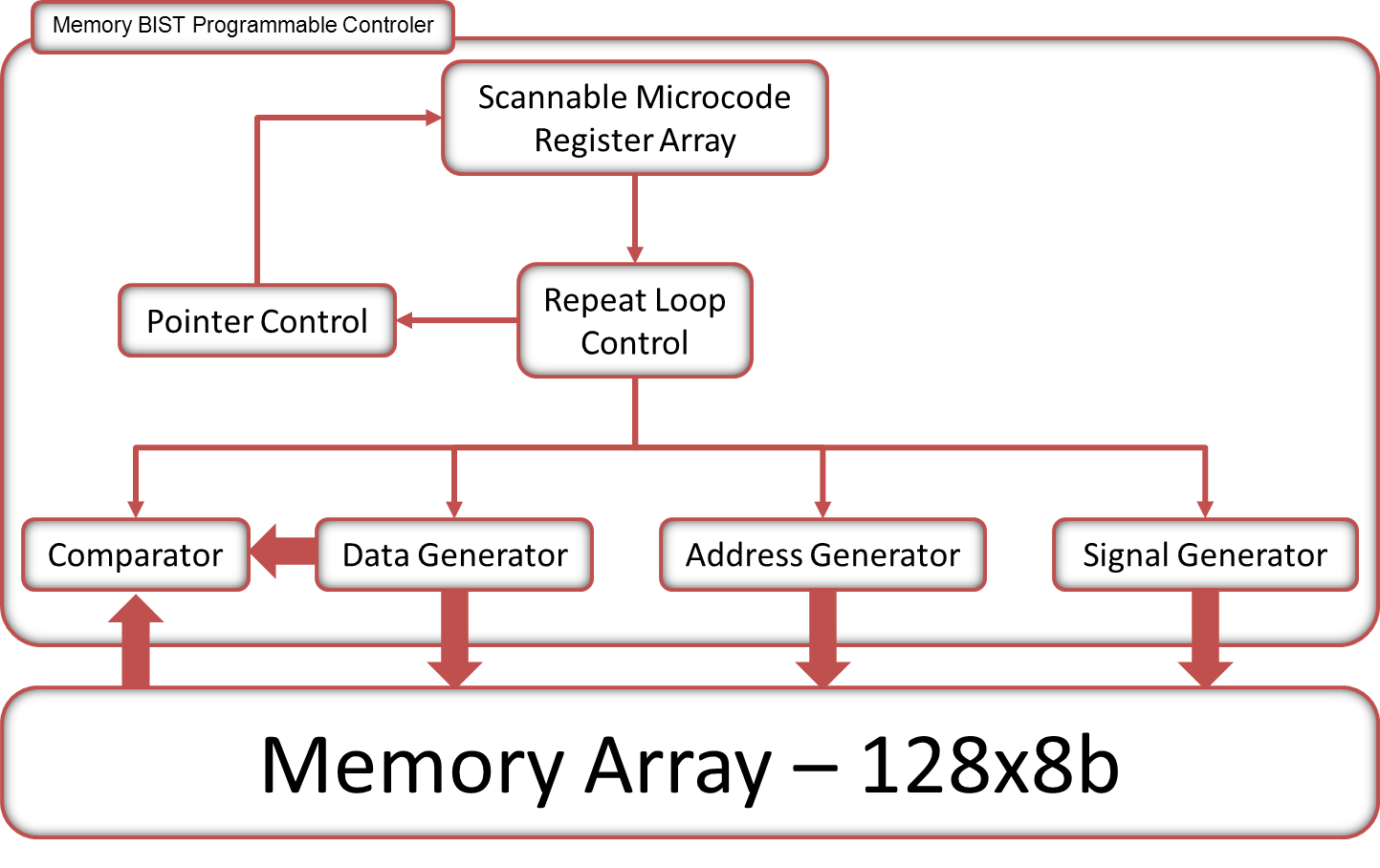


Figure 8. Memory BIST scheme for exercise 1

## first_page.JPG

## VLSI Testing Procedures



## bok_carbon.jpgProcedure 1: Running ATPG (simplified flow for combinational circuits)

1. *Tessent Shell* needs the following files to be specified:
   * *Gate-level netlist* – contains Verilog gate-level design description.
   * *ATPG library*  -- needed if the design contains modules not found in the netlist file.
   * *Dofile*  – contains a series of commands to be executed in batch mode (this file is optional, all commands may be also invoked in an interactive mode)*.*
   * Logfile – a logfile of messages (optional), add –replace to overwrite the previous logfile
2. Run Tessent Shell using the following command:  
   $TESSENT\_PATH/tessent –shell –dofile <dofile\_name> /  
   -log <logfile\_name> -replace
3. Invoke test pattern generation and simulation context:

set\_context patterns -scan

1. Load design model:

read\_verilog <circuit\_filename.v>

1. Load ATPG library file if needed for this design:
2. read\_cell\_library <atpg.lib>Indicate the top module in the design:

set\_current\_design <top\_module>

1. Switch to analysis mode:   
   set\_system\_mode analysis
2. Analyze Flattening, *DRC and, Circuit Learning reports*.

Notice that all available commands have a manual page that can be invoked as follows:  
help <command\_name>

To close Tessent Shell use command: exit -d

Procedure 2: Test pattern generation

After running *Tessent Shell* and analysis mode [Procedure 1],

1. Indicate stuck-at fault model:

set\_fault\_type stuck

1. Add a targeted list of faults or all faults:add\_faults -all
2. Run *Pattern Generation*:  
   create\_patterns
3. Analyze the results [Procedure 4].

Procedure 3: Fault simulation with external test pattern set

After running *Tessent Shell* and analysis mode [Procedure 1], you can run *Fault Simulation* of a previously prepared external test patterns set.

1. Indicate stuck-at fault model:

set\_fault\_type stuck

1. Add all faults:add\_faults –all
2. Indicate the path to external test patterns file (choose *Ascii test patterns file format*):

set\_pattern\_source external <patterns\_filename.ascii> -ascii

1. Run fault simulation:

simulate\_patterns

1. Analyze the results using command line or *DFTVisualizer* [Procedure 4].



Procedure 4: Fault coverage and detected fault analysis

A: using command line

In order to analyze results of ATPG or fault simulation use the following commands in theanalysis mode.

1. report\_statistics
2. report\_faults –class <class\_name>
3. class names:
   * FULL – all fault classes
   * RE – redundant faults
   * DS – faults detected by simulation
   * EQ – equivalent faults

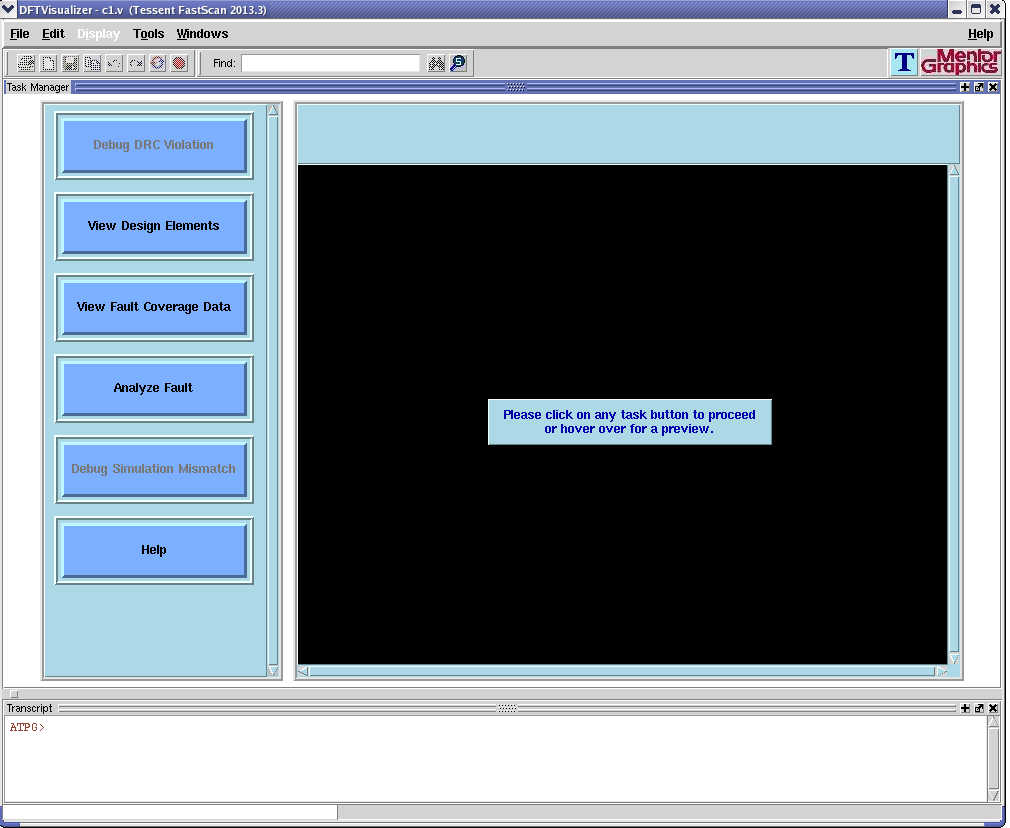


Figure 9: *DFTVisualizer - View Fault Coverage Data* window

B: using DFTVisualizer

1. In order to analyze results of ATPG or fault simulation open a *DFTVisualizer* window using the command open\_visualizer in theanalysis mode.
2. In the *DFTVisualizer* window select the *View Fault Coverage Data* option from the Task Manager and do the following (Figure 9).

* Setup the necessary test statistics reported in the *Browser* window by clicking appropriate buttons (*Fault Coverage*, *Test Coverage*, *Fault By Category*).
* From the main menu select *Tools* and click *Analyze Faults…*
* In the *Faults & Statistics Options* select the *Entire Design*.
* In the *Report Faults Options* select the *Both of the Above*.
* In the *Fault Class* indicate the targeted fault class (Figure 10)[Appendix 5].
* In the *Reported Data* click the *Report Faults*.
* Analyze the list of faults [Appendix 5].

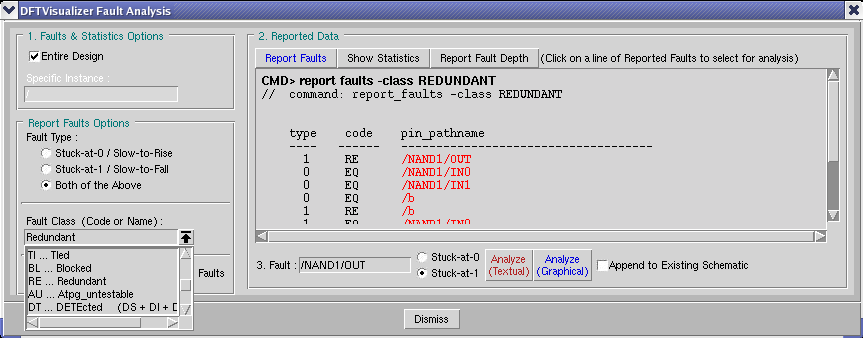
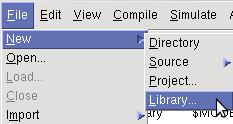
****

Figure 10: *Fault Class Choices* window

Procedure 5: HDL simulation

* 1. Run *ModelSim*:  *vsim &*.
  2. Create a design library and map to a physical library.  
     From *File* menu choose *New,* and next *Library…*.



* In the *Create a New Library* window select *a new library and a logical mapping to it*.

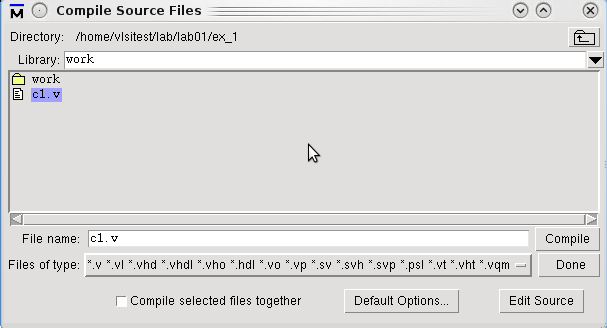
new_lib_and_mapping.png

* Type the name of your library in Library Name field, e.g. *my\_work\_lib* or leave empty and use the default library *work*.
* Click *OK*.
  1. Compile HDL design files.
* From *Compile* menu choose *Compile*…

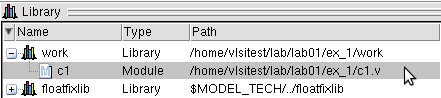




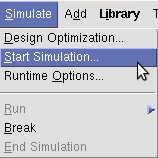
* In the *Compile HDL Source File* window select top-level file from a design directory and click *Compile*.



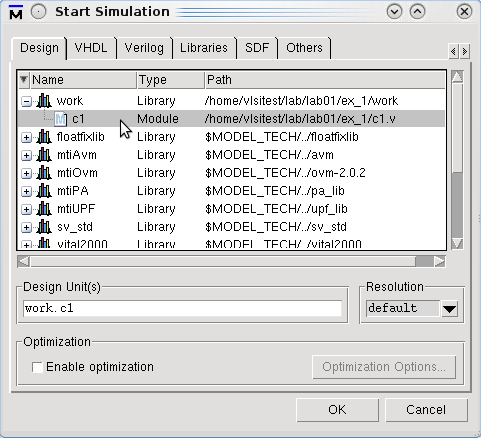
* Click *Done* to finish.
* Review the messages on the *ModelSim Console* window, check compiled modules and expand the working library in the *Library pane*.



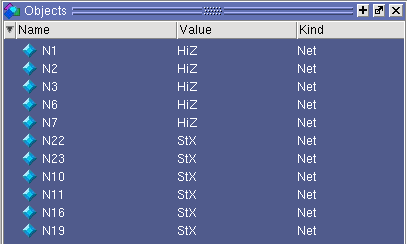
* 1. Simulate.
* From *Simulate* menu choose *Start Simulation*...



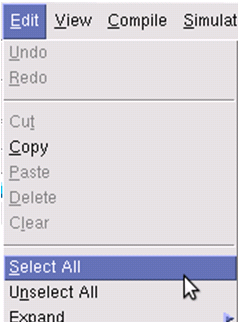
* In the *Start Simulation* window choose *Design* tab and from the available libraries select the top-level unit, disable optimization, and click *OK*.

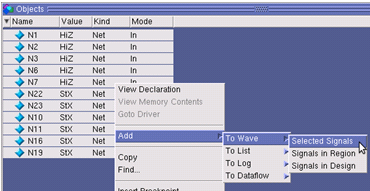


* Review messages printed to the *ModelSim Console*.
* Review signals of the design in the *Objects* window.



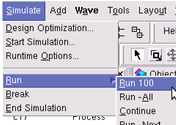
* 1. Open the waveform viewer and analyze the observable signals.
* Add all required signals in the top-level unit to the waveform viewer. In the *Object* window from the *Edit* menu choose *Select All*.



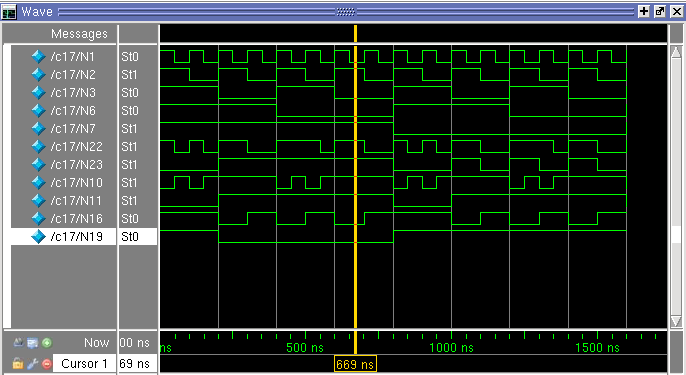
* Right-click on the highlighted signals and next select *Add to Wave*.  
    
   
* To force the required state of input signals right-click on the appropriate signal and select *Force…* or *Clock…*. Depending on the selection you can set different properties of the excitation signal.

|  |  |
| --- | --- |
| ms12.png | ms13a.png |
| ms13b.png |

* To run simulation select *Run* and *Run 100* from the *Simulate* menu.





* + - In the *Wave* window review simulated signals of the design.  
        
      
    - To restart simulation select *Run* and *Restart* from *Simulate* menu.

****Procedure 7: **Synthesizing HDL using Leonardo Spectrum**

1. *Leonardo Spectrum* needs three files to be specified as shown in Figure 11:
   * *RTL model* – synthesizable HDL description of your design.
   * *Synthesis library* – contains a description of testable models for logical gates.
   * *Batch file* – contains a series of commands to be executed by *Leonardo Spectrum*.

*b. Leonardo Spectrum* results:

* *Technology specific netlist* – a new netlist synthesized in accordance with a given technology.

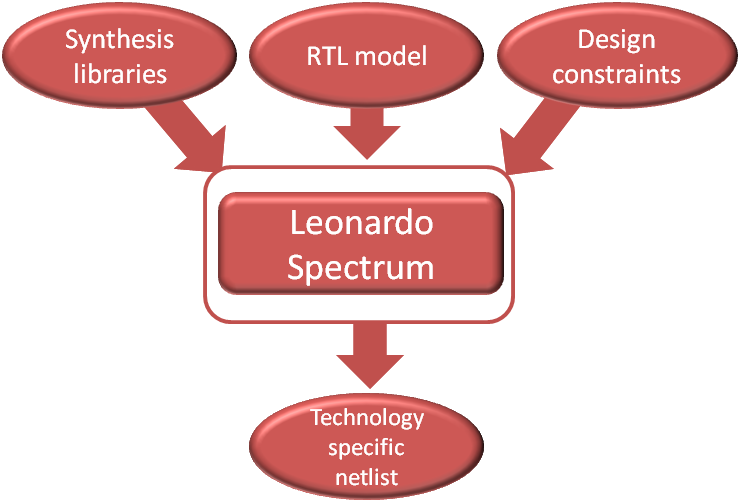
****

Figure 11: Leonardo Spectrum synthesis flow

c. First, you need to create a similar file and a directory structure as shown in the frame below:

<LAB\_#>/ -- lab directory   
spectrum\_synth/ -- Leonardo Spectrum work directory

run\_spectrum -- Leonardo Spectrum run script

spectrum.batch -- Leonardo Spectrum batch file  
netlists/ -- HDL source files  
libs/ -- libraries subdirectory   
log/ -- reports subdirectory

d. Second, create the batch file <*spectrum.batch*> for *Leonardo Spectrum*:

//Load the library  
load\_library <tsmc035\_typ>  
//Read the HDL files  
read { abmux.vhd alu.vhd control.vhd datamux.vhd top.vhd }  
//Optimize the design to a technology  
optimize  
//Optionally set timing constraints   
set\_attribute -net clk -name clock\_cycle -value 10  
optimize\_timing

//Generate reports  
report\_delay  
report\_area -cell

//Write out the netlist  
set vhdl\_write\_use\_packages {library ieee,adk; use ieee.std\_logic\_1164.all; use adk.adk\_components.all;}

apply\_rename\_rules -ruleset VHDL  
apply\_rename\_rules -ruleset VERILOG  
auto\_write ../netlist/design.v

e. Next, to run *Leonardo Spectrum* and synthesize RTL model, create the <*run\_spectrum*> file according to the following call syntax:

$LEONARDO\_SPECTRUM\_PATH/spectrum \

-file <spectrum.batch> \

-logfile <logfile\_name>

f. Finally, run pre-generated batch file: *prompt > ./run\_spectrum*. Examine the logfile and verify if the netlist has been written.

Procedure 8: **Scan chains insertion using Tessent Scan (batch mode)**

This procedure introduces a scan chains insertion flow that you may use to create scan chains in your design at RTL level using *Tessent* *Scan* batch mode. Figure 12 illustrates the scan chains insertion flow.

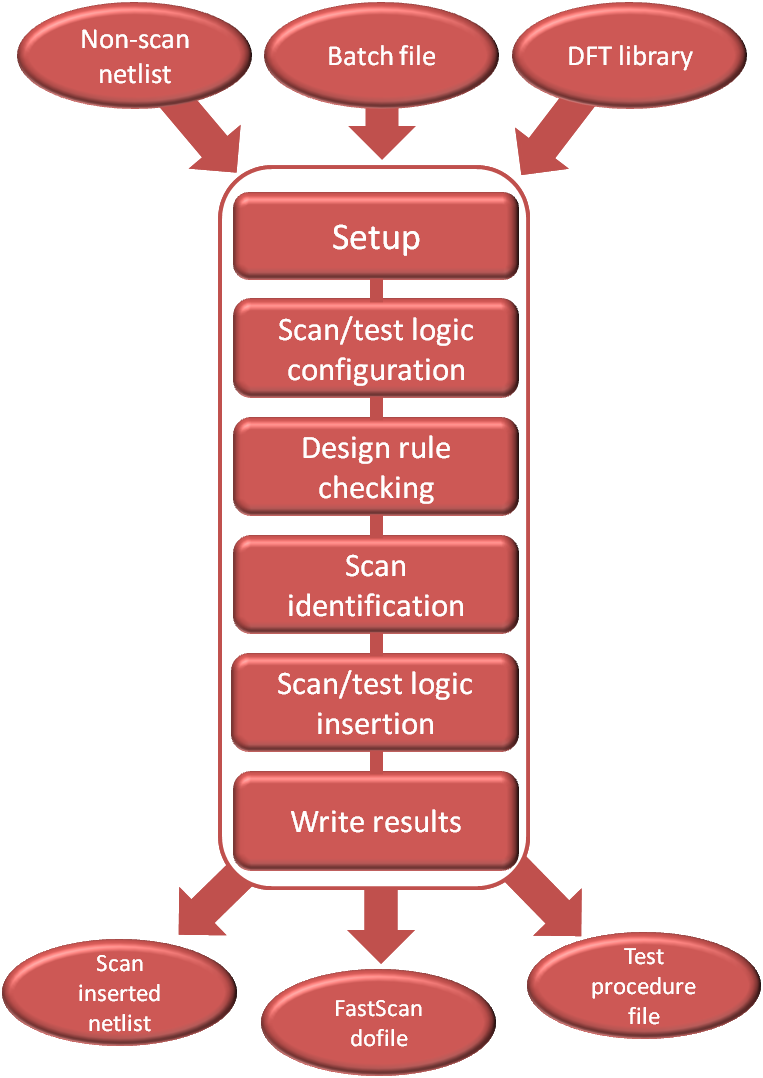


Figure 12: Tessent Scan scan chains insertion flow

1. *Tessent Scan* needs three files to be specified.
   * *Non-scan netlist* – synthesizable HDL description of your design.
   * *DFT library* – contains a description of testable models for logic gates.
   * *Dofile* – contains a series of commands to be executed by *Tessent* *Scan*.
2. *Tessent* Scan results.
   * *Scan inserted netlist* – a new netlist with inserted scan chains.
   * *FastScan dofile* –provides circuit setup and scan circuitry information used by *FastScan* for ATPG.
   * *Test procedure file* –contains cycle-based procedures and timing definitions used by *FastScan* to operate the scan structures within a design.
3. First, you need to create a similar file and a directory structure as shown in the frame below:

<LAB\_#>/ -- lab directory   
scan\_ins/ -- Tessent Scan work directory

run\_scan -- scan insertion run script

scan.do -- scan insertion dofile  
netlists/ -- HDL source files  
libs/ -- libraries subdirectory   
log/ -- reports subdirectory



1. Second, create the dofile <scan.do> for *Tessent* *Scan*:

set\_context dft –scan // Go to DFT mode  
read\_verilog <verilog\_netlist>  
analyze\_control\_signals –auto   
// Setting parameters for the scan chains logic – 8 scans  
insert\_test\_logic -number 8  
// Save scan inserted netlist  
write\_design –output\_file <verilog\_netlist\_with\_scans> –verilog –replace  
// Save test procedure and setup files with the filename prefix "design"  
write\_atpg\_setup <scan\_setup\_filename>

1. Next, to run *Tessent* *Scan* and generate a scan chains inserted design file create a <run\_scan> file according to the following call syntax:

$TESSENT\_PATH/tessent -shell \

-dofile <dofile\_name> \

-logfile <logfile\_name> -replace

1. Finally, run pre-generated dofile: *prompt > ./run\_scan*. Examine the transcript or logfile and verify that the netlist, dofile, and test procedure files have been written.

****Procedure 9: **Test pattern generation using Tessent Shell FastScan or TestKompress**

This procedure introduces a test pattern generation flow shown in Figure 13 that you may use to create patterns for your design using *Tessent FastScan* or *TestKompress*.

When using *TestKompress,* test patterns are generated in a fashion similar to that of traditional ATPG (e.g., *FastScan*). In fact, *TestKompress* commands are compatible with most *FastScan* commands. During logic creation, a dofile containing setup information for the pattern generation phase is automatically generated (similarly to how *Tessent* *Scan* creates files describing the operation of the inserted scan chain circuitry).

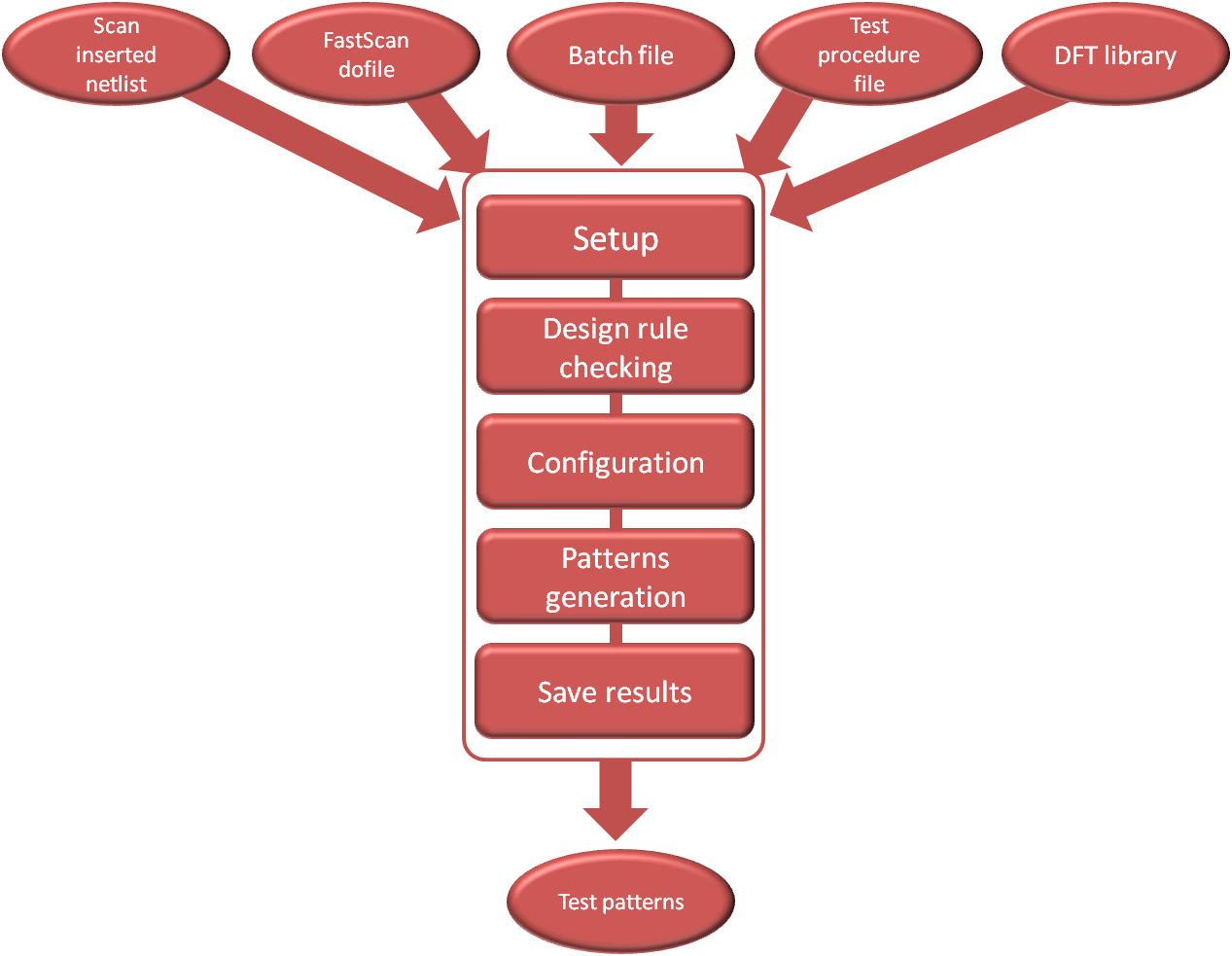


Figure 13: FastScan (TestKompress) test patterns generation flow

1. *FastScan* (*TestKompress*) needs five files to be specified:
   * *Scan inserted netlist* – a netlist with inserted scan chains.
   * *FastScan setup (created by Tessent Scan)* – it provides circuit setup and scan circuitry information.
   * *Test procedure file –* contains cycle-based procedures and timing definitions, used by *FastScan* to operate the scan structures within a design.
   * *Dofile*  – contains a series of commands to be executed by *FastScan.*
   * *DFT library* – contains a description of testable models for logic gates.
2. *FastScan (TestKompress)* results:
   * *Test patterns* – test patterns description file[Appendices 1 and 2]*.*
3. First, you need to create the same file and directory structure as shown in the frame below:

<LAB\_#>/ -- lab directory

fs\_pat\_gen/ -- FastScan work directory

run\_fs -- FastScan run script

fs.dofile -- FastScan dofile  
netlists/ -- HDL source files  
libs/ -- libraries subdirectory   
patt/ -- test patterns subdirectory  
log/ -- reports subdirectory

1. Second, create the dofile <fs.do> for *FastScan* (*TestKompress*):

set\_context patterns –scan

read\_verilog <verilog\_netlist\_with\_scans>read\_cell\_library <atpg.library>

set\_current\_design <top\_design\_name>

// Read scan setup

dofile <scan\_setup\_filename>

// Go to atpg mode

set\_system\_mode analysis

// Pattern generation, default is stuck-at

create\_patterns

report\_statistics

write\_patterns ../patt/par.v –verilog –parallel –replace

write\_patterns ../patt/ser.v –verilog –serial -replace

exit

1. Next, to run *FastScan* (*TestKompress*) and generate test patterns create a run\_fs (run\_tk) file according to the following call syntax:

$TESSENT\_PATH/tessent -shell \

-dofile <dofile\_name> \

-logfile <logfile\_name> -replace

Procedure 10: **EDT logic insertion using TestKompress**

This procedure presents EDT logic insertion flow that you may use to implant test compression logic into your design and obtain initial test coverage and compression estimates by using *TestKompress*.

1. *TestKompress* needs five files to be specified:
   * *Scan inserted netlist* *–* a netlist with inserted scan chains.
   * *TestKompress dofile – (created by Tessent Scan)* provides circuit setup and scan circuitry information.
   * *Test procedure file – (created by Tessent Scan)* contains cycle-based procedures and timing definitions, used by *TestKompress* to operate the scan structures within the design.
   * *Dofile* *–* contains a series of commands to be executed by *TestKompress.*
   * *DFT library* – contains a description of testable models for logic gates.
2. *TestKompress* results:
   * *Design\_edt.v – TestKompress circuitry (Verilog RTL).*
   * *Design\_edt\_top.v – top level wrapper (instantiates core and TestKompress logic).*
   * *Design\_core\_blackbox.v – blackbox description of the core (used for synthesis).*
   * *Design\_dc\_script.scr – Design Compiler synthesis script.*
   * *Design\_edt.dofile – dofile for TestKompress (test pattern generation).*
   * *Design\_edt.testproc – test procedure file (test pattern generation).*
   * *Design\_bypass.dofile – dofile for bypass mode.*
   * *Design\_bypass.testproc detected* – *procedure file for bypass mode.*
3. First, you need to create the same file and a directory structure as shown in the frame below:

<LAB\_#>/ -- lab directory   
tk\_ip/ -- TestKompress work directory

run\_tk\_ip -- run script

tk\_ip.do -- TestKompress dofile  
netlists/ -- HDL source files  
libs/ -- libraries subdirectory   
patt/ -- test patterns subdirectory  
log/ -- reports subdirector

1. Second, create the dofile *<tk\_ip.do>* for *TestKompress*:

set\_context dft -edt

read\_verilog <verilog\_netlist\_with\_scans>

set\_current\_design <top\_design\_name>

read\_cell\_library <atpg.library>

//Use dofile from Tessent Scan run

dofile design.dofile

//Settings for EDT

set\_edt\_options -input\_channels 4 -output\_channels 4

//Run DRCs and go to ATPG mode

set\_system\_mode analysis

//Create Patterns as a sanity check to make sure

//that you are getting the same coverage as FastScan

create\_patterns

//Save the EDT logic and synthesis scripts

//with the filename prefix "design".

write\_edt\_files ./design -verilog -replace

//Exit TestKompress

exit -d

1. Next, to run *TestKompress* and insert test logic create a *<run\_tk\_ip>* file according to the following call syntax:

$TESSENT\_PATH/tessent -shell \

-dofile <dofile\_name> \

-logfile <logfile\_name> -replace

Procedure 11: **Diagnosis using Tessent Diagnosis**

This procedure presents how to invoke *Tessent Diagnosis* on a failure report generated on automatic test equipment (ATE). These failures can be analyzed by *Tessent Diagnosis* and report the suspected defect locations and failure modes.

1. *Tessent Diagnosis* needs five files to be specified:
   * *Flat model netlist – a netlist with test procedure information for pattern generation.*
   * *Dofile – contains a series of commands to be executed by Tessent Diagnosis.*
   * *Test patterns – test patterns description file used in testing process.*
   * *Failure file – a report file as derived from the ATE.*
2. *Tessent Diagnosis* results:
   * *Diagnosis report – a file with various diagnosis statistics (defect location, type, score).*
3. First, you need to create the same file and a directory structure as shown in the frame below:

<LAB\_#>/ -- lab directory   
diagnosis/ -- *Tessent Diagnosis* work directory

run\_diag -- run script

ya.do -- *Tessent Diagnosis* dofile  
netlists/ -- flat model netlist  
patt/ -- test patterns subdirectory

fail/ -- failure files subdirectory  
log/ -- reports subdirectory

1. Second, create the dofile *<diag.do>* for *Tessent Diagnosis*:

set\_context patterns –scan\_diagnosis

read\_flat\_model ../netlists/design.flat

// read external pattern file

set\_pattern\_source external ./patt/pat\_core.ascii –ascii

// read failure file

read\_failure ./fail1.log

// run diagnosis and save report

diagnose\_failure -output fail1.rep -repl

1. Next, to use *Tessent Diagnosis* and run diagnosis create a *<run\_diag>* file according to the following call syntax:

$MGC\_HOME/bin/tessent \  
-dofile diag.do \   
-log ../log/diag.log –rep

1. Additional commands:
   * *write\_failures* – emulates the defect and produces a new failure file,
   * *create\_diagnosis\_patterns -diagnosis\_report <diagnosis report file>* – creates additional internal test patterns to improve a diagnostic resolution.

## logo.pngfirst_page.JPG

## VLSI Testing Appendices

****Appendix 1: Test pattern description file for combinational designs

*FastScan* handles several application-dependent storage formats for test patterns. One of them (ASCII test pattern file for f/1 from Lab 1, Ex 1) is presented below.

ASCII\_PATTERN\_FILE\_VERSION = 2;

SETUP =

declare input bus "PI" = "/a", "/b", "/c", "/d", "/e";

declare output bus "PO" = "/z";

end;

SCAN\_TEST =

pattern = 0;

force "PI" "11011" 0;

measure "PO" "1" 1;

end;

Appendix 2: Test pattern description file for sequential designs

*FastScan* handles several application-dependent storage formats for test patterns. One of them (ASCII) is presented below.

SETUP =

TESCYCLE\_WIDTH = 3;

DECLARE INPUT BUS "ibus" = "/H", "/INP(0)", "/INP(1)", "/INP(2)",

"/INP(3)";

DECLARE OUTPUT BUS "obus\_3" = "/OUTP(0)";

CLOCK "/H" =

OFF\_STATE = 0;

END;

END;

CYCLE\_TEST =

CYCLE = 0;

FORCE "ibus" "01111" 0;

FORCE "ibus" "11111" 1;

FORCE "ibus" "01111" 2;

MEASURE "obus\_3" "1" 3;

CYCLE = 1;

FORCE "ibus" "01100" 0;

FORCE "ibus" "11100" 1;

FORCE "ibus" "01100" 2;

MEASURE "obus\_3" "1" 3;

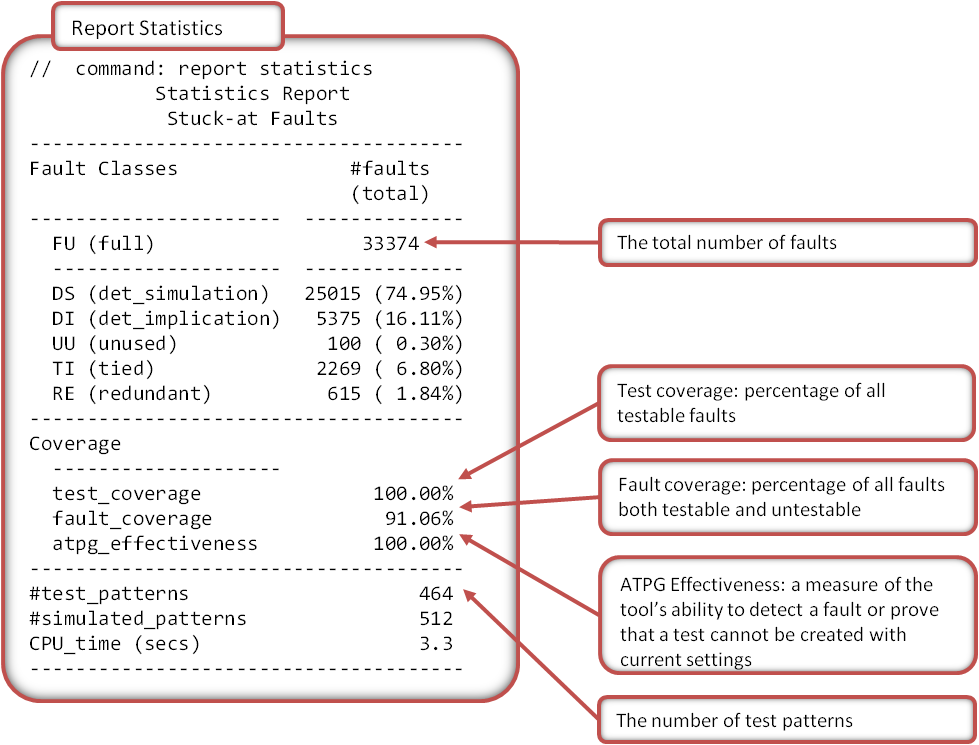
END;

****Appendix 3: Primitive polynomials of degree up to 50**[[1]](#footnote-1)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **5 coefficients** | **7 coefficients** | **9 coefficients** |
| **9** | 97420 | 9753210 | 986543210 |
| **10** | 106520 | 10864210 | 1097643210 |
| **11** | 118520 | 11975310 | 1197643210 |
| **12** | 127430 | 12 1096420 | 12 108764210 |
| **13** | 139730 | 13 1086420 | 13 119764210 |
| **14** | 149720 | 14 11 107520 | 14 12 10975310 |
| **15** | 15 11630 | 15 1297420 | 15 13 11975320 |
| **16** | 16 10740 | 16 13 12 9 6 3 0 | 16 15 13 1086420 |
| **17** | 17 12840 | 17 14 118530 | 17 14 12 1086420 |
| **18** | 18 15940 | 18 14 129630 | 18 16 13 1196420 |
| **19** | 19 13940 | 19 16 139630 | 19 16 13 1086420 |
| **20** | 20 13950 | 20 17 13 10630 | 20 17 15 1297420 |
| **21** | 21 17 11 5 0 | 21 17 13 10630 | 21 18 15 12 107520 |
| **22** | 22 16 12 5 0 | 22 17 12 10630 | 22 19 17 14 118520 |
| **23** | 23 17 11 5 0 | 23 19 15 11730 | 23 19 15 12 106420 |
| **24** | 24 20 11 5 0 | 24 19 16 13850 | 24 20 17 15 129630 |
| **25** | 25 18 12 6 0 | 25 22 17 13840 | 25 22 18 15 129630 |
| **26** | 26 17 13 6 0 | 26 21 17 13940 | 26 23 20 16 139630 |
| **27** | 27 20 13 7 0 | 27 24 19 14 10 5 0 | 27 23 20 16 129630 |
| **28** | 28 21 15 7 0 | 28 25 20 15 10 5 0 | 28 26 22 18 14 10630 |
| **29** | 29 20 14 8 0 | 29 24 19 14 1050 | 29 25 21 17 14 10630 |
| **30** | 30 20 13 8 0 | 30 24 19 14950 | 30 27 22 18 139630 |
| **31** | 31 23 15 7 0 | 31 25 20 15 10 5 0 | 31 27 23 19 15 11730 |
| **32** | 32 25 15 7 0 | 32 27 21 16 10 5 0 | 32 28 23 20 17 12840 |
| **33** | 33 25 16 8 0 | 33 29 23 17 11 5 0 | 33 29 24 20 16 12840 |
| **34** | 34 24 15 7 0 | 34 27 20 16 10 5 0 | 34 30 26 21 16 12840 |
| **35** | 35 27 17 8 0 | 35 28 23 17 10 5 0 | 35 31 26 22 17 12840 |
| **36** | 36 25 17 8 0 | 36 29 24 18 12 6 0 | 36 31 27 22 17 13840 |
| **37** | 37 28 18 9 0 | 37 31 25 18 12 6 0 | 37 32 27 23 18 13950 |
| **38** | 38 28 20 9 0 | 38 33 26 20 12 6 0 | 38 33 28 23 18 14940 |
| **39** | 39 28 18 9 0 | 39 32 26 19 13 7 0 | 39 34 29 24 19 14950 |
| **40** | 40 29 21 10 0 | 40 34 27 19 12 6 0 | 40 36 30 26 20 15 1050 |
| **41** | 41 29 19 10 0 | 41 34 27 20 13 6 0 | 41 36 31 26 20 15 10 5 0 |
| **42** | 42 31 19 10 0 | 42 34 28 20 14 7 0 | 42 36 31 26 21 16 11 5 0 |
| **43** | 43 33 21 10 0 | 43 36 28 20 14 7 0 | 43 37 31 25 20 15 10 5 0 |
| **44** | 44 31 22 11 0 | 44 37 29 21 14 7 0 | 44 38 32 27 23 17 11 5 0 |
| **45** | 45 32 22 10 0 | 45 37 31 23 16 8 0 | 45 39 33 27 22 16 11 5 0 |
| **46** | 46 33 23 10 0 | 46 39 30 21 14 7 0 | 46 40 34 28 23 17 11 5 0 |
| **47** | 47 35 24 11 0 | 47 38 29 21 14 7 0 | 47 41 35 29 23 17 11 5 0 |
| **48** | 48 38 26 13 0 | 48 41 34 25 16 7 0 | 48 43 36 30 25 19 12 6 0 |
| **49** | 49 35 23 12 0 | 49 40 31 24 16 8 0 | 49 43 37 31 24 18 12 6 0 |
| **50** | 50 39 24 12 0 | 50 43 34 24 16 8 0 | 50 44 37 30 25 18 12 6 0 |

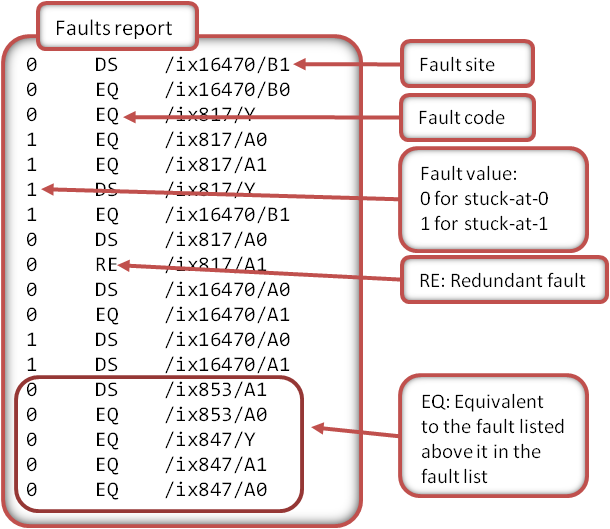
****Appendix 4: Test coverage report

Use the *report statistic* command to generate coverage statistics.



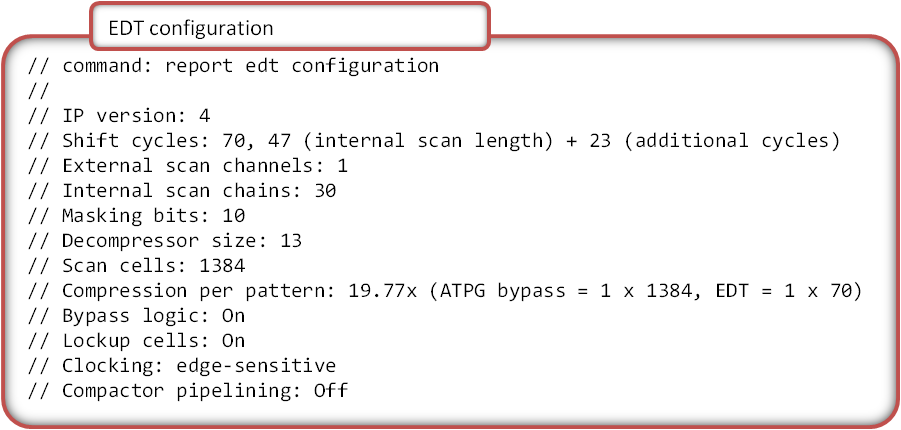
Appendix 5: Faults report

Use the *REPort faults* command in the *FastScan* or *TestKompress* dofile to generate the list of the detected faults.



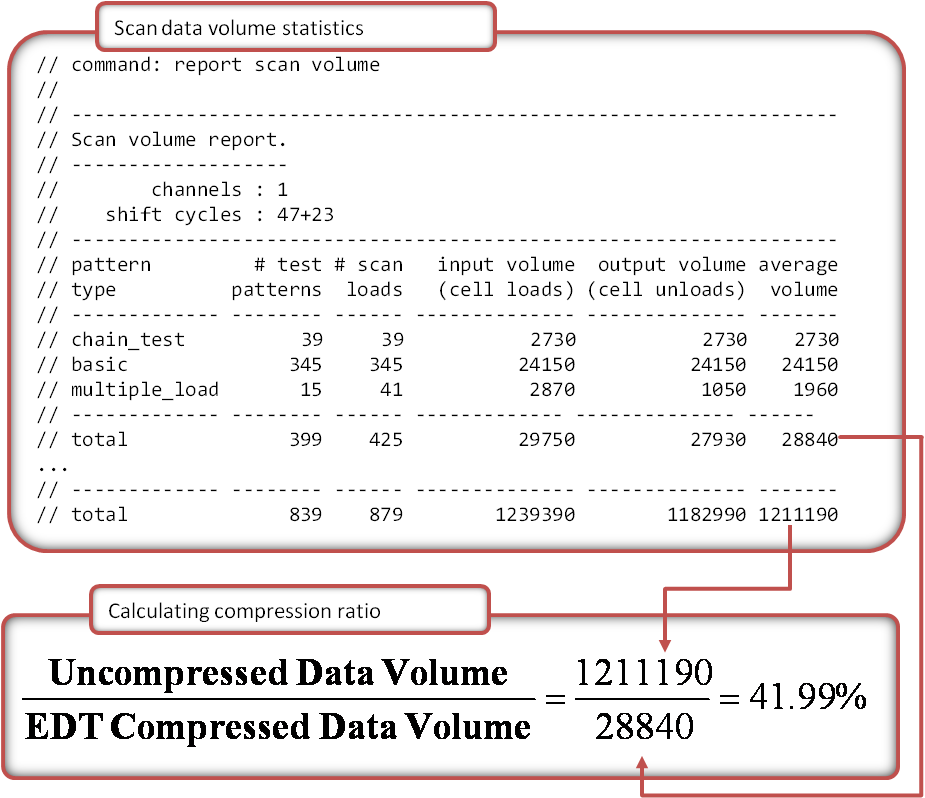
Appendix 6: EDT configuration report

Use the *REPort EDT Configuration* command in the *FastScan* or *TestKompress* dofile to generate the basic EDT configuration report.

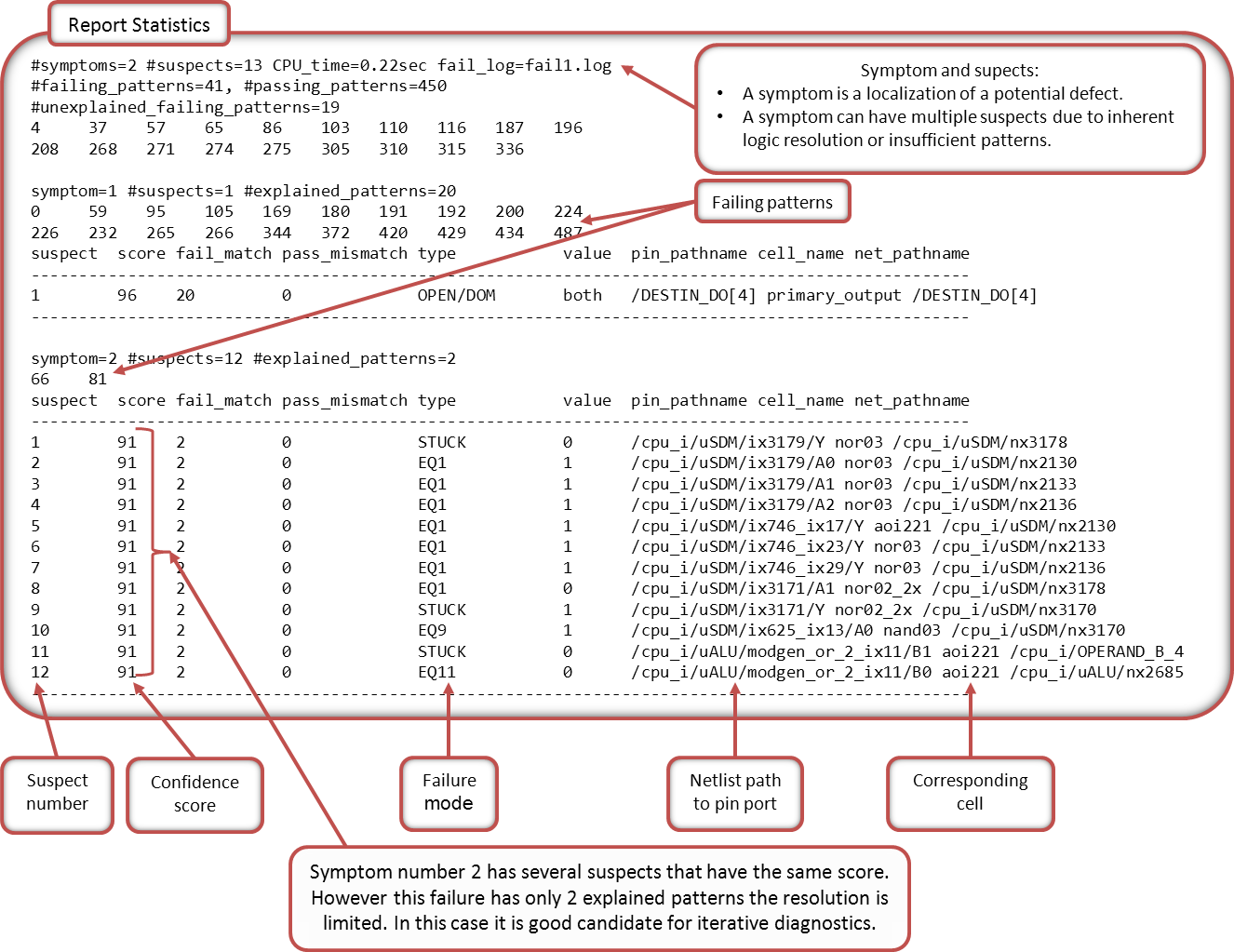


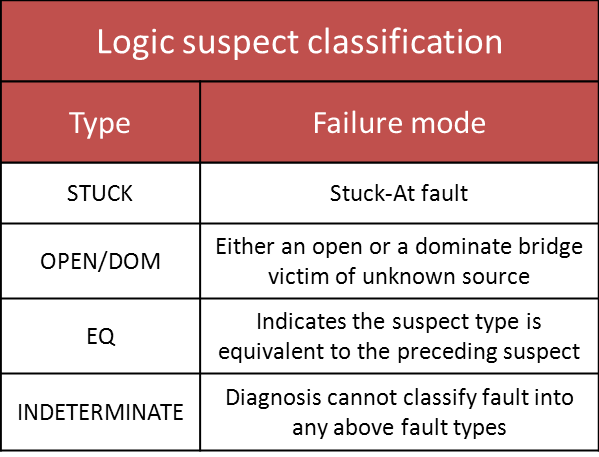
Appendix 7: Compression ratio

Use the *REPort Scan Volume* command to generate compressed and uncompressed data volume statistics.

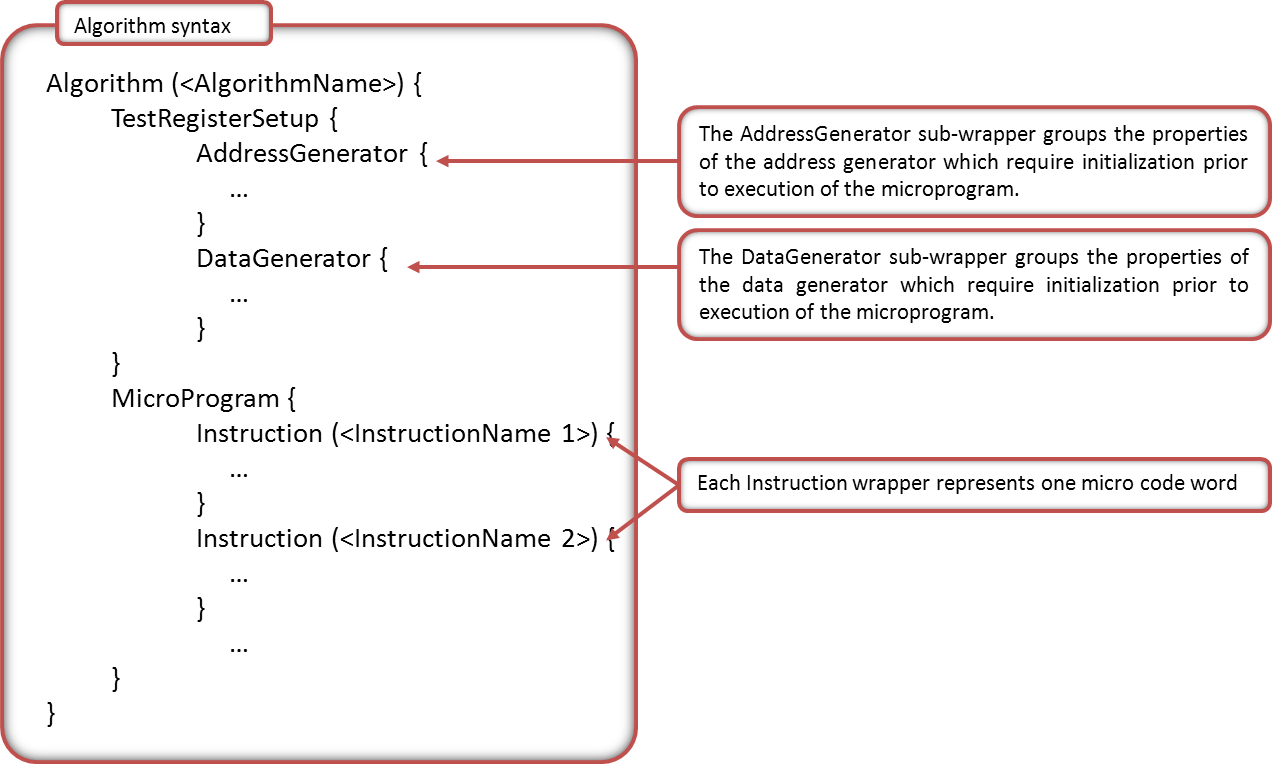


Appendix 8: Diagnosis report





Appendix 9: Tessent MemoryBIST algorithm syntax



1. J. Rajski, J. Tyszer, “Primitive Polynomials Over GF(2) of Degree up to 660 with Uniformly Distributed Coefficients,” *JETTA*, vol. 19, 2003. [↑](#footnote-ref-1)