Test Pattern Validation User Guide

Version K-2015.06-SP4, December 2015

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx. All other product or company names may be trademarks of their respective owners. Inc.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

Preface

This preface is comprised of the following sections:

About This User Guide

The Test Pattern Validation User Guide describes MAX Testbench and PowerFault. You use these tools to validate generated test patterns. This manual assumes you understand how to use TetraMAX® ATPG to generate test patterns as described in the *TetraMAX ATPG User Guide*.

You can obtain more information on TetraMAX ATPG features and commands by accessing TetraMAX ATPG Online Help.

Audience

This manual is intended for design engineers who have ASIC design experience and some exposure to testability cone timepts and strategies.

This manual is also useful for test engineers who incorporate the test vectors produced by TetraMAX ATPG into test programs for a particular tester or who work with DFT netlists.

Related Publications

For additional information about TetraMAX ATPG, see Documentation on the Web, which is available through SolvNet® at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to read the documentation for the following related Synopsys products: DFTMAX™ and Design Compiler®.

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the TetraMAX ATPG Release Notes on the SolvNet site.

To see the TetraMAX ATPG Release Notes:

1. Go to the SolvNet Download Center located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select TetraMAX ATPG, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as pin1 [pin2 ... pinN]
|	Indicates a choice among alternatives, such as low | medium | high. (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_environment_viewer`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a

wide range of Synopsys online services including software downloads, documentation on the Web, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to the SolvNet site at http://solvnet.synopsys.com, clicking Support, and then clicking “Open a Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at: http://www.synopsys.com/Support/GlobalSupportCenter/Pages

1

Introduction

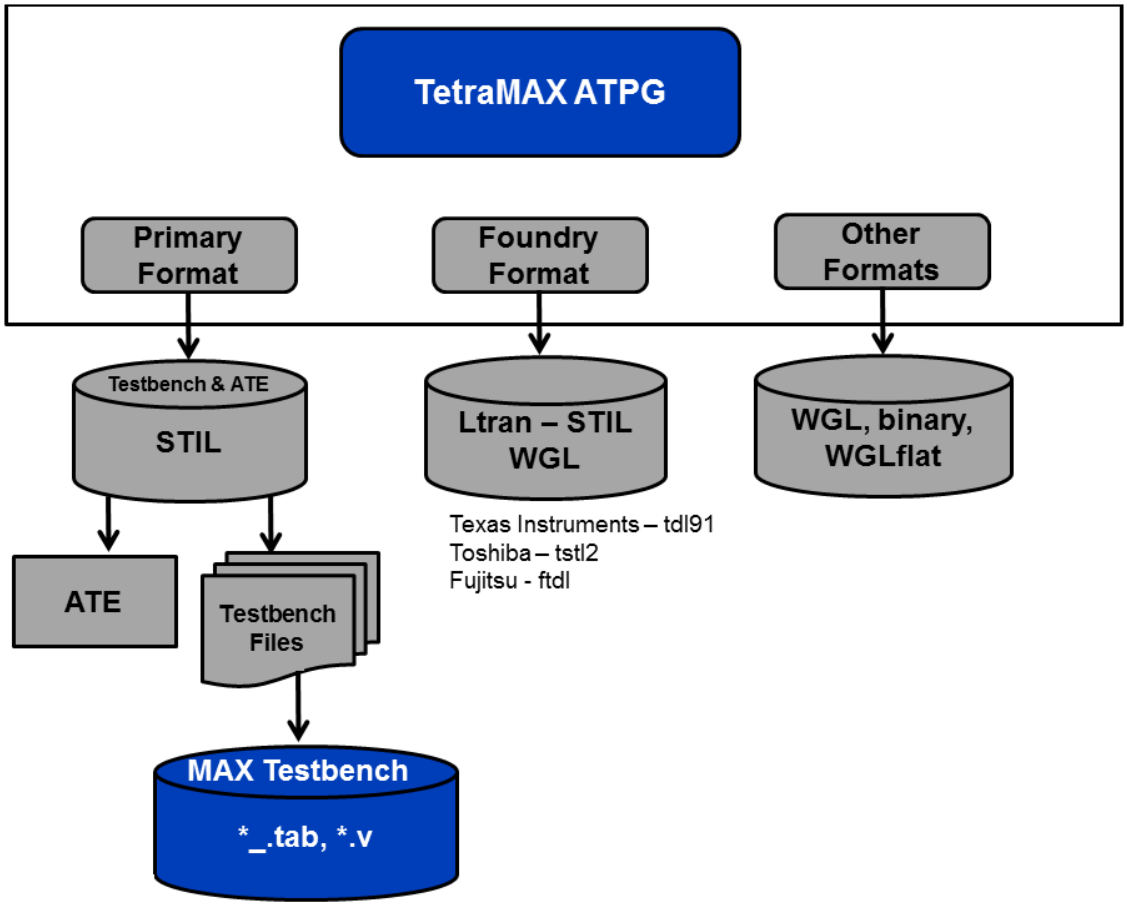
The Test Pattern Validation User Guide describes the Synopsys tools you can use to validate generated test patterns. This includes MAX Testbench, which validates STIL patterns created from TetraMAX ATPG, and PowerFault, which validates IDDQ patterns created from TetraMAX ATPG.

The following sections provide an introduction to this user guide:

TetraMAX Pattern Format Overview

Figure 1 shows an overview of the TetraMAX pattern formats.

Figure 1 TetraMAX ATPG Pattern Formats



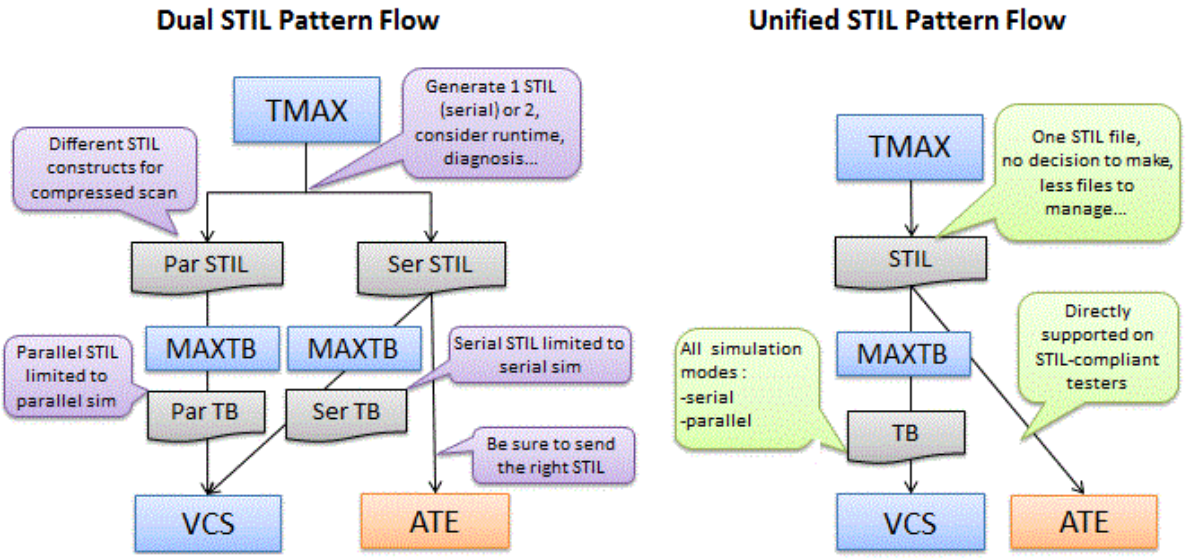
Writing STIL Patterns

TetraMAX ATPG creates unified STIL patterns by default. This simplifies the validation flow considerably because only a single STIL file is required to support all simulation modes (you do not need to write both serial and a parallel formats).

You can use unified STIL patterns in MAX Testbench. This avoids many of the issues presented by the dual STIL flow, and is based only on the actual STIL file targeted for the tester.

You can use a single unified STIL pattern file to perform all types of simulation, including parallel and mixed serial and parallel.

Figure 2 Comparing Combined Pattern Validation Flows



The `write_patterns` command includes several options that enable TetraMAX ATPG to produce a variety of pattern formats.

The `-format stil` option of the `write_patterns` command writes patterns in the proposed IEEE-1450.1 Standard Test Interface Language (STIL) for Digital Test Vectors format. For more information on the proposed IEEE-1450.1 STIL for Digital Test Vectors format (extension to the 1450.0-1999 standard), see Appendix E STIL Language Format in the TetraMAX ATPG User Guide. This format can be both written and read. However, only a subset of the language written by TetraMAX ATPG is supported for reading back in.

The `-format stil99` option of the `write_patterns` command writes patterns in the official IEEE-1450.0 Standard Test Interface Language (STIL) for Digital Test Vectors format. This format may be both written and read, but only the subset of the language written by TetraMAX ATPG is supported for reading back in.

Note: You must use a 1450.0-compliant DRC procedure as input when to write output in stil99 format.

The syntax generated when using the `-format stil` option is part of the proposed IEEE 1450.1 extensions to STIL 1450-1999.

If you use the `-format stil` or `stil99` options, TetraMAX ATPG generates a STIL file with a name in the filename `<pfile>.<ext>` in which you specified `write_patterns pfile>.<ext>.`

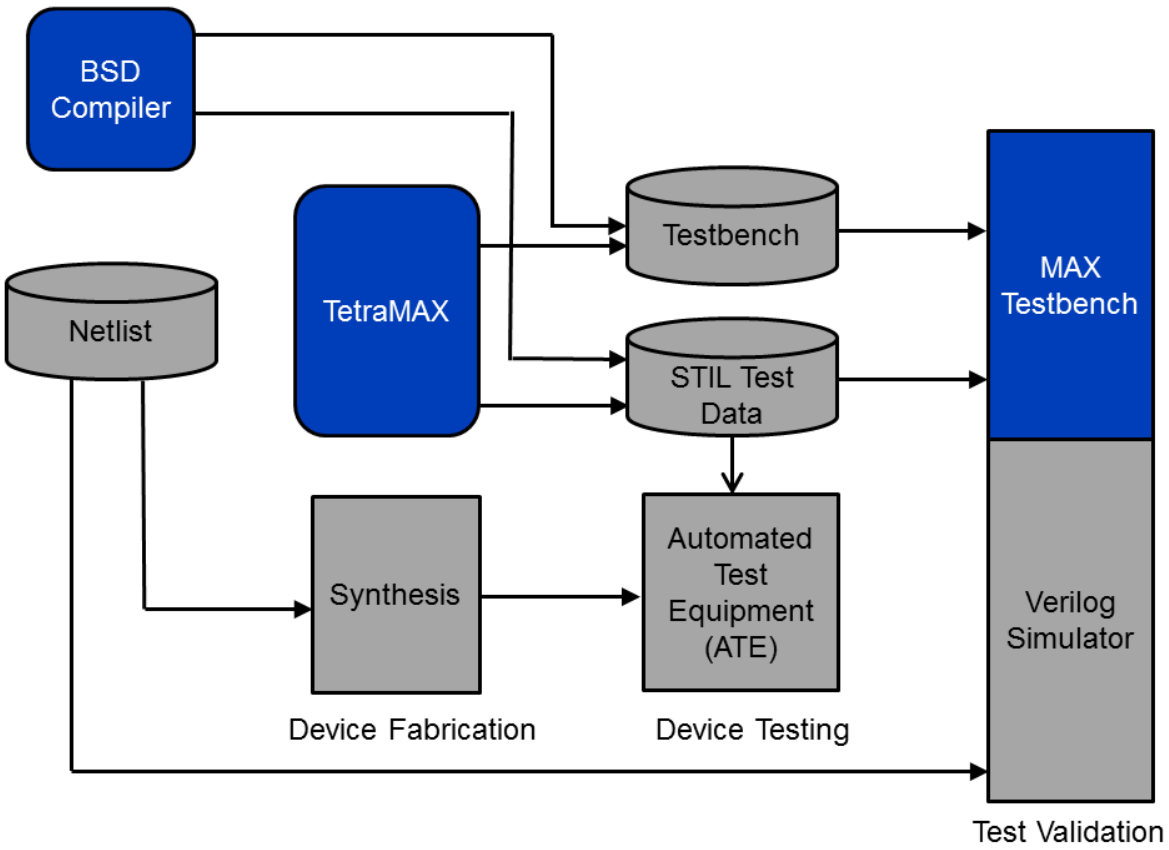
When you use the `-format stil` or `-format stil99` options, you can also use the `-serial` or `-parallel` options to specify TetraMAX ATPG to write patterns in serial

(expanded) or parallel form. See the description of the write_patterns command in TetraMAX Help for detailed information on using these options.

Design to Test Validation Flow

Figure 3 shows the validation flow using MAX Testbench. In this flow, test simulation and manufactured-device testing use the same STIL-format test data files.

Figure 3 Design-to-Test Validation Flow



When you run the Verilog simulation, MAX Testbench applies STIL-formatted test data as stimulus to the design and validates the design's response against the STIL-specified expected data. The simulation results ensure both the logical operation and timing sensitivity of the final STIL test patterns generated by TetraMAX ATPG.

MAX Testbench validates the simulated device response against the timed output response defined by STIL. For windowed data, it confirms that the output response is stable within the windowed time region.

Installation

The tools described in this manual can be installed as standalone products or over an existing Synopsys product installation (an “overlay” installation). An overlay installation shares certain support and licensing files with other Synopsys tools, whereas a standalone installation has its own independent set of support files. You specify the type of installation you want when you install the product.

You can obtain installation files by downloading them from Synopsys using electronic software transfer (EST) or File Transfer Protocol (FTP).

An environment variable called `SYNOPSYS` specifies the location for the TetraMAX ATPG installation. You need to set this environment variable explicitly.

Complete installation instructions are provided in the *Installation Guide* that comes with each product release.

Specifying the Location for TetraMAX Installation

TetraMAX ATPG requires the `SYNOPSYS` environment variable, a variable typically used with all Synopsys products. For backward compatibility, `SYNOPSYS_TMAX` can be used instead of the `SYNOPSYS` variable. However, TetraMAX ATPG looks for `SYNOPSYS` and if not found, then looks for `SYNOPSYS_TMAX`. If SYNOPSYS_TMAX is found, then it overrides `SYNOPSYS` and issues a warning that there are differences between them.

The conditions and rules are as follows:

- `SYNOPSYS` is set and `SYNOPSYS_TMAX` is not set. This is the preferred and recommended condition.
- `SYNOPSYS_TMAX` is set and `SYNOPSYS` is not set. The tool will set `SYNOPSYS` using the value of `SYNOPSYS_TMAX` and continue.
- Both `SYNOPSYS` and `SYNOPSYS_TMAX` are set. `SYNOPSYS_TMAX` will take precedence and SYNOPSYS is set to match before invoking the kernel.
- Both `SYNOPSYS` and `SYNOPSYS_TMAX` are set, and are of different values, then a warning message is generated similar to the following:

```
WARNING: $SYNOPSYS and $SYNOPSYS_TMAX are set differently,
using $SYNOPSYS_TMAX
WARNING: SYNOPSYS_TMAX = /mount/groucho/joeuser/tmax
WARNING: SYNOPSYS = /mount/harpo/production/synopsys
WARNING: Use of SYNOPSYS_TMAX is outdated and support for this
is removed in a future release. Use SYNOPSYS instead.
```

2

Using MAX Testbench

MAX Testbench is a pattern validation tool that converts TetraMAX STIL test vectors for physical device testers into Verilog simulation vectors.

The following sections describe how to use MAX Testbench:

Overview

MAX Testbench simulates and validates STIL test patterns used in an ATE environment. These patterns are used in an ATE environment.

MAX Testbench reads a STIL file generated from TetraMAX ATPG, interprets its protocol, applies its test stimulus to the DUT, and checks the responses against the expected data specified in the STIL file. MAX Testbench is considered a genuine pattern validator because it uses the actual TetraMAX ATPG STIL file used by the ATE as an input to test the DU.

MAX Testbench supports all STIL data generated by TetraMAX ATPG, including:

- All simulation mechanisms (serial, parallel and mixed serial/parallel)
- All type of faults (SAF, TF, DFs, IDDQ and bridging)
- All types of ATPG (Basic ATPG, Fast and Full Sequential)
- STIL from BSDC
- All existing DFT structures (e.g., normal scan, adaptive scan, PLL including on-chip clocking, shadow registers, differential pads, lockup latches, shared scan-in …)

MAX Testbench does not support DBIST/XDBIST or core integration.

Adaptive scan designs run in parallel mode only when translating from a parallel STIL format written from TetraMAX ATPG. Likewise, for serial mode, adaptive scan designs run only when translating from a serial STIL format written from TetraMAX ATPG.

Licenses

MAX Testbench requires the "Test-Validate" production key. The SYNOPSYS environment variable is used to recover the license system paths (this variable is also used to point to the stil.err file path).

Installation

The command setup and usage for MAX Testbench is as follows:

```
alias stil2Verilog 'setenv SYNOPSYS /install_area/latest;
$SYNOPSYS/
platform/syn/bin/stil2Verilog'
```

Then execute the following:

```
stil2Verilog -help
```

Obtaining Help

To access help information, specify the `-help` option on the tool command line. This command will print the description of all options.

There is no specific man page for each error or warning. The messages that are printed if errors occur are clear enough to enable you to adjust the command line to continue.

See Also

Writing ATPG Patterns in TetraMAX Help

Running MAX Testbench

You can run the MAX Testbench using either the `write_testbench` command or the `stil2Verilog` command. The `write_testbench` command enables you to run MAX Testbench without leaving the TetraMAX environment, and the `stil2Verilog` command is a standalone executable.

The MAX Testbench flow consists of the following basic steps:

1. Use TetraMAX ATPG to write a STIL pattern file.

   ```
   TEST-T> write_patterns STIL_pat_file -format STIL
   ```

   For details on using the `write_patterns` command, see "Writing ATPG Patterns in the *TetraMAX User Guide*."

2. Specify the `write_testbench` or `stil2Verilog` command using the STIL pattern file generated from the `write_patterns` command.

   Examples:

   ```
   % write_testbench -input stil_pattern_file.stil \
       -output Verilog_testbench.v

   % stil2Verilog stil_pattern_file.stil Verilog_testbench.v
   ```

   Two files are generated:

   - The first file is the Verilog principal file, which uses the following convention: *`Verilog_Testbench_filename`*`.v`.
   - The second generated file is a data file named *`Verilog_Testbench_filename`*`.dat`.

   An example of the output printed after running the `stil2Verilog` command is as follows:

   ```
   ######################################################
   #
   # STIL2VERILOG #
   #
   # Copyright (c) 2007-2014 SYNOPSYS INC. ALL RIGHTS RESERVED #
   #
   ######################################################
   maxtb> Parsing command line...
   maxtb> Checking for feature license...
   maxtb> Parsing STIL file "comp_usf.stil" ...
   ... STIL version 1.0 ( Design 2005) ...
   ```

```
... Building test model ...
... Signals ...
... SignalGroups ...
... Timing ...
... ScanStructures : "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
"sccompin0" "sccompin1" "sccompout0" "sccompout1" "sccompout2"
"sccompout3" "sccompin2" "sccompin3" ...
... PatternBurst "ScanCompression_mode" ...
... PatternExec "ScanCompression_mode" ...
... ClockStructures "ScanCompression_mode": pll_controller ...
... CompressorStructures : "test_U_decompressor_
ScanCompression_mode" "test_U_compressor_ScanCompression_mode"
...
... Procedures "ScanCompression_mode": "multiclock_capture"
"allclock_capture" "allclock_launch" "allclock_launch_capture"
"load_unload" ...
... MacroDefs "ScanCompression_mode": "test_setup" ...
... Pattern block "_pattern_" ...
... Pattern block "_pattern_ref_clk0" ...

maxtb> Info: Event ForceOff (Z) interpreted as CompareUnknown
(X) in the event waves of WFT "_multiclock_capture_WFT_"
containing both compare and force types (I-007)
maxtb> STIL file successfully interpreted (PatternExec:
""ScanCompression_mode"").
maxtb> Total test patterns to process 21
maxtb> Detected a Scan Compression mode.
maxtb> Test data file "comp_usf.dat" generated successfully.
maxtb> Test bench file "comp_usf.v" generated successfully.
maxtb> Info (I-007) occurred 2 times, use -verbose to see all
occurrences.
maxtb> Memory usage: 6.9 Mbytes. CPU usage: 0.079 seconds.
maxtb> End.
```

3. Run the simulation.

   Invoke the VCS simulator using the following command line:

   ```
   % vcs Verilog_testbench_file design_netlist \
      -v design_library
   ```

   Note the following:

   - When running zero-delay simulations, you must use the `+delay_mode_zero` and `+tetramax` arguments.

See Also

Configuring MAX Testbench
Predefined Verilog Options

Using the write_testbench Command

The syntax for the `write_testbench` command is as follows:

```
write_testbench
-input [stil_filename | {-split_in \
    {list_of_stil_files_for_split_in\}}]
-output testbench_name
[-generic_testbench]
[-patterns_only]
[-replace]
[-config_file config_filename]
[-parameters {list_of_parameters}]
```

The options are described as follows:

`-input [stil_filename | {-split_in \ {list_of_stil_files_for_split_in\}}]`

The *stil_filename* argument specifies the path name of the previous TetraMAX ATPG-generated STIL file requested by the equivalent Verilog testbench. You can use a previously generated STIL file as input. This file can originate from either the current session or from an older session using the `write_patterns` command.

The following syntax is used for specifying split STIL pattern files as input (note that backslashes are required to escape the extra set of curly braces):

```
{-split_in \{list_of_stil_files_for_split_in\}}
```

The following example shows how to specify a set of split STIL pattern files:

```
write_testbench -input {-split_in
    \{patterns_0.stil patterns_1.stil\}} -output pat_mxtb
```

`-output testbench_name`

Specifies the names used for the generated Verilog testbench output files. Files are created using the naming convention `<testbench_name>.v` and `<testbench_name>.dat`.

`-generic_testbench`

Provides special memory allocation for runtime programmability. Used in the first pass of the runtime programmability flow, this option is required because the Verilog 95 and 2001 formats use static memory allocation to enable buffers and arrays to store and manipulate .dat file information. For more information on using this command, see "Runtime Programmability."

`-patterns_only`

Used in the second pass, or later, run of the runtime programmability flow, this option initiates a light processing task that merges the new test data in the test data file. This option also enables additional internal instructions to be generated for the special test data file. For more information on using this command, see "Runtime Programmability."

`-replace`
Forces the new output files to replace any existing output files. The default is to not allow a replacement.

`-config_file` *`config_filename`*
Specifies the name of a configuration file that contains a list of customized options to the MAX Testbench command line. See "Customized MAX Testbench Parameters Used in a Configuration File with the write_testbench Command" for a complete list of options that can be used in the configuration file. You can use a configuration file template located at $SYNOPSYS/auxx/syn/ltran.

`-parameters {`*`list_of_parameters`*`}`
Enables you to specify additional options to the MAX Testbench command line. See "MAX Testbench Command-Line Parameters Used with the write_ testbench Command" for a complete list of parameters you can use with the `-parameters` option.

If you use the `-parameters` option, make sure it is the last specified argument in the command line, otherwise you might encounter some Tcl UI conversion limitations.

A usage example for this option is as follows:

```
write_testbench -parameters { -v_file \"design_file_names\" -v_
lib \"library_file_names\" -tb_module module_name -config_file
config1}
```

Note the following:

- All the parameters must be specified using the Tcl syntax required in the TMAX shell. For example: `-parameters {param1 param2 -param3 \"param4\"}`
- quotation marks must have a backslash, as required by Tcl syntax, to be interpreted correctly and passed directly to the MAX Testbench command line.
- Parameters specified within a `-parameters {}` list are order-dependent. They are parsed in the order in which they are specified, and are transmitted directly to the MAX Testbench command line. These parameters must follow the order and syntax required for the MAX Testbench command line.

Using the stil2Verilog Command

The syntax for the `stil2Verilog` command is as follows:

`stil2Verilog [`*`pattern_file`*`] [`*`tbench_file`*`] [`*`options`*`]`

The syntax descriptions are as follows:

`pattern_file`
Specifies the ATPG-generated STIL pattern file used as input. This file must be specified, except when the `-split_in` option is used.

`tbench_file`
Specifies the name of the testbench file to generate. When the *`tb_file_name`* is specified, a `.v` extension is added when generated the protocol file, and a `.dat` extension is used when generating the test data file. You should use only

the root name with the command line, for example, `stil2erilogpat.stil tbench`, that generates tbench.v and tbench.dat files in the current working directory. This argument is optional when the `-generate_config` or `-report` options are specified.

Other optional arguments can be specified, as shown in the following syntax. The defaults are shown in bold enclosed between parentheses.

```
-config_file TB_config_file
-first d
-force_enhanced_debug
-generate_config config_file_template
-generic_testbench
-help [msg_code]
-last d
-log log_file
-parallel
-patterns_only
-replace
-report
-run_mode (go-nogo) | diagnosis
-sdf_file sdf_file_name
-serial
-ser_only
-sim_script <= [ [vcs] | [mti] | [nc] | [ xl] ]
-split_in { 1.stil, 2.stil… }  | { dir1 /*.stil } testbench_name
-split_out pat_intervstil_filetestbench_name
-tb_format <= (v95) | v01 | sv
-tb_module module_name
-verbose
-version
-v_file { design_file_names }
-v_lib { library_file_names }
```

The descriptions for the optional syntax items are as follows:

`-config_file TB_config_file`

MAX Testbench can be configured at several levels. At the top of the MAX Testbench configuration file, you can edit the `set cfg_*` variables to define the various testbench defaults, such as the progress message interval time and the simulation time unit. The second half of the configuration file contains a set of editable setup parameters for the VCS/MIT/Cadence simulation script file. The *TB_config_file* parameter specifies the name of the configuration file used to set up the testbench at generation time. See "Example of the Configuration Template".

`-first d`

Specifies the first pattern number that TetraMAX ATPG writes. The default is to begin with pattern 0. **Note:** For Full-Sequential patterns, this option might cause simulation

mismatches.

`-force_enhanced_debug`

Forces MAX Testbench to halt if any errors are encountered when processing the parallel strobe data (PSD) file. The default is to not force MAX Testbench to stop. For more information on the PSD file, see "Understanding the PSD File."

`-generate_config` *`config_file_template`*

MAX Testbench can generate a configuration file template that you can edit and modify. The *`config_file_template`* parameter specifies the path where the configuration file template is written.

`-generic_testbench` (or `-streaming_patterns`)

Generates a generic testbench that can load future test patterns (.dat files) without recompiling. For more information on using this command, see "Runtime Programmability."

`-help [`*`msg_code`*`]`

Shows all possible options, and the complete `stil2Verilog` syntax and exits. If *`msg_code`* is specified, then prints the help page corresponding to that code *`msg_code`* syntax: '1-letter'-'3-digit code' where letter can be '`E`', '`W`' or '`I`' and the 3-digit code must correspond to a valid code in the range [000-999] For example: `E-001`, `W-010`

`-last` *`d`*

Specifies the last pattern number for the patterns to be written. The default is to end with the last available pattern.

`-log`

Generates a log file.

`-parallel`

Specifies the parallel load mode for simulation, which is the default.

`-patterns_only`

Generates test patterns only (.dat file) to be used with an existing equivalent testbench (.v file). For more information on using this command, see "Runtime Programmability."

`-replace`

Forces MAX Testbench to overwrite the testbench files, the configuration file template, and simulation script.

`-report`

Displays the configuration setting and test pattern information. It has the following parameters (note that multiple parameters can be specified if separated by commas):

`all` — displays all the information (default in verbose mode)

`config` — displays the configuration setting

`dft` — displays DFT structure information

`drc` — displays DRC warnings

`flow` — displays STIL pattern flow

`macro` — displays macro information

`nb_patterns` — displays the total number of patterns to be executed

`proc` — displays procedure information

`sigs` — displays all the signal information

`sig_groups` — displays all the signal groups information

`wft` — displays WaveformTable information

`-run_mode go-nogo | diagnosis`

Allows the targeting of either Go-nogo mode (the default) or diagnosis mode. For details, see "Setting the Run Mode."

`-sdf_file` *`sdf_file_name`*

Specifies the SDF file name used for back annotation.

`-serial`

Specifies the serial load mode simulation. The default simulation scan load is parallel. The same behavior can be obtained by using the `+define+tmax_serial` compiler directive to force the simulation of all patterns to be serial. If `+tmax_serial=N` is used, MAX Testbench forces serial simulation of the first `N` patterns, and then starts parallel simulation of the remaining patterns

`-ser_only`

Generates the testbench file for serial load mode only. This allows a reduction in the size of the testbench and speeds up the simulation.

`-shell`

Runs the tool in shell mode.

`-sim_script vcs | mti | nc | xl`

The `sim_script <simulator>` option specifies a simulation script to be generated together with the testbench file. You also must provide the `v_file` and `v_lib` options. Note that only VCS scripts are supported; the other simulator scripts that are generated conform to the simulator script generated by TMAX (`write_patterns` command). The argument specifies the target simulator:

- `vcs` — VCS simulator command shell script
- `mti` — ModelSim simulator command shell script
- `xl` — Cadence XL simulator command shell script
- `nc` — Cadence NCVerilog simulator command shell script

Note the specification of several arguments at the same time to target all of the simulators is supported as repetitive entries "`-sim_script vcs -sim_script mti -sim_script xl`"

The output name of the generated script file is: `<name_of_testbench_file>_<simulator>.sh`.

`-split_in { 1.stil, 2.stil… } | { dir1 /*.stil }`

Specifies MAX Testbench to use split STIL files based on either a detailed list of STIL files or a generic list description using the wildcard (*) symbol. In the generic list format, the files are recognized in alphabetical order. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket, as shown in the following example:

```
stil2Verilog -split_in { bill.patt.stil.ts_and_chain
bill.patt_0.stil bill.patt_1.stil bill.patt_2.stil bill.patt_
3.stil bill.patt_4.stil bill.patt_5.stil bill.patt_6.stil
bill.patt_7.stil bill.patt_8.stil bill.patt_9.stil bill.patt_
10.stil } bill.pat_stil.v -replace
```

Note that you can also specify multiple files in the configuration file. For more information on this option, see "Using Split STIL Pattern Files".

`-split_out` *`pat_intervalstil_file`*

Specifies MAX Testbench to split STIL files, The *`pat_interval`* argument specifies the maximum number of patterns that a given .dat file will contain. For more information on this option, see "Splitting Large STIL Files".

`-tb_format v95 | v01 | sv`

Specifies the testbench format applied to the *`tbench_file`* specification. The default is `v95`, and is currently the only supported option. Formats:

`v95` — Verilog 1995

`v01` — Verilog 2001

`sv` — SystemVerilog

`-tb_module` *`module_name`*

Specifies the module name for the top-level module of the Verilog testbench.

`-verbose`

Activates verbose mode.

`-version`

Prints the stil2Verilog banner, including the version.

`-v_file { `*`design_file_names`*`}`

Specifies design netilist source files (the DUT description) required to run the simulation. It is required when using the `sim_script` option. Wild characters are supported. Note that *`design_file_name1`* and *`design_file_nameN`* must be separated with spaces. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket (you can also specify multiple files in the configuration file).

`-v_lib { `*`library_file_names`*` }`

Specifies the library file (the DUT related technology library) required to run the simulation. This option is required when using `sim_script` option. Note that *`library_file_name1`* and *`library_file_nameN`* must be separated with spaces. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket, as shown in the following example:

```
stil2Verilog pats.stil maxtb -replace -v_lib { lib1.v lib2.v }
```

Note that you can also specify multiple files in the configuration file. Wildcard characters are supported for simulation script generation.

Setting the Run Mode

There are two basic run modes you can set when starting MAX Testbench using the stil2Verilog command: Go-nogo and Diagnosis.

The Go-nogo mode is set using the `-run_mode go-nogo` option. In this mode, MAX Testbench does the following:

- Sets the verbosity level to 0 (equivalent to using `+define+tmax_msg=0` at VCS compilation time)
- Makes the testbench reporting the beginning of each 5 patterns (equivalent to using `+define+tmax_rpt=5` at VCS compilation time)
- Initializes the file name for the collection of diagnostics failures to `<testbench_name>.diag`.

The Diagnosis mode is set using the `-run_mode diagnosis` option. In this mode, MAX Testbench saves the mismatches in the `<testbench_name>.diag file` in a pattern-based format compatible with the TetraMAX `run_diagnosis` command.

For example, the mismatches are recorded in the following manner:

```
30 test_so2 10 (exp=0, got=1)  // chain , V=313, T=31240.00 ns
30 test_so3 10 (exp=0, got=1)  // chain , V=313, T=31240.00 ns
30 test_so4 10 (exp=0, got=1)  // chain , V=313, T=31240.00 ns
```

These failures can be used by the TetraMAX diagnostics to identify the failing scan chain. You can print a report using the command `run_diagnosis -only_report_failures`.

The failures log file name default can be changed at the time the simulation is executed by using the following compiler directive:

```
% vcs ... +define+tmax_diag_file=\"<file_name>\"
```

The default can also be changed at the time the testbench is generated using the configuration file parameter `cfg_diag_file`.

See Also

Understanding the Failures File
Using the Failures File

Configuring MAX Testbench

Table 1 shows the possible configurations for MAX Testbench.

Table 1 MAX Testbench Behaviors

Config. Type	Config. File Option	Sim. Predefine Option

Pre-defined Verilog code that affects the simulator script generation.
Initial N serial (flattened scan) vectors.
`+tmax_serial=N`
	`set define_<user_def1> 0`
	Example:
	`set define_tmax_serial 0`
		`+tmax_serial` or `tmax_serial=N`
		Example:
		`+define+tmax_serial`

Pre-defined Verilog code that affects the simulator script generation.
Parallel scan access with N serial vectors.
`+tmax_parallel=N`
	`set define_<user_def1> 0`
	Example:
	`set define_tmax_parallel 0`
		`+tmax_parallel=N`
		Example:
		`+define+tmax_parallel`

Pre-defined Verilog code that affects the simulator script generation.
Number of patterns to simulate.
	`set define_<user_def1> 0`
	Example:
	`set define_ tmax_n_pattern_sim 10`
		`tmax_n_pattern_sim=N`
		Example
		`+define+tmax_n_pattern_sim=10`

Pre-defined Verilog code that affects the simulator script generation.
Generates a delay (a "dead period") for parallel scan access to align parallel load timing with serial load timing
	`set define_tmax_serial_timing`
	See Also:
	`cfg_serial_timing`
		`+define+tmax_serial_timing`

Top-level module	`#set tb_module_name <"new_name">`	

Table 1 MAX Testbench Behaviors (Continued)

Config. Type	Config. File Option	Sim. Predefine Option
Sets the severity level. NOTE: The command `drcw_severity` requires two mandatory parameters: `<rule_name>`: TetraMAX rule name (wild-card character '*' is supported) `<severity>`: severity level ("ignore"| "warning"|"error") Example: `set drcw_severity C11 warning`	`set drcw_severity <rule_name> <severity>`	
Overcomes the size optimization and generates an extended testbench. Setting of 1 creates a compact testbench.	`set cfg_tb_format_extended 0`	
Maximum number of patterns loaded simultaneously in the simulation process	`cfg_patterns_read_interval`	
Specifies the interval of the progress message (0 is disabled, N is every Nth pattern is reported) .	`cfg_patterns_report_interval`	`tmax_rpt=N`
Defines the verbose level. (See the Verbose Level section below.)	`cfg_message_verbosity_level`	`tmax_msg=N` (could be `0`, `1`, `2`, `3` and `4`)
Generates an extended-VCD of the simulation run	`cfg_evcd_file "evcd_file"`	

Table 1 MAX Testbench Behaviors (Continued)

Config. Type	Config. File Option	Sim. Predefine Option
Changes the failure log file's default name at the time the simulation is executed.	`cfg_diag_file "diag_file"` Causes the testbench to override the name in the testbench file.	`tmax_diag_ file=\"<file>\"` Affects the simulation runtime.
Configures the miscompare in pattern-based (N=1)format or cycle-based (N=2) format format		`+tmax_diag=N (N could be 1 or 2)`
Generates a delay for parallel scan access to align parallel load timing with serial load timing	`cfg_serial_timing` Affects the testbench only.	`tmax_serial_timing` Affects the simulation runtime.

Specifies the simulation time unit (i.e., time scale)	`cfg_time_unit`
	Example:
	`set cfg_time_unit "1ps"`
		N/A

Specifies the simulation time precision (i.e., time precision)	`cfg_time_precision`	N/A
Defines the DUT Module name (use only if tool asks for this parameter) .	`cfg_dut_module_name`	N/A

Table 1 MAX Testbench Behaviors (Continued)

Config. Type	Config. File Option	Sim. Predefine Option

Delays the release of all forced scan cells in the load_unload procedure to the next cycle by the specified time. The delay starts from the beginning of next cycle. This option is supported for the parallel dual STIL flow, but is not currently supported for the unified STIL flow.	`cfg_parallel_release_time`
	Must add units.
	Example:
	`cfg_parallel_release_time 50000ps`
		N/A

Reports the instance name of the failing cells during the simulation of a parallel-formatted STIL file. To enable the report, you must set the boolean variable to '1". The default, 0, turns off this reporting. Note that this feature impacts simulation memory consumption.	`cfg_parallel_stil_report_cell_name`
	Example:
	`cfg_parallel_stil_report_cell_name 1`
		N/A

Note the following:

- The “Command Line Option" column contains the `stil2Verilog` commands.
- The “Configuration File Option" column contains those variables available inside the configuration file when used in conjunction with the `-config_file <file_name>` option during the `stil2Verilog` execution.
- The “Simulator Predefine Option" column contains those options that can be used in a simulator script or also defined in the `-config_file <file_name>` option in the section titled "variables only affecting the simulator script generation".

  For example:

  A “Simulator Predefined Option” can be changed at the time the simulation is executed by using the following compiler directive:

  ```
  % vcs ... +define+tmax_serial=1
  ```
- In the first two rows of Table 1, the special case of `define_<user_def>` is used for any user-defined simulator variable. However, it is also used for variables that are hard-coded into the testbench, such as `tmax_serial` and `tmax_parallel`.

The default of `define_<user_def>` can also be changed at the time the testbench is generated using the `-sim_script vcs|mti|xl|nc` option along with defining the `-config_file <file_name>` option. With the line "`set define_<user_def1> 0`" modified as "`set define_tmax_serial=1`" inside the configuration file.

Example of the Configuration Template

You can generate the template file shown in Example 1 using the following command:

```
stil2Verilog -generate_config  TB_config_file
```

Example 1 Configuration Template Example

```
## STIL2VERILOG CONFIGURATION FILE TEMPLATE (go-nogo default
values) ##

# uncomment out the setting statement to use predefined variables
# the "set cfg_*" variables only affect the testbench definition

# cfg_patterns_read_interval: specifies the maximum number of
patterns loaded simultaneously in the simulation process
#set cfg_patterns_read_interval 1000

# cfg_patterns_report_interval: Specifies the interval of the
progress message
#set cfg_patterns_report_interval 5

#  cfg_message_verbosity_level: control for a prespecified set of
trace options
#set cfg_message_verbosity_level 0

# cfg_evcd_file evcd_file: generates an extended-VCD of the
simulation run
#set cfg_evcd_file "evcd_file"
# cfg_diag_file: generates a failures log file compliant with
TetraMAX diagnostics. This overrides the name in the tb file.
#set cfg_diag_file "diag_file"
# cfg_serial_timing: generates a delay for parallel scan access to
align parallel
load timing with serial load timing
#set cfg_serial_timing 0
# cfg_time_unit: specifies the simulation time unit
#set cfg_time_unit "1ns"
# cfg_time_precision: specifies the simulation time precision
#set cfg_time_precision "1ns"
#  cfg_dut_module_name: specifies the DUT module name to be tested
(variable to be used only when the tool asks for it)
#set cfg_dut_module_name "dut_module_name"
```

```
#### TB file formatting section
# cfg_tb_format_extended: specifies whether an extended TB file is
needed
#set cfg_tb_format_extended 0

# set drcw_severity <rule_name> <severity>
# The command "drcw_severity" needs two mandatory parameters:
#             - <rule_name>: TetraMAX rule name (wild-card
character '*' is supported)
#             - <severity>:  severity level
("ignore"|"warning"|"error")
#set drcw_severity C11 warning


### variables only affecting the simulator script generation
# define_<preprocessor_define>: specifies the preprocessor
definitions for the simulator
#set define_<user_def1> 0
#set define_<user_def2> "TRUE"
#design_files: specifies all source files required to run the
simulation
#set design_files "netlist1.v netlist2.v"
# lib_files: specifies all library source files required to run
the simulation
#set lib_files "lib1.v lib2.v"
# vcs_options: specifies the user VCS command line options
#set vcs_options "VCSoption1 VCSoption2"
# nc_options: specifies the user NCSim command line options
#set nc_options "NCoption1 NCoption2"
# mti_options: specifies the user ModelSim command line options
#set mti_options "MTIoption1 MTIoption2"
# xl_options: specifies the user Verilog XL command line options
#set xl_options "XLoption1 XLoption2"
```

An example configuration file is shown in Example 2 below.

Example 2 Example Configuration Rile

```
###### STIL2VERILOG CONFIGURATION FILE #####

# Specifies the maximum number of patterns
# loaded simultaneously in the simulation process
set cfg_patterns_read_interval 1000

# Specifies the interval of the progress message
set cfg_patterns_report_interval 5

# Control for a prespecified set of trace options
set cfg_message_verbosity_level 3
```

```
# Generates a failures log file compliant with
# TetraMAX diagnostics
set cfg_diag_file "diag_file"

# Specifies the DUT module name to be tested
#set cfg_dut_module_name "dut_module_name"

# Specifies all source files required to run the simulation
#set design_files "netlist1.v netlist2.v"

# other configurations…
```

To assign a value to a configuration parameter, you should use the following syntax:

```
set <config_parameter_name> <value>
```

Note: Every comment line must begin with "#".

See Also

Runtime Programmability
Predefined Verilog Options

Setting the Verbose Level

You can use the `[tmax_msg=N]` argument to set four different levels of verbosity output for MAX Testbench. Each level prints a specific set of data to help to follow the simulation execution. The levels are defined as follows:

- **Level 0** — The default. It prints the Header + Start + End information + Errors (if any)
- **Level 1** — Prints information from level 0 and adds the pattern information according to the value of `tmax_rpt` compile time option. This information includes the current time and vector and some basic information regarding the files/settings.
- **Level 2** — Prints information from level 1 and adds any Macro/Procedure execution. The Macro/Procedure information includes time, vector information and Shift statement
- **Level 3** — Prints information from level 2 and adds all executed statements in the pattern block (not only procedures and macros).
- **Level 4** — Prints information from level 3 and adds vector (a per cycle report).

See Also

Displaying Instance Names

Understanding the Failures File

When you set the `-run_mode diagnosis` option of the `stil2Verilog` command, MAX Testbench prints all miscompare messages to a file used with the `run_diagnosis` command for diagnostics. The format of this file is dependent of the pattern type (legacy scan, adaptive scan, or serializer), the simulation mode (serial or parallel), and the STIL type (dual or unified).

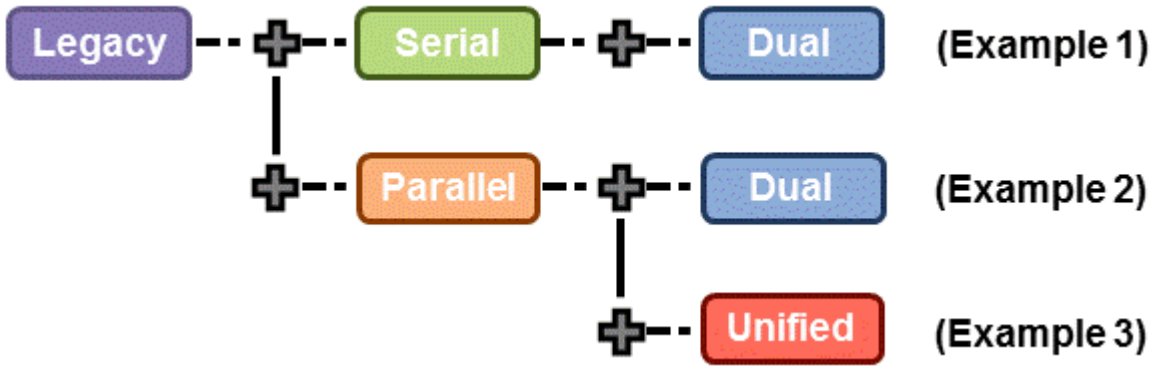
The following sections describe the relationship of the failures formats for each pattern type:

- Legacy Scan Failures
- Adaptive Scan Failures
- Serializer Scan Failures

MAX Testbench and Legacy Scan Failures

In legacy scan, given the serial/parallel and dual/unified types, the failure formats are the same. A failure contains the cycle count of the failure (`V=`), the expected data (`exp=`), the data captured (`got=`), the chain name (`chain`), the scan output pin name (`pin`), and the scan cell position (`scan cell`). Figure 1 describes the relationship of the failures for legacy scan.

Figure 1 Relationship of Failures Format for Legacy Scan



Example 1, Example 2, and Example 3 are reports for the same failure printed during the simulation of the patterns.

Example 1

```
Error during scan pattern 32 (detected during unload of pattern
31)

At T=49240.00 ns, V=493, exp=0, got=1, chain 4, pin test_so4, scan
cell 10
```

Example 2

```
Error during scan pattern 32 (detected during parallel unload of
pattern 31)

At T=16240.00 ns, V=163, exp=0, got=1, chain 4, pin test_so4, scan
cell 10
```

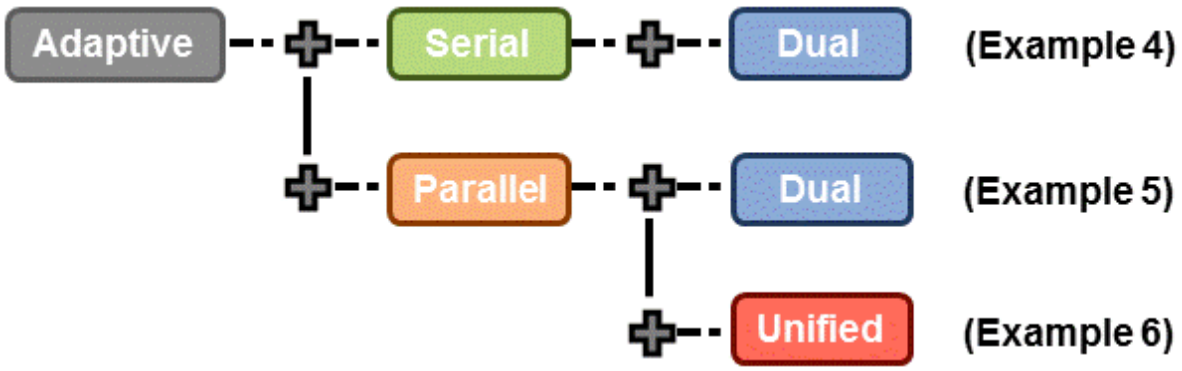
Example 3

```
Error during scan pattern 32 (detected during parallel unload of
pattern 31)

At T=16240.00 ns, V=163, exp=0, got=1, chain 4, pin test_so4, scan
cell 10
```

MAX Testbench and Adaptive Scan Failures

In adaptive scan the failure formats are not the same. A failure contains the cycle count of the failure (`V=`), the expected data (`exp=`), the data captured (`got=`), the chain name (`chain`) only for dual STIL flow parallel, the scan output pin name (`pin`) for dual STIL flow serial mode and unified STIL flow parallel mode. The pin information for dual STIL flow for parallel mode is the pin pathname of the failing scan cell output. The report also contains the scan cell position (`scan cell`).

Figure 2 Relationship of Failures Format for Adaptive Scan



Example 4, Example 5, and Example 6 are reports for the same failure printed during the simulation of the patterns.

Example 4

```
Error during scan pattern 31 (detected during unload of pattern
30)

At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so2, scan
cell 10
```

```
At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so3, scan
cell 10

At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so4, scan
cell 10
```

Example 5

```
Error during scan pattern 31 (detected during parallel unload of
pattern 30)

At T=15740.00 ns, V=158, exp=0, got=1, chain 10, pin

snps_micro.mic0.pc0.prog_counter_q_reg[11] .QN, scan cell 10
```

Note: In the case of dual STIL flow parallel mode for adaptive scan patterns, MAX Testbench, reports the failing scan chain and failing scan cell position. But, for performance reasons, the scan cell instance name for the failing position is not reported. However, it does report the scan cell instance name with position 0 for the failing scan chain.

Example 6

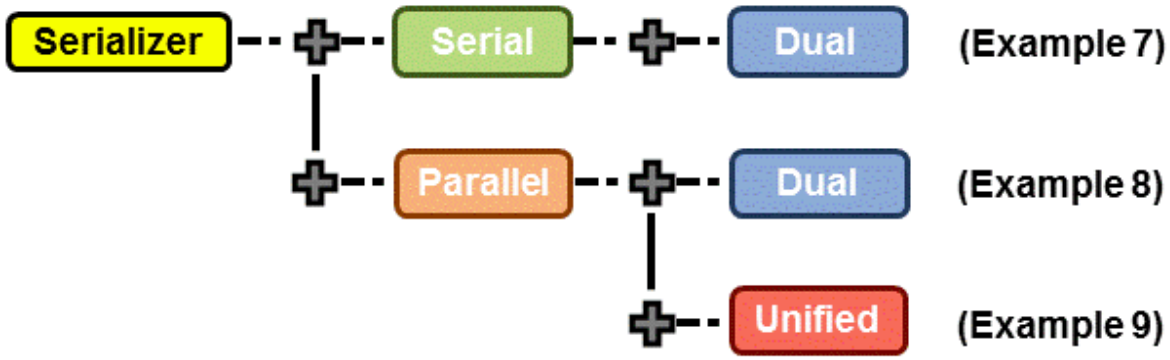
```
Error during scan pattern 31 (detected during parallel unload of
pattern 30)

Error during scan pattern 31 (detected during parallel unload of
pattern 30)

At T=15740.00 ns, V=158, exp=0, got=1, pin test_so3, scan cell 10

Error during scan pattern 31 (detected during parallel unload of
pattern 30)

At T=15740.00 ns, V=158, exp=0, got=1, pin test_so4, scan cell 10
```

Note: In the case of Unified STIL flow parallel mode for adaptive scan patterns, MAX Testbench reports the failing scan cell position only. The failing scan chain name and the failing scan cell instance name are not provided. You can use TetraMAX diagnostics to retrieve the failing scan chain name.

MAX Testbench and Serializer Scan Failures

Figure 3 describes the relationship of serializer scan failures.

Figure 3 Relationship of Failures Format for Serializer



Example 7

```
Error during scan pattern 5 (detected during unload of pattern 4)

At T=28340.00 ns, V=284, exp=0, got=1, chain , pin test_so1, scan
cell 2, serializer index 1

At T=28440.00 ns, V=285, exp=0, got=1, chain , pin test_so1, scan
cell 2, serializer index 2

At T=28540.00 ns, V=286, exp=1, got=0, chain , pin test_so1, scan
cell 2, serializer index 3
```

Note: In the case of the dual STIL flow parallel mode for serializer patterns, MAX Testbench reports the failing scan chain and failing scan cell position. But, for performance reasons, the scan cell instance name for the failing position is not reported. However, it does report the scan cell instance name of position 0 for the failing scan chain.

Example 8

```
Error during scan pattern 5 (detected during parallel unload of
pattern 4)

At T=6640.00 ns, V=67, exp=1, got=0, chain 1, pin

snps_micro.mic0.alu0.accu_q_reg[4] .Q, scan cell 2
```

Note: In the case of unified STIL flow parallel mode for serializer patterns, MAX Testbench reports the failing scan cell position only. The failing scan chain and the failing scan cell instance name are not provided. The failing scan chain name could be retrieved using the diagnostics in TetraMAX ATPG.

Example 9

```
Error during scan pattern 5 (detected during unload of pattern 4)

At T=28340.00 ns, V=284, exp=0, got=1, chain , pin test_so1, scan
cell 2, serializer index 1
```

```
At T=28440.00 ns, V=285, exp=0, got=1, chain , pin test_so1, scan
cell 2, serializer index 2

At T=28540.00 ns, V=286, exp=1, got=0, chain , pin test_so1, scan
cell 2, serializer index 3
```

Using the Failures File

You can configure and use the failures file printed by MAX Testbench for diagnosis. To use this file, you need to set the `+tmax_diag` option.

By default, the diagnosis file name is `<tbenchname>.diag`. The default names of the diagnosis file when the `-split_out` option is used are `<tbenchname>_0.diag`, `<tbenchname>_1.diag`, etc., for the different partitions. You can change the default using the `+tmax_diag_file` option.

The setting `+tmax_diag=1` reports the pattern-based failure format. The setting `+tmax_diag=2` reports the cycle-based failure format.

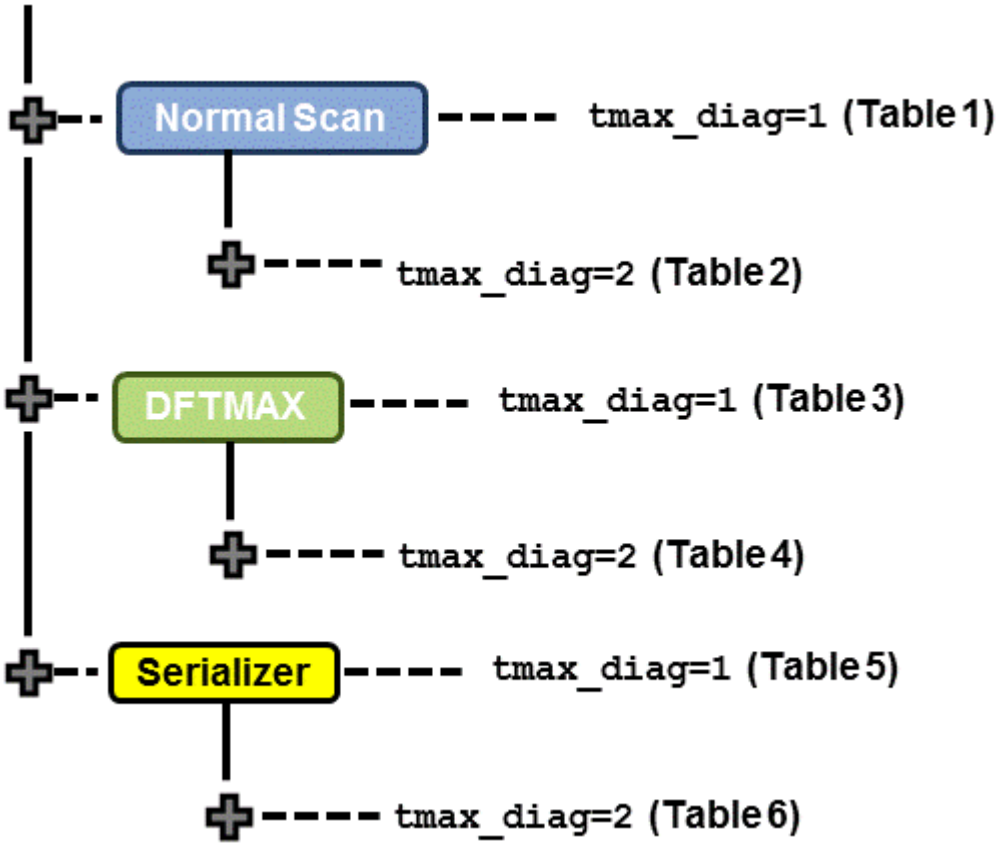
Note the following limitations:

- You cannot run the diagnosis directly if all the partitions are simulated sequentially. This is because the failures are created in separate failure log files. Before running the diagnosis, you must manually append the failure log files into a single file.
- You cannot run the diagnosis if the entire partitions are simulated sequentially and the cycle-based format is used (`+tmax_diag=2`). This is because the recorded cycles are reset for each partition simulation.

Both settings offer a way to generate a failure log file that can be used for a diagnostic if a fault is injected in a circuit and its effect simulated. You can also use these settings to validate the detection of a fault by TetraMAX diagnostics. In addition, they can be used for per-cycle pattern masking or for TetraMAX diagnostics to find the failing scan chain and cell for a unified STIL flow miscompare.

Figure 2 summarizes the formats and applications possible for failures printed using the `+tmax_diag` option.

Figure 2 Summary of Uses for Failures File



The format names and their descriptions are as follows:

- **Format A** = `<pat_num> <pin_name> <shift_cycle> (exp=%b, got=%b)`
- **Format B** = `<pat_num><chain_name> <cell_index> (exp=%b, got=%b)`
- **Format C** = `<pat_num><pin_name> (exp=%b,got=%b)`
- **Format S** = `<pat_num>pat_num> <pin_name> <unload_shift_cycle> <shift_position> (exp=%b, got=%b)`
- **Format D** = `C <pin_name> <vect_nbr> (exp=<exp state>, got=<got state>)`

Note the following:

- The USF and DSF serial simulation modes have the same format and capability. Thus, only the USF parallel is present in the tables. The USF serial is not displayed in the tables.
- The cycle-based format is printed only for serial simulation. This is because the simulation in parallel has less cycles than serial simulation. Thus, the cycles reported by parallel simulation are not valid. If `+tmax_diag=2` is used for a parallel simulation mode, the simulation is not stopped, but the testbench automatically changes the `+tmax_diag` setting to 1. A warning message is also printed in the simulation log. Then, as shown in the following tables, the following statement is printed for all parallel simulation DSF and USF modes: "Not Supported."

The following tables describe the failures file format and their usage in detail.

Table 1 Failures Format and Usage for Normal Scan and tmax_diag=1

		Normal Scan	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	D	Not Supported	Not Supported
	Capture	D	Not Supported	Not Supported
Good for Diagnostics		Yes	No	No
Good for Masking		Yes	No	No

Table 2 Failures Format and Usage for Normal Scan and tmax_diag=2

		Normal Scan	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	D	Not Supported	Not Supported
	Capture	D	Not Supported	Not Supported
Good for Diagnostics		Yes	No	No
Good for Masking		Yes	No	No

Table 3 Failures Format and Usage for DFTMAX Compression and tmax_diag=1

		DFTMAX	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	A	B	A
	Capture	C	C	C
Good for Diagnostics		Yes	Yes*	Yes
Good for Masking		Yes	Yes	Yes

* Failures are usable for TetraMAX diagnostics provided that the command `set_diagnosis -dftmax_chain_format` is used

Table 4 Simulation Failures Format and Usage for DFTMAX Compression and tmax_diag=2

		DFTMAX	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	D	Not Supported	Not Supported
	Capture	D	Not Supported	Not Supported
Good for Diagnostics		Yes	No	No
Good for Masking		Yes	No	No

Table 5 Failures Format and Usage for Serializer and tmax_diag=1

		Serializer	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	S	B	S
	Capture	C	C	C
Good for Diagnostics		Yes	Yes*	Yes
Good for Masking		Yes	Yes	Yes

* If the `set_diagnosis -dftmax_chain_format` command is specified, failures can be used for TetraMAX diagnostics.

Table 6 MAX Testbench Simulation Failures Format and Their Usage for Serializer and tmax_diag=2

		Serializer	Patterns Simulation	Mode
		Dual STIL Serial	Dual STIL Parallel	Unified STIL Parallel
Failure Format for:	Shift	D	Not Supported	Not Supported
	Capture	D	Not Supported	Not Supported
Good for Diagnostics		Yes	No	No
Good for Masking		Yes	No	No

See Also

Diagnosing Manufacturing Test Failures in the *TetraMAX User Guide*

Displaying the Instance Names of Failing Cells

MAX Testbench can display the instance name of the failing cells during the simulation of a parallel-formatted STIL file. To enable this feature, you need to set the boolean variable `cfg_parallel_stil_report_cell_name` in the configuration file. When this variable is set to '`1`', it enables the reporting of the failing scan cell instance names ('`0`' is the default). **Note:** This feature impacts simulation memory consumption.

Note the following examples:

Normal Scan design:

```
cfg_parallel_stil_report_cell_name=0 (default)
Error during scan pattern 9 (detected during parallel unload of
pattern 8)
At T=4640.00 ns, V=47, exp=0, got=1, chain chain1, pin out[4],
scan cell 2

cfg_parallel_stil_report_cell_name=1 ➡ cell name added
Error during scan pattern 9 (detected during parallel unload of
pattern 8)
At T=4640.00 ns, V=47, exp=0, got=1, chain chain1, pin
out[4], scan cell 2, cell name out_reg[2]
```

Scan Compression design:

```
cfg_parallel_stil_report_cell_name=1 ➡ cell name added
Error during scan pattern 28 (detected during parallel unload of
pattern 27)
At T=33940.00 ns, V=340, exp=0, got=1, chain 35, scan cell 1, cell
name U_CORE.dd_d.o_tval_reg
At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell
7, cell name U_CORE.dd_d.o_data_reg_3_
At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell 9, cell
name U_CORE.dd_d.o_data_reg_1_ cfg_parallel_stil_report_cell_
name=0 (default)

Error during scan pattern 28 (detected during parallel unload of
pattern 27)

 At T=33940.00 ns, V=340, exp=0, got=1, chain 35, scan cell 1

At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell 7

 At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell 9
```

Table 7 Summary of the DFT and STIL support for cfg_parallel_stil_report_cell_name

DFT Architecture	STIL Format	Testbench (XTB+cfg: MAX Testbench used with cfg_parallel_stil_report_cell_name variable)	Print Instance Name
Legacy	DSF Serial	DPV	Yes
		XTB	No
		XTB+cfg	No
	DSF Parallel	DPV	Yes
		XTB	No
		XTB+cfg	Yes
	USF Parallel	DPV	Yes
		XTB	No
		XTB+cfg	Yes
DFTMAX	DSF Serial	DPV	No
		XTB	No
		XTB+cfg	No
	DSF Parallel	DPV	Yes
		XTB	No
		XTB+cfg	Yes
	USF Parallel	DPV	No
		XTB	No
		XTB+cfg	No
Serializer	DSF Serial	DPV	No
		XTB	No
		XTB+cfg	No
	DSF Parallel	DPV	Yes
		XTB	No
		XTB+cfg	Yes
	USF Parallel	DPV	No
		XTB	No
		XTB+cfg	No

See Also

Configuring MAX Testbench

Using Split STIL Pattern Files

You can use the `-split_in` option of the `stil2Verilog` command to specify the use of split STIL pattern files. This option has two different formats:

- The `-split_in { 1.stil, 2.stil… }` format uses split STIL files based on a detailed list of STIL files.
- The `-split_in { dir1/*.stil }` format uses split STIL files based on a generic list description using the wildcard (*) symbol.

Note the following:

- The input STIL files from both the detailed list format and the generic list format are assumed to belong to the same pattern set (split patterns of the same original patterns). Multiple files must be specified within curly brackets, with a space before and after each bracket.
- The input STIL files all have the same test protocol (procedures, signals, WFTs, etc). The only difference between these STIL files is the content of the "Pattern" block, which contains test data. Max Testbench takes the first STIL file it encounters as a representative to the other STIL files and extracts and interprets the protocol information from it.
- You must ensure that the input STIL files correspond to the same split patterns. You must also avoid any form of mixing with other STIL files in the list (using the detailed list format) or mixing within the directory (using the generic list format).

Execution Flow for -split_in Option

When the `-split_in` option is specified, the testbench is generated using a single execution. One testbench (.v) file is generated for all STIL files. The number of .dat files directly correlates to the number of input STIL files.

The following example shows a MAX Testbench report:

```
maxtb> Parsing STL procedure file "pat1.stil" ...
maxtb> Parsing STIL data file "pat1.stil, pat2.stil, pat3.stil…"…
maxtb> STIL file successfully interpreted (PatternExec: "").
maxtb> Detected a Normal Scan mode.
maxtb> Test bench files " xtb_tbench.v", "xtb_tbench1.dat”…
“xtb_tbench3.dat" generated successfully.
maxtb> Test data file mapping :
pat1.stil ?? xtb_tbench1.dat (patterns <X1> to <Y1>)
pat2.stil ?? xtb_tbench2.dat (patterns <X2> to <Y2>)
pat3.stil ?? xtb_tbench3.dat (patterns <X3> to <Y3>)
```

The header of each .dat file identifies the STIL partition that was used to generate it, as shown in the following example line:

```
// Generated from original STIL file : ./pat1.stil
```

Using this information, you can link various simulations to the original STIL partitions, regardless of the order of the STIL files specified by the `-split_in` option. You can also combine the existing `-sim_script` option with the `-split_in` option to generate a validation script that enables automatic management of the validation step when using different simulation modes.

See Also

Reading a Split Patterns File in the *TetraMAX User Guide*

Splitting Large STIL Files

You can use the `-split_out` option of the `stil2Verilog` command to specify MAX Testbench to split large STIL files. For example, for a STIL file with ten patterns, the following command generates one testbench file and three .dat files:

```
stil2Verilog -split_out 4 mypat.stil my_tb
```

The first .dat file contains four patterns (0 to 3), the second .dat file contains four patterns (#4 to #7), and the third .dat file contains two patterns (patterns #8 and #9).

The splitting process is based on a user-specified interval. Therefore, you should avoid splitting between two interdependent patterns.

The following sections describe how to split large STIL files:

- Why Split Large STIL files?
- Executing the Partition Process
- Example Test

Why Split Large STIL Files?

The ability to split STIL files is useful for two situations:

- When the number of patterns in a .dat file is so large that it cannot be simulated because it exceeds the system memory capacity. For example, to simulate two million patterns, the size of the .dat file contains 24 million lines, which corresponds to all instructions for all patterns. In this case, the simulator (VCS) runs out of memory before completing the simulation.
- Even if the system memory can accommodate the entire simulation, the excessive memory consumption drastically impacts the performance of the simulation. This can occur when the use of memory swapping and memory resources prevent other applications from using that machine.

When it splits the STIL files, MAX Testbench can resolve a completely blocked simulation, or optimize the memory and runtime simulation. This capability also allows MAX Testbench to serially run a set of patterns as if these patterns were split from TetraMAX ATPG in different STIL files.

Executing the Partition Process

You use the `-split_out` option to define the maximum number of patterns to include in each partition. Based on your specification, MAX Testbench generates a testbench (.v) file and a set of partitioned data (.dat) files from a single STIL file.

When splitting large STIL files, MAX Testbench uses the following equation to determine the number of partitions (or .dat files) to create:

$$\text{Number of Partitions} = \frac{\text{Total Number of Patterns}}{\text{Number of Patterns in a Partition}}$$

The partitioning process is as follows:

1. The first partition (partition 0) starts as normal and stops at the execution of the last pattern of the partition.
2. The second partition (partition 1) starts by reproducing the test_setup macro and the `Condition` statement to restore the context of the last pattern of the first partition (partition 0).

   The second partition contains a duplication of the last pattern of the previous partition, except that all unload states are masked. The strobe of the states corresponds to the second-to-last pattern of the previous partition. This strobe is ensured by the first partition, so you do not need to replicate it. All subsequent partitions follow the architecture of the second partition.
3. Use VCS to create a simulation executable for MAX Testbench, then use the simulation executable and the `+tmax_part=`*`partition_number`* option to simulate each partition, as shown in the following example:

   ```
   simv +tmax_part=0
   simv +tmax_part=1
   simv +tmax_part=2
   ```

Example Test

Note the following example test:

```
./simv +tmax_part=0 | tee run_vcs_par_usf_split_simv0.log
./simv +tmax_part=1 | tee run_vcs_par_usf_split_simv1.log

./simv +tmax_part=0 | tee run_vcs_par_usf_split_simv0.log
Chronologic VCS simulator copyright 1991-2013
Contains Synopsys proprietary information.
################################################################
MAX TB
Test Protocol File generated from original file "pattn/pattn_comp_
USF_par.stil"
STIL file version: 1.0
```

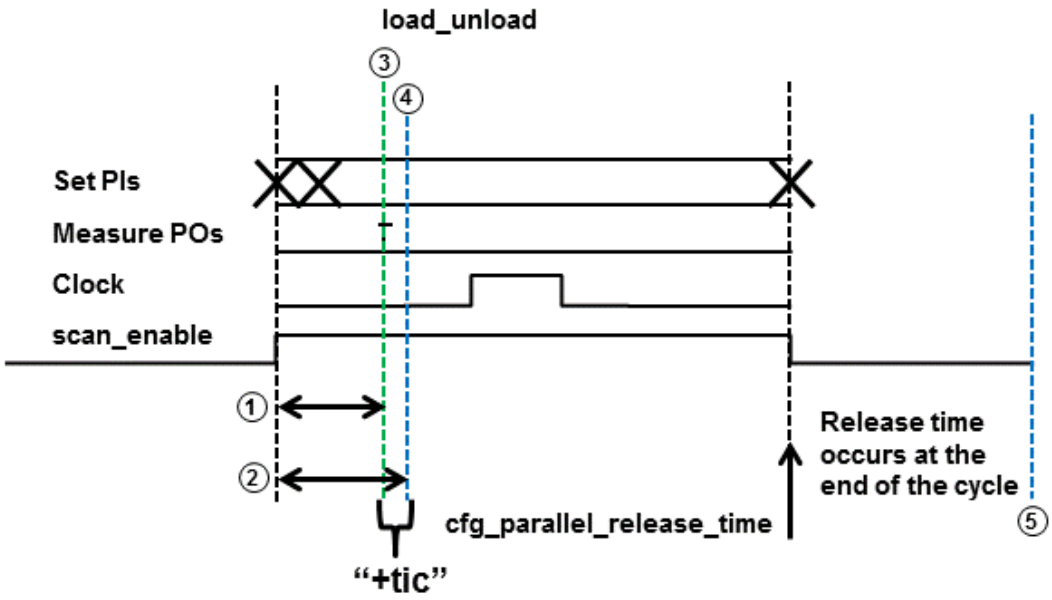
```
Enhanced Runtime Version: use <sim_exec> +tmax_help for available
runtime options
################################################################

XTB: Reading partition 0 (test data file /TEST_split/pattn/pattn_
comp_USF_par_split_0.dat)
XTB: Enabling Enhanced Debug Mode. Using mode 1 (conditional
parallel strobe).
XTB: Starting parallel simulation of 6 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 10 (T=1700.00 ns, V=18)
XTB: Begin parallel scan load for pattern 10, unload 2 (T=2000.00
ns, V=21)
XTB: Begin parallel scan load for pattern 5 (T=1700.00 ns, V=18)
XTB: Simulation of 6 patterns completed with 0 mismatches (0
internal mismatches) (time: 2000.00 ns, cycles: 20)

V C S S i m u l a t i o n R e p o r t
Time: 2000000 ps
```

```
./simv +tmax_part=1 | tee run_vcs_par_usf_split_simv1.log
Chronologic VCS simulator copyright 1991-2013
Contains Synopsys proprietary information.
################################################################
MAX TB
Test Protocol File generated from original file "pattn/pattn_comp_
USF_par.stil"
STIL file version: 1.0
Enhanced Runtime Version: use <sim_exec> +tmax_help for available
runtime options
################################################################

XTB: Reading partition 1 (test data file /TEST_split/pattn/pattn_
comp_USF_par_split_1.dat)
XTB: Enabling Enhanced Debug Mode. Using mode 1 (conditional
parallel strobe).
XTB: Starting parallel simulation of 6 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 5 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 10 (T=1700.00 ns, V=18)
XTB: Simulation of 6 patterns completed with 0 mismatches (0
internal mismatches) (time: 2200.00 ns, cycles: 22)


V C S S i m u l a t i o n R e p o r t
Time: 2200000 ps
```

Force Release and Strobe Timing in Parallel Load Simulation

The timing for parallel load simulation differs from a serial load simulation when the data is driven directly on the flip-flops. Figure 1 shows how the parallel load MAX Testbench works in terms of force, release, and strobe times.

Figure 1 Timing For Parallel Load MAX Testbench



① The time of the scan-enable to the first output measure performed for scan-outs only

② The time of the scan-enable to the first clock of each scan element performed for every scan cell. The input to the scan cell in scan mode must be stable before the effective edge of the clock pulse, and if the scan enable affect that stability, it needs to be checked as a critical path.

③ Parallel placement strobe time check

④ Parallel Force Event occurs "+tic" after strobe

⑤ Release time can be controlled using the "cfg_parallel_release_time" config file option.

See Also

Defining the load_unload Procedure in the *TetraMAX User Guide*

MAX Testbench Runtime Programmability

MAX Testbench supports a runtime programmability flow that enables you to specify a series of runtime simulation options that use the same compiled executable in different modes.

For example, you can compile a single executable using one or more runtime options, such as `+tmax_msg`, `+tmax_rpt`, `+tmax_serial`, `+tmax_parallel`, `+tmax_n_pattern_sim`, and `+tmax_test_data_file`. You can then specify any of these options at runtime using the same executable.

You can also use a set of options to change test patterns. For example, if you want to write out patterns with different chain tests. **Note:** The flow for using split patterns is different than the flow for regular patterns. For details, see "Runtime Programmability for Patterns."

The following sections describe how to configure and execute runtime programmability in MAX Testbench:

- Basic Runtime Programmability Simulation Flow
- Runtime Programmability for Patterns
- Example: Using Runtime Predefined VCS Options
- Limitations

See Also

Configuring MAX Testbench
Predefined Verilog Options

Basic Runtime Programmability Simulation Flow

The basic simulation flow for runtime programmability is as follows:

1. Generate a STIL-based testbench. For details, see "Running MAX Testbench."
2. Configure the compile-time options, as needed.
3. Compile the testbench, design, and libraries, and produce a single default simulation executable. You only need to compile the executable one time, using minimal configuration.
4. Run the simulation, for example:

   ```
   <sim_exec> +<runtime_option>
   ```

   Note that you can use any of the following runtime options:

   - `tmax_msg`
   - `tmax_rpt`

- tmax_serial
- tmax_parallel
- tmax_n_pattern_sim
- tmax_test_data_file

For details on these options, see the "MAX Testbench Configuration" section.

5. If you encounter a new behavior, or need a new report or test patterns, specify the appropriate runtime option and rerun the simulation without recompiling the executable. For example:

```
<simv_exec> <+tmax_test_data_file="myfile.dat">
```

In the previous example, myfile.dat is the newly generated data (.dat) file to be used with the existing testbench file.

Note the following:

- If you specify the tmax_serial option at compile time and the +parallel option at runtime, the resulting simulation is a parallel simulation.
- The msg and rpt options affect the simulation report by providing different verbosity levels. Their defaults are 0 and 5, respectively. Setting up values different than these values, either at compile-time or runtime, is automatically reported by the testbench at simulation time 0. The runtime options override their compilation-time counterparts.
- The n_pattern_sim option overrides the equivalent tmax_n_pattern_sim option, if the latter option is specified. Otherwise, it overrides the default initial set of patterns (the entire set in the STIL file, or the set generated by Max Testbench using the -first and -last options).

Runtime Programmability for Patterns

You can use the -generic_testbench and -patterns_only options with the write_testbench or stil2Verilog commands to configure runtime programmability for patterns.

Note: Do not confuse the use of regular patterns and the use of split patterns for runtime programmability. You cannot simultaneously use the -generic_testbench and -patterns_only options for split patterns. See "Using Split Patterns" for details

The following sections describe how to use runtime programmability for patterns:

- Using the -generic_testbench Option
- Using the -patterns_only Option
- Executing the Flow
- Using Split Patterns

Using the -generic_testbench Option

The `-generic_testbench` option, used in the first pass of the flow, provides special memory allocation for runtime programmability. This is required because the Verilog 95 and 2001 formats use static memory allocation to enable buffers and arrays to store and manipulate .dat information. This type of data storage cannot be handled by a standard .dat file. Also, it is expected that .dat files will continue to expand as they store an increasing number of vectors and atomic instructions.

The `-generic_testbench` option runs a task that detects the loading of the .dat file, and then allocates an additional memory margin. If, at some point, the data exceeds this allocated capacity, an error message, such as the following, will appear.

```
XTB Error: size of test data file <file_name>.dat exceeding
testbench memory allocation. Exiting...
(recompile using -pvalue+design1_test.tb_part.MDEPTH=<###>).
```

As indicated in the message, you will need to recompile the testbench using the suggested Verilog parameter to adjust the memory allocation.

Using the -patterns_only Option

The `-patterns_only` option is used for a second pass, or later, run. It initiates a light processing task that merges the new test data. This option also enables additional internal instructions to be generated for the special .dat file. For example, it includes a computation of the capacity for later usage by the testbench for memory management.

If you are running an updated pattern file, and have specified the `-pattern_only` option, you will see the following message:

```
XTB: Setting test data file to "<file_name>.dat" (at runtime).
Running simulation with new database...
```

Executing the Flow

The flow for runtime programmability for patterns is as follows:

1. Generate the tesbench in generic mode using the first available STIL file. For example:

   ```
   write_testbench -input pats.stil -output runtime_1 \
       -replace -parameter {-generic_testbench \
       -log mxtb.log -verbose}
   Executing 'stil2Verilog'...
   ```

2. Compile and simulate this testbench (along with other required source and library files).
3. When a new pattern set is required, generate a new STIL file, while keeping the same STIL procedure file for the DRC (same test protocol).

4. Rerun MAX TestBench against the newly generated STIL file to generate only new the test data file, as shown in the following example:

```
write_testbench -input pats_new.stil -output runtime_2 \
    -replace -parameter { -patterns_only -log mxtb_2.log \
    -verbose}
```

5. Attach the newly generated .dat file to the simulation executable and rerun the simulation (without recompilation), as shown in the following example:

```
simv +tmax_test_data_file="<new_pattern_filename>.dat"
Command: ./simv +tmax_test_data_file=runtime_2.dat
################################################
MAX TB Version H-2013.03
Test Protocol File generated from original file " pats_
new.stil"
STIL file version: 1.0
################################################
XTB: Setting test data file to "runtime_2.dat" (at runtime).
Running simulation with new database...
XTB: Starting parallel simulation of 5 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Simulation of 5 patterns completed with 0 errors (time:
2700.00 ns, cycles: 27)
V C S S i m u l a t i o n R e p o r t
```

6. Repeat steps 3 to 5, as needed, to include a new STIL file.

Using Split Patterns

The following examples show how to split patterns for runtime programmability.

This example uses the `stil2Verilog` command:

```
stil2Verilog input_stil_file_name output_testbench_name \
    -tb_module < > -split_out 32 -generic -replace \
    -log translation.log
```

The next example uses the `write_testbench` command:

```
write_testbench -input input_stil_file_name -out output_testbench_
name \
   -parameters {-split_out 32 -tb_module < > -generic \
   -log mxtb.log}
```

The next set of examples show the process of splitting pattern files using the `write_patterns` command and a series of `write_testbench` commands. Note that you do not need to use the `-patterns_only` option to create the first split file. In this case, the first split file is created using the `-generic` option in the first `write_testbench` command of the command sequence.

```
write_patterns ./pattern/top_scan.stil -format stil -replace \
    -split 5

write_testbench -input ./pattern/top_scan_0.stil
    -output ./pattern/top_scan_maxtb -replace \
    -parameter {-generic -log mxtb_generic_split_0.log \
    -verbose }

write_testbench -input ./pattern/top_scan_1.stil \
    -output ./pattern/top_scan_maxtb_1 -replace \
    -parameter {-patterns_only -log mxtb_split_1.log \
    -verbose }

write_testbench -input ./pattern/top_scan_2.stil \
    -output ./pattern/top_scan_maxtb_2 -replace \
    -parameter {-patterns_only -log mxtb_split_2.log \
    -verbose }

write_testbench -input ./pattern/top_scan_3.stil \
    -output ./pattern/top_scan_maxtb_3 -replace \
    -parameter {-patterns_only -log mxtb_split_3.log \
    -verbose }

write_testbench -input ./pattern/top_scan_4.stil \
    -output ./pattern/top_scan_maxtb_4 -replace \
    -parameter {-patterns_only -log mxtb_split_4.log \
    -verbose }

write_testbench -input ./pattern/top_scan_5.stil \
    -output ./pattern/top_scan_maxtb_5 -replace \
    -parameter {-patterns_only -log mxtb_split_5.log \
    -verbose }
```

Example: Using Runtime Predefined VCS Options

The following example shows how to use runtime predefined VCS options:

```
%> ./simv_usf +tmax_msg=3 +tmax_n_pattern_sim=1 +tmax_rpt=3
###############################################
MAX TB Version H-2013.03
Test Protocol File generated from original file "runtime.stil"
STIL file version: 1.0
###############################################
XTB: Setting runtime option "tmax_n_pattern_sim" to 1.
XTB: User requesting simulating patterns 0 to 1
XTB: Setting runtime option "tmax_msg" to 3.
XTB: Setting runtime option "tmax_rpt" to 3.
XTB: Starting parallel simulation of 2 patterns
```

```
XTB: Using 0 serial shifts
XTB: Processed statement: WFTStmt
XTB: Processed statement: ConditionStmt
XTB: Starting macro test_setup..., T=0.00 ns, V=1
XTB: Processed statement: test_setupStmt
XTB: Processed statement: SetPat
XTB: Starting proc load_unload..., T=200.00 ns, V=3
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: (parallel) shift, at 300.00 ns
XTB: Processed statement: load_unloadStmt
XTB: Starting proc capture..., T=400.00 ns, V=5
XTB: Processed statement: captureStmt
XTB: Processed statement: IncPat
XTB: Starting proc load_unload..., T=500.00 ns, V=6
XTB: (parallel) shift, at 600.00 ns
XTB: Processed statement: load_unloadStmt
XTB: Starting proc capture_clk..., T=700.00 ns, V=8
XTB: Processed statement: capture_clkStmt
XTB: Processed statement: IncPat
XTB: Simulation of 2 patterns completed with 0 error (time:
1000.00 ns, cycles: 10)
V C S S i m u l a t i o n R e p o r t
```

Runtime Programmability Limitations

The following limitations apply to runtime programmability:

- The following runtime options are not supported: `tmax_vcde`, `tmax_serial_timing`, `tmax_diag_file`, `tmax_diag`.
- You cannot change between the `+delay_mode_zero`, `+typdelays`, `+mindelays`, and `+maxdelays` options.
- You cannot use a different test_setup procedure at runtime.
- You cannot change the width of a variable.
- You cannot make changes to the STIL procedure file before generating second-pass patterns.
- You cannot change compile-time switches.
- You cannot add $dumpvars statements
- You cannot use different versions of VCS.

MAX Testbench Support for IDDQ Testing

IDDQ testing detects circuit faults by measuring the amount of current drawn by a CMOS device in the quiescent state (a value commonly called “IddQ”). If the circuit is designed correctly, this amount of current is extremely small. A significant amount of current indicates the presence of one or more defects in the device.

You can use the following methods in MAX Testbench to configure the IDDQ testing:

- Compile-Time Options
- Configuration File Settings
- Generating a VCS Simulation Script

See Also

Generating IDDQ Test Patterns in the *TetraMAX User Guide*

Compile-Time Options for IDDQ

MAX Testbench has two compile-time options that support IDDQ testing and are specified at the command line when starting a simulation. Note that these compile-time options cannot be specified in the configuration file:

- `tmax_iddq`

  This option enables IDDQ testing during PowerFault simulation. The default behavior is not to use the IDDQ test mode. The following example enables IDDQ testing from the VCS command line:

  ```
  % vcs ... +define+tmax_iddq
  ```

- `tmax_iddq_seed_mode=<0|1|2>`

  This option changes the fault seeding for IDDQ testing to one of three modes:

  - `0` for automatic seeding (default)
  - `1` for seeding from a fault file only
  - `2` for both automatic seeding and file seeding

  When the seeding mode is set to `1` or `2`, the testbench assumes the existence of a fault list file (or its symbolic link) in the current directory named *`tb_module_name`*`.faults`. If this file is not found, the simulation stops and an error is issued.

  Note: You can override the default fault list name in the configuration file (see the next section).

See Also

Predefined Verilog Options

IDDQ Configuration File Settings

You can make several IDDQ test-related specifications in a dedicated subsection of the configuration file. Note that there are no command-line equivalences to these settings since they are testbench file-specific commands.

`cfg_iddq_seed_file` *`fault_list_file`*

This parameter overrides the default *tb_module_name*.faults file when faults are seeded from an external fault list file. The default *tb_module_name*file in Max Testbench is *DUT_name*_test.

The following example specifies faults seeded from a file called `my_dut_test`:

```
set cfg_iddq_seed_file my_dut_test
```

`cfg_iddq_verbose 0 | 1`

This parameter enables or disables the PowerFault verbose report. The default is 1, which enables the verbose report. Specify a value of 0 to disable the verbose report.

The following example disables the PowerFault verbose report:

```
set cfg_iddq_verbose 0
```

Note: You can use the `+define+tmax_msg=4` simulation option to report file names that are used during the simulation process.

`cfg_iddq_leaky_status 0 | 1`

This parameter enables or disables the PowerFault leaky nodes report printed in the *tb_name*.leaky file. The default is 1, which enables the leaky nodes report. Specify a value of 0 to disable this report.

The following example disables the PowerFault leaky nodes report:

```
set cfg_iddq_leaky_status 0
```

`cfg_iddq_seed_faul_model 0 | 1`

This parameter specifies the PowerFault fault model used for external fault seeding. The default is 0, which specifies SA faults. Specify a value of 1 for bridging faults.

The following example specifies bridging faults for automatic seeding:

```
set cfg_iddq_seed_faul_model 1
```

`cfg_iddq_cycle` *`value`*

Use this parameter to set the initial counter value for IDDQ strobes. The default is 0.

The following example sets the initial counter value to 1:

```
set cfg_iddq_cycle 1
```

See Also

Configuring MAX Testbench

Generating a VCS Simulation Script

You can use MAX Testbench to generate a script that sets up required information for IDDQ test simulation. This information is required to enable the PLI access option functions (`+acc`), the path to the archive PowerFault PLI library (`libiddq_vcs.a`), and the path to the PLI function interface (`iddq_vcs.tab`).

Note that automatic simulation script generation for IDDQ testing is limited to the VCS simulator only.

The following example is a basic script generated by MAX Testbench using the `-sim_script` option (without using any available parameters from the configuration file) when IDDQ test mode is enabled:

```
#!/bin/sh
LIB_FILES="my_lib.v ${IDDQ_HOME}/lib/libiddq_vcs.a -P${IDDQ_HOME}
/lib/iddq_vcs.tab"
DEFINES=""
OPTIONS="+tetramax +acc+2"
NETLIST_FILES="my_netlist.v"
TBENCH_FILE="new_i021_s1_s.v"
SIMULATOR="vcs"
${SIMULATOR} -R ${DEFINES} ${OPTIONS} ${TBENCH_FILE} ${NETLIST_
FILES} ${LIB_FILES}
SIMSTATUS=$?
if [ ${SIMSTATUS} -ne 0 ]
then echo "WARNING: simulation command returned error status
${SIMSTATUS}"; exit ${SIMSTATUS};
fi
```

Note the following:

- When generating the script, MAX Testbench assumes that the IDDQ_HOME environment variable points to the location of an existing PowerFault PLI.
- You must have a valid Test-IDDQ license to run the PowerFault PLI.

Understanding MAX Testbench Parallel Miscompares

The following example shows the VCS script used for parallel simulation for MAX Testbench:

```
vcs -full64 -R \
      -l parallel_stil.log \
      +delay_mode_zero +tetramax

      par.v \
      -v ../lib/class.v \
      ../1_dftc/result/lt_timer_flat.v \
```

```
+define+tmax_rpt=1 \
+define+tmax_msg=10
```

How MAX Testbench Works

The Verilog writer for MAX Testbench is essentially an algorithm that browses the data structure and retrieves the appropriate information according to the order and the form determined by the Verilog testbench template.

MAX Testbench does not parse the netlist file. It retrieves the DUT interface (its hierarchical name and its primary I/O) from the STIL file. Therefore, it is the responsibility of the STIL provider (TetraMAX ATPG) to make sure that this interface corresponds effectively to the one described in the netlist. The testbench file (test protocol) contains all the details of the STIL file, whereas the test data file translates the execution part (Pattern blocks). See Figure 4 and Figure 5.

Figure 4 Relationship of Files in MAX Testbench Flow

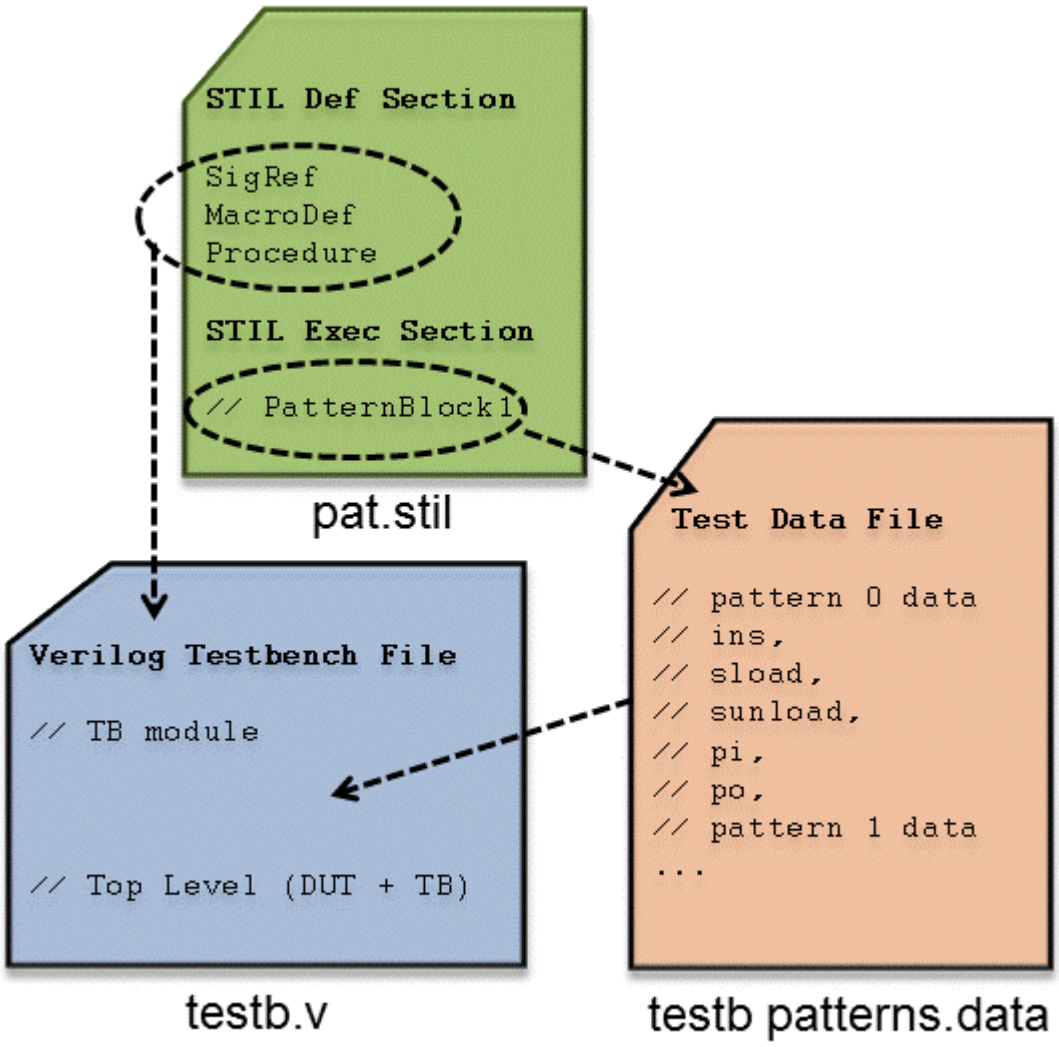
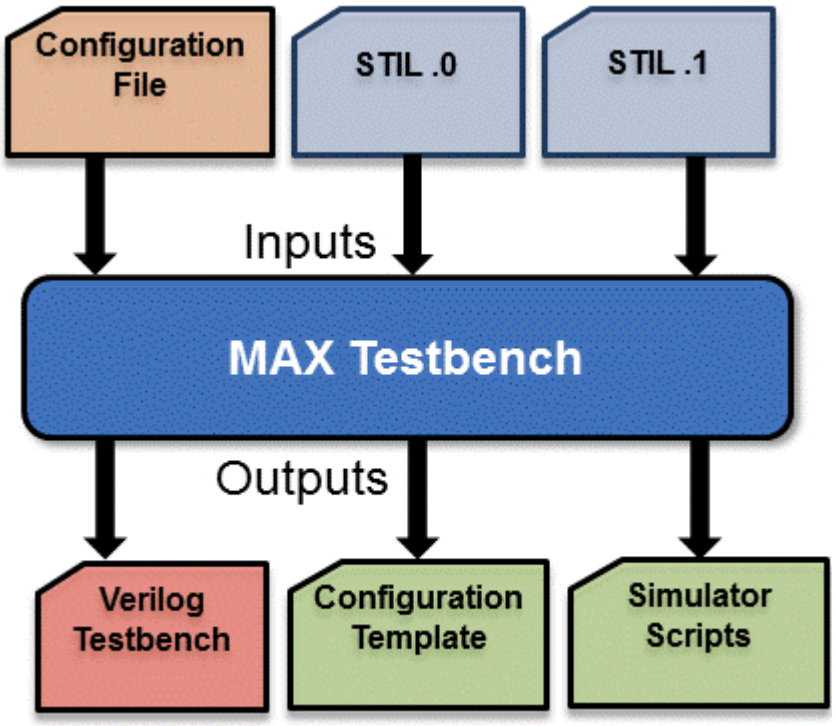


Figure 5 MAX Testbench Flow



See Also

Editing the STIL Procedure File

Predefined Verilog Options

Table 1 describes a set of predefined Verilog options. When specified on the VCS compile line, these options must be preceded by the '+`define`' statement

Table 1 Predefined Verilog options

Verilog Option	Description

`+tmax_help`	Used with the `simv` executable, this option reports the available runtime options, which are:
	`+tmax_n_pattern_sim`
	`+tmax_serial`
	`+tmax_parallel`
	`+tmax_msg`
	`+tmax_rpt`
	`+tmax_test_setup_only_one time`
	`+tmax_test_data_file`

`+tmax_serial=N`	Initial N serial (flattened scan) vectors
`+tmax_parallel=N`	Parallel scan access with N serial vectors
`+tmax_rpt=N`	Specifies the interval of the progress message
`+tmax_msg=N`	Control for a prespecified set of trace options
`+tmax_vcde`	Generates an extended VCD of the simulation run
`+tmax_serial_ timing`	Generates a delay (a "dead period") for parallel scan access.
`+tmax_test_setup_ only_one time`	Simulates the `test_setup` macro only one time when using split patterns with MAX Testbench. This option is useful when you are using multiple STIL pattern files and want to avoid multiple simulations of the `test_setup` macro. It can be used for both compile time and runtime during a simulation.

The `+tmax_rpt` option controls the generation of a statement on entry to every TetraMAX ATPG pattern unit during the simulation. This statement is printed during the simulation run, and provides an indication of progress during the simulation run. This progress statement has two forms, depending on whether the next scan operation is executed in serial or parallel fashion:
`Starting Serial Execution of TetraMAX ATPG pattern N, time NNN, V`

```
#NN

Starting Parallel Execution of TetraMAX ATPG pattern N, time NNN,
V #NN

Starting Serial Execution of TetraMAX pattern N (load N), time
NNN, V #NN
Starting Parallel Execution of TetraMAX pattern N (load N), time
NNN, V #NN
```

By default, the pattern reporting interval is set to every 5 patterns. This value can be changed by specifying the interval value to the `+tmax_rpt` option. For instance, `+define+tmax_rpt=1` on the VCS compile line generates a message for each TetraMAX ATPG pattern executed. All pattern reporting messages can be disabled by setting `+define+tmax_rpt=0`.

The `+tmax_msg` option controls a pre-defined set of trace options, using the values `1` through `4` to specify tracing, where '`1`' provides the least amount of trace information and '`4`' traces everything. These values activate the trace options as follows:

- `0` — disables all tracing (except progress reports with +tmax_rpt)
- `1` — traces entry to each Procedure and Macro call
- `2` — adds tracing of WaveformTable changes
- `3` — adds tracing of Labels
- `4` — adds tracing of Vectors

The `+tmax_msg` option is set to 0 by default.

These two options `+tmax_rpt` and `+tmax_msg` provide a single control of tracing information, established as the simulation environment is started. By editing the testbench file, additional options can be specified during the simulation run.

The option `+tmax_evcd` supports generation of an extended VCD file for the instance of the design under test (dut). The name of this file is "sim_vcde.out". The option `+tmax_serial_timing` causes an interval of no events to be generated for each parallel scan access operation. This period aligns the overall simulation time of parallel scan access with the same time required for a normal serial shift operation. This "dead period" is described in "Parallel Scan Access". By default, this dead period is not present and the parallel scan access simulation occupies a single cycle period for the entire scan operation. For designs that can accept this dead period, this option facilitates coordinating times between parallel and serial simulations, and facilitates identifying the physical runtime of a pattern set with parallel scan access operation present. Some designs might not support this dead period, for instance certain styles of PLL models might lose synchronization for intervals without clock events present. These designs should not use this option.

The `+tmax_diag` option controls the generation of miscompare messages formatted for TetraMAX ATPG diagnostics during the simulation.

See Also

Configuring MAX Testbench

MAX Testbench Limitations

The following limitations appliy when using MAX Testbench:

- MAX Testbench does not support DBIST/XDBIST, or core integration. XDBIST and CoreTest are EOL (End-Of-Life) tools.
- For script generation, predefined options are supported only for a VCS script.

See Also

Runtime Programmability Limitations

3

MAX Testbench Error Messages and Warnings

The following sections list and describe the various error messages and warnings associated with MAX Testbench:

- Error Message Descriptions
- Warning Message Descriptions
- Informational Message Descriptions

Note: You can access a detailed description for a particular message by specifying either of the following commands:

```
stil2Verilog -help [message_code]
```

or

```
write_testbench -help [message_code]
```

Error Message Descriptions

Table 1 lists all MAX Testbench error messages and their descriptions.

Table 1 Error Message Descriptions

Error Message	Description	What Next
`E-001- No license found for this site`	The license file specified in the SYNOPSYS installation does not contain a valid license for this site.	Check the SYNOPSYS environment variable or contact SYNOPSYS to get a valid license.
`E-002- No threads associated with the first PatternExec`	The tool automatically searches for the first PatternExec statement in the specified STIL file. Its name is displayed in the verbose mode execution. This message occurs when the STIL interpretation process failed to retrieve any execution threads corresponding to the detected PatternExec statement.	Check the validity of the STIL file and its first PatternExec statement.
`E-003 - Multiple PatList found, not fully supported yet (only one at a time or in parallel but with PLL like patterns)`	The PatList statement is not yet fully supported. The tool only supports for now only simple PatList representations, like the PLL like patterns.	Generate a STIL that uses the supported PatList syntax and patterns .
`E-006- Cannot recover signal <name> from the STIL structures, last label <name>`	Respective signal cannot be found in the Signals list of the STIL file.	Check the STIL file syntax

Error Message	Description	What Next
E-007- Unsupported event %s in wave of cluster "%c" of signal %s in WFT "%s"	The tool currently does not support the following event types:WeakDown, WeakUp, CompareLowWindow, CompareHighWindow, CompareOffWindow, CompareValidWindow, LogicLow, LogicHigh, LogicZ, Marker, ForcePrior	Generate a STIL that uses only the supported event types
E-008- The event waves of cluster <name> of signal <name> in WFT <name> have incompatible types (force and compare simultaneously, not yet supported)	The cluster of reported signal contains both force and compare event waves simultaneously. The tool does not support this yet .	Generate a STIL that does not use this type of event waves in the WaveForm description
E-010- Can't find definition for <name> in the STIL structures	The specified Procedure or Macro cannot be found in the STIL structures. That can be caused by an incomplete STIL file.	Check the syntax of the STIL file
E-011- Too many signal references in the Equivalent statement %s, not yet supported	The tool only supports one to one equivalences for now and the input STIL file contains Equivalent statements with multiple signal specifications.	Generate a STIL that contains only one to one equivalences
E-013- Invalid Equivalent statement <location>	The tool only supports one to one equivalences for now and the specified. Equivalent statement does not respect this rule.	Generate a STIL that contains correct Equivalent statements

Error Message	Description	What Next
E-014- Loop Data statement in <name> not yet supported	Only the simple Loop statement is currently supported. The Loop Data is not yet supported.	Generate a STIL that does not contain Loop Data
E-015 - The requested help page does not exist	A message code was specified that does not correspond to an existing help page.	Check the correctness of the message code
E-017- Duplicate definition for <name>	There is more than one definition for a specified Procedure/Macro in the input STIL file. This represents a bad STIL syntax and should be corrected.	Check the syntax of the input STIL file
E-018- Multiple specification of -log option	The command line -log option has been specified more than one time. Only one specification is allowed to avoid confusion.	Check and edit the command line to have a single -log specification
E-019- Missing "log" option value	The command line -log option has an mandatory argument that specifies the name of the file which is used to write the transcription of the tool execution. This argument is absent.	Check and edit the command line to add a file name as argument for -log
E-021- Error during the consistency checking of the command line parameters and options	The error message indicates which parameter/option is cone timerned.	Modify the command line according to the error message. Check the user documentation for more details

Error Message	Description	What Next
E-023- cannot write file <file_name> as it already exists, specify -replace if you want to overwrite it	When the tool is about to generate a file it checks if the respective file name already exists on disk. In this case, to avoid accidental lost of user important data the tool asks the user for a confirmation, more specific the user has to provide the -replace option in the command line to confirm that this is the desired behavior.	If the overwriting of the respective file is desired then add the -replace option in the command line
E-024- Ambiguous option <name>, can match multiple options like <enum>	The specified command line option match more than one command line option. The command line processing allows for incomplete option name specifications, but a minimal specification is required to avoid ambiguity.	Edit the command line and clearly specify your options to avoid ambiguity
E-025- <file/directory_ name> No such file or directory	The specified file(s) or folder(s) cannot be found on disk. This usually is caused by a wrong specification of the design/library files generated from the command line or from the config file.	Specify correct file/folder names

Error Message	Description	What Next
E-028- <value> is not a valid cfg_ time_unit or cfg_ time_precision value (Valid integer are 1, 10 and 100. Units of measurement are s, ms, us, ns, ps and fs)	Specified value for cfg_ time_unit or cfg_time_ precision is invalid. This usually occurs in the config file consistency checking process.	Edit the invalid values with correct ones
E-029- It is illegal to set the time precision larger than the time unit	Value specified for time precision is too big.	Specify a lower value for time precision, lower or equal with the time unit
E-030- Cannot generate Verilog testbench neither for serial nor for parallel load mode...	Specified testbench generation mode is not possible with the given STIL file. This might happen when you specify the parallel_only or serial_ only configuration.	Specify a different simulation mode
E-031- Cannot open <file_name> file.	Specified file name is not accessible. It may be a config file name, a log file name, design file name, library file name, test data file, protocol file, etc.	Check the existence, the location, or the permission of the specified file
E-032- Error during the consistency checking of config_ file data	The error message indicates which config file field is affected.	Modify the config file according to the error message
E-033- Error reading Tcl file <file_name> at line <#>. Only comments and variable settings allowed	The config file only supports a limited Tcl syntax, such as variable settings, comments and empty lines.	Modify the config file by removing the unsupported syntax.

Error Message	Description	What Next
E-035- Cannot retrieve DUT module name in STIL file. Set the "cfg_dut_ module_name" in the config file to avoid the problem	The tool automatically extracts the DUT module name from the specified STIL file.	Use a config file to specify it by setting the cfg_dut_module_ name parameter. A template config file can be generated using the -generate_ config option
E-036- Detected an unsupported multi-vector Shift construct.	The tool detected a STIL Shift block that includes multiple Vector statements – some of which are not consuming data without a pound (#) sign .	Make sure the vectors are not intended to be post-amble (or preamble) vectors that need to defined after (or before) the Shift block. If so, correct the STIL file accordingly. If not, contact Synopsys support.
E-037- Detected an unsupported multi-vector Loop construct.	The tool detected a STIL Loop block that includes multiple Vector statements.	USF Parallel simulation is not supported for STIL files using these type of constructs.
E-038- Cannot process MISR outputs. Theratio between the number of compressors and the number of SERDES MISR outputs is not supported. Parallel simulation may fail	The tool detected a situation in which it can't determine the assignment between the compressor outputs and the SERDES MISR output	If possible, use a number of compressors that can divide with the number of SERDES MISR outputs. The simulation may fail otherwise.
E-039- Shift statement can only be called from Procedures	Shift statements are only supported when they are called inside a Procedure.	Generate a STIL file that respects this syntax

Error Message	Description	What Next
E-040 - Wrong values for -first and/or -last options	The first and last options need to be positive integers and in increasing order (last > first). First and last must both be less than max_patterns.	Set the appropriate values.
E-041 - Parallel simulation mode for loop block within procedure "proc"	Parallel simulation for a STIL file with a loop block consuming scan data within a load_unload procedure is not supported.	Regenerate a "serial_only" STIL version from TetraMAX ATPG or use the -ser_only MAXTestbench option (in case of USF STIL) to generate the appropriate testbench and run the simulation in serial mode.
E-042 - Error during the consistency checking of the input STIL file	Identifies a missing structure or field in the STIL file.	Add the missing structure or field in the input STIL file.
E-043 - Enhanced Debug Mode for Combined Pattern Validation (EDCPV)	Due to some consistency checks, EDCPV mode cannot be activated. As a result, the generated testbench cannot pinpoint the exact failing scan cell in parallel simulation mode.	Refer to the requirements described in "Debugging Parallel Simulation Failures Using Combined Pattern Validation."
E-044 -Detected an invalid multibit scan cell. Simulation cannot be performed in parallel mode	MAX Testbench detected multibit scan cells that are incorrectly described. In this case, parallel mode simulation is not possible, since the respective scan cell cannot be correctly identified in the design.	Check the input STIL file and the TetraMAX parameters for errors.

Warning Message Descriptions

Table 2 lists all MAX Testbench warning messages and their descriptions.

Table 2 Warning Message Description

Warning Message	Description	What Next
`W-000 - Failed to initialize error file <file_name>, no STIL syntax error messages are available`	This message occurs when the reported error filename is invalid, does not exists or the user does not have access rights to it. This does not affect the tool execution, but the eventual STIL syntax error messages will not be displayed.	If this is not the expected behavior, then check the file path and the SYNOPSYS environment variable
`W-001 - Multiple assignments for signal <name> (old value <value>), proceeding with <value>, last label <name>`	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing. This message is displayed only in verbose mode.	If this is not the expected behavior, then check the STIL file

Warning Message	Description	What Next
W-002 - Multiple assignments for signal <name> in signal group <name>, proceeding with <value>, last label <name>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the locationwhere this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing. This message is displayed only in verbose mode.	If this is not the expected behavior, then check the STIL file
W-003 - Multiple assignments for inout signal <name> in signal group <name> without a WFCMap specified (<values>), last label <name>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the locationwhere this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the nameof the last Label observed during processing.	If this is not the expected behavior, then check the STIL file

Warning Message	Description	What Next
W-004 - Insufficient data for signal group <name>, ignoring signal <name>	This message occurs for signal groups when the length of the data assigned to it is less then the length of the signal group itself. In this case the signals for which there is no data to be assigned are ignored. This is usually caused by an incorrect STIL.	If this is not the expected behavior, then check the STIL file
W-005 - Multiple assignments for sig <name>, proceeding with <value>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the locationwhere this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the nameof the last Label observed during processing. This message is displayed only in verbose mode.	If this is not the expected behavior, then check the STIL file
W-006 - Cannot build testbench in parallel load mode (no scan chains found)	This message occurs when the tool did not detect any scan chains in the input STIL file. Without the full description of the scan chains a parallel load mode testbench cannot be generated.	Check the STIL file syntax or regenerate it using the latest versions of DFT Compiler and Tetra MAX

Warning Message	Description	What Next
W-007 - SYNOPSYS and SYNOPSYS_TMAX environment variables have different values, SYNOPSYS_TMAX is considered in this case	This message occurs then both SYNOPSYS and SYNOPSYS_TMAX environment variables are specified but with different values. In this case the values specified by the SYNOPSYS_TMAX environment variable is considered.	If this is not the desired behavior, re-specify correctly the environment variables
W-008 - Failed to retrieve WFC <wfc> of signal <name> from WFT <name>, processing its string value, last label <name>	This message occurs when a signal is assigned a WFC that is not described in the current WFT In this case the tool will try to interpret the WFC behavior using its string value instead of the WFT. This message is displayed only in verbose mode.	If this is not the expected behavior, then check the STIL file
W-009 - Failed to retrieve WFC <wfc> for signal <name> of group <name> in WFT <name>, processing its string value, last label <name>	This message occurs when a signal inside a signal group is assigned a WFC that is not described in the current WFT. In this case the tool will try to interpret the WFC behavior using its string value insteadof the WFT. This message this displayed only in verbose mode when the cone timerned signal is of type Pseudo.	If this is not the expected behavior, then check the STIL file

Warning Message	Description	What Next
W-010 - Cannot build testbench in parallel load mode (no cells specified in <name> scan chain)	This message occurs when the tool did not detect any scan cells in the respective scan chain. Without the full description of the scan chains a parallel load mode testbench cannot be generated.	Check the STIL file syntax or regenerate it using the latest versions of DFT Compiler and Tetra MAX
W-011 - Multiple assignments for signal <name> in Vector stmt, proceeding with <value>, last label <name>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the locationwhere this happens, the parent Macro/Procedure name (if any), and the name of the last Label observed during processing.	If this is not the expected behavior, then check the STIL file
W-012 - Cannot generate simulation script file (DUT module name missing)	This message occurs when the tool was not able to automatically detect the name of the DUT module and a simulation script is requested. In this case the script file will not be generated.	Specify the DUT module name using the command line or the configuration file

Warning Message	Description	What Next
W-013 - NETLIST_ FILES variable in the simulation script file is empty (design files missing)	This message occurs as a simulation script have been requested but no design files have been specified, neither using the command line -v_file option nor the design_ files variable in the configuration file. In this case the script file is not completed.	Specify the design files by editing the generated simulation script file
W-014 - LIB_FILES variable in the simulation script file is empty (library files missing)	This message occurs as a simulation script have been requested but no library files have been specified, neither using the command line -v_file option nor the lib_files variable in the configuration file. In this case the script file is not completed.	Specify the library files by editing the generated simulation script file
W-015 - Parallel option ignored as -serial_only testbench requested	When a serial_only testbench is requested then, as expected, all the parallel options are ignored. The user is warned to avoid any confusion.	If this is not the expected behavior, then change the testbench generation mode
W-018 - Specified time precision <value> too large. This can cause errors during simulation	The value specified for cfg_time_precision in the config file may be too large.	If this is the case, then edit the config file and change the value accordingly

Warning Message	Description	What Next
W-019 - Parallel nshift parameter not supported for scan compression designs. Ignored.	In the case of scan compression designs, the tool can generate a testbench for parallel load mode simulation with nshift only when the input STIL file supports the Unified STIL flow.	Regenerate the STIL file using the default mode of the write_patterns command.
W-020 - <name> parameter not yet supported (ignored)	Certain parameters enumerated in the config file example are not yet supported.	A full list of the supported ones may be found in the user guide. If specified, these parameters are ignored
W-021 - Test bench module name already defined in command line. "cfg_tb_ module_name" variable in the configuration file ignored	The testbench module name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification
W-022 - Design files already defined in command line. "design_files" variable in the configuration file ignored	The design file name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification

Warning Message	Description	What Next
W-023 - Library files already defined in command line. "lib_files" variable in the configuration file ignored	The library file name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification
W-024 - Unknown <name> variable (ignored)	The reported variable name is not part of the configuration file syntax.	To find the correct syntax of this file you can generate a config file template using the -generate_config option or consult the user manual
W-025 - Configuration file <file_name> does not contain any variable setting	The specified input configuration file does not contain any variable settings.	Check the configuration file content or path if that is not the expected behavior
W-026 - Invalid load/unload chains or groups of ctlCompressor <name>	The ctlCompressor block is not valid because the load/unload chains or groups are not correct (i.e.: some scan chains are specified in the groups but are undefined or empty). Since the ctlCompressor block is wrong, it is not possible to run a parallel simulation from a serial formatted STIL file.	Check the STIL file and rerun DFT Compiler and/or TetraMAX ATPG if necessary.

Warning Message	Description	What Next
W-030 - Detected Serial Only test patterns, the generated testbench can only be run in serial simulation mode	This occurs either when the user intentionally requested a serial only testbench or when the provided STIL file does not contain enough information to allow a parallel load mode simulation also.	Check the STIL file, TMAX script and the options of the write_patterns command and the DFT script used with DFT compiler and make sure that this is the desired behavior.
W-031 - Detected Parallel Only test patterns, the generated testbench can only be run in parallel simulation mode	This message occurs when the provided STIL file contains pure parallel patterns, specially formatted for a parallel simulation. These patterns can't be simulated serially.	Check the STIL file, TMAX script and the options of the write_ pattern command and the DFT script used with DFT compiler and make sure that this is the desired behavior.
W-032 - Parallel nshift parameter too small (minimum <value> serial shift required)	This message occurs when the user specifies a parallel nshift parameter too small. A wrong nshift parameter value might cause the simulation to fail.	Change the parallel nshift parameter using the -parallel command line option of MaxTestBench or the -parallel option of the write_patterns command of TetraMAX ATPG.
W-033 - Unified STIL Flow for Serializer is not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support Unified STIL Flow mode for Serializer architecture.	Contact Synopsys for the next available release supporting Unified STIL Flow mode for Serializer.

Warning Message	Description	What Next
W-034 - Unified STIL Flow for multiple shifts load/unload protocol not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support Unified STIL Flow mode for multiple shifts load/unload protocol.	Contact Synopsys for the next available release supporting Unified STIL Flow mode for multiple shifts load/unload protocol.
W-035 - Parallel load mode simulation of multi bit cells not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support parallel load mode simulation of multi bit cells.	Contact Synopsys for the next available release supporting parallel load mode simulation of multibit cells.
W-036 - Scan cell with multiple input ports not yet supported: parallel load mode simulation might fail	The current version of MAX Testbench does not support scan cell with multiple input ports. Since the tool cannot force all the specified input ports, parallel load mode simulation might fail.	Contact Synopsys for the next available release supporting parallel load mode simulation of multiple inputs.
W-037 - Unified STIL Flow for Sequential Compression is not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support the Unified STIL Flow mode for Sequential Compression architecture.	Contact Synopsys for the next available release supporting Unified STIL Flow mode for Sequential Compression.

Warning Message	Description	What Next
`W-038 - Testbench data file requiring very large memory, automatically using/updating -split_out to <value>`	MAX Testbench has detected that the testbench data file size required a memory buffer larger than the one supported currently by Verilog 1995 (the default testbench output). To avoid a Verilog simulation failure, the pattern data has been written out in multiple .dat files; each file will contain a maximum number of patterns specified by the `-split_out` value. A mapping with all the created partitions is reported at the end of Max Testbench execution. Use this map to simulate the desired partition. For example, `simv +tmax_part=0`	
`W-039- Delayed release time (cfg_parallel_release_time) set in configuration file <> ignored (valid only for DSF parallel STILs).`	The configuration option cfg_parallel_release_time is not supported for a USF STIL, nor for a serial-only STIL file	No action required. This message is just a notification that the set value is not considered by MAX Testbench.
`W-040- Unified STIL Flow for Scalable Adaptive Scan is not yet supported. Mode forced to serial only simulation.`	The current version of MAX Testbench does not support the Unified STIL Flow mode for Scalable Adaptive Scan architecture.	Contact Synopsys for the next available release supporting Unified STIL Flow mode for Scalable Adaptive Scan

Warning Message	Description	What Next
W-041 - Disabling the Enhanced Debug Mode for Unified STIL Flow (EDUSF)	Due to some consistency checks, EDUSF mode cannot be activated. The generated testbench will not be able to pinpoint the exact failing scan cell in parallel simulation mode.	
W-042 - Pattern-based failure data format in serial load mode simulation is not compliant with the TetraMAX diagnosis tool.	The pattern-based failure data format of DFTMAX Ultra Chain Test in serial load mode simulation is not compliant with the TetraMAX diagnosis tool.	Use a cycle-based failure data format in serial load mode simulation for DFTMAX Ultra Chain Test in serial load mode simulation. Contact Synopsys for the next available release with the full support of pattern-based failure data format.
W-044 - Detected invalid multibit scan cell, simulation cannot be performed in parallel mode.	MaxTestbench detected multibit scan cells that were incorrectly described. In this case, a parallel mode simulation is not possible since the respective scan cell can't be correctly identified in the design.	Check the input STIL file and the TetraMAX parameters for errors.

Informational Message Descriptions

Table 3 lists all MAX Testbench informational messages and their descriptions.

Table 3 Informational Message Descriptions

Info Message	Description	What Next
`I-001 - nshift parameter is greater or equal than the maximum scan chain length (%d in the current design)`	This message indicates that the value specified for the nshift parameter is greater or equal than the maximum scan chain length. In this situation, as expected, the simulation becomes a serial one.	This is an expected behavior.
`I-002- Time unit sets to <value>`	This is a message to inform the user that he is about to overwrite the automatic setting for this parameter with a specified value using the cfg_time_unit parameter from the configuration file.	This is an expected behavior
`I-003- Time precision sets to <value>`	This is a message to inform the user that he is about to overwrite the automatic setting for this parameter with a specified value using the cfg_time_precision parameter from the configuration file.	This is an expected behavior

Table 3 Informational Message Descriptions (Continued)

Info Message	Description	What Next
I-004- Multiple assignments for signal <name> in signal group <name>, using WFCMap and proceeding with <value>, last label <name>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), the WFCMap resulting value, and the name of the last Label observed during processing. This message is displayed only in verbose mode.	This is an expected behavior
I-005- Event ForceOff (Z) interpreted as CompareUnknown (X) in the event waves of cluster "X" of Signal "%s" in WFT "%s"	This message describes how the tool interprets certain 'unusual' constructs found in the waveform table. These constructs are usually encountered when processing older versions of STIL.	This is an expected behavior

Table 3 Informational Message Descriptions (Continued)

Info Message	Description	What Next
I-006- Multiple assignments for sig <name>, using WFCmap and proceeding with <value>	This message occurs when a signal is assigned multiple values inside a statement. The signal may be part of a SignalGroup or all the assignments may be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), the WFCMap resulting value, and the name of the last Label observed during processing. This message is displayed only in verbose mode.	This is an expected behavior
I-007- Event ForceOff (Z) interpreted as CompareUnknown (X) in the event waves of WFT "%s" containing both compare and force types	This message informs the user about how the tool interprets certain 'unusual' constructs found in the waveform table. Usually encountered when processing older versions of STIL.	This is an expected behavior
I-008- Requesting <name> EVCD file generation (use "tmax_vcde" simulator compiler definition to enable file generation)	User specified a EVCD file in the configuration file. The tool will update the testbench but the simulation will not generate the EVCD file by default.	Specify the "tmax_ vcde" simulator compiler definition to enable file generation

Table 3 Informational Message Descriptions (Continued)

Info Message	Description	What Next
`I-009- Updated Serializer Tail Pipeline internally to zero due to shorter Serializer data length`	In the case of DFTMAX Serializer with slow pipelines (core pipelines), for some configurations TetraMAX does not consider the Serializer Tail pipeline stages as expected by MAX Testbench. When this occurs, MAXTestbench attempts to compensate for this behavior.	This situation rarely occurs.
`I-011- The following clocks will not be pulsed during the parallel Shift: list_of_clocks`	Lists the clocks that will not be used during the parallel shift simulation.	See the I-011 manpage for additional details.

4

Debugging Parallel Simulation Failures Using Combined Pattern Validation

This section describes how to debug parallel simulation failures using the combined pattern Validation (CPV) flow. Using this flow, you can precisely debug patterns by reporting the exact failing scan cell for scan compression architectures.

This debug capability is an enhancement to the existing unified STIL flow (USF) and includes interoperability between TetraMAX ATPG, MAX Testbench, and VCS.

The following sections describe how to debug parallel simulation:

See Also

Writing STIL Patterns

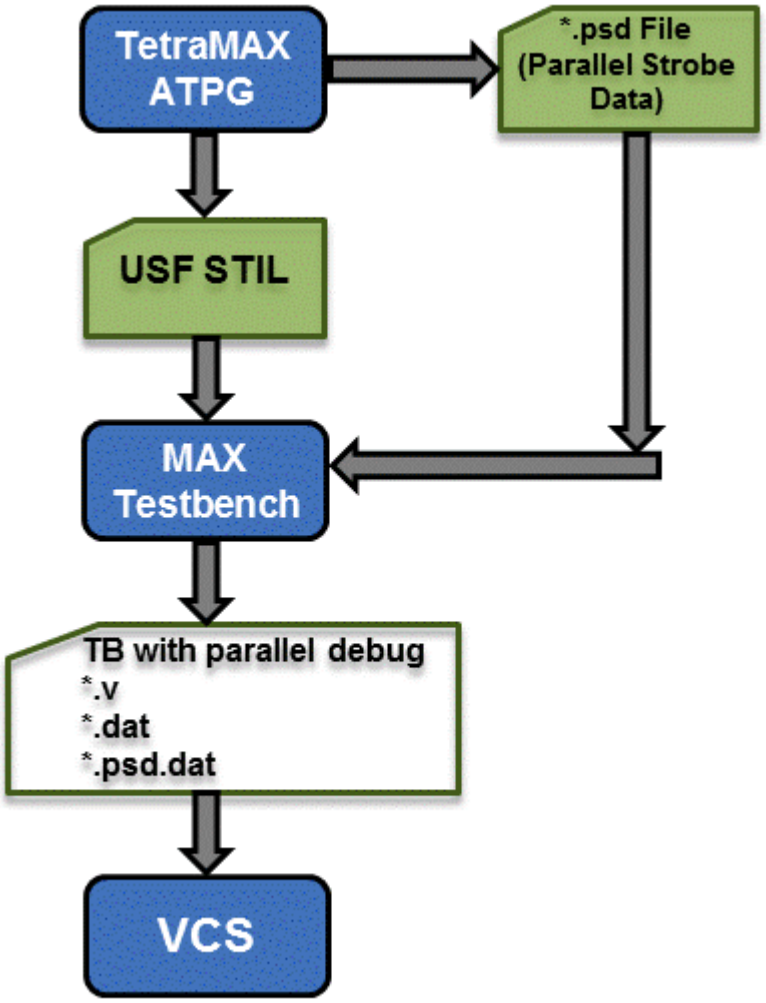
Overview

The CPV parallel simulation failure debug flow is similar to the debug flow used by the unified STIL flow (USF). However, the USF has limited support for debugging parallel simulation failures. For more information on both the DSF and USF, see "Writing STIL Patterns."

The USF simulation report lists the pattern number, scan output pin, and the shift index for each failure, but it not does not include the particular scan cell that failed. For diagnosing manufacturing defects, this information is sufficient, since you usually only need to pinpoint the exact fault site (the location of the faulty gate or pin). However, for parallel simulation pattern debugging, you usually need to identify the exact failing scan cell and instance name.

Using the CPV parallel simulation failure debug flow, you can conveniently debug failures without using TetraMAX ATPG to identify the chains and cell instance names with issues. This flow also provides the flexibility to use your own debug tools. Figure 1 shows the basic CPV parallel simulation failure debug flow.

Figure 1 CPV Parallel Simulation Failure Debug Flow



As shown in Figure 1, TetraMAX ATPG saves the parallel test data to the parallel strobe data (PSD) file in the working directory. You then write the STIL pattern files, and MAX Testbench uses the USF file and the PSD file to generate a testbench and test data file.

MAX Testbench also generates another test data file that holds only the parallel strobe data used during the simulation miscompare activity of the simulator. This additional MAX Testbench output file (*.dat.psd) is used during the load_unload procedure as golden (expected) data, which provides comparison data at the scan chain level and failure information at the scan cell resolution level.

See Also

Using MAX Testbench
Setting the Run Mode

Understanding the PSD File

The PSD file is a binary format file that contains additional parallel strobe data required for debugging parallel simulation failures. You can create a separate PSD file for each pattern unload. Without compression, this file can be four to ten times larger than the original DSF parallel STIL file. You can compress the PSD file as needed using the gzip utility.

The data in the PSD file corresponds to the expected strobe (unload scan chain) data. It is coded using two bits to model states 0, 1 and X, as shown in the following example:

```
Pattern 1 (fast_sequential)
Time 0: load c1 = 0111
Time 1: force_all_pis = 0000000000 00000ZZZZ
Time 2: pulse clocks ck2 (1)
Time 3: force_all_pis = 0000100100 00000ZZZZ
Time 4: measure_all_pos = 00ZZZZ
Time 5: pulse clocks ck1 (0)
Time 6: unload c1 = 0000
```

The History section of the USF file contains attributes that link the PSD file and USF pattern file. This information uses STIL annotation, as shown in the following example:

```
Ann {* PSDF = last_100 *}
Ann {* PSDS = 1328742765 *}
Ann {* PSDA = #0#0/0 *}
```

Note the following:

- PSDF — Identifies the PSD file name and location.
- PSDS — Identifies the unique signature (composed of a date and specific ID number) of the PSD file corresponding to the USF file.
- PSDA — Identifies the number of partitions when more than one PSD file is used.

TetraMAX ATPG does not use the STIL `Include` statement to establish the USF to PSD file link. This means the additional parallel strobe data does not need to use the STIL syntax, which could overload the USF file with large amounts of test information.

Figure 2 shows examples of the attributes in the USF file and the corresponding hex data in the PSD file.

Figure 2 USF File and PSD File Example

USF File with PSD Fields in Bold

```
STIL 1.0 { Design 2005; }
Header {
...
 History {
   Ann {*Wed May  5 03:00:07 2010  *}
   Ann {*PSDF = ./my_psd.bin *}
   Ann {*PSDS = <date><ses_id>*}
...
Signals {}
SignalGroups {}
...
Pattern "_pattern_" {
```

PSD File (Hex format)

```
// PSD File generated by TetraMAX V. XXXX
// <signature>
// <total_pat_nb>

<pat_id> // PSD for pattern 0
<nb_loads>
// PSD for load 1
<chain_1_id>

<parallel_strobe_data>
<chain_2_id >

<parallel_strobe_data>
...
<chain_N_id >

<parallel_strobe_data>
// PSD for load 2
<chain_1_id>

<parallel_strobe_data>
...
// PSD for load L
<pat_id>  // PSD for pattern 1
<nb_loads>
// PSD for load 1
....
```

Creating a PSD File

There are two ways to create a PSD file:

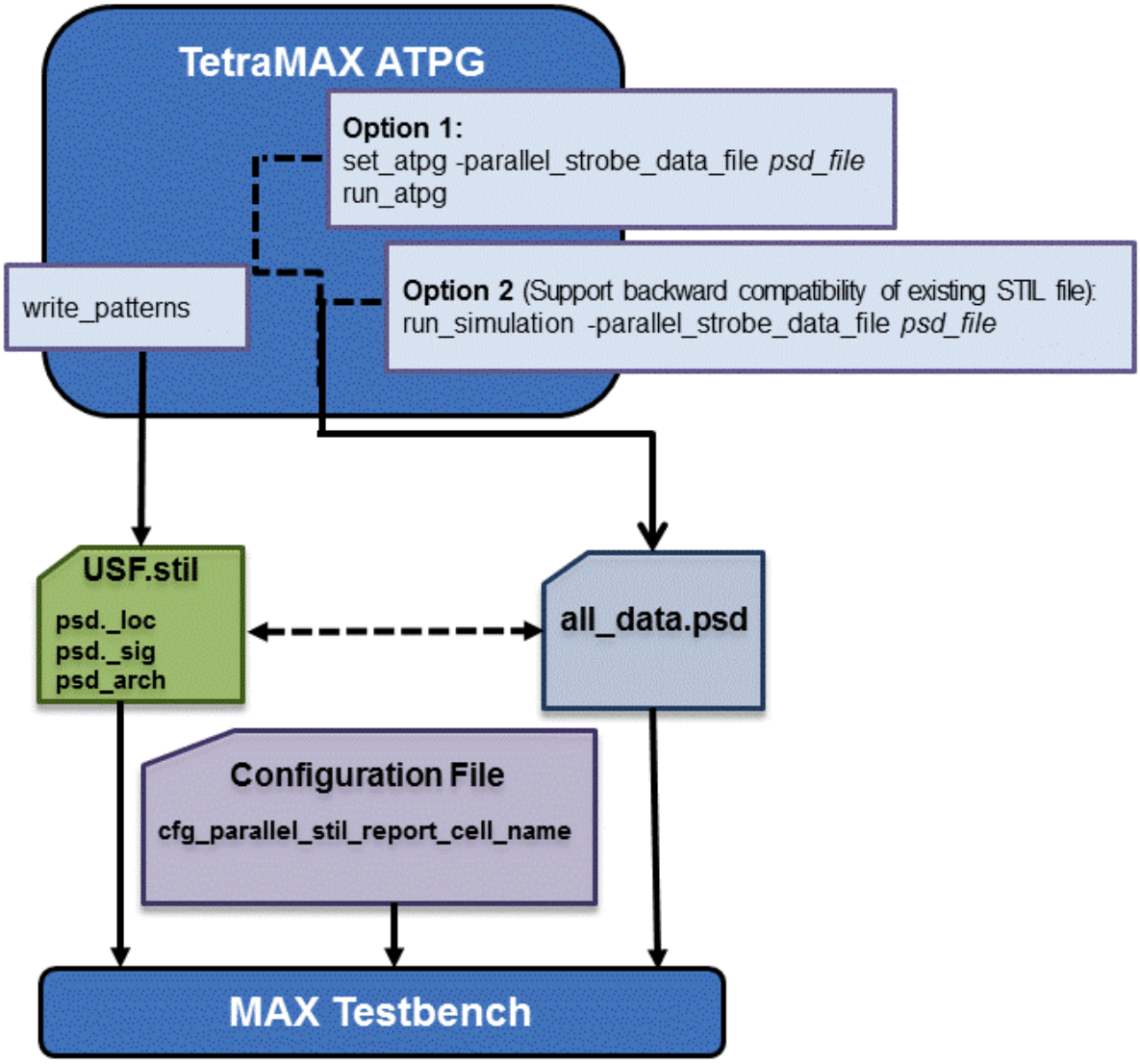
- Using the ATPG flow

  Specify the `-parallel_strobe_data_file` option of the `set_atpg` command and the `run_atpg` command. This process is described in "Using the run_atpg Command to Create a PSD File."
- Using the Run Simulation flow

  Specify the `-parallel_strobe_file` option of the `run_simulation` command to create a PSD file and support the backward compatibility of an existing STIL file. This process is described in "Using the run_simulation Command to Create a PSD File."

Figure 3 shows these options in a flow.

Figure 3 Options for Creating a PSD File



Using the run_atpg Command to Create a PSD File

To generate a PSD file during the ATPG flow, you need to specify the `-parallel_strobe_data_file` option of the `set_atpg` command and the `run_atpg` command.

You can also specify the `report_settings atpg` command to print the settings in the PSD file.

The following example shows how to generate a PSD file using the `run_atpg` command:

```
TEST-T> set_atpg -parallel_strobe_data_file psd_file \
    -replace_parallel_strobe_data_file
```

```
TEST-T> report_settings atpg
atpg = parallel_strobe_data_file=psd_file,
timing_exceptions_au_analysis=no, num_processes=0;
TEST-T> run_atpg
TEST-T> write_patterns out.stil -format stil
TEST-T> write_testbench -input usf.1040.stil \
  -output usf.1040 -replace -parameter \
   { -first 10 -last 40 -config config.file -verbose \
    -log mxtb.log}
Executing 'stil2Verilog'...
maxtb> Starting from test pattern 10
maxtb> Last test pattern to process 40
maxtb> Total test patterns to process 31
maxtb> Detected a Scan Compression mode.
maxtb> Generating Verilog testbench for both serial and parallel
load mode...
```

Note the following:

- When you invoke MAX Testbench, the PSD file specified in the `set_atpg` command is automatically used. If you do not want to include the PSD file, specify the following option during simulation compilation:

  `tmax_usf_debug_strobe_mode=0`

- The `write_testbench` command in the previous example references a configuration file called my_config. This file contains the following command:

  `set cfg_parallel_stil_report_cell_name 1`

  This command is described in detail in "Displaying Instance Names."

Using the run_simulation Command to Create a PSD File

You use the `-parallel_strobe_file` option of the `run_simulation` command to create a PSD file that supports the backward compatibility of an existing STIL file.

The following example shows how to use the `run_simulation` command to create a PSD file:

```
set_atpg -noparallel_strobe_data_file

set_patterns -external usf.stil -delete
Warning: Internal pattern set is now deleted. (M133)
End parsing STIL file usf.stil with 0 errors.
End reading 22 patterns, CPU_time = 23.00 sec, Memory = 0MB

report_patterns -summary
            Pattern Summary Report
-----------------------------------------------
#internal patterns 0
#external patterns (usf.stil) 22
#fast_sequential patterns 22
```

```
-----------------------------------------------

run_simulation -parallel_strobe_data_file \
    test_tr_resim.psd -replace
Created parallel strobe data file 'test_tr_resim.psd'
Begin good simulation of 22 external patterns.
Simulation completed: #patterns=22, #fail_pats=0(0), #failing_
meas=0(0), CPU time=11.00
Total parallel strobe data patterns: 22, external patterns: 22

write_patterns usf_resim.stil -format stil -replace -external
Warning: STIL patterns defaulted to parallel simulation mode.
(M474)
Patterns written reference 158 V statements, generating 802 test
cycles
End writing file 'usf_resim.stil' with 22 patterns, File_size =
1531782, CPU_time = 23.0 sec.

report_patterns -summary
           Pattern Summary Report
-----------------------------------------------
#internal patterns                          0
#external patterns (usf.stil) 22
#fast_sequential patterns 22
-----------------------------------------------

write_testbench -input usf_resim.stil -output usf_resim \
   -replace -parameter { -log mxtb_resim.log -verbose \
   -config my_config }
```

Note the following:

- For TetraMAX-generated ATPG patterns, you should use the `run_simulation` command without any additional options. In this case, TetraMAX automatically uses the appropriate simulation algorithm based on the type of pattern input. TetraMAX recognizes patterns produced using Basic Scan or Fast-Sequential mode, but Full-Sequential mode patterns are not supported in this flow.
- Using the `run_simulation` command results in longer runtimes. Therefore, whenever possible, you should use the flow with the `set_atpg -parallel_strobe_data_file` command.
- You can also improve the performance using the `-num_processes` option of the `set_simulation` command. This option specifies the use of multiple CPU cores. For example, the `set_simulation -num_processes 4` command specifies the use of 4 cores. You can then generate the parallel patterns using the `write_patterns`*`file_name`* `-parallel` command.
- The `write_testbench` command in the previous example references a configuration file called my_config. This file contains the following command:

  ```
  set cfg_parallel_stil_report_cell_name 1
  ```

This command is described in detail in the next section, "Displaying Instance Names."

- When you invoke MAX Testbench, the PSD file specified in the `run_simulation` command is automatically used. If you don't want to include the PSD file, specify the following option during simulation compilation:

  `tmax_usf_debug_strobe_mode=0`

Displaying Instance Names

You can configure MAX Testbench to print the instance names of failing cells during the simulation of a parallel-formatted STIL file. To do this, specify the following command in the MAX Testbench configuration file:

```
set cfg_parallel_stil_report_cell_name 1
```

This configuration file command impacts simulation memory consumption. If you do not want to display instance names, specify the following command:

```
set cfg_parallel_stil_report_cell_name 0
```

You also must enable the use of the configuration file by specifying "1" in the User Control Section of the header of the *.dat file generated by MAX Testbench. In this case, you do not need to regenerate the testbench files. The following example shows the User Control Section:

```
// MAX TB Test Data File, generated by MAX TB
// Module under test: snps_micro
// Generated from original STIL file : ./patterns_
config/pats.usf.stil
// STIL file version: "1.0"

/////////////// User Control Section ///////////////
// Total pattern count to simulate (48), set the new value in
binary radix
110000
// Enhanced Debug for CPV. Set to 0 to disable
1
```

The following example shows the message that prints when the parallel simulation failure debug mode is enabled:

```
XTB: Enabling Enhanced Debug Mode.
XTB: Starting parallel simulation of 48 patterns
XTB: Using 0 serial shifts
```

Flow Configuration Options

In the example flow shown in Figure 3, MAX Testbench uses as input a PSD file created from TetraMAX ATPG and a configuration file that specifies the reporting of instance names. Depending on your debugging needs and simulation resources, you can use different combinations of this input to MAX Testbench.

For example, if you do not want to reference the instance names in the simulation miscompare messages, you can exclude this information from the configuration file as described in "Displaying Instance Names." Or, if you do not want to reference the strobe data provided in the PSD file (see "Understanding the PSD File"), you can exclude this file.

Table 1 shows a summary of MAX Testbench mismatch debug support.

Table 1 MAX Testbench Simulation Mismatch Support

Mismatch Debug Instance name	MAX Testbench			
	Dual STIL Flow		Unified STIL Flow	
	Serial	Parallel	Serial	Parallel
Legacy	NA	With CFG File	NA	With CFG File
Compression & serializer	NA	With CFG File	NA	With CFG File & PSD File

The following section, "Example Simulation Compare Messages," shows examples of these reporting options.

Example Simulation Miscompare Messages

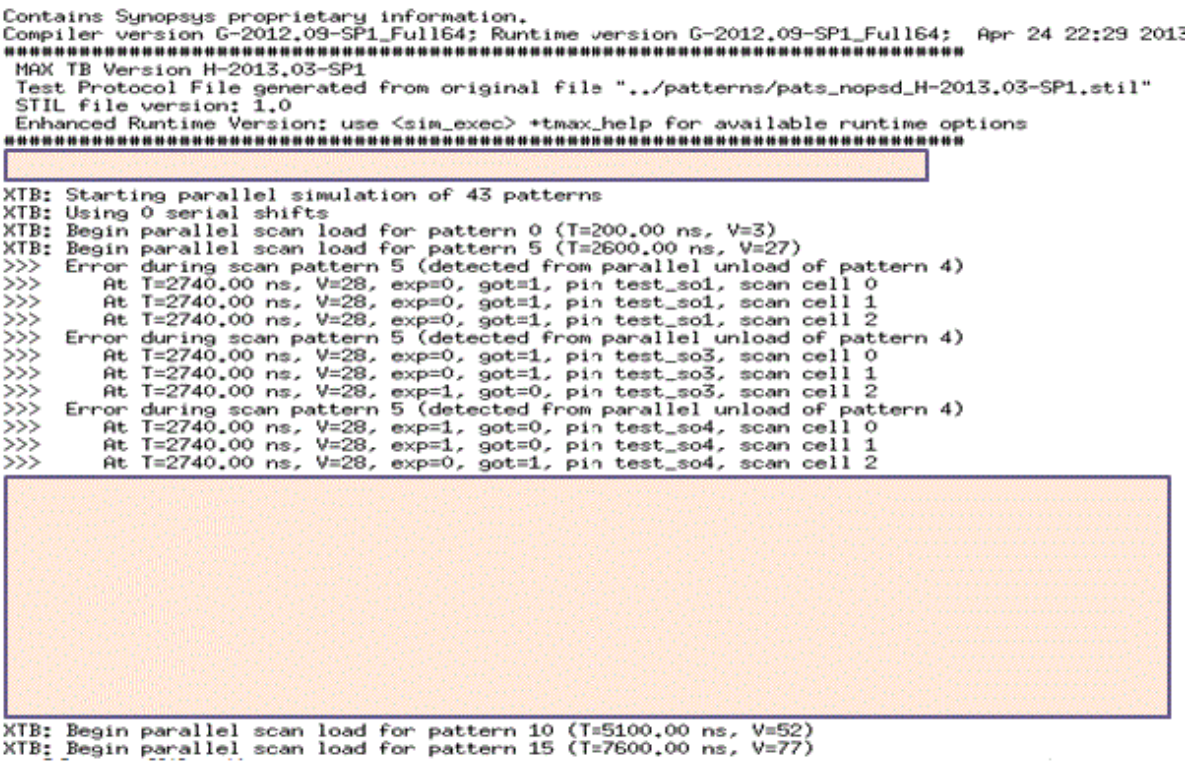
You can use different configuration combinations of input to report various simulation miscompare messages.

The following sections show examples of the various miscompare messages:

- Example 1 shows messages that appear when neither a PSD file or a configuration file is used as input to MAX Testbench.
- Example 2 shows messages that appear when a PSD file is used, but not a configuration file.
- Example 3 shows messages that appear when you use both a PSD file and a configuration file as input.
- Verbosity Setting Examples shows messages with the trace reporting verbosity level set to 0 (the default) using the `+tmax_msg` runtime option.

Example 1

Example 1 Messages That Appear With No PSD File and No Configuration File



```
Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1_Full64: Runtime version G-2012.09-SP1_Full64;  Apr 24 22:29 2013
********************************************************************************
 MAX TB Version H-2013.03-SP1
 Test Protocol File generated from original file "../patterns/pats_nopsd_H-2013.03-SP1.stil"
 STIL file version: 1.0
 Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime options
********************************************************************************

XTB: Starting parallel simulation of 43 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 2
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so3, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so3, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so4, scan cell 2

XTB: Begin parallel scan load for pattern 10 (T=5100.00 ns, V=52)
XTB: Begin parallel scan load for pattern 15 (T=7600.00 ns, V=77)
```

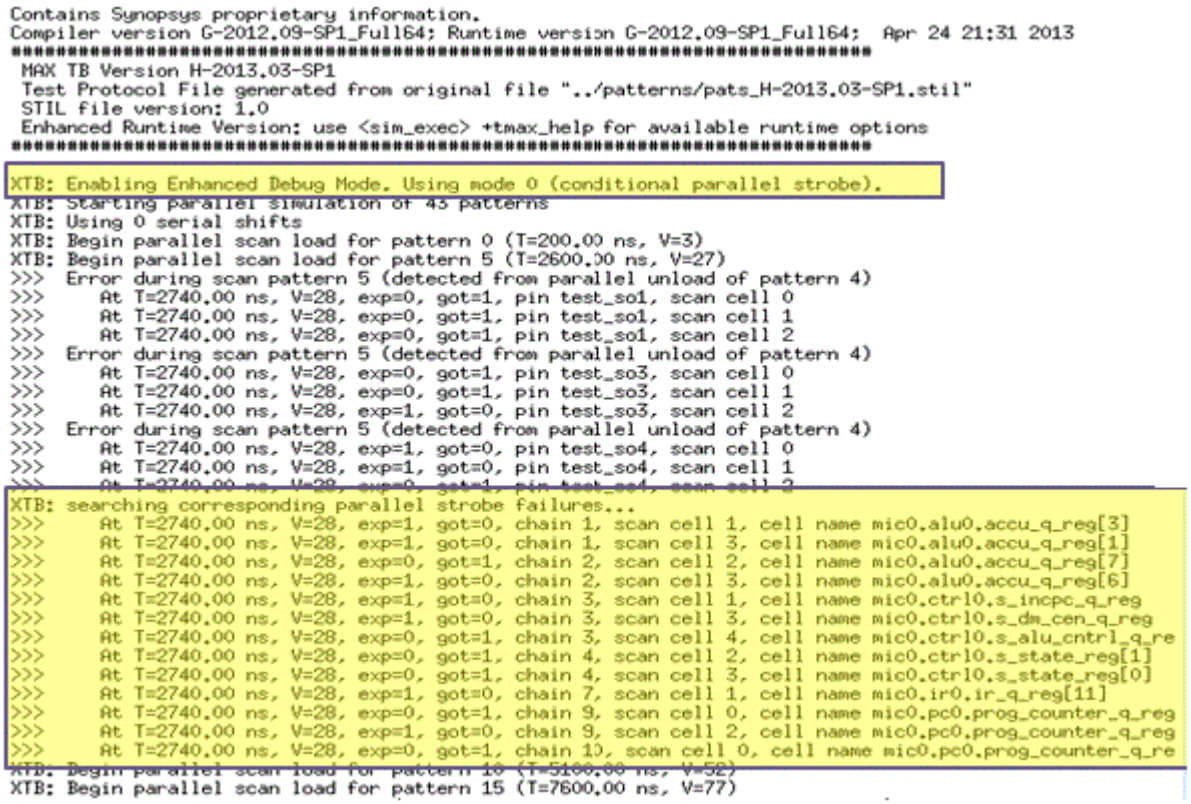
Example 2

Example 2 Messages That Appear With a PSD File and No Configuration File

```
Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1_Full64; Runtime version G-2012.09-SP1_Full64;  Apr 24 21:31 2013
********************************************************************************
 MAX TB Version H-2013.03-SP1
 Test Protocol File generated from original file "../patterns/pats_H-2013.03-SP1.stil"
 STIL file version: 1.0
 Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime options
********************************************************************************

XTB: Enabling Enhanced Debug Mode. Using mode 0 (conditional parallel strobe).
XTB: Starting parallel simulation of 43 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 2
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so3, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so3, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>>  Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=0, got=1, pin test_so4, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 3
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 2, scan cell 2
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 3
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 3, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 3, scan cell 3
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 3, scan cell 4
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 3
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 7, scan cell 1
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 0
>>>     At T=2740.00 ns, V=28, exp=1, got=0, chain 9, scan cell 2
>>>     At T=2740.00 ns, V=28, exp=0, got=1, chain 10, scan cell 0
XTB: Begin parallel scan load for pattern 10 (T=5100.00 ns, V=52)
XTB: Begin parallel scan load for pattern 15 (T=7600.00 ns, V=77)
```

Example 3

Example 3 Messages That Appear With a PSD File and a Configuration File



Verbosity Setting Examples

You can further control the reporting of simulation miscompare messages by specifying the `+tmax_msg` runtime option, or by setting the `cfg_message_verbosity_level` command in the MAX Testbench configuration file. For details on the `+tmax_msg` option, see "Setting the Verbose Level."

The following examples show how the messages appear when you set the verbosity level to 0 (the default) using the `+tmax_msg` runtime option.

Example 4 Using a PSD File and No Configuration File With Verbosity Level 0

```
#################################################################
MAX TB
Test Protocol File generated from original file "pats.usf.stil"
STIL file version: 1.0
NO CONFIGURATION FILE
#################################################################
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
```

```
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 2
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 2
```

Example 5 Using a PSD File and Configuration File with Verbosity Level 0

```
################################################################
MAX TB
Test Protocol File generated from original file "pats.usf.stil"
STIL file version: 1.0
USING THE CONFIGURATION FILE
################################################################
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 2,
cell name mic0.alu0.accu_q_reg[7]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2,
cell name mic0.ctrl0.s_state_reg[1]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 2,
cell name mic0.pc0.prog_counter_q_reg[5]
```

Example 6 Using a Configuration File and No PSD file with Verbosity Level 0:

```
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
>>> Error during scan pattern 7 (detected from parallel unload of
pattern 6)
```

Debug Modes for Simulation Miscompare Messages

You can specify modes for reporting various levels of details of simulation runtime miscompare messages for scan compression technology. To do this, use the `+tmax_usf_debug_strobe_mode` predefined simulation command option. The syntax for this option is as follows:

`+tmax_usf_debug_strobe_mode=<0, 1, 2, 3>`

Each mode is described as follows:

0 - Disables parallel simulation failure debug and generates normal error messages related only to the scan output. This mode is useful for increasing simulation performance when you only want to quickly determine the pass/fail status of very large designs.

1 - Specifies the default mode, referred to as the "Conditional parallel strobe mode." This mode generates miscompare simulation messages using parallel strobe data that is applied only to USF failures.

2 - This mode, referred to as the "Unconditional parallel strobe mode," concurrently activates the USF and the CPV parallel strobe data for generating miscompare messages for each pattern.

3 - Generates miscompare messages only for internal errors using parallel strobe data applied to each pattern. The messages generated from this mode do not indicate if a parallel strobe failure is propagated to the primary scan output (after the compressor).

Note: You can also specify this option as a command in the Runtime field of the testbench (*.v) file produced by MAX Testbench. However, the simulation command line always overrides the default specification of the testbench file.

Table 2 summarizes the errors reported for each mode. An "Error" is actually a reported mismatch message generated during scan-unload processing."Normal IO Errors" refer to error messages generated during scan that report errors relative to the scan output. "Internal Errors" refer to error messages generated during scan that report the error relative to an internal scan cell.

Table 2 Debug Modes and Reported Errors

Mode	Normal IO Errors	Internal Errors
Mode=0	Yes	No
Mode=1	Yes	Yes
Mode=2	Yes	Yes
Mode=3	No	Yes

Note that in serial simulation, the Internal error field is not available. Only the normal I/O errors are recorded, as if you received tester failures at the I/O of the device.

The following examples show how messages for the various modes appear in the log file:

MODE 0 Log File Example

```
jv_comp_parallel_mode0.log:XTB: Enhanced Debug Mode disabled (user
request).
jv_comp_parallel_mode0.log:XTB: Simulation of 7 patterns completed
with 6 mismatches (time: 2700.00 ns, cycles: 27)
-------------------------------------------------------------
```

MODE 1 Log File Example

```
jv_comp_parallel_mode1.log:XTB: Enabling Enhanced Debug Mode.
Using mode 1 (conditional parallel strobe).
jv_comp_parallel_mode1.log:XTB: Simulation of 7 patterns completed
with 6 mismatches (1672 internal mismatches) (time: 2700.00 ns,
cycles: 27)
-------------------------------------------------------------------
```

MODE 2 Log File Example

```
jv_comp_parallel_mode2.log:XTB: Enabling Enhanced Debug Mode.
Using mode 2 (unconditional parallel strobe).
jv_comp_parallel_mode2.log:XTB: Simulation of 7 patterns completed
with 6 mismatches (10569 internal mismatches) (time: 2700.00 ns,
cycles: 27)
-----------------------------------------------------------------
```

MODE 3 Log File Example

```
jv_comp_parallel_mode3.log:XTB: Enabling Enhanced Debug Mode.
Using mode 3 (only parallel strobe).
jv_comp_parallel_mode3.log:XTB: Simulation of 7 patterns completed
with (10569 internal mismatches) (time: 2700.00 ns, cycles: 27)
```

Pattern Splitting

MAX Testbench stores key simulation miscompare activity for the parallel strobe data in a *psd.dat file. This data is used during the load_unload procedure as golden (expected) data. By default, the *psd.dat file contains a maximum of 1000 patterns. When more than 1000 patterns are used, MAX Testbench automatically splits the contents of the PSD file and generates a set of corresponding set of *_psd.dat files.

You can manually specify pattern splitting in TetraMAX ATPG or MAX Testbench using any of the following flow options:

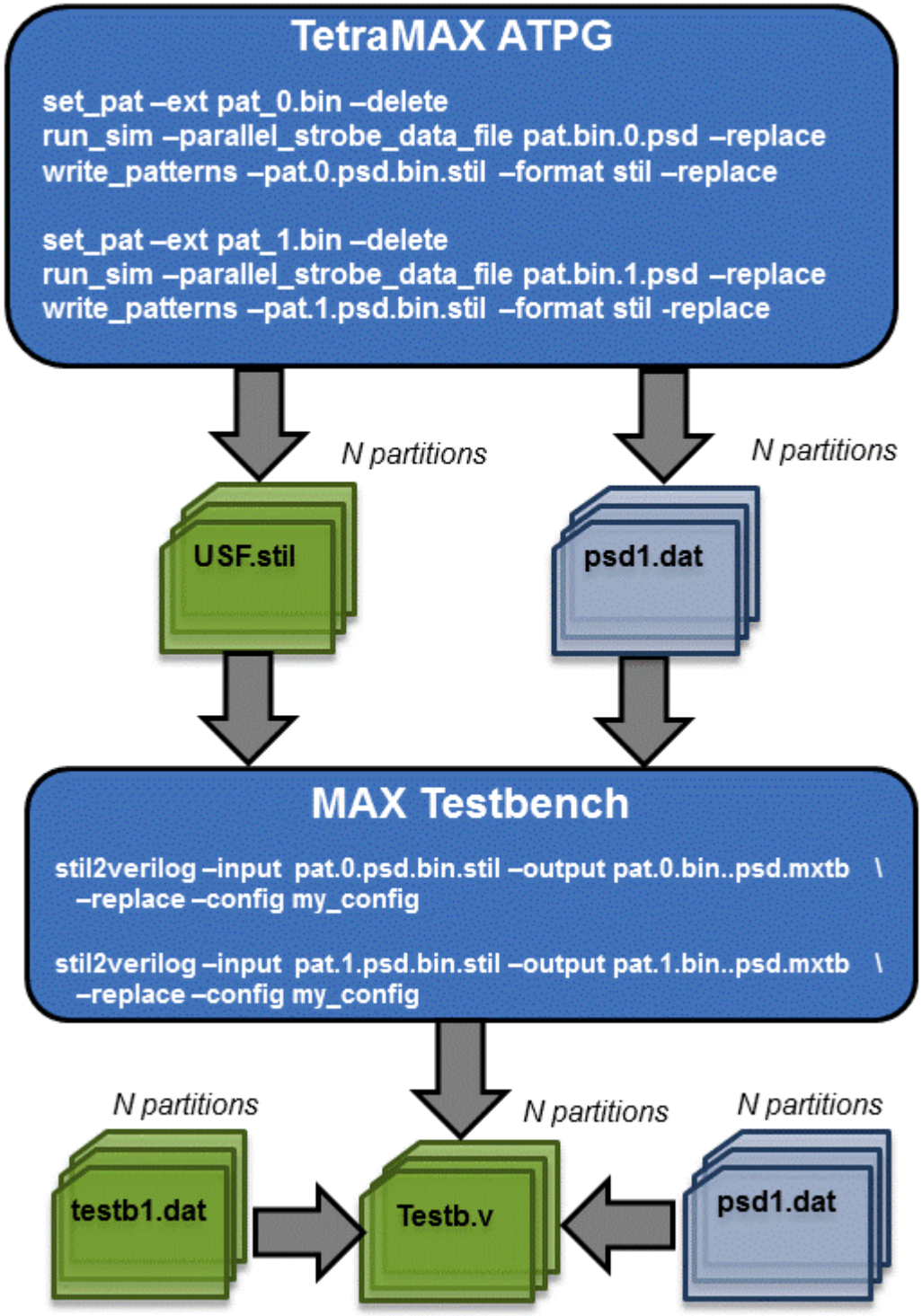
- Split the patterns using the `write_patterns` command in TetraMAX ATPG before they are used by MAX Testbench. This process is described in "Splitting Patterns Using TetraMAX."
- Use the `-split_out` option of the `write_testbench` or `stil2Verilog` commands to split the patterns in MAX Testbench. This flow is described in "Splitting Patterns Using MAX Testbench."
- Use the `run_simulation` command flow and the `-first` and `-last` options of the `write_testbench` or `stil2Verilog` commands to address only the failing VCS pattern sets in MAX Testbench. This flow is described in "Specifying a Range of Split Patterns Using MAX Testbench."

Splitting Patterns Using TetraMAX

You can split patterns using the `write_patterns` command in TetraMAX ATPG before using the patterns in MAX Testbench. For example, you might want TetraMAX ATPG to write out 500 patterns per file. To do this, read each split STIL pattern file into TetraMAX ATPG and then specify the `run_simulation -parallel_strobe_data_file` command for each pattern file.

Figure 5 shows the flow for using the `write_patterns` command to split patterns before using MAX Testbench. For examples of this flow, see "Examples Using TetraMAX for Pattern Splitting."

Figure 5 Debugging Flow Using Split Patterns in Binary Format



Examples Using TetraMAX For Pattern Splitting

The following examples show pattern splitting using the `write_patterns` command in TetraMAX ATPG:

- Set Up Example
- Example Using Pattern File From write_patterns Command
- Example Using Split USF STIL Pattern Files

Set Up Example

The following example writes out split binary patterns from the same ATPG run:

```
run_atpg -auto
write_patterns pats.bin -format binary -replace -split 3
```

Example Using Pattern File From write_patterns Command

This example uses split binary pattern files from the `write_patterns` commands in the previous example, then writes out USF STIL patterns:

```
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_0.bin -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.bin.0.psd -replace
write_patterns pat.0.psd.bin.stil -format stil -replace -external
report_patterns -summary

set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_1.bin -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.bin.1.psd -replace
write_patterns pat.1.psd.bin.stil -format stil -replace -external
report_patterns -summary
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_2.bin -delete
report_patterns -summary

run_sim -parallel_strobe_data_file pat.bin.2.psd -replace
write_patterns pat.2.psd.bin.stil -format stil -replace -external
report_patterns -summary

write_testbench -input pat.0.psd.bin.stil -output \
    pat.0.bin..psd.mxtb -replace -parameters \
    {-log mxtb_bin.0.log -verbose -config my_config}
write_testbench -input pat.1.psd.bin.stil -output
pat.1.bin..psd.mxtb \
    -replace -parameters {-log mxtb_bin.1.log -verbose \
    -config my_config}
```

```
write_testbench -input pat.2.psd.bin.stil -output \
    pat.2.bin..psd.mxtb -replace -parameters \
    {-log mxtb_bin.2.log -verbose -config my_config}
```

Example Output Files:

```
pat.bin.2.psd
pat.bin.1.psd
pat.bin.0.psd
pat.2.psd.bin.stil
pat.1.psd.bin.stil
pat.0.psd.bin.stil
mxtb_bin.2.log
pat.2.bin..psd.mxtb.v
pat.2.bin..psd.mxtb.dat
pat.2.bin..psd.mxtb_psd.dat
mxtb_bin.1.log
pat.1.bin..psd.mxtb.v
pat.1.bin..psd.mxtb.dat
pat.1.bin..psd.mxtb_psd.dat
mxtb_bin.0.log
pat.0.bin..psd.mxtb.v
pat.0.bin..psd.mxtb.dat
pat.0.bin..psd.mxtb_psd.dat
```

Example Using Split USF STIL Pattern Files

The following example uses split USF STIL pattern files:

```
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_0.stil -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.usf.0.psd -replace
write_patterns pat.usf.0.psd.stil -format stil -replace -external
report_patterns -summary

set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_1.stil -delete
report_patterns -summary

run_sim -parallel_strobe_data_file pat.usf.1.psd -replace
write_patterns pat.usf.1.psd.stil -format stil -replace -external
report_patterns -summary

set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_2.stil -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.usf.2.psd -replace
write_patterns pat.usf.2.psd.stil -format stil -replace -external
report_patterns -summary

write_testbench -input pat.usf.0.psd.stil -output \
```

```
    pat.usf.0.psd.mxtb -replace -parameters \
    {-log mxtb_usf.0.log -verbose -config my_config}
write_testbench -input pat.usf.1.psd.stil -output \
    pat.usf.1.psd.mxtb -replace -parameters \
     {-log mxtb_usf.1.log -verbose -config my_config}
write_testbench -input pat.usf.2.psd.stil -output \
    pat.usf.2.psd.mxtb -replace -parameters {-log \
    mxtb_usf.2.log -verbose -config my_config}
```

Example Output Files:

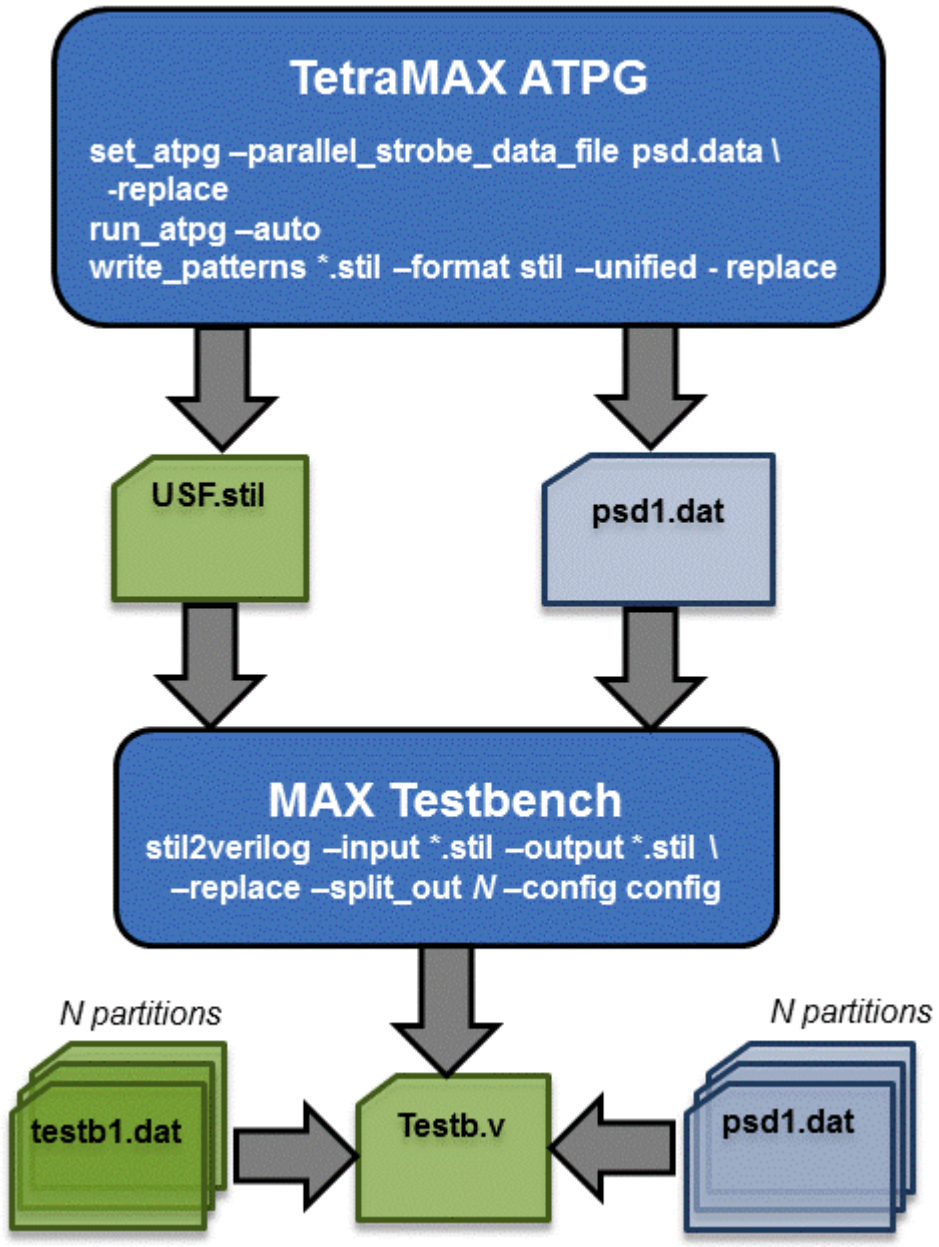
```
pat.usf.2.psd
pat.usf.1.psd
pat.usf.0.psd
pat.usf.2.psd.stil
pat.usf.1.psd.stil
pat.usf.0.psd.stil
mxtb_usf.2.log
pat.usf.2.psd.mxtb.v
pat.usf.2.psd.mxtb.dat
pat.usf.2.psd.mxtb_psd.dat
mxtb_usf.1.log
pat.usf.1.psd.mxtb.v
pat.usf.1.psd.mxtb.dat
pat.usf.1.psd.mxtb_psd.dat
mxtb_usf.0.log
pat.usf.0.psd.mxtb.v
pat.usf.0.psd.mxtb.dat
pat.usf.0.psd.mxtb_psd.dat
```

Splitting Patterns Using MAX Testbench

You can manually specify pattern splitting in MAX Testbench using the `-split_out` option of the `write_testbench` or `stil2Verilog` commands.

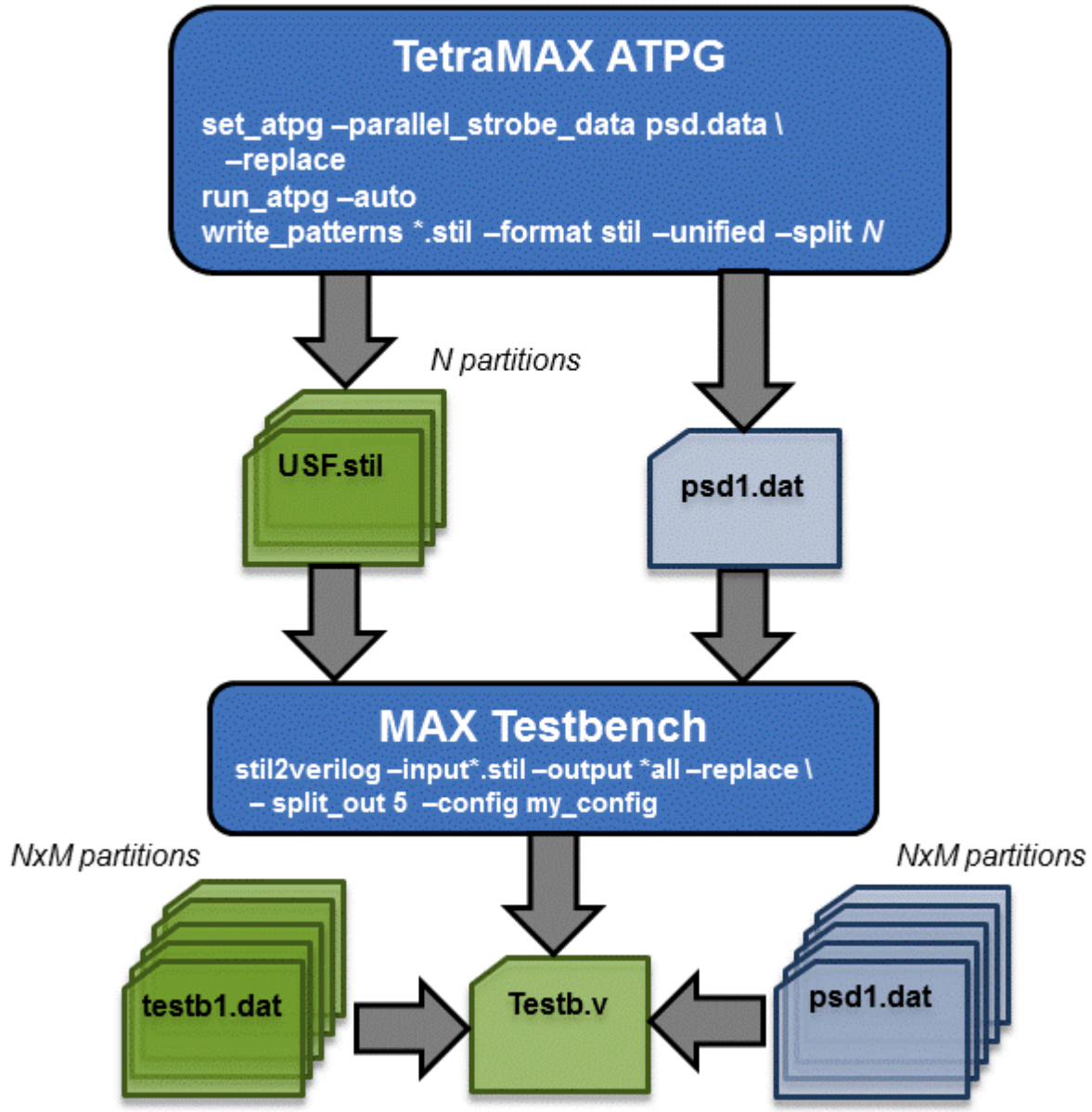
Figure 6 shows the flow for splitting patterns using MAX Testbench.

Figure 6 Flow for Using MAX Testbench to Split Patterns



You can also split patterns in both TetraMAX ATPG and MAX Testbench. This flow is described in Figure 7.

Figure 7 Flow For Using Both TetraMAX and MAX Testbench to Split Patterns

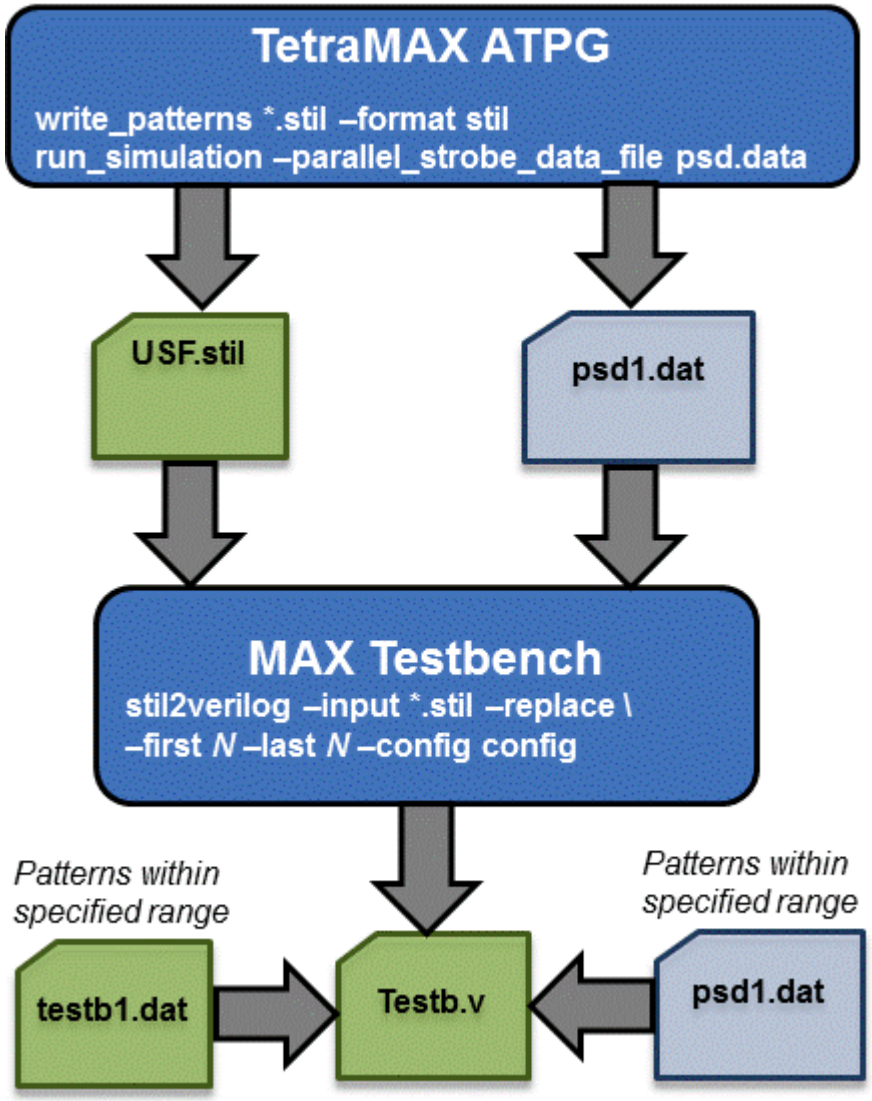


Specifying a Range of Split Patterns Using MAX Testbench

You can split a specified range of split patterns in MAX Testbench so you can better focus your debugging efforts. To do this, use the standard `run_simulation` command flow and read back only the set of binary or STIL patterns that failed in simulation, then produce the PSD file (for details, see “Using the run_simulation Command to Create a PSD File”).

Next, use the `-first` and `-last` options of the `stil2Verilog` or `write_testbench` commands to produce a selected set of pattern files, then resimulate these files in VCS . This flow is described in Figure 8.

Figure 8 Flow for Splitting a Selected Range of Patterns



MAX Testbench and Consistency Checking

When you run MAX Testbench, it automatically detects and processes the PSD file, and issues the following message:

```
maxtb> Detected STIL file with Enhanced Debug for CPV (EDCPV)
Capability (PSD file: psdata). Processing...
```

MAX Testbench performs a series of consistency checks between the contents of the USF file and PSD file. If any issues are detected, it generates a testbench file without the parallel strobe data, and issues the following warning message:

```
Warning: Disabling the Enhanced Debug Mode for Combined Pattern
Validation (EDCPV) corrupted PSD file due to bad file signature
(1329175245). Make sure the PSD file corresponds to the generated
STIL file (W-041)
```

The following message is specific to the debugging parallel simulation failures using the Combined Pattern Validation (CPV) flow:

```
W-041: Disabling the Enhanced Debug Mode for Unified STIL Flow
(EDUSF)
```

This message is issued when the debug mode for parallel simulation failures cannot be activated because of consistency checking failures. As a result, the generated testbench is not be able to pinpoint the exact failing scan cell in parallel simulation mode. MAX Testbench continues to generate the testbench files without the parallel strobe data file.

See Also

MAX Testbench Error Warnings and Messages

Limitations

Note the following limitations related to debugging simulation failures using CPV:

- The Full-Sequential mode is not supported.
- The `set_patterns` and `run_simulation` commands are not supported for multiple contiguous runs (see "Creating a PSD File"). Also, update and masking flows are not supported, including pattern restore from binary and new pattern write flows, multiple pattern read back, and single merged pattern write.
- The `-first`, `-last`, `-sorted`, `-reorder`, and `-type` options of the `write_patterns` command are not supported.
- The `-sorted`, `-reorder`, and `-type` options of the `write_testbench` and `stil2Verilog` commands are not supported.

- A PSD file cannot be generated by the `write_patterns` command.
- Multicore simulation is not supported in the `run_simulation` flow.
- The `-last` option of the `run_simulation` command is not supported.

5

Troubleshooting MAX Testbench

The following sections describe how to resolve MAX Testbench-generated errors:

Introduction

You can run a design against a set of predefined stimulus and check (validate) the design response against an expected response. This process mimics the behavior of the tester against a device under test.

Problems might occur with

- incorrect or incomplete STIL data
- incorrect connections of the device to this stimulus in the testbench
- incorrect device response due to structural errors or timing problems inside the design

Ultimately, the goal of using a testbench is to validate that the device response, often with accurate internal timing, does match the response expected in the STIL data.

There are alternative and additional troubleshooting strategies to what is presented in this section. The most important aspects when testing are knowledge of the design and remembering the fundamental characteristics of the test you're troubleshooting.

Troubleshooting Compilation Errors

This section describes some of the typical error messages you encounter during compilation when using VCS or Ncsim. These error messages are related to the following parameters or issues:

- FILELENGTH Parameter
- NAMELENGTH Parameter
- Memory Allocation

FILELENGTH Parameter

The following error message appears if you exceed the maximum file length:

```
XTB Error: cannot open /disk/path.to.a.large.file.name.maxtb_
psd.dat PSD file. Disabling Enhanced Debug USF mode...
```

By default, the FILELENGTH parameter in MAX Testbench is set to 1024 characters, which corresponds to the 1024 character limit imposed by NCSIM. In some cases, you can set this parameter to a higher limit at the compilation stage either in the testbench file or at the simulation command line.

You can use the following MAX Testbench parameter to change the maximum file length:

```
parameter FILELENGTH = 1024; // max length for file names
```

If you are using a set of long paths, you can set the Verilog FILELENGTH parameter in the testbench, using the following syntax:

```
-pvalue+tb_name. FILELENGTH=your_value
```

You also might encounter the following error:

```
Warning-[STASKW_CO] Cannot open file
/disk/some.path.name.to.a.very.large.file.name.maxtb.Verilog.gz,
8535
The file
/disk/some.path.name.to.a.very.large.file.name.maxtb.maxtb_
psd.dat'
could not be opened. No such file or directory.
Ensure that the file exists with proper permissions.
XTB Error: cannot open
/disk/some.path.name.to.a.very.large.file.name.maxtb_psd.dat PSD
file. Disabling Enhanced Debug USF mode...
```

For exceptionally long paths, you can override the Verilog parameter in the testbench and specify an extended file length at the simulation recompile command line using the following syntax:

```
vcs -pvalue+tb_name. FILELENGTH=your_value
```

NAMELENGTH Parameter

For parallel strobe data (PSD) files, the default filename length is 800 characters. If you exceed this length, the following message appears:

```
Warning-[STASKW_CO] Cannot open file
./LongName.p.maxtb.v, 1278
The file 'ReallyLongName.p.maxtb_psd.dat' could not be opened. No
such file or directory.
Ensure that the file exists with proper permissions.
XTB Error: cannot open ReallyLongName.p.maxtb_psd.dat PSD file.
Disabling Enhanced Debug USF mode...
```

To correct this error, you can set the NAMELENGTH parameter in the testbench or at the simulation recompile line using the following syntax:

```
vcs -pvalue+tb_name.NAMELENGTH=800
```

Memory Allocation

The following error message identifies a memory allocation error:

```
XTB Error: size of test data file filename.dat exceeding testbench
memory allocation. Exiting...

(recompile using -pvalue+design1_test.tb_part.MDEPTH=<###>).
```

In this case, you need to recompile the testbench using the following Verilog parameter to adjust the memory allocation:

```
-pvalue+design1_test.tb_part.MDEPTH=depth)
```

For more information, see "MAX Testbench Runtime Programmability."

Troubleshooting Miscompares

The following sections describe the process of debugging failures (miscompares) detected when simulating a design using MAX Testbench and a set of generated STIL pattern data:

- Handling Miscompare Messages
- Localizing a Failure Location
- Adding More Fingerprints

These sections also present some techniques for using MAX Testbench to assist in the analysis of simulation mismatch messages when they occur during a simulation run. These techniques start with the direct approach:

- Understanding the simulation mismatch message completely
- Proceeding to some advanced options to assist in debugging the overall simulation behavior
- Miscompares are most commonly the misapplication of STIL data and caused by either incorrect design constructs for this data
- STIL constructs for the design or the context of the application

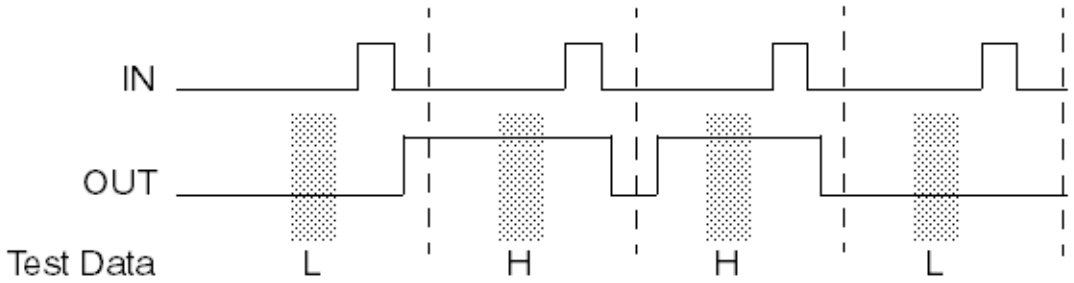
Handling Miscompare Messages

Test data is sampled at distinct points in the test pattern, which are called test strobes. Test strobes indicate whether the device is operating properly or not in response to the stimulus provided by the test data.

In general, miscompares happen only on outputs (or bidirectional signals in the output mode). This limits the visibility into both the device operation and the test data expectations, which can make analyzing these failures more complicated. Furthermore, these output measurements are placed to occur at locations of a stable device response to assure repeatable test operation. And finally, output strobe miscompares often identify an internal failure that might have happened some time in the past. All of these issues complicate the analysis process.

In Figure 1, the limited visibility into the design behavior is shown by output strobe data on signal “out” that indicates this signal remains high between two test Vectors, although the actual device operation has a period of a low state between these two measurements. This is not incorrect, in fact it is probably expected design operation.

Figure 1 Measurement Points on "OUT"



This section details the four forms of miscompare messages generated by Verilog DPV and the information that each message contains.

Miscompare Message 1

```
STILDPV: Signal SOT expected to be 1 was X
         At time 1240000, V# 5
         With WaveformTable "_default_WFT_"
         At Label: "pulse"
         Current Call Stack: "capture_CLK"
```

This miscompare message is generated from a STIL Vector when an output response does not match the expected data present in the test data. The message contains a fingerprint of information to consider when analyzing this failure. It reports the nature of the error and where it happened, but does not indicate why.

- The expected state in the STIL test data, and the actual state seen in the simulation during this test strobe.
- Both the simulation time index and the STIL vector number, to cross-reference this failure in simulation time with the test data.
- The current WaveformTable name active in this vector, to help correlate this failure with the STIL data and identify what timing was active at this failure.
- The last STIL vector label seen during execution of the STIL test data. Again, this helps to correlate the failure with the STIL data. Be aware that the label might be the last one seen if there is no label on this vector (the message reports "Last Label Seen:" if the label is not on this vector itself).
- The procedure and macro call stack, if this failure happens from inside a procedure or macro call (or series of calls).

Both the labels and the call stack information might be lists of multiple entries. Verilog DPV separates multiple entries with a backslash (\) character.

Miscompare Message 2

```
STILDPV: Signal SOT expected to be 1 was X
         At time 9640000, V# 97
         With WaveformTable "_default_WFT_"
         Last Previous Label (6 cycles prior): pulse"
```

```
        Current Call Stack: load_unload”
        TetraMAX pattern 7, Index 5 of chain c1 (scancell A1)
```

If the failure occurs during an identified unload of scan data during the simulation with the simulation executing serial scan simulation, then the failure message will contain an additional line of information that identifies:

- The failing pattern number from the TetraMAX ATPG information.
- The index into the Shift operation that reported the failure.
- The name of the failing scan chain.
- The name of the scan cell that aligns with this index.

The index specified in this message is relative to the scan cell order identified in the ScanStructures section of the STIL data; index 1 = the first scan cell in the ScanStructures section and so on.

Miscompare Message 3

```
STILDPV: Parallel Mode Scancell A1 expected to be 1 was X
         At time 9040100, V# 91
         With WaveformTable "_default_WFT_"
         TetraMAX pattern 7, Index 5 of chain c1
```

If the failure occurs during an identified unload of scan data during the simulation with the simulation executing parallel scan simulation, then the failure message is formatted differently. It identifies:

- The failing scan cell, and the expected and actual states of that cell.
- The time that this failure was detected (beware: in parallel mode this is the time that the parallel-measure operation was performed. This is inside the Shift operation being performed, but it might not correlate with a strobe time inside a Vector, because the scan data must be sampled before input events occur).
- The WaveformTable active for this Shift.
- The failing pattern number from the TetraMAX information.
- The index into the Shift operation that reported the failure.
- The name of the failing scan chain.

Like miscompare message 2, the index specified in this message is relative the scan cell order identified in the `ScanStructures` section of the STIL data; index 1 = the first scan cell in the `ScanStructures` section and so on.

Miscompare Message 4

```
STILDPV: Signal SOT changed to 1 in a windowed strobe at time
940000
```

Output strobes can be defined to be instantaneous samples in time, or “window strobes” that validate an output remains at the specified state for a region of time.

When window strobes are used, an additional error might be generated if an output transitions inside the region of that strobe. This error message identifies the signal, the state it transitioned to, and the simulation time that this occurred.

For an example of the scenario that generates this message, see Figure 5.

Localizing a Failure Location

When a failure occurs, your first debugging step is to localize the failure in the STIL data file. The following sections describe how to localize a failure by interpreting the fingerprint information:

- Resolving the First Failure
- Miscompare Fingerprints
- Additional Troubleshooting Help

When the failure is localized, you need to determine if it's reasonable to test this output signal at this location.

With STIL constructs, an output remains in the last specified operation (the last WaveformCharacter asserted) until that operation (WaveformCharacter) is changed on that signal.

In the example that follows, a signal called “tdo” is being tested in a Vector after a Shift operation. But in the two Vectors, “tdo” is not included, because it is expected that this signal should remain in the last tested state, or should this signal have been set to an untested value (generally an “X” WaveformCharacter for TetraMAX tests). Notice that the “tck=P” signal is repeated in the last two vectors, because it does not remain in the last tested state.

```
load_unload  {
   W _default_WFT_;
   ...
   Shift { V { tdi=#; tdo=#; tck=P; }}
   V { tdi=#; tdo=#; tck=P; tms=1; }
   V { tck=P; tms=1; }
   V { tck=P; tms=0; }
 }
```

Resolving the First Failure

Subsequent failures can be caused by cascading effects; the very first error is the best error to start examining. Because basic scan patterns, starting with a scan load and ending with a scan unload, are self-contained units, failures in one scan pattern do not typically propagate—unless the failure is indicative of a design or timing fault that persists throughout the patterns (or the patterns have sequential behavior).

Don’t take “first” literally as the first printed mismatch, all mismatches that happen at the same time step (or even at different times, but in the same STIL vector), are all a consequence of a problem that was functionally detected at this point. Any error generated in the first failing vector is a good starting point.

Miscompare Fingerprints

The following sections explain how to interpret the information contained in the miscompare messages and how to troubleshoot various situations:

- Expected versus Actual States
- Current Waveform Table
- Labels and Calling Stack

Expected versus Actual States

The first piece of data to analyze is the expected state (specified in the test data), and the actual state present from the simulation run.

Are all the actual states an “X” value? This can indicate initialization issues, or the loss of the internal design state during operation caused by glitches or transient events. If an “X” is found in the simulation, start tracing it backward in both the design and in simulation time—where did that X come from?

Are the mismatches hard errors? For example is a “1” expected, but it is actually a “0”? This could be caused by one of the following:

- Timing problems in the design
- Strobe positioning
- Extra or missing clocks
- Glitches, or transients

Current Waveform Table

The next piece of data in the mismatch message to analyze is the WaveformTable reference.

What are the event times specified for this strobe? What are the event times on the other inputs? Are the event relationships proper—was the test developed with the strobe events after (or before) the input events and is that timing relationship maintained in this WaveformTable?

Is there enough time between the input events and the output strobes? Does the design have time to settle before the strobe measurement?

TetraMAX ATPG has distinct event ordering requirements, and the timing specified in the WaveformTable needs to be compatible with the test generation. In particular, the strobe times must be placed before the clock pulse (pre-clock measure) or after the clock pulse (post-clock measure).

The name of the WaveformTable can sometimes help locate the failure as well. In particular for path delay environments, the name of the WaveformTable can identify the launch, capture, or combined events and isolate the failing Vector that uses that named WaveformTable.

Labels and Calling Stack

The final piece of information to analyze in the mismatch message is the referenced label and the current call stack at the failing location. This can often isolate the location of a mismatch by the presence of the label or the name of the procedure currently active when this mismatch occurred.

What activity is happening here? Is it a capture or scan operation? Is an output strobe expected here?

Additional Troubleshooting Help

Sometimes the information contained in the mismatch message is not sufficient to localize the failure in the STIL data. When this happens, the first thing to do is to activate the tracing options to get more information about what was being simulated when the failure occurred. The next section describes how to activate the MAX Testbench trace options.

Sometimes tracing might not get clearly to the failing location either. The last recourse is to edit the STIL data itself and add more information.

Adding More Fingerprints

If you cannot identify the location of a failure, you might need to edit the STIL data and add additional information. The most helpful construct to add is the Label statements to a Vector that did not have distinct labels (see following example). Because the previous label is always printed in the miscompare message, adding labels directly can eliminate ambiguity in identifying that failing location.

```
load_unload  {
   W _default_WFT_;
   ...
   Shift { V { tdi=#; tdo=#; tck=P; }}
   V { tdi=#; tdo=#; tck=P; tms=1; }
 1_u_post_2: V { tck=P; tms=1; }
 1_u_post_3: V { tck=P; tms=0; }
 }
```

Labels might be added in STIL data files generated by TetraMAX ATPG or might be added to the procedure definitions (if the label is added to a procedure) defined in the STL procedure file data sent to TetraMAX ATPG as well, if TetraMAX ATPG is used to regenerate the STIL data test.

Debugging Simulation Mismatches Using the write_simtrace Command

This section describes the process for using the `write_simtrace` command to assist in debugging ATPG pattern miscompares found during a Verilog simulation. You can use this command in conjunction with sinmulation miscompare information to create a new Verilog module to monitor additional nodes. A typical flow using TetraMAX ATPG and VCS is also provided.

The following topics are covered in this section:

Overview

Analyzing simulation-identified mismatches of expected behavior during the pattern validation process is a complex task. There are many reasons for a mismatch, including:

- Response differences due to internal design delays
- Differences due to effects of the “actual” timing specified
- Formatting errors in the stimulus
- Fundamental errors in selecting options during ATPG

Each situation might contribute to the causes for a mismatch. The only evidence of a failure is a statement generated during the simulation run that indicates that the expected state of an output generated by ATPG differs from the state indicated by the simulation. Unfortunately, there is minimal feedback to help you identify the cause of the situation.

To understand the specific cause of the mismatch, you need to compare two sets of simulation data: the ATPG simulation that produced the expected state and the behavior of the Verilog simulation that produced a different state.

After you identify the first difference in behavior, there are still several more steps in the analysis process. You will need to trace back this first difference through the design elements (and often back through time) to identify the cause of the difference. The process of tracing back through time involves re-running the simulation data to produce additional data; as a result, the analysis of this issue is an iterative process,

The key to identifying the discrepancies between the environments is to correlate the information between the Verilog simulation and the TetraMAX ATPG simulation. TetraMAX ATPG includes a graphical schematic viewer (GSV) with simulation annotation capability. Verilog also has several mechanisms to provide access to internal simulation results that are common to all Verilog simulators.

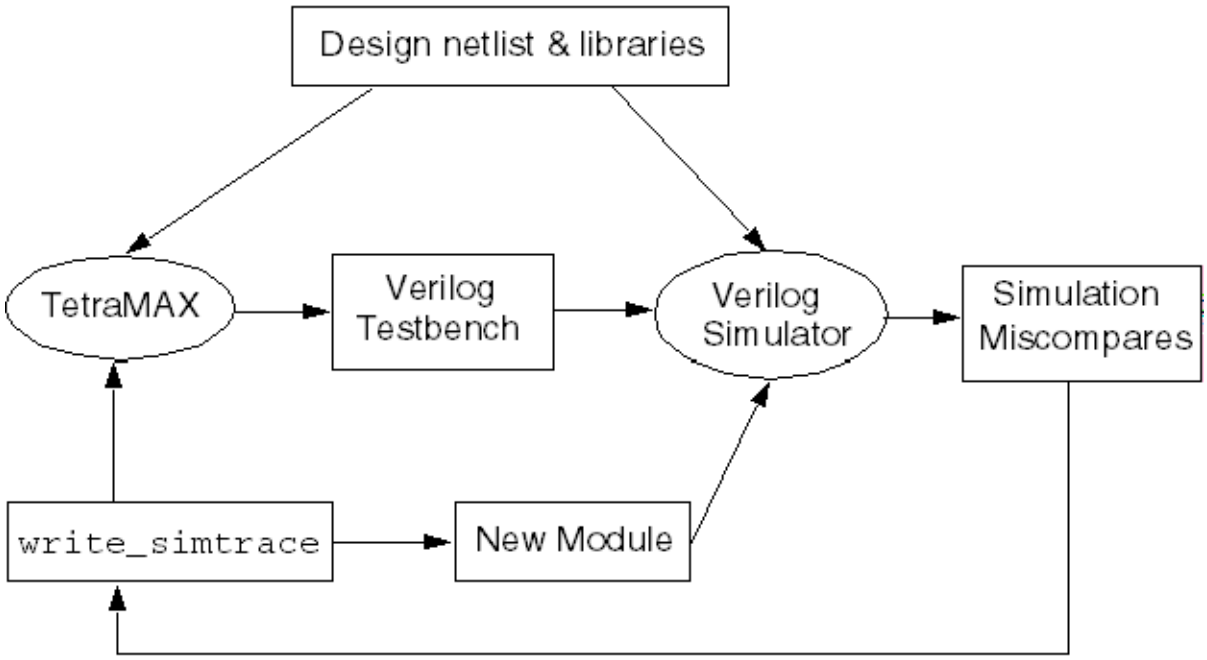
The `write_simtrace` command facilitates the creation of a Verilog module by adding simulation data to be used for comparison with TetraMAX ATPG.

Debugging Flow

Figure 1 shows a typical flow for debugging simulation miscompares using the `write_simtrace` command.

Note: This flow assumes that you are familiar with Verilog simulation. It also assumes that you are using a common netlist for both the Verilog and TetraMAX ATPG environments, and that you have executed the `run_simulation` command after ATPG with no failures.

Figure 1 Debugging Simulation Miscompares Using write_simtrace



Note in Figure 1 that a Verilog testbench is written out after the TetraMAX ATPG process, and is simulated. The simulation log file shows miscompares on scan cells or primary outputs. For each miscompare, you will need to analyze the relevant nodes in the TetraMAX GSV to find their source. The `write_simtrace` command is used to generate a new Verilog module with these additional signals and read it into the simulator. If you monitor the changes on the nodes in the simulation environment at the time the miscompares occur, and correlate that data with the same pattern in TetraMAX ATPG, you will eventually see some differences between the two environments that led to the divergent behavior.

The overall process of analyzing simulation miscompares is iterative. You can use the same TetraMAX session for ATPG by analyzing with the GSV and running `write_simtrace`. On the other hand, the simulation would need to be rerun with the new module to monitor the specified signals.

If you do not want to keep the TetraMAX session running (due to license or hardware demands, for example), it is recommended that you write out an image after DRC and save the patterns in binary format. This will ensure that you can quickly re-create the TetraMAX state used for debugging.

Input Requirements

To leverage the functionality of this feature, you need to supply a common or compatible netlist for both TetraMAX ATPG and the Verilog simulator.

You also need to provide a MAX Testbench format pattern. Additional testbench environments produced by Synopsys tools are supported but might require additions or modifications depending on the naming constructs used to identify the DUT in the testbench. Usage outside these flows is unsupported.

A TetraMAX scan cell report, as produced by the following command, is useful for providing the instance path names of the scan cells:

```
report_scan_cells -all > chains.rpt
```

To avoid rerunning TetraMAX ATPG from scratch, it is recommended that you create an image of the design after running DRC and then save the resulting ATPG patterns in binary format. This ensures that the TetraMAX environment can be quickly recovered for debugging simulation miscompares if the original TetraMAX session cannot be maintained.

Depending on the context and usage of Verilog, you might need to edit the output simtrace file to add a `timescale` statement. In addition, this file can be modified to identify an offset time to start the monitoring process.

You also need to modify the Verilog scripts or invocation environment to include the debug file as one of the Verilog source files to incorporate this functionality in the simulation.

Using the write_simtrace Command

The `write_simtrace` command generates a file in Verilog syntax that defines a standalone module that contains debug statements to invoke a set of Verilog operations. This debugging process references nodes specified by the `-scan` and `-gate` options. Because this is a standalone module, it references these nets as instantiated in the simulation through the testbench module; there are dependencies on these references based on the naming convention of the top module in the testbench module.

After running the `write_simtrace` command, if all nodes specified were found and the file was written as expected, TetraMAX ATPG will print the following message:

```
End writing file 'filename' referencing integer nets, File_size =
integer
```

This statement identifies how many nets were placed in the output file to be monitored. Note that the file name will not include the path information.

Understanding the Simtrace File

The format of the output simtrace file is shown below:

```
// Generated by TetraMAX(TM)
// from command: < simtrace_command_line >
`define TBNAME AAA_tmax)testbench_1_16.dut
// `define TBNAME module_test.dut
module simtrace_1;
  initial begin
//  #<time_to_start> // uncomment and enter a time to start
    $monitor("%t: <scan_data>; <gate_data>", $time(), <list of
net
references>);
  end
```

```
endmodule // simtrace_1
```

The name of this module is the name of the file without an extension. The module consists of a Verilog initial block that contains an annotation (commented-out) that you can uncomment and edit to identify the time to start this trace operation.

The default trace operation uses the Verilog `$monitor` statement, which is defined in the Verilog standard and supported (with equivalent functionality) across all Verilog clones.

Each `-scan` and `-gate` option identifies a set of monitored nets in the display. Each of these sets is configured as identified below. A semicolon is placed between each different set of nodes in the display to emphasize separate options.

The `<scan_data>` is explained as follows:

If the scan reference contains a chain name and a cell name, for example, " `c450:23` ", then the display will contain this reference name, followed by 3 state bits that represent the state of the scan element before this reference, the state of this reference, and the state after this reference. The states before and after are enclosed in parentheses. If there is no element before (this is the first element of the chain) or no element after (this is the last element of the chain), then the corresponding state will not be present. Following the 3 states, each non-constant input to this cell is listed as well. This allows tracing of scan enable and scan clock behavior during the simulation. For example, for a cell in the middle of a chain:

```
C450:23 (0)1(1), D:0 SI:1 SE:0 CK:0
```

The `<gate_data>` is formatted similarly to the `<scan_data>` with a port name specified. The name of the signal or net is printed, followed by the resolved state of that net. For example: `Clk1:0` .

If the gate_reference is to a module, then the information printed looks very similar to the information for scan_data, with one output state of the module, followed by a listing of all non-constant inputs.

All names may be long and might traverse through the design hierarchy. By default, only the last twenty characters of the name are printed in the output statement. The `-length` option may be specified to make these names uniformly longer or shorter.

You need to read the simtrace file into a Verilog simulation by adding this filename to the list of Verilog source files on the Verilog command line or during invocation.

Error Conditions and Messages

The output file is notgenerated if there are errors on the `write_simtrace` command line. All errors are accompanied by error messages of several forms, which are described as follows:

- A standard TetraMAX error message is issued for improper command arguments, missing arguments, or incomplete command lines (no arguments).

  In addition, M650 messages might be generated, with the following forms:
- `Cannot write to simulation debug file <name>. (M650)`

  `No nodes to monitor in simulation debug file <name>. (M650)`

These two messages indicate a failure to access a writable file, or that there were no nodes to monitor from the command line. Both of these situations mean that an output file will not be generated.

Example Debug Flow

The following use case is an example of how to use the debug flow.

After running ATPG and writing out patterns in legacy Verilog format, the simulation of the patterns results in the following lines in the Verilog simulation log file:

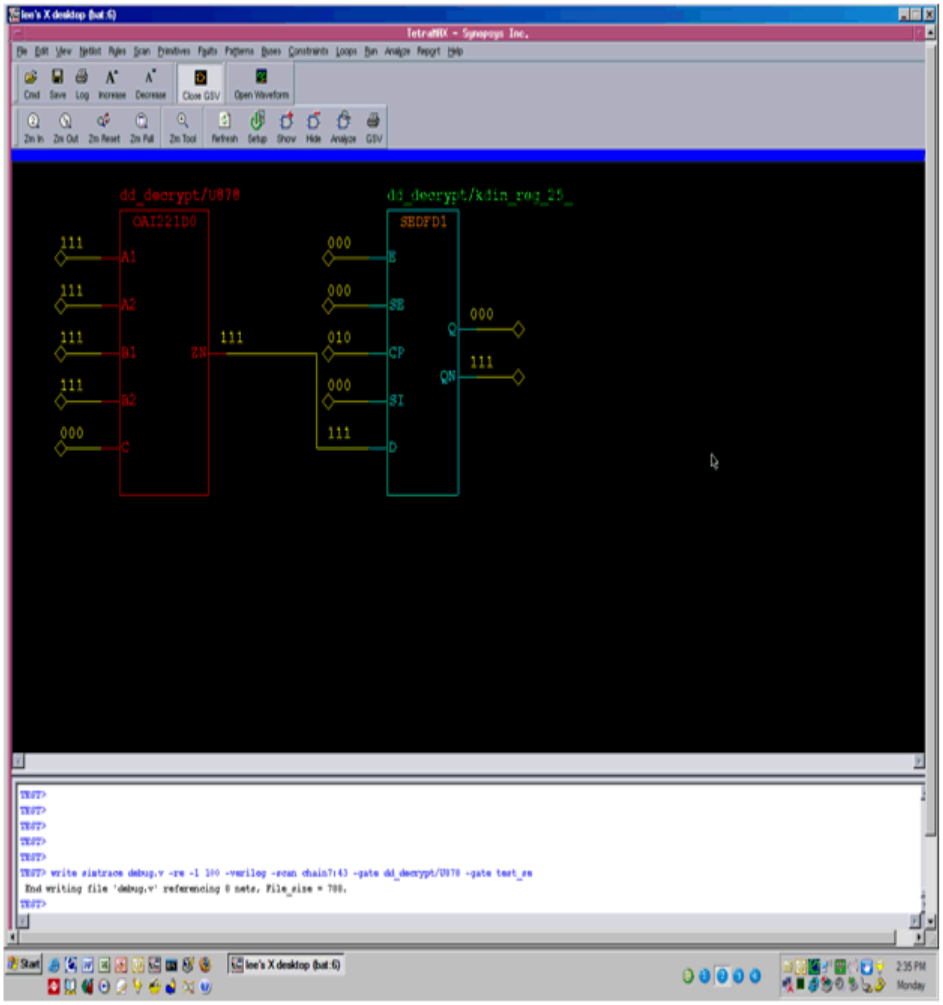
```
       37945.00 ns; chain7:43 0(0) INs: CP:1 SI:0; dd_
decrypt.U878.ZN:0,
 INs: A1:1 A2:1 B1:1 B21 C:1;
       37955.00 ns; chain7:43 0(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:0,
 INs: A1:1 A2:1 B1:1 B21 C:1

 // *** ERROR during scan pattern 4 (detected during final pattern
unload)
   4 chain7 43 (exp=0, got=1)  // pin SO_7, scan cell 43, T=
 38040.00 ns
 //        40000.00 ns : Simulation of 5 patterns completed with 1
errors
```

From the TetraMAX scan cells report:

```
 chain7     43  MASTER    NN   10199  dd_decrypt/kdin_reg_25_
(SEDFD1)
```

The miscompared gates and patterns are displayed in the TetraMAX GSV, as shown in Figure 2.

Figure 2 Display of miscompared gates and patterns



To create the debug module in TetraMAX ATPG, specify the following `write_simtrace` command:

```
TEST-T> write_simtrace debug.v -re -l 100 -scan chain 7:43 \
-gate { dd_decrypt/U878 test_se }
End writing file 'debug.v' referencing 8 nets, File_size = 788.
```

After rerunning the simulation with the debug.v module, the following information is now included in the Verilog simulation log file:

```
       37945.00 ns; chain7:43 1(0) INs: CP:1 SI:0; dd_
decrypt.U878.ZN:1,
 INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;

       37955.00 ns; chain7:43 1(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:1,
 INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;

 // *** ERROR during scan pattern 4 (detected during final pattern
unload)
```

```
   4 chain7 43 (exp=0, got=1)  // pin SO_7, scan cell 43, T=
 38040.00 ns
```

To correlate the information that appears in the TetraMAX GSV for pattern 4, look at the values in the simulation log file at the time of the capture operation. To do this, search backward from the time of the miscompare to identify when the scan enable port was enabled:

```
       33255.00 ns; chain7:43 0(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:0,
 INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;

       33300.00 ns; chain7:43 0(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:0,
 INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:0;

       33545.00 ns; chain7:43 1(0) INs: CP:1 SI:0; dd_
decrypt.U878.ZN:1,
 INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:0;

       33555.00 ns; chain7:43 1(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:1,
 INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:0;

       33600.00 ns; chain7:43 1(0) INs: CP:0 SI:0; dd_
decrypt.U878.ZN:1,
 INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:1;
```

This example shows that the D input of the scan cell will capture the output of `dd_decrypt.U878`. Notice that there is a difference between the TetraMAX value and the simulator value for `dd_decrypt.U878.C`. If you can identify the cause of this discrepancy, you will eventually find the root cause of the miscompare. By tracing the logic cone of `dd_decrypt.U878.C` in the TetraMAX GSV to primary inputs or sequential elements, the additional objects to be monitored in simulation can be easily extracted and their values compared against TetraMAX ATPG.

Restrictions and Limitations

Note the following usage restrictions and limitations:

- Encrypted netlists for TetraMAX ATPG or the Verilog simulator are not supported because the names provided by this flow will not match in both tools.
- Non-Verilog simulators are not supported.

6

PowerFault Simulation

PowerFault simulation technology enables you to validate the IDDQ test patterns generated by TetraMAX ATPG.

The following sections describe PowerFault simulation:

- PowerFault Simulation Technology
- IDDQ Testing Flows
- Licensing

Note: PowerFault-IDDQ might not work on unsupported operating platforms or simulators. See the *TetraMAX ATPG Release Notes* for a list of supported platforms and simulators.

PowerFault Simulation Technology

PowerFault simulation technology verifies quiescence at strobe points, analyzes and debusg nonquiescent states, selects the best IDDQ strobe points for maximum fault coverage, and generates IDDQ fault coverage reports.

Instead of using the IDDQ fault model, you can use the standard stuck-at-0/stuck-at-1 fault model to generate ordinary stuck-at ATPG patterns, and then allow PowerFault to select the best patterns from the resulting test set for IDDQ measurement. The PowerFault simulation chooses the strobe times that provide the highest fault coverage.

PowerFault technology uses the same Verilog simulator, netlist, libraries, and testbench used for product sign-off, helping to ensure accurate results. The netlist and testbench do not need to be modified in any way, and no additional libraries need to be generated.

You run PowerFault after generating IDDQ test patterns with TetraMAX ATPG, as described in "Quiescence Test Pattern Generation" in the *TetraMAX ATPG User Guide*.

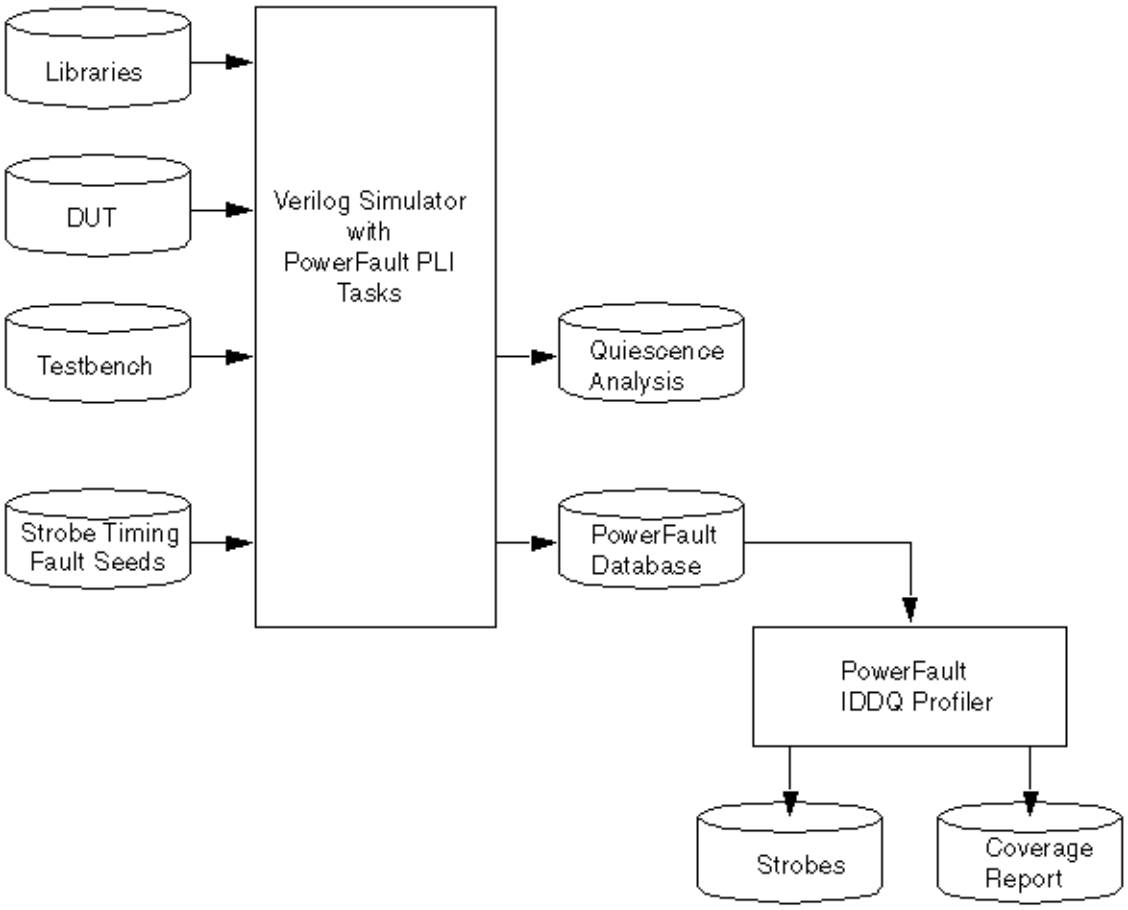
You perform IDDQ fault detection and strobe selection in two stages:

1. Run a Verilog simulation with the normal testbench, using the PowerFault tasks to specify the IDDQ configuration and fault selection, and to evaluate the potential IDDQ strobes for quiescence.

   The inputs to the simulation are the model libraries, the description of the device under test (DUT), the testbench, and the IDDQ parameters (fault locations and strobe timing information).

   The simulator produces a quiescence analysis report, which you can use to debug any leaky nodes found in the design, and an IDDQ database, which contains information on the potential strobe times and corresponding faults that can be detected.
2. Run the IDDQ Profiler, IDDQPro, to analyze the IDDQ database produced by the PowerFault tasks. The IDDQ Profiler selects the best IDDQ strobe times and generates a fault coverage report, either in batch mode or interactively.

Figure 1 Data Flow for PowerFault Strobe Selection



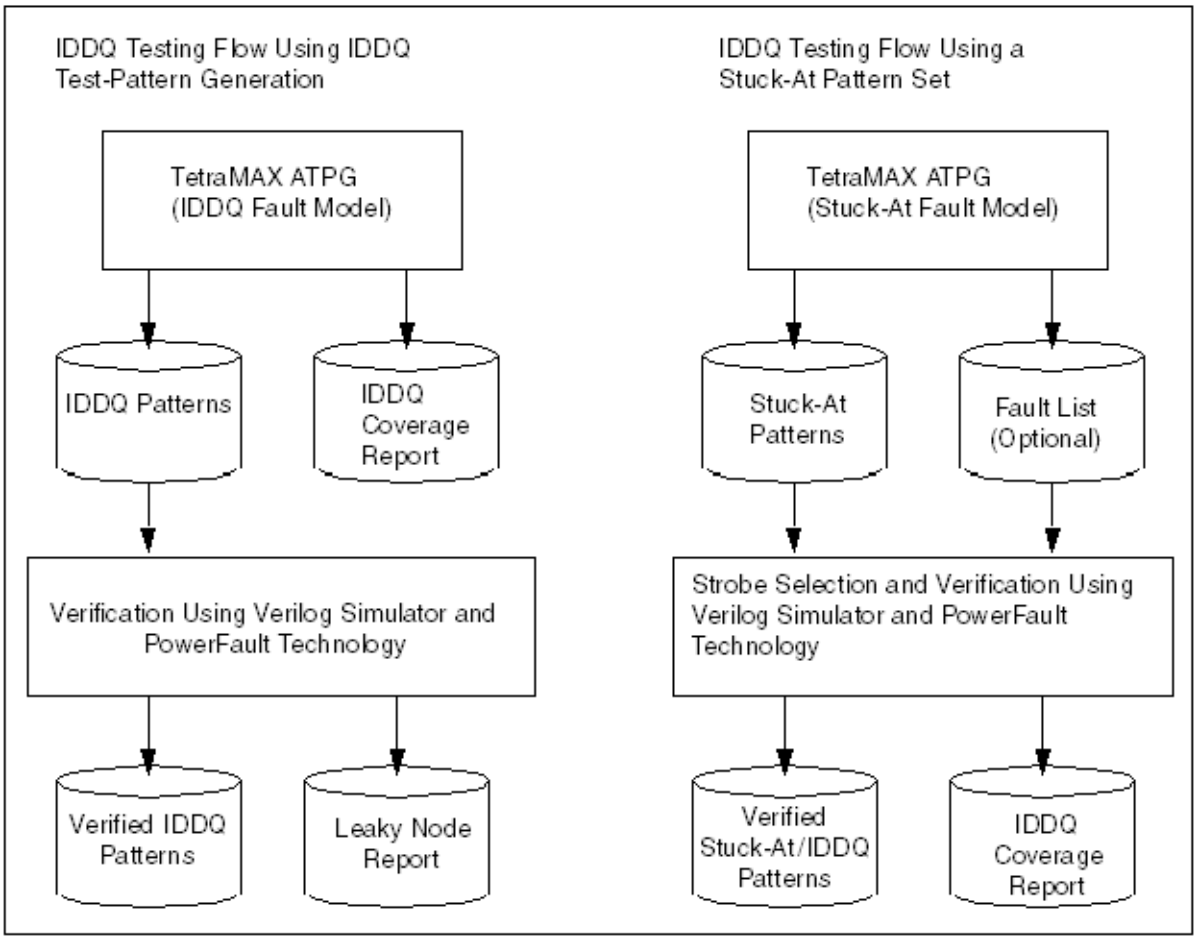
IDDQ Testing Flows

There are two recommended IDDQ testing flows:

- IDDQ Test Pattern Generation — TetraMAX ATPG generates an IDDQ test pattern set targeting the IDDQ fault model rather than the usual stuck-at fault model.
- IDDQ Strobe Selection From an Existing Pattern Set — Use an existing stuck-at ATPG pattern set and let the PowerFault simulation select the best IDDQ strobe times in that pattern set.

Figure 2 shows the types of data produced by these two IDDQ test flows.

Figure 2 IDDQ Testing Flows



IDDQ Test Pattern Generation

In the IDDQ testing flow shown in Figure 2, TetraMAX ATPG generates a test pattern that directly targets IDDQ faults. Instead of attempting to propagate the effects of stuck-at faults to the device outputs, the ATPG algorithm attempts to sensitize all IDDQ faults and apply IDDQ strobes to test all such faults. TetraMAX ATPG compresses and merges the IDDQ test patterns, just like ordinary stuck-at patterns.

While generating IDDQ test patterns, TetraMAX ATPG avoids any condition that could cause excessive current drain, such as strong or weak bus contention or floating buses. You can override the default behavior and specify whether to allow such conditions by using the set_iddq command.

In this IDDQ flow, TetraMAX ATPG generates an IDDQ test pattern and an IDDQ fault coverage report. It generates quiescent strobes by using ATPG techniques to avoid all bus contention and float states in every vector it generates. The resulting test pattern has an IDDQ strobe for every ATPG test cycle. In other words, the output is an IDDQ-only test pattern.

After the test pattern has been generated, you can use a Verilog/PowerFault simulation to verify the test pattern for quiescence at each strobe. The simulation does not need to perform strobe selection or fault coverage analysis because these tasks are handled by TetraMAX ATPG.

Having TetraMAX ATPG generate the IDDQ test patterns is a very efficient method. It works best when the design uses models that are entirely structural. When the design models transistors, capacitive discharge, or behavioral elements in either the netlist or library, the ATPG might be either optimistic or pessimistic because it does not simulate the mixed-level data and signal information in the same way as the Verilog simulation module. Consider this behavior when you select your IDDQ test flow.

IDDQ Strobe Selection From an Existing Pattern Set

In the IDDQ testing flow shown in Figure 1, PowerFault selects a near-optimum set of strobe points from an existing pattern set. The existing pattern can be a conventional stuck-at ATPG pattern or a functional test pattern. The output of this flow is the original test pattern with IDDQ strobe points identified at the appropriate times for maximum fault coverage.

In order for valid IDDQ strobe times to exist, the design must be quiescent enough of the time so that an adequate number of potential strobe points exist. You need to avoid conditions that could cause current to flow, such as floating buses or bus contention.

The specification of faults targeted for IDDQ testing is called fault seeding. There are a variety of ways to perform fault seeding, depending on your IDDQ testing goals. For example, to complement stuck-at ATPG testing done by TetraMAX ATPG, you can initially target faults that could not be tested by TetraMAX ATPG, such as those found to be undetectable, ATPG untestable, not detected, or possibly detected. For general IDDQ fault testing, you can seed faults automatically from the design description, or you can provide a fault list generated by TetraMAX ATPG or another tool.

The Verilog/PowerFault simulator determines the quiescent strobe times in the test pattern (called the qualified strobe times) and determines which faults are detected by each strobe. Then the IDDQ Profiler finds a set of strobe points to provide maximum fault coverage for a given number of strobes.

You can optionally run the IDDQ Profiler in interactive mode, which lets you select different combinations of IDDQ strobes and examine the resulting fault coverage for each combination. This mode lets you navigate through the hierarchy of the design and examine the fault coverage at different levels and in different sections of the design.

Licensing

The Test-IDDQ license is required to perform Verilog/PowerFault simulation. This license is automatically checked out when needed, and is checked back in when the tool stops running.

By default, the lack of an available Test-IDDQ license causes an error when the tool attempts to check out a license. You can have PowerFault wait until a license becomes available instead, which lets you queue up multiple PowerFault simulation processes and have each process automatically wait until a license becomes available.

PowerFault supports license queuing, which allows the tool to wait for licenses that are temporarily unavailable. To enable this feature, you must set the `SNPSLMD_QUEUE` environment variable to a non-empty arbitrary value (“1”, “TRUE”, “ON”, “SET”, etc.) before invoking PowerFault:

```
unix> setenv SNPSLMD_QUEUE 1
```

Existing Powerfault behavior with `SSI_WAIT_LICENSE` will continue to be supported for backward functionality of existing customer scripts.

```
% setenv SSI_WAIT_LICENSE
```

Note: If the license does not exist or was not installed properly, then the Verilog/PowerFault simulation will hang indefinitely without any warning or error message.

7

Verilog Simulation with PowerFault

PowerFault simulation technology operates as a standard programmable language interface (PLI) task that you add to your Verilog simulator. You can use PowerFault to find the best IDDQ strobe points for maximum fault coverage, to generate IDDQ fault coverage reports, to verify quiescence at strobe points, and to analyze and debug nonquiescent states.

The following sections describe Verilog simulation with PowerFault:

Preparing Simulators for PowerFault IDDQ

PowerFault includes two primary parts:

- A set of PLI tasks you add to the Verilog simulator
- The IDDQ Profiler, a program that reads the IDDQ database generated by the PowerFault's IDDQ-specific PLI tasks

Before you can use PowerFault, you need to link the PLI tasks into your Verilog simulator. The procedure for doing this depends on the type of Verilog simulator you are using and the platform you are running. The following sections provide instructions to support the following Verilog simulators (platform differences are identified with the simulator):

- Synopsys VCS
- Cadence NC-Verilog®
- Cadence Verilog-XL®
- Model Technology ModelSim®

These instructions assume basic installation contexts for the simulators. If your installation differs, you will need to make changes to the commands presented here. For troubleshooting problems, refer to the vendor-specific documentation on integrating PLI tasks. Information about integrating additional PLI tasks is not presented here.

```
setenv SYNOPSYS root_directory
set path=($SYNOPSYS/bin $path)
```

For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see "Specifying the Location for TetraMAX Installation."

Then, to simplify the procedures, set the environment variable `$IDDQ_HOME` to point to where you installed Powerfault IDDQ. For example, in a typical Synopsys installation using `csh`, the command is:

```
setenv IDDQ_HOME $SYNOPSYS/iddq/
```

Note the following:

- sparc64 and hp64 should be used only in specific 64-bit contexts.
- PowerFault features dynamic resolution of its PLI tasks. This means that one time a simulation executable has been built with the PowerFault constructs present (following the guidelines here), this executable does *not* need to be rebuilt if you change to a different version of PowerFault. Changing the environment variable `$IDDQ_HOME` to the desired version will load the runtime behavior of that version of PowerFault dynamically into this simulation run.

Using PowerFault IDDQ With Synopsys VCS

To generate the VCS simulation executable with PowerFault IDDQ, invoke VCS with the following arguments:

- Command-line argument `+acc+2`
- When running zero-delay simulations, you must use the `+delay_mode_zero` and

`+tetramax` arguments.

- Command-line argument `-P $IDDQ_HOME/lib/iddq_vcs.tab` to specify the PLI table (or merge this PLI table with other tables you might already be using), a reference to `$IDDQ_HOME/lib/libiddq_vcs.a`. Note: do not used the `-P` argument with any non-PLI Verilog testbenches.

In addition, you must specify:

- Your design modules
- Any other command-line options necessary to execute your simulation

If your simulation environment uses PLIs from multiple sources, you might need to combine the tab files from each PLI, along with the file `$IDDQ_HOME/lib/iddq_vcs.tab` for PowerFault operation, into a single tab file. See the VCS documentation for information about tab files.

The following command will enable the ssi_iddq task to be invoked from the Verilog source information in model.v:

```
vcs model.v +acc+2 -P $IDDQ_HOME/lib/iddq_vcs.tab \
$IDDQ_HOME/lib/libiddq_vcs.a
```

For VCS 64-bit operation, if you specify the `-full64` option for VCS 64-bit contexts, you must also set `$IDDQ_HOME` to the appropriate 64-bit build for that platform: either `sparc64` for Solaris environments or `hp64` for HP-UX environments. If you do not specify the `-full64` option, then `sparcOS5` or `hp32` should be specified. Since the `-comp64` option affects compilation only, `$IDDQ_HOME` should reference `sparcOS5` or `hp32` software versions as well.

For difficulties that you or your CAD group cannot handle, contact Synopsys at:

Web: https://solvnet.synopsys.com Email: support_center@synopsys.com Phone: 1-800-245-8005 (inside the continental United States)

Additional phone numbers and contacts can be found at:
http://www.synopsys.com/Support/GlobalSupportCenters/Pages

For additional VCS support, email vcs_support@synopsys.com.

Using PowerFault IDDQ With Cadence NC-Verilog

The following sections describe how to prepare for and run a PowerFault NC-Verilog simulation:

Setup

You can use the Powerfault library with NC-Verilog in many ways. The following example describes two such flows. For both flows, set these NC-Verilog-specific environment variables:

```
setenv CDS_INST_DIR <path_to_Cadence_install_directory>
setenv INSTALL_DIR  $CDS_INST_DIR
setenv ARCH <platform>  //sun4v for solaris, lnx86 for linux.
```

```
setenv LM_LICENSE_FILE <>
```

32-bit Setup

```
setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools:${LD_LIBRARY_PATH} //
32-bit
set path=($CDS_INST_DIR/tools/bin $path) // 32-bit
```

64-bit Setup

Note: Use `+nc64` option when invoking

```
setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools/64bit:${LD_LIBRARY_
PATH} //
64-bit
set path=($CDS_INST_DIR/tools/bin/64bit $path) // 64-bit
```

Note: For the 64-bit environments use the *cds_pic.a libraries

Creating the Static Executable

The following steps describe how to create the static executable:

1. Create a directory to build the ncelab and ncsim variables and navigate to this directory. Create an environment variable to this path to access it quickly.

   ```
   mkdir nc
   cd nc
   setenv LOCAL_NC "/<this_directory_path>"
   ```

   If PowerFault is the only PLI being linked into the Verilog run, then go to step 2. If additional PLIs are being added to your Verilog environment, then go to step 3.

2. Run two build operations using your Makefile.nc

   ```
   make ncelab $IDDQ_HOME/lib/sample_Makefile.nc
   make ncsim $IDDQ_HOME/lib/sample_Makefile.nc
   ```

   Go to step 6.

3. Copy the PLI task and the sample makefile into the nc directory. The makefile contains the pathname of the PowerFault object file `$IDDQ_HOME/lib/libiddq_cds.a`.

   ```
   cp $IDDQ_HOME/lib/veriuser_sample_forNC.c .
   cp $IDDQ_HOME/lib/sample_Makefile.nc .
   ```

4. Edit the example files to define additional PLI tasks.
5. Run two build operations using your `Makefile.nc`

   ```
   make ncelab -f sample_Makefile.nc
   make ncsim -f sample_Makefile.nc
   ```

6. Ensure the directory you created is located in your path variable before the instances of these tools under the directory: `$CDS_INST_DIR`.

   ```
   set path=($LOCAL_NC $path)
   ```

Running Simulation

```
ncvlog <design data and related switches>
ncelab -access +rwc <related switches>
```

```
ncsim <testbench name and related switches>
```

Make sure that the executables `ncelab` and `ncsim` picked up in the previous steps are the ones created in `$LOCAL_NC` directory, not the ones in the cadence installation path.

You can also use the single-step `ncVerilog` command as follows:

```
ncVerilog +access+rwc +ncelabexe+$LOCAL_NC/ncelab
+ncsimexe+$LOCAL_NC/ncsim <design data and other switches>
```

Note: If using 64-bit binaries, use the "`+nc64`" option with the ncVerilog script

Creating a Dynamic Library

This section describes a flow to create a dynamic library libpli.so and update the path, `LD_LIBRARY_PATH` to include the path to this library. In this flow, TetraMAX ATPG resolves PLI functional calls during simulation. There are two ways to build the dynamic library: either use vconfig, as in the first flow below, or use the sample_Makefile.nc, with the target being libpli.so.

1. Create a directory in which to build the libpli.so library and navigate to this directory. Set an environment variable to this location to access it quickly.

   ```
   mkdir nc
   cd nc
   setenv LIB_DIR "/<this_directory_path>"
   ```

2. Copy the PLI task file into the directory.

   ```
   cp $IDDQ_HOME/lib/veriuser_sample_forNC.c .
   ```

3. Edit the sample files to define additional PLI tasks.
4. Use the vconfig utility and generate the script fo create the libpli.so library. You can also use the cr_vlog template file shown at the end of this step.
   - Name the output script `cr_vlog.`
   - Select `Dynamic PLI libraries only`
   - Select `build libpli`
   - Ensure that you include the user template file veriuser.c in the link statement.

     This is the `$IDDQ_HOME/lib/veriuser_sample_forNC.c` file that you copied to the $LIB_DIR directory.

     Link the Powerfault object file from the pathname, $IDDQ_HOME/lib/libiddq_cds.a

     The vconfig command displays the following message after it completes:
   - `*** SUCCESSFUL COMPLETION OF VCONFIG`
     `****** EXECUTE THE SCRIPT: cr_vlog TO BUILD: Dynamic PLI library.`
   - Add another linking path:`$IDDQ_HOME/lib` to the first compile command in the cr_vlog script.

     The cr_vlog script is as follows:

     ```
     cc -KPIC -c ./veriuser_sample_forNC.c -I$CDS_INST_
     DIR/tools/Verilog/include -I$IDDQ_HOME/lib
     ```

```
ld -G veriuser_sample_forNC.o $IDDQ_HOME/lib/libiddq_cds.a
 -o libpli.so
```

- Change the cr_vlog script to correspond the architecure of the machine on which it runs.
- To compile on a 64-bit machine, use the `-xarch=v9` value with the cc command.
- For Linux, use `-fPIC` instead of `-KPIC`. Also, you might need to replace ld with gcc or use `-lc` with `ld` on Linux.

5. Run the cr_vlog script to create libpli.so library. Ensure the directory $LIB_DIR you create is in the path, LD_LIBRARY_PATH.

```
setenv LD_LIBRARY_PATH ${LIB_DIR}:${ LD_LIBRARY_PATH}
```

Note: You must edit the generated cr_vlog script to add a reference to 64-bit environment on the veriuser.c compile (add -xarch=v9), and the -64 option to the `ld` command.

Running Simulation

```
ncvlog <design data and related switches>
ncelab -access +rwc <related switches>
ncsim <testbench name and related switches>
Equivalently, single-step ncVerilog command can also be used
as
follows.
ncVerilog  +access+rwc <design data and other switches>
```

Using PowerFault IDDQ With Cadence Verilog-XL

The following sections describe how to setup and run a PowerFault Cadence Verilog-XL simulation:

Setup

To access user-defined PLI tasks at runtime, create a link between the tasks and a Verilog-XL executable using the vconfig command. The vconfig command displays a series of prompts and creates another script called cr_vlog, which builds and links the ssi_iddq task into the Verilog executable.

This is a standard procedure for many Verilog-XL users. You only need to do it only one time for a version of Verilog-XL, and it should take about 10 minutes. Cadence uses this method to support users that need PLI functionality.

After you create a link between the PowerFault IDDQ constructs and the Verilog-XL executable, you can use them each time you run the executable. The PowerFault IDDQ functions do not add overhead to a simulation run if the run does not use these functions. The functions are not loaded unless you use PowerFault IDDQ PLIs in the Verilog source files.

You do not need any additional runtime options for a Verilog-XL simulation to use PowerFault IDDQ after you create a link to it.

To create a link between the tasks and a Verilog-XL executable, do the following:

1. Set the Verilog-XL specific environment variables:

```
setenv CDS_INST_DIR <path_to_Cadence_install_directory>
setenv INSTALL_DIR  $CDS_INST_DIR
setenv TARGETLIB .
setenv ARCH <platform>
setenv LM_LICENSE_FILE <>
setenv LD_LIBRARY_PATH $CDS_INST_DIR/
tools:${LD_LIBRARY_PATH}
set path=($CDS_INST_DIR/tools/bin $CDS_INST_DIR/tools/
bin $path)
```

2. Create a directory to hold the Verilog executable and navigate into it. Set an environment variable to this location to access it quickly.

```
mkdir vlog
cd vlog
setenv LOCAL_XL "/<this_directory_path>"
```

3. Copy the sample veriuser.c file into this directory:

```
cp $IDDQ_HOME/lib/veriuser_sample_forNC.c .
```

4. Edit the veriuser_sample_forNC.c file to define additional PLI tasks.
5. Run the vconfig command and create the cr_vlog script to link the new Verilog executable. The vconfig commands displays the following prompts. Respond to each prompt as appropriate; for example,

   Name the output script `cr_vlog`. Choose a `Stand Alone` target. Choose a `Static with User PLI Application` link. Name the Verilog-XL target `Verilog`.

   You can answer `no` to other options.

   Create a link between your user routines and Verilog-XL. The cr_vlog script includes a section to compile your routines and include them in the link statement.

   Ensure that you include the user template file veriuser.c in the link statement. This is the $IDDQ_HOME/lib/veriuser_sample_forNC.c file that you copied to the vlog directory.

   Ensure that you include the user template file vpi_user.c in the link statement. The pathname of this file is `$CDS_INST_DIR/Verilog/src/ vpi_user.c`. The vconfig command prompts you to accept the correct path.

   Create a link to Powerfault object file as well. The pathname of this file is $IDDQ_ HOME/lib/libiddq_cds.a

   After it completes, the vconfig command completes:

```
*** SUCCESSFUL COMPLETION OF VCONFIG ***
*** EXECUTE THE SCRIPT: cr_vlog TO BUILD: Stand Alone

Verilog-XL
```

6. Add to the option `-I/$IDDQ_HOME/lib` to the first compile command in the cr_vlog file, which compiles the sample veriuser.c file.
7. Do the following before running the generated cr_vlog script:

   Note for HP-UX 9.0 and 10.2 users:

   The cr_vlog script must use the -Wl and -E compile options. Change the cc command from `cc -o Verilog` to `cc -Wl,-E -o Verilog`.

   If you are using either HPUX 9.0 or Verilog-XL version 2.X, you must also create a link to the -ldld library. The last lines of the cr_vlog script must be similat to:

   ```
   +O3 -lm -lBSD -lcl -N -ldld
   ```

   If you use a link editor (such as ld) instead of the cc command to create the final link, make sure you pass the -W1 and -E options as shown previously.

   Note for Solaris users:

   You must create a link between the cr_vlog script and the -lsocket, -lnsl, and -lintl libraries.

   Check the last few lines of script and ensure these libraries are included.
8. Run the cr_vlog script. The script creates a link between the ssi_iddq task and the new Verilog executable (Verilog) in the current directory.
9. Verify that the Verilog directory appears in your path variable before other references to an executable with the same name, or reference this executable directly when running Verilog. For example,

   ```
   set path=(./vlog $path)
   ```

Running Simulation

Before running simulation, ensure that the executable Verilog used to run simulation is the executable that you created in the $LOCAL_XL directory and not the executable in the Cadence installation path.

To run simulation, use the following command:

```
Verilog +access+rwc <design data and related switches>
```

Running Verilogxl

There is no command line example due to the interpreted nature of this simulation. You do not need any runtime options to enable the PLI tasks after you create a link between them and the Verilog-XL executable.

Using PowerFault IDDQ With Model Technology ModelSim

User-defined PLI tasks must be compiled and linked in ModelSim to create a shared library that is dynamically loaded by its Verilog simulator, vsim.

The following steps show you how to compile and link a ModelSim shared library:

1. Create a directory where you want to build a shared library and navigate to it; for example,

   ```
   mkdir MTI
   cd MTI
   ```

2. Copy the PLI task into this directory as "veriuser.c"; for example,

   ```
   cp $IDDQ_HOME/lib/veriuser_sample.c veriuser.c
   ```

3. Edit veriuser.c to define any additional PLI tasks.
4. Compile and link veriuser.c to create a shared library named "libpli.so"; for example,

   ```
   cc -O -KPIC -c -o ./veriuser.o \
      -I<install_dir_path>/modeltech/include \
      -I$IDDQ_HOME/lib -DaccVersionLatest ./veriuser.c
   ld -G -o libpli.so veriuser.o \
      $IDDQ_HOME/lib/libiddq_cds.a -lc
   ```

   Note:

   > For compiling on a 64-bit machine, use `-xarch=v9` with cc. For Linux, use `-fPIC` instead of `-KPIC`.

5. Identify the shared library to be loaded by vsim during simulation. You can do this in one of three ways:
   - Set the environment variable PLIOBJS to the path of the shared library; for example,

     ```
     setenv PLIOBJS <path_to_the_MTI_directory>/libpli.so
     vlog ...
     vsim ...
     ```

   - Pass the shared library to vsim in its -pli argument; for example,

     ```
     vlog ...
     vsim -pli <path_to_the_MTI_directory>/libpli.so ...
     ```

   - Assign the path to the shared library to the Veriuser variable in the "modelsim.ini" file, and set the environment variable `MODELSIM` to the path of the modelsim.ini file; for example,

     In the modelsim.ini file:

     ```
     Veriuser = <path_to_the_MTI_directory>/libpli.so
     ```

     On the command line:

```
setenv MODELSIM <path_to_modelsim.ini_file/modelsim.ini
vlog ...
vsim ...
```

PowerFault PLI Tasks

The following sections describe the various PowerFault PLI tasks:

- Getting Started
- PLI Task Command Summary Table
- PLI Task Command Reference

Getting Started

The first step in using PowerFault technology is to run a Verilog simulation using your normal testbench, combined with the PowerFault tasks to seed faults and evaluate potential IDDQ strobes.

A task called `ssi_iddq` executes PowerFault commands in the Verilog file that configures the Verilog simulation for IDDQ analysis. Some of the commands are mandatory and some are optional. The commands must at least specify the device under test, seed the faults, and apply IDDQ strobes.

For example, preparation for IDDQ testing can be as simple as adding a module similar to the following to your Verilog simulation:

```
module IDDQTEST();
  parameter CLOCK_PERIOD = 10000;
  initial begin
    $ssi_iddq( "dut tbench.M88" );
    $ssi_iddq( "seed SA tbench.M88" );
  end
  always begin
     fork
     # CLOCK_PERIOD;
     # (CLOCK_PERIOD -1) $ssi_iddq( "strobe_try" );
     join
   end
endmodule
```

This example contains three PowerFault commands. The first one specifies the device under test (DUT) to be `tbench.M88`. The second one seeds the entire device with stuck-at (SA) faults. Inside the `always` block, the third one invokes the `strobe_try` command to evaluate the device for IDDQ strobing at one time unit before the end of each cycle.

The order of commands in the Verilog file is important because the PLI tasks must be performed in the following order:

1. Specify the DUT module or modules (mandatory).
2. Specify other simulation setup parameters (optional).
3. Specify disallowed leaky states (optional).
4. Specify allowed leaky states (optional).
5. Specify fault seed exclusions (optional).
6. Specify fault models (optional).
7. Specify fault seeds (mandatory).
8. Run testbench and specify strobe timing (mandatory).

PLI Task Command Summary Table

Table 1 provides a quick summary of the PowerFault commands that you can use in Verilog files to perform PLI tasks. For detailed information on each command, see the next section, "PLI Task Command Reference." If you are viewing this document in online form, you can click the page number reference in the table to jump to the detailed description of the command.

Table 1 PLI Task Command Summary

Simulation Setup Commands	
`dut`	Specifies the DUT modules
`output`	Names the IDDQ database
`ignore`	Specifies black box nets and modules
`statedep_float`	Specifies the primitives that can block floating nodes
`io`	Specifies DUT ports
`measure`	Specifies the rail for IDDQ measurement
`verb`	Turns verbose mode on or off (off by default)
Leaky State Commands	
`allow`	Allows user-specified leaky states
`disable SepRail`	Forces all top-level pullups and pulldowns in contention to be identified as leaky, see
`disallow`	Disallows user-specified leaky states

Fault Seeding Commands

`seed SA`	Seeds stuck-at faults automatically
`seed B`	Seeds bridging faults automatically
`scope`	Sets the scope for faults seeded by read commands
`read_bridges`	Seeds bridging faults from a file
`read_tmax`	Seeds faults from a TetraMAX fault list
`read_verifault`	Seeds faults from a Verifault fault list
`read_zycad`	Seeds faults from a Zycad fault origin file

Fault Seed Exclusion Command

`exclude`	Excludes module instances from fault seeding

Fault Model Commands

`model SA`	Configures operation of the seed SA command
`model B`	Configures operation of the seed B command

Strobe Commands

`strobe_try`	Performs an IDDQ strobe evaluation if the chip is quiet; see
`strobe_force`	Forces an IDDQ strobe evaluation
`strobe_limit`	Limits the number of IDDQ strobe evaluations
`cycle`	Sets the internal cycle count

Circuit Examination Commands

`status`	Prints a report on leaky nets
`summary`	Prints a nodal analysis summary

PLI Task Command Reference

The following sections describe the syntax and functions of the PowerFault commands:

Note: Each command description includes the Backus-Naur form (BNF) syntax and a description of the command behavior.

Conventions

The following conventions apply to the PLI task command descriptions:

Special-Purpose Characters

Several special-purpose characters are used in the command syntax descriptions, as described in Table 2.

Table 2 Special Characters in Command Syntax

Character	Purpose
+	Plus-sign suffix indicates repetition of one or more
*	Asterisk suffix indicates repetition of zero or more
[ ]	Square brackets enclose an optional element
( )	Parentheses indicate grouping
|	Vertical bar separates alternative choices

When you use Verilog escaped identifiers in a command, each escape character must itself be escaped. For example, to use the name tbench.dut.\IO(23) with the `allow` command, use the following syntax:

```
$ssi_iddq( "allow float tbench.dut.\\IO(23)" );
```

Module Instances and Entity Models

A number of commands accept either *module-instance* or *entity-model* as a parameter. A *module-instance* is a full path name of an instantiated module, such as the module name `tbench.au.ctrl`?. An *entity-model?* is the definition name (not instance name) of a module. For example, `tbench.au.ctrl` might be one instance of the `IOCTRL` entity model. When you specify an entity model in a command, it applies to all instances of that model.

Cell Instances

The commands for fault seeding refer to Verilog cells. A cell instance is a module instance that has either of these characteristics:

- The module definition appears between the compiler directives `` `celldefine `` and `` `endcelldefine ``??.
- The module definition is in a model library, and the `+nolibcell` option has not been used. A library is a collection of module definitions contained in a file or directory that are read by library invocation options (such as the `-y` option provided by most Verilog simulators).

If you use the `+nolibcell` option when you invoke the Verilog simulator, only modules meeting the first condition above are considered cells.

By default, PowerFault treats cells as fault primitives. It seeds faults only at cell boundaries, not inside of cells. However, some design environments generate netlists that mark very large blocks as cells. To make PowerFault seed inside those cells, use the `model SA seed_inside_cells` command or the `model B seed_inside_cells` command.

Port and Terminal References

The commands for allowing and disallowing leaky states refer to *connection references*. A connection reference describes a port of a module or a terminal of a primitive. You can refer to a port by its name. You can also refer to ports and terminals by their index numbers, with 0 indicating the first port or terminal. For example, `port.0` refers to the first port of a module; *`term`*`.0` refers to the first terminal (the output terminal) of a primitive.

Simulation Setup Commands

The following simulation setup commands set up the general operating parameters for the PowerFault simulation, such as the name of the device under test (DUT), the name of the generated IDDQ database, and the names of the DUT ports:

- dut
- output
- ignore
- io
- statedep_float
- measure
- verb

dut

dut *module-instance+*

This command is required and must be the first ssi_iddq-task command executed. It specifies which instances represent the device under test. The arguments are the full path names of one or more module instances.

Here are some examples of valid dut commands:

```
$ssi_iddq( "dut tbench.core" );

$ssi_iddq( "dut tbench.slave tbench.master" );
```

output

output [*mode*] [*label*] *database-name*
mode ::= (create|append|replace=*testbench-number*)
label ::= label=*string*

This command specifies the name of the generated IDDQ database. The database is a directory that PowerFault uses to store simulation results. During the Verilog simulation, the ssi_iddq-task commands fill the database with information for the IDDQ Profiler. You run the IDDQ Profiler after the Verilog simulation to select strobes and generate IDDQ reports.

The following command makes the ssi_iddq task create a database named /cad/sim/M88/iddq.db1?:

```
$ssi_iddq( "output /cad/sim/M88/iddq.db1" );
```

The default *mode* is create?, which creates the database if it does not already exist. If the database already exists, its entire contents are cleared before the new simulation results are stored.

When you use the append mode, the simulation results are appended to the specified database. The append mode allows the simulation results from multiple testbenches for a circuit to be saved into one database, as described in the “Combining Multiple Verilog Simulations” section.

The replace mode replaces one specified testbench result in a multiple set of results saved using the append mode. For the testbench number, specify 1 to overwrite the first results saved, 2 to overwrite the second results saved, and so on.

The *label* option assigns a string label to represent the current testbench. This is useful when the database is used to store results from multiple testbenches. When the IDDQ Profiler selects strobes, it uses the label to identify the testbench from which the strobe was selected.

The append mode is useful for a circuit that has multiple testbenches. It is much more efficient to append the results from multiple testbenches to one database, rather than create a separate database for each testbench. For details, see “Combining Multiple Verilog Simulations”.

Do not use the append mode with multiple concurrent simulations. For example, you cannot start four Verilog simulations at the same time and try to have each one append to the same database. If you have multiple testbenches for a circuit, you need to run them serially.

ignore

ignore net *module-or-prim-instanceconn-ref*
ignore net *entity-modelconn-ref*
ignore (all|core|ports) *module-or-prim-instance*

```
ignore (all|core|ports) entity-model
conn-ref ::= port-name | port.port-index |
   term.term-index
port-name ::= scalar-port-name | vector-port-name
   [port-index]
```

The `ignore` command describes which nodes in your circuit should be ignored for IDDQ testing. Ignored nodes are excluded from analysis, fault seeding, and status reports. The same effect can be produced by using the `exclude`?, `allow fight`?, and `allow float` commands together, but using the `ignore` command is more efficient. This command overrides all built-in checkers and all custom checkers defined with the `disallow` command.

In the first two forms of the command, *`conn-ref`* describes which node to ignore. For example, the following command causes the node connected to the port named `INTR` in the module `tbench.core.busarb` to be ignored:

```
$ssi_iddq( "ignore net tbench.core.busarb INTR" );
```

The following command causes the node connected to the fifth port of `tbench.core.busarb` to be ignored:

```
$ssi_iddq( "ignore net tbench.core.busarb port.5" );
```

The following command causes the nodes connected to the `INTR` port of all instances of the `ARB` module to be ignored:

```
$ssi_iddq( "ignore net ARB INTR" );
```

In the last two forms of the command, the `(all|core|ports)` option describes how the command is applied to nodes of a particular module or primitive. For example, the following command causes all nodes connected to ports of the `tbench.core.pll` module to be ignored:

```
$ssi_iddq( "ignore ports tbench.core.pll" );
```

The following command causes all nodes inside `tbench.core.pll` to be ignored:

```
$ssi_iddq( "ignore core tbench.core.pll" );
```

The following command causes all nodes connected to ports and all nodes inside `tbench.core.pll` to be ignored:

```
$ssi_iddq( "ignore all tbench.core.pll" );
```

io

```
io net-instance+
```

This command lists any primary inputs and outputs (I/O pads) that are not connected to ports of the DUT modules. PowerFault assumes that each port of a DUT module is connected to an I/O pad. If your chip has I/O pads that are not connected to a port of a DUT module, you can optionally specify them with this command. Doing so might allow PowerFault to find better strobe points.

statedep_float

```
statedep_float #-and-ins#-nand-ins#-nor-ins#-or-ins
```

This command specifies the types of primitives that can block floating nodes. The default setting is:

```
$ssi_iddq( "statedep_float 3 3 2 0" );
```

By default, AND and NAND gates with up to three inputs and NOR gates with up to two inputs can block floating nodes. These primitives are commonly used to “gate” a three-state bus so that it does not cause a leakage current. For more information on this topic, see “State-Dependent Floating Nodes”.

If your foundry implements two-input OR gates so that they can block floating nodes, use this command:

```
$ssi_iddq( "statedep_float 3 3 2 2" );measure (0|1)
```

measure

```
measure (0|1)
```

This command specifies which power rail to use for IDDQ measurement. By default, PowerFault assumes that IDDQ measurements are made on the VDD (power) rail; this is the most common test method. If your automated test equipment (ATE) is configured to measure ISSQ, the current flowing through the VSS (ground) rail, use the following command:

```
$ssi_iddq( "measure 0" );
```

verb

```
verb (on|off)
```

This command turns verbose mode on and off. In verbose mode, the `ssi_iddq` task echoes every command before execution, and it also prints the result (qualified or unqualified) of every `strobe_try` command. By default, verbose mode is initially off. To turn on verbose mode, use this command:

```
$ssi_iddq( "verb on" );
```

Leaky State Commands

PowerFault has powerful algorithms for determining quiescence. By default, it recognizes two types of leaky states: floating inputs (“float”) and drive contention (“fight”). It is also configurable; the `allow`, `disable SepRail`, and `disallow` commands let you modify the algorithms for determining quiescence.

The following sections describe the leaky state commands:

- allow
- disable SepRail
- disallow

allow

The `allow` command specifies the types of leaky states that are to be ignored. The `disallow` command defines new leaky states that would normally be unrecognized, such as leaky behavioral and external models (for more information, see “Behavioral and External Models”). The `allow` command tells PowerFault how to ignore leaky states it normally recognizes; the `disallow` command tells PowerFault how to identify leaky states it does not normally recognize.

There are several different forms of this command. These are the forms that apply to specified nets, instances, or entity models:

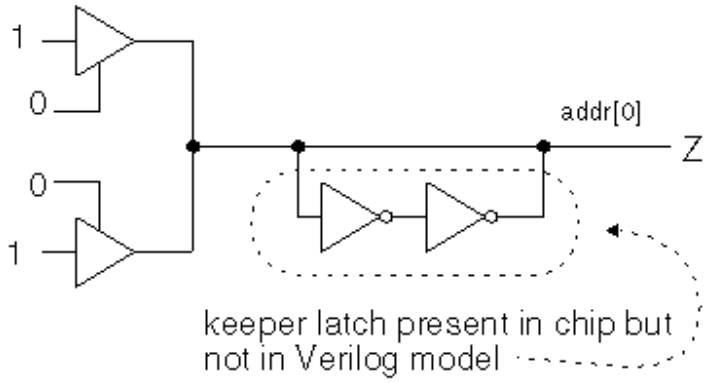
```
allow (float|fight) net-instance
allow (float|fight) module-or-prim-instance [conn-ref]
allow (float|fight) entity-model [conn-ref]
```

```
conn-ref ::= port-name | port.port-index | term.term-index
port-name ::= scalar-port-name|
vector-port-name[port-index]
```

These commands specify which leaky states in the design to allow (ignored by PowerFault). You can use them to have PowerFault ignore leaky states that are not present in the real chip.

Incomplete Verilog models can cause misleading leaky states, which PowerFault should ignore. For example, consider a chip that has an internal three-state bus with a keeper latch like the one shown in Figure 1.

Figure 1 Three-State Bus With Keeper Latch



When the bus is fabricated on the chip, the keeper latch prevents the bus from floating. However, the Verilog model for the bus does not include the keeper latch. As a result, when the bus floats (has a Z value) during the Verilog simulation, PowerFault considers it a possible cause of high current and disqualifies any strobe try at that time.

To tell PowerFault that the bus addr[0] does not cause high current when it floats during the simulation, use a command like the following:

```
$ssi_iddq( "allow float tbench.iob.addr[0]" );
```

When you use a module (primitive) instance name, the `allow` command applies to all nets declared inside the instance, including those inside of submodules, and to all nets attached to the instance's ports (terminals). For example, to allow nets to float inside of and connected to `tbench.au.ctlr`?, use this command:

```
$ssi_iddq( "allow float tbench.au.ctlr" );
```

If you use an entity-model name, the command applies to every instance of that entity model. For example, to allow all nets to float inside of and connected to the instances of the `IOCTL` module, use the following command:

```
$ssi_iddq( "allow float IOCTL" );
```

By using the optional connection reference, you can make the command apply to a specific port or terminal. For example, if `IOCTL` has a port named `out2?`, then the following command allows the nets attached to the `out2` port of all `IOCTL` instances to float:

```
$ssi_iddq( "allow float IOCTL out2" );
```

The following command allows the nets attached to the output terminal of all `bufif0` instances to float:

```
$ssi_iddq( "allow float bufif0 term.0" );
```

To globally allow leaky states, use this command:

```
allow (all|poss) (fight|float)
```

This form of the `allow` command turns on global options that apply to every net. The `all` option makes PowerFault ignore all true and all possibly leaky states. The `poss` option makes PowerFault ignore just the possibly leaky states; true leaky states are still disallowed. For a description of true and possibly floating nodes, see “Floating Nodes and Drive Contention”.

This form of the `allow` command is most useful for verifying strobe timing and debugging test vectors. For example, if you want to find vectors that definitely have drive contention (so that you can measure it on your ATE), use these commands:

```
$ssi_iddq( "allow poss fight" );
$ssi_iddq( "allow all float" );
```

In this case, only vectors with true drive contention are disqualified because all floating nodes and all nodes with possible drive contention are ignored.

Here is the form of the command for allowing leaky states inside cells:

```
allow cell (fight|float)
```

This form of the `allow` command applies to every net that is internal to a cell. Nets connected to cell ports and nets outside of cells are not affected. The `fight` option makes PowerFault ignore all true and possible drive contention on nets inside of cells. The `float` option makes PowerFault ignore all true and possibly floating nets inside of cells. For a description of true and possibly floating nodes, see “Floating Nodes and Drive Contention”.

This form of the `allow` command is most useful when your cell libraries have many internal nets that are erroneously flagged as floating or contending. This most commonly happens when cells use dummy local nets (nets not present in the real chip) for the purpose of timing checks. If you know that all the nets internal to your cells are always quiescent, you can use these commands:

```
$ssi_iddq( "allow cell fight" );
$ssi_iddq( "allow cell float" );
```

disable SepRail

Current measurements, performed at test, are subject to the configuration of the test equipment when considering current contributions. Typically, many test environments use separate power supplies for the device signals (often referred to as "pin electronics") from the primary power supply for the device itself.

Because of these separate supplies, some leaky conditions might not contribute current that is measured from the device rails or primary power supply. In particular, out-of-state pullups or pulldowns on the IO of the device might not contribute to measured IDDQ current. Eliminating test vectors that do not contribute leaky current can reduce the overall effectiveness of a set of IDDQ tests. Remember, only pullups and pulldowns that are associated with the top-level signals of the design are considered here. Internally, all current-generating situations are considered.

By default, IddQTest will not identify all out-of-state pull conditions on top-level IO signals as leaky. Certain situations are allowable. In particular, internal pulls (pullups or pulldowns that are part of the internal device definition) that are pulling to the opposite state of the measured rail (for example, internal pulldowns for IddQ measurements) will not be identified as leaky. External pulls (pullups or pulldowns that are external to the device referenced with the `dut` command) that are pulling to the same state as the measured rail (for example, external pullups for IddQ measurements) will also not be identified as leaky.

To override this default behavior, and force *all* out-of-state conditions with pullups and pulldowns at the top level of the design to be identified as leaky, the option `disable SepRail` must be specified. This can be specified as:

```
$ssi_iddq( "disable SepRail" );
```

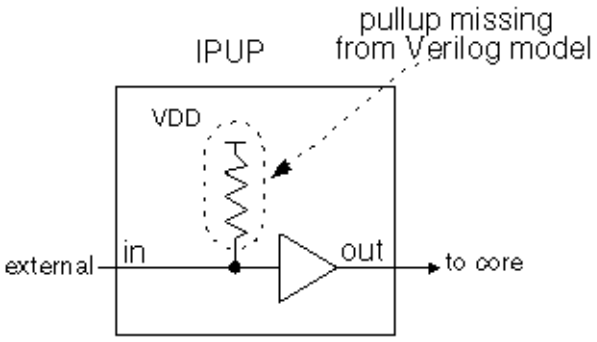
disallow

```
disallow module-or-prim-instanceleaky-condition
disallow entity-modelleaky-condition
leaky-condition ::= expr
expr ::= ( expr ) | expr && expr | expr || expr
| conn-ref == value | conn-ref != value
conn-ref ::= port-name | port.port-index | term.term-index
port-name ::= scalar-port-name |
vector-port-name[bit-index]
value ::= 0|1|Z|X
```

This command describes specific leaky states that would not otherwise be recognized. At every strobe try, PowerFault examines your entire netlist for leaky states. If your chip has leaky states that cannot be detected by analyzing the Verilog netlist, you might need to use the `disallow` command.

For example, consider the case where the input pads on your chip have pullups as shown in Figure 2, but these pullups are missing from your Verilog models.

Figure 2 Input Macro With Pullup



If `IPUP` is the entity model for your input pad and its input port is named `in`?, use the following command to tell PowerFault that the DUT is leaky when the input is 0:

```
$ssi_iddq( "disallow IPUP in == 0" );
```

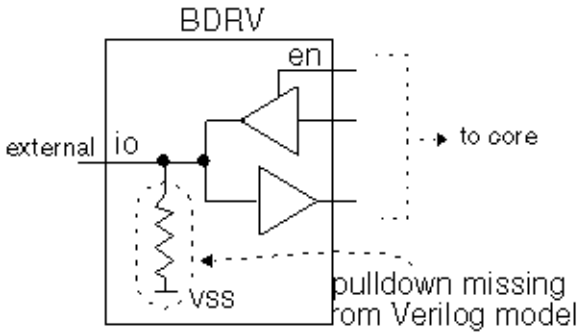
You can also refer to a port or terminal by its index number. Index numbers start at zero. For example, if port `in` is the second port in the `IPUP` port list, then the preceding command example is equivalent to the following command:

```
$ssi_iddq( "disallow IPUP port.1 == 0" );
```

The *`leaky-condition`* argument specifies an entity model or a particular instance that is nonquiescent. This condition is a Boolean expression describing the combination of port values or terminal values that make the chip leaky. If you specify an entity model, the condition applies to all instances of the entity model.

For example, assume the bidirectional pads on your chip have pulldowns as shown in Figure 3, but those pulldowns are missing from your Verilog model.

Figure 3 Bidirectional Macro With Pulldown



To tell PowerFault that `BDRV` is an entity model that is leaky when port `io` is high and port `en` is high, use this command:

```
$ssi_iddq( "disallow BDRV ( io == 1 ) && ( en == 1 )" );
```

The `==` and `!=` operators differ from their Verilog counterparts. The expression `(conn-ref == value)` is true only if the values match exactly. For example, if `io` is X, then the expression `(io == 1)` is *not* true.

The following form of the `disallow` command turns on global options, which apply to every net:

```
disallow (Xs|Zs|Caps)
```

Turning on these options makes PowerFault follow pessimistic rules for determining quiescence. By default, nets at three-state (`Z`), unknown (`?X?`), and capacitive (`Caps`) values are allowed as long as they do not cause leakage. In other words, a net can be at Z if it does not have any loads.

To make PowerFault compatible with less-sophisticated IDDQ tools that disallow every X or Z, use these commands:

```
$ssi_iddq( "disallow Xs" );
$ssi_iddq( "disallow Zs" );
```

Using these `disallow` commands, no Xs or Zs are allowed because a single X or Z implies nonquiescence and disqualifies an IDDQ strobe try. Because PowerFault analyzes the netlist in detail, if your chip is modeled structurally (the logic is implemented with Verilog user-defined primitives and ordinary primitives), you probably do not need to use this form of the `disallow` command. It is better to describe only the specific leaky states, so that more strobe times are allowed.

Fault Seeding Commands

At the beginning of the simulation, before using the `strobe_try` command to evaluate strobes for IDDQ testing, you need to tell PowerFault where to seed faults. For this purpose, you can use `seed` commands to seed faults automatically, or `read` commands to seed faults from an existing fault list.

The `seed` and `read` commands are cumulative. If you want to seed some faults automatically and seed some faults from a fault list, use both the `seed` and `read` commands.

The following sections describe the various seeding commands:

seed SA

`seed SA` *`module-instance+`*
`seed SA` *`net-instance+`*

This command seeds both stuck-at-0 and stuck-at-1 faults in each of the specified instances or nets. For module instances, PowerFault performs automatic hierarchical seeding of each module and all its lower-level modules. The placement of fault seeds (ports, terminals, and so on) is determined by the current fault model. For more information, see "Fault Model Commands".

Here are some examples of valid `seed SA` commands:

```
$ssi_iddq( "seed SA tbench.M88.IO tbench.M88.CORE" );

$ssi_iddq( "seed SA tbench.M88.IO.CO tbench.M88.IO.IRDY" );
```

seed B

`seed B` *`module-instance+`*
`seed B` *`net-instancenet-instance`*

This command automatically seeds bridging faults throughout the specified instances or between two specified nets. For module instances, PowerFault performs automatic hierarchical seeding of each module and all its lower-level modules. The placement of fault seeds (between ports, terminals, and so on) is determined by the current fault model. For more information, see "Fault Model Commands".

Here are some examples of valid `seed B` commands:

```
$ssi_iddq( "seed B tbench.M88.IO" );
$ssi_iddq( "seed B tbench.M88.IO.SHF0 tbench.M88.IO.SHF1" );
```

scope

`scope` *`module-instance`*

This command sets the scope for the faults seeded by subsequent `read_` type commands. By default, PowerFault expects full path names for all fault entries. Some ATPG environments generate fault entries that have incomplete path names (for example, without the testbench module name). For those environments, use the `scope` command to specify a prefix for all path names.

For example, the following four commands tell PowerFault to do the following: seed faults from files `tbench.core` and `tbench.io`?, consider all names in `U55.flist` to be relative to `tbench.core`?, and consider all names in `U24.flist` to be relative to `tbench.io`?:

```
$ssi_iddq( "scope tbench.core" );
$ssi_iddq( "read_tmax U55.flist" );
$ssi_iddq( "scope tbench.io" );
$ssi_iddq( "read_tmax U24.flist" );
```

read_bridges

```
read_bridges file-name
```

This command reads the names of net pairs from a file (one pair per line) and seeds a bridging fault between each listed pair. For example, a file containing the following two lines would seed bridging faults in the `tbench.M88` module between `TA` and `TB`?, and between `PA` and `PB`?:

```
tbench.M88.TA tbench.M88.TB
tbench.M88.PA tbench.M88.PB
```

read_tmax

```
read_tmax [strip] fault-classes* file-name
fault-classes ::=(DS|DI|AP|NP|UU|UO|UT|UB|UR|AN|NC|NO|-- )
```

This command reads fault entries from a TetraMAX fault list. By default, only fault entries in the AP, NP, NC, and NO classes are seeded. If you want to seed faults in other classes, use the *`fault-classes`* argument to specify the fault classes. For definitions of these fault classes, refer to the *TetraMAX ATPG User Guide*.

For example, the following command seeds faults in the `fa1` file that are in the following classes: possibly detected (AP, NP), undetectable (UU, UT, UB, UR), ATPG untestable (AN), and not detected (NC, NO):

```
$ssi_iddq( "read_tmax AP NP UU UT UB UR AN NC NO fa1" );
```

By default, PowerFault remembers all the comment lines and unseeded faults in the fault list, so that when it produces the final fault report, you can easily compare the report to the original fault list. If you do not want to save this information (it requires extra disk space), use the `strip` option:

```
$ssi_iddq( "read_tmax strip AP NP UU UT UB UR AN NC NO fa1" );
```

read_verifault

```
read_verifault [strip] status-types* file-name
status-types ::=(detected|potential|undetected|
   drop_task|drop_active|drop_looping|drop_detected |
   drop_potential|drop_pli|drop_hyper_active|
   drop_hyper_mem|untestable )
```

This command reads fault seeds from a Verifault-XL fault list. By default, only fault descriptors without status or with the status `undetected` or `potential` are seeded. If you want to seed faults with other status types, use the *`status-types`* argument to specify the status types.

For example, the following command seeds all faults with status `potential`, `undetected`, or `untestable` from the file `M88.flist`?:

```
$ssi_iddq( "read_verifault potential undetected untestable
```

```
   M88.flist" );
```

By default, PowerFault remembers all the comment lines and unseeded faults in the fault list, so that when it produces the final fault report, you can easily compare the report to the original fault list. If you do not want to save this information (it requires extra disk space), use the `strip` option:

```
$ssi_iddq( "read_verifault strip potential undetected
   untestable M88.flist" );
```

read_zycad

`read_zycad [strip]` *`fault-types* result-types* file-name`*

`fault-types``::= (i|o|n)`

`result-types``::= (C|D|H|I|M|N|O|P|U)`

This command reads fault seeds from a Zycad fault origin file. By default, only fault origins with the node type (`n`) and the undetected (`U`) or not run yet (`N`) or possibly (`P`) result are seeded. If you want to seed other fault types or results, use the *`fault-types`* and *`result-types`* arguments to specify them.

For example, the following command seeds all input and output faults with the impossible (`I`) and possibly (`P`) result from the file `M88.fog?`:

```
$ssi_iddq( "read_zycad i o I P M88.fog" );
```

exclude

`exclude` *`module-instance+`*
`exclude` *`primitive-instance+`*
`exclude` *`entity-model+`*

The `exclude` command excludes specified parts of the design from fault seeding. This command specifies instances and entities that are to be excluded from the fault seeding performed by the `seed?`, `read_tmax?`, `read_verifault?`, and `read_zycad` commands.

For example, to exclude all instances of the `BRAM16` entity from fault seeding, use the following command:

```
$ssi_iddq( "exclude BRAM16" );
```

To exclude individual instances, specify the full path name of each instance:

```
$ssi_iddq( "exclude tbench.M88.io tbench.M88.mem" );
```

The `exclude` command excludes only instances from seeding. It does not exclude them from being checked for leaky states. If you need to ignore a leaky state, use the `allow` command, described in “Leaky State Commands”.

Fault Model Commands

The `model` commands determine where the `seed` commands will place faults. Therefore, if you use a `model` command, you must execute it before the `seed` command. When you specify a module instance name in the `seed` command, the seeding algorithm performs a hierarchical traversal of the instance, seeding faults on the ports and terminals specified by the current fault model. By default, this traversal stops at cell boundaries.

The settings made with a `model` command are not cumulative. The current model is based only on the most recent `model` command. In other words, each `model` command overwrites the settings made by the previous `model` command.

The following sections describe the fault model commands:

- model SA
- model B

model SA

```
model SA directionsa-placement [seed_inside_cells]
direction ::= (port_IN|port_OUT|term_IN|term_OUT)+
sa-placement ::= (all_mods|leaf_mods|cell_mods|prims)+
```

This command specifies where the `seed SA` command seeds stuck-at faults. Table 3 summarizes the command options.

Table 3 Options for Stuck-At Fault Models

Direction Options	
`port_IN`	Enables stuck-at faults on input ports of chosen modules
`port_OUT`	Enables stuck-at faults on output ports of chosen modules
`term_IN`	Enables stuck-at faults on input terminals of primitives
`term_OUT`	Enables stuck-at faults on output terminals of primitives
Stuck-At Placement Options	
`all_mods`	Chooses all modules for port stuck-at faults
`leaf_mods`	Chooses leaf modules for port stuck-at faults
`cell_mods`	Chooses cell modules for port stuck-at faults
`prims`	Chooses primitives for terminal stuck-at faults
Seed Inside Cells Option	
`seed_inside_cells`	Enables fault seeding inside cells

The default stuck-at fault seeding behavior is equivalent to the following `model SA` command:

```
model SA port_IN port_OUT term_IN term_OUT
   leaf_mods cell_mods prims
```

With the default stuck-at fault model, faults are seeded on input and output ports of cell and leaf modules, and on input and output terminals of every primitive, but not inside cells. Primitives and

modules found inside of cells are ignored. A leaf module is a module that does not contain any instances of submodules.

If you want to seed inside cells, include the `seed_inside_cells` option. For example, these two lines seed stuck-at faults on output terminals of every primitive, including those inside cells:

```
$ssi_iddq( "model SA term_OUT prims seed_inside_cells" );
$ssi_iddq( "seed SA tbench.M88" );
```

For detailed examples showing how the `model SA` command options affect the placement of fault seeds, see Options for PowerFault-Generated Seeding.

model B

```
model B bridge-placement [seed_inside_cells]
bridge-placement ::= (cell_ports|fet_terms|
   gate_IN2IN|gateIN2OUT|vector)+
```

This command specifies where the `seed B` command seeds bridging faults. A bridging fault is a short circuit between two different functional nodes in the design. A fault of this type is considered detected by an IDDQ strobe when one node is at logic 1 and the other is at logic 0.

Table 4 summarizes the `model B` command options.

Table 4 Options for Bridging Fault Models

Bridge Placement Options	
`cell_ports`	Enables bridging faults between adjacent ports of cells and between each input and output port of cells (if the cell has two or fewer output ports)
`fet_terms`	Enables bridging faults between all pairs of terminals of field effect transistor (FET) switches
`gate_IN2IN`	Enables bridging faults between adjacent input terminals of non-FET primitives (including UDPs)
`gate_IN2OUT`	Enables bridging faults between all pairs of input and output terminals of non-FET primitives (including UDPs)
`vector`	Enables bridging faults between adjacent bits of expanded vectors

Seed Inside Cells Option

`seed_inside_cells`	Enables fault seeding inside cells

The default bridging fault seeding behavior is equivalent to the following `model B` command:
`model B cell_ports fet_terms gate_IN2IN gate_IN2OUT vector`

With the default bridging fault model, bridging faults are seeded between the ports of cells, the terminals of primitives, and the bits of expanded vectors. No seeding is performed inside cells.

To seed other types of bridging faults, specify them with the `model B` command. For example, these two lines seed bridging faults between the ports of all cells inside `tbench.M88`?:

```
$ssi_iddq( "model B cell_ports" );
$ssi_iddq( "seed B tbench.M88" );
```

For detailed examples showing how the `model B` command options affect the placement of fault seeds, see "Options for PowerFault-Generated Seeding".

Strobe Commands

After you specify the DUT modules and seed the faults, you need to describe the IDDQ strobe timing. When the testbench is running, it must use either the `strobe_try` or `strobe_force` command to indicate when it is appropriate to apply an IDDQ strobe.

The following sections describe the various strobe commands:

- strobe_try
- strobe_force
- strobe_limit
- cycle

strobe_try

`strobe_try`

You should have the testbench invoke the `strobe_try` command at as many potential strobe times as possible. The `strobe_try` command tells PowerFault that the circuit is stable and can be tested for quiescence.

For example, you can use the following line just before the end of each cycle:

```
$ssi_iddq("strobe_try")
```

At each occurrence of this line, PowerFault determines whether the circuit is quiescent, allowing an IDDQ strobe to be applied. If the `verb on` command has been executed, the simulator reports the result of each `strobe_try`?, allowing you to identify nonquiescent strobe times.

You should use the `strobe_try` command one time per tester cycle, and it should be the last event of the cycle. For example, if you have delay paths that take multiple clock cycles, do not use the command when those paths are active.

strobe_force

`strobe_force`

This command turns off quiescence checking and allows PowerFault to consider all strobe times. Use this command only if you are sure the chip is quiescent. For example, you can use it if your technology provides an `IDDQ_OK` signal that forces the chip into quiescence.

If you know the quiescent points in your simulation, you can use the `strobe_force` command rather than the `strobe_try` command to reduce the simulation runtime. With the `strobe_force` command, PowerFault does not need to check the entire chip for quiescence at each strobe try.

strobe_limit

`strobe_limit` *`max-strobes`*

This command terminates the Verilog simulation when *`max-strobes`* qualified strobe points have been found.

For example, the following command stops the simulation after 100 qualified strobe points have been found:

```
$ssi_iddq( "strobe_limit 100" );
```

cycle

`cycle` *`cycle-number`*

This command sets the initial PowerFault cycle number, an internal counter maintained by PowerFault. The cycle number has no affect on finding or selecting IDDQ strobes. It is used during Verilog simulations and during strobe selection to report a cycle number along with the simulation time of each strobe.

By default, the cycle number begins at 1 and is incremented after every strobe try. If your test program does not strobe on every cycle, you can use the `cycle` command to synchronize PowerFault with the cycle count of your test program. For example, if your cycle count begins at 0 instead of 1, use this command:

```
$ssi_iddq( "cycle 0" );
```

The `cycle` command can also accept a nonstring argument, allowing you to set the cycle number to the value of a simulation variable. For example:

```
always @testbench.CYCLE
   $ssi_iddq( "cycle", testbench.CYCLE );
```

Circuit Examination Commands

The circuit examination commands, `status` and `summary`?, provide information on the location and cause of IDDQ testing problems found in the design. The following sections describe the circuit examination commands:

- status
- summary

status

`status [drivers]`
`(leaky|nonleaky|both|all_leaky)[`*`file-name`*`]`

This command determines why your circuit is quiescent or nonquiescent at a particular simulation time. It is most useful when you are having difficulty producing qualified strobe points.

If there is a persistent leaky node in your circuit (for example, caused by an always-active pulldown), PowerFault will not be able to find quiescent strobe points. Fortunately, the `status leaky` command can quickly identify any leaky nodes, allowing you to improve your test program so that it produces more quiescent strobe points.

Use the following command to print out all the net conditions that imply that the circuit is not quiescent:

```
$ssi_iddq( "status leaky" );
```

The command prints out the name of each leaky net and the reason that the net's value implies that the circuit is not quiescent. There are two possible causes for a leaky node: a floating input or drive contention.

Here is an example of a report generated by the `status` command:

```
Time 35799
top.dut.ioctl.stba is leaky. Re: float
top.dut.ioctl.addr[0] is leaky. Re: fight
top.dut.ioctl.addr[1] is leaky. Re: possible fight
>
```

Note: If you use the `status` command and the `strobe_try` command in the same simulation run, and you want the status report to include the first strobe, you must execute the first `status` command before the first `strobe_try` command.

Use the following command to print out all the net conditions that imply that the circuit is quiescent:

```
$ssi_iddq( "status nonleaky" );
```

Use the following command to print out all the net conditions that imply that the circuit is or is not quiescent:

```
$ssi_iddq( "status both" );
```

The output of the `status` command can be quite long because it can contain up to one line for every net in the chip. You can direct the output to a file instead of to the screen. For example, to write the leaky states into a file named `bad_nets`?, use the following command:

```
$ssi_iddq( "status leaky bad_nets" );
```

The simulator creates the `bad_nets` file the first time it executes the `status` command. When it executes the `status` command again in the same simulation run, it appends the output to the `bad_nets` file, together with the current simulation time. This creates a report of the leaky states at every disqualified strobe time.

By default, the `leaky` option reports only the first occurrence of a leaky node. If the same leaky condition occurs at different strobe times, the report says "All reported" at each such strobe time after the report of the first occurrence. To get a full report on all leaky nodes, including those already reported, use the `all_leaky` option instead of the `leaky` option, as in the following example:

```
$ssi_iddq( "status all_leaky bad_nodes" );
```

This can produce a very long report.

The `drivers` option makes the `status` command print the contribution of each driver. However, it reports only gate-level driver information. For example, consider the following command:

```
$ssi_iddq( "status drivers leaky bad_nodes" );
```

The command produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
```

```
  St0<- top.dut.mmu.UT344
  St1<- top.dut.mmu.UT366
  StX<- resolved value
top.dut.mmu.TDATA is leaky: Re: float
  HiZ<- top.dut.mmu.UT455
  HiZ<- top.dut.mmu.UT456
```

In this example, `top.dut.mmu.DIO` has a drive fight. One driver is at strong 0 (`St0`) and the other is at strong 1 (`St1`). The contributing value of each driver is printed in Verilog strength/value format (described in section 7.10 of the IEEE 1364 Verilog LRM).

The same `status` command without the `drivers` option produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
top.dut.mmu.TDATA is leaky: Re: float
```

summary

`summary` *`file-name`*

When you use the `summary` command, PowerFault prints a summary at the end of the simulation that describes problem nodes. It lists the nodes reported by the `status` command and also lists the nodes that were not reported but might cause problems.

The summary for each node is reported in this format:

`net-instance-name``:` *`property`*`+`

The `summary` command merges simulation information reported by the `status` command with static information from the formal analyzer. For example, consider the case where the `status` command produces the following output:

```
Time 3999
tbench.M88.SELM.RESET is leaky: Re: float
tbench.M88.VEE[0] is leaky: Re: float
  HiZ <- tbench.M88.CB.vee0.out
  HiZ <- tbench.M88.LB.vee0.out
Time 12999
tbench.M88.DIO[1] is leaky: Re: possible fight
  St0 <- tbench.M88.dpad1_clr
  StX <- tbench.M88.dpad1_snd
  StX <- resolved value
tbench.M88.BIO is leaky: Re: disallowed X
tbench.M88.U244 is leaky: Re: ARAM (WR_EN == 1 && DATA[0]
   == Z)
```

The corresponding summary might look like this:

```
Summary of problem nodes:
tbench.M88.SELM.RESET: did float : unconnected
tbench.M88.VEE[0]: did float : not muxed
tbench.M88.DIO[1]: did fight : can float : not muxed
tbench.M88.BIO: disallowed value
tbench.M88.U244: disallow ARAM (WR_EN == 1 && DATA[0] == Z)
tbench.M88.APP.POW: constant fight
```

The summary lists nodes that can cause problems for IDDQ testing. It might also identify node properties that are considered design problems. For example, if floating nodes are illegal in your

design environment, you should check to see whether any nodes have the “did float” or “can float” property.

The more your circuit is modeled at the gate level, the more accurate the summary is.

Table 5 lists and describes the node properties reported by the `summary` command.

Table 5 Node Properties Reported by summary Command

Node Property	Description
did float	The node was reported as floating (or possibly floating) during simulation.
did fight	The node was reported as having (or possibly having) drive contention during the simulation.
did pull	The node was reported as having (or possibly having) an active pullup/pulldown during simulation.
disallowed value	The node was reported as violating a simple `disallow` command during the simulation.
disallow *expr*	The node was reported as violating a compound `disallow` command during the simulation. *expr* contains the text of the `disallow` command.
can float	The node can float, but was not reported as floating during the simulation.
can fight	The node can have drive contention, but was not reported as having this condition during the simulation.

Node Property	Description
can pull	The node has pullups/pulldowns, but they were not active during the simulation.
not muxed	The node has multiple drivers that are not multiplexed. In other words, the control logic for the drivers does not always enable one and only one driver at a time.
unconnected	The node is an unconnected input.
constant fight	The node has a constant current. In other words, it has both a pullup and a pulldown.

Disallowed/Disallow Value Property

A node with the “disallowed value” property violated a simple `disallow` command at some time during the simulation. Here are some examples of simple `disallow` commands:

```
$ssi_iddq( "disallow tbench.M88 (BIO == X)" );
$ssi_iddq( "disallow BUF3I (out == 0)" );
```

A node with the “disallow *expr*” property violated a compound `disallow` command at some time during the simulation. Here are some examples of compound `disallow` commands:

```
$ssi_iddq( "disallow ARAM (WR_EN == 1 && DATA[0] == Z)" );
$ssi_iddq( "disallow PHMX (in == 1 && en != 0)" );
```

Can Float Property

Each node with the “can float” property requires special consideration because it can cause high current. Each such node was never reported as floating during the simulation because of one or more of these conditions:

- The node never floated.
- The node floated but was blocked.
- The node floated but did not have a load (it was not connected to a gate-level input).

See Also

Floating Nodes and Drive Contention

8

Faults and Fault Seeding

The process of specifying fault locations for IDDQ testing is called *fault seeding*. You can have PowerFault seed faults automatically from the design description, or you can use a fault list generated by TetraMAX ATPG or another tool.

The following sections describe faults and fault seeding:

Fault Models

The TetraMAX ATPG and Verilog/PowerFault environments support several different types of fault models which are described in the following sections:

- Fault Models in TetraMAX
- Fault Models in PowerFault

Fault Models in TetraMAX

In TetraMAX ATPG, the term “fault model” refers to the type of fault used for test pattern generation.

- For IDDQ testing, there are two choices: stuck-at and IDDQ. The stuck-at fault model is the standard, default model most often used to generate test patterns.
- The IDDQ fault model is used to generate test patterns specifically for IDDQ testing.

There are two types of IDDQ fault models, the pseudo-stuck-at model and the toggle model.

The fault model choice in TetraMAX ATPG determines how the ATPG algorithm operates. For the stuck-at model, TetraMAX ATPG attempts to propagate the effects of faults to the scan elements and device outputs. For the IDDQ model, TetraMAX ATPG attempts to control all nodes to 0 and 1 while avoiding conditions that violate quiescence.

For more information on TetraMAX fault models, see the *TetraMAX ATPG User Guide* or consult the TetraMAX online help.

Fault Models in PowerFault

In the PowerFault environment, the term “fault model”? refers to the algorithm used to seed faults in the design when you use the `seed SA` command to seed stuck-at faults or the `seed B` command to seed bridging faults.

Stuck-At Faults

A stuck-at-0 fault is considered detected when the node in question is placed in the 1 state, the circuit is quiescent, and an IDDQ strobe occurs. Similarly, a stuck-at-1 fault is considered detected when the node is placed in the 0 state, the circuit is quiescent, and an IDDQ strobe occurs.

To seed stuck-at faults from a TetraMAX fault file, use the `read_tmax` command. Similar commands are available to seed faults from a Verifault or Zycad fault list. To seed stuck-at faults automatically throughout the design based on the locations of the modules, cells, primitives, ports, and terminals in the design, use the `model SA` and `seed SA` commands.

Untestable faults are ignored during fault detection and strobe selection, but they are still listed in fault reports for reference. Faults untestable by PowerFault include stuck-at-0 faults on supply0 wires and stuck-at-1 faults on supply1 wires.

Bridging Faults

A bridging fault involves two nodes. The fault is considered detected when one node is placed in the 1 state, the other is placed in the 0 state, the circuit is quiescent, and an IDDQ strobe occurs. For an accurate fault model, the two nodes in question must be physically adjacent in the fabricated device, so that actual bridging between the nodes is possible in a defective device.

You can seed bridging faults by reading them from a list (which could be generated by an external tool) by using the `read_bridges` command. You can also seed bridging faults automatically between adjacent cell ports, between terminals of field effect transistor (FET) switches, between the terminals of gate primitives, and between adjacent vector bits. In this case, *adjacent* means "right next to each other in the Verilog description." To seed bridging faults in this manner, use the `model B` and `seed B` commands.

Fault Seeding

At the beginning of the Verilog/PowerFault simulation, before using the `strobe_try` command to evaluate strobes for IDDQ testing, you need to tell PowerFault where to seed faults. To do this, you can use `seed` commands to seed faults automatically or the `read_tmax` command to seed faults from an existing fault list.

The `seed` and `read_tmax` commands are cumulative. If you want to seed some faults automatically and seed some faults from a fault list, you can use both the `seed` and `read_tmax` commands, and all of the faults seeded by the two commands are used.

The following sections describe fault seeding:

- Seeding From a TetraMAX Fault List
- Seeding From an External Fault List
- PowerFault-Generated Seeding

Seeding From a TetraMAX Fault List

To seed the design with stuck-at faults from a TetraMAX fault list, use the `read_tmax` command. In this command, you specify the TetraMAX fault file name, and optionally, the detectability classes of faults to be seeded.

In TetraMAX ATPG, you create a fault file upon completion of test pattern generation by using the `write_faults` command. Typically, you write a complete fault list using a command similar to the following:

```
write_faults mylist.faults -replace -all
```

Before you generate the fault list, you need to set the hierarchical delimiter character in TetraMAX ATPG. PowerFault expects the delimiter character to be a period. By default, TetraMAX ATPG uses the forward slash (/) character. To generate the fault list in a compatible format, use the following `set_build` command before you build the model:

```
set_build -hierarchical_delimiter .
```

The generated fault file describes each fault in terms of type (stuck-at-0 or stuck-at-1), detectability class, and location in the design. For example:

```
sa0    DS    .testbench.fadder.co
```

```
sa1   DS   .testbench.fadder.co
sa0   DS   .testbench.fadder.sum
sa1   DS   .testbench.fadder.sum
...
```

Each fault class and each hierarchical group of fault classes has a two-character abbreviation. For example, DS stands for “detected by simulation.”

The TetraMAX fault classes are defined in the following list:

```
   DT - detected
      DS - detected by simulation
      DI - detected by implication
   PT - possibly detected
      AP - ATPG untestable, possibly detected
      NP - not analyzed, possibly detected
   UD - undetectable
      UU - undetectable, unused
UO - undetectable, unobservable
      UT - undetectable, tied
      UB - undetectable, blocked
      UR - undetectable, redundant
   AU - ATPG untestable
      AN - ATPG untestable, not detected
   ND - not detected
     NC - not controlled
      NO - not observed
```

TetraMAX ATPG places each fault into one of the bottom-level fault classes. For more information about fault classes, refer to the *TetraMAX ATPG User Guide*.

By default, the PowerFault command `read_tmax` seeds faults in the AP, NP, NC, and NO classes. If you want to seed faults belonging to classes other than the default set, you need to specify the classes in the `read_tmax` command. For example, the following command seeds faults in the `fa1` file that belong to the following classes: possibly detected (AP, NP), undetectable (UU, UT, UB, UR), ATPG untestable (AN), and not detected (NC, NO):

```
$ssi_iddq( "read_tmax AP NP UU UT UB UR AN NC NO fa1" );
```

One way to use this command is to target undetectable and possibly detected faults in TetraMAX ATPG. In this way, PowerFault complements TetraMAX ATPG to obtain the best possible overall test coverage. If adequate coverage of these faults is obtained with just a few IDDQ strobes and if your tester time budget allows it, you can then seed faults throughout the design with the `seed SA` command and generate additional IDDQ strobes to obtain even better IDDQ test coverage.

Seeding From an External Fault List

If you use the Verifault-XL fault simulator, you can seed the design with faults from a Verifault fault list or fault dictionary. Similarly, if you use the Zycad fault simulator, you can seed the design with faults from the Zycad fault origin file.

To seed faults from these types of files, use the `read_verifault` command, described in “read_verifault” or the `read_zycad` command, described in “read_zycad” in the "PowerFault PLI Tasks" section.

To seed the design with bridging faults from a file-based list, use the `read_bridges` command. For details, see “read_bridges” in the "PowerFault PLI Tasks" section.

PowerFault-Generated Seeding

To have PowerFault automatically seed the design, use the `seed SA` command to seed stuck-at faults or the `seed B` command to seed bridging faults. To specify how these seeding algorithms operate, use the `model SA` and `model B` commands. For details, see “Fault Model Commands” in the "PowerFault PLI Tasks" section.

Options for PowerFault-Generated Seeding

For PowerFault-generated seeding, use the `seed SA` and `seed B` commands. The `model SA` and `model B` commands specify the behavior of the seeding algorithms.

The following sections provide some specific examples showing how you can use the `model SA` and `model B` command options to control the seeding of faults in the design:

- Stuck-At Fault Model Options
- Bridging Faults

For basic information on using the `model SA` or `model B` command, see “model SA” or “model B” in " PowerFault PLI Tasks" section.

Stuck-At Fault Model Options

The `model SA` command determines where the `seed SA` command seeds stuck-at faults. Table 1 lists and describes the fault model options available in the `model SA` command.

Table 1 Options for Stuck-At Fault Models

Direction Options	
`port_IN`	Enables stuck-at faults on input ports of chosen modules
`port_OUT`	Enables stuck-at faults on output ports of chosen modules
`term_IN`	Enables stuck-at faults on input terminals of primitives
`term_OUT`	Enables stuck-at faults on output terminals of primitives
Stuck-At Placement Options	

`all_mods`	Chooses all modules for port stuck-at faults
`leaf_mods`	Chooses leaf modules for port stuck-at faults
`cell_mods`	Chooses cell modules for port stuck-at faults
`prims`	Chooses primitives for terminal stuck-at faults

Seed Inside Cells Option

`seed_inside_cells`	Enables fault seeding inside cells

The `all_mods`?, `leaf_mods`?, and `cell_mods` options specify which types of modules will have port faults. The `port_IN` and `port_OUT` options specify which types of ports from those modules are seeded with stuck-at faults.

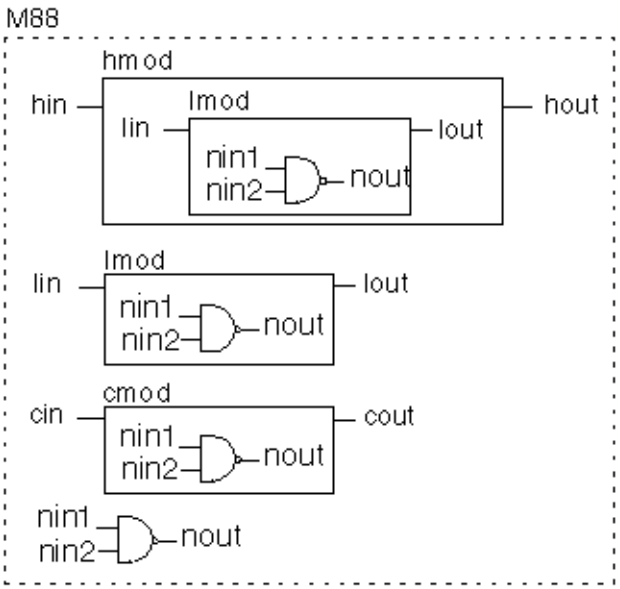
The `prims` option specifies that any primitive instance found within a seeded module will have terminal faults. The `term_IN` and `term_OUT` options specify which types of terminals from those primitives are seeded with stuck-at faults.

Here is a specific example to help demonstrate how these options work. Assume that you have the following Verilog description of a testbench module called tbench.M88:

```
module M88();
  hier hmod( hout, hin );
  leaf lmod( lout, lin );
  cell cmod( cout, cin );
  nand( nout, nin1, nin2 );
endmodule

module hier( out, in );
output out;
input in;
  leaf lmod( lout, lin );
endmodule

module leaf( out, in );
output out;
input in;
  nand( nout, nin1, nin2 );
endmodule

`celldefine
module cell( out, in );
output out;
input in;
  nand( nout, nin1, nin2 );
endmodule
`endcelldefine
```

At the top level of hierarchy, this testbench module contains a hierarchical module (?hmod?), a leaf-level module (?lmod?), a module that has been defined as a cell (?cmod?), and a primitive gate (?nand?). Figure 1 shows a circuit diagram corresponding to this Verilog description.

Figure 1 Circuit Example for Stuck-At Fault Seeding



Default Stuck-At Fault Seeding

By default, the `seed SA` command seeds port faults on leaf and cell modules and seeds terminal faults on primitives. The default behavior is equivalent to using the following `model SA` command:

```
model SA port_IN port_OUT term_IN term_OUT
   leaf_mods cell_mods prims
```

Suppose that you start stuck-at seeding using the default model:

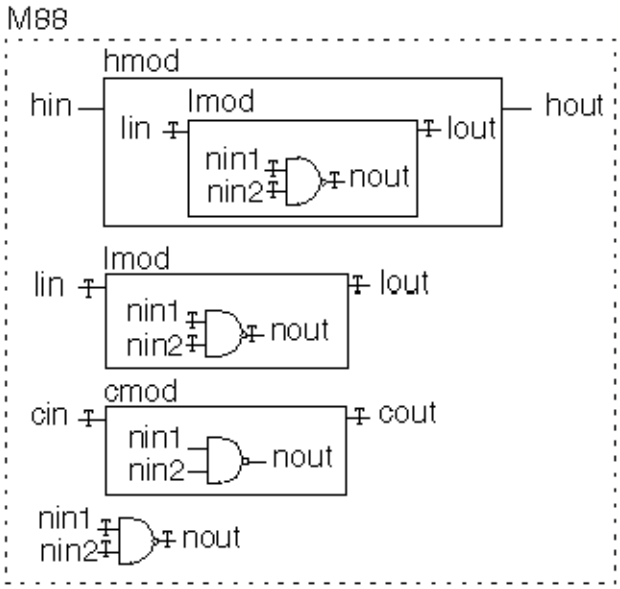
```
$ssi_iddq( "seed SA tbench.M88" );
```

This command seeds stuck-at faults on the following nets:

```
tbench.M88.lmod.lin
tbench.M88.lmod.lout
tbench.M88.hmod.lmod.nin1
tbench.M88.hmod.lmod.nin2
tbench.M88.hmod.lmod.nout
tbench.M88.lin
tbench.M88.lout
tbench.M88.lmod.nin1
tbench.M88.lmod.nin2
tbench.M88.lmod.nout
tbench.M88.cin
tbench.M88.cout
tbench.M88.nin1
tbench.M88.nin2
tbench.M88.nout
```

Figure 2 shows the circuit diagram with each seeded fault marked with an asterisk (*).

Figure 2 Seed Locations: Default Stuck-At Fault Model



all_mods

The `all_mods` option chooses all modules for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of all modules inside tbench.M88?:

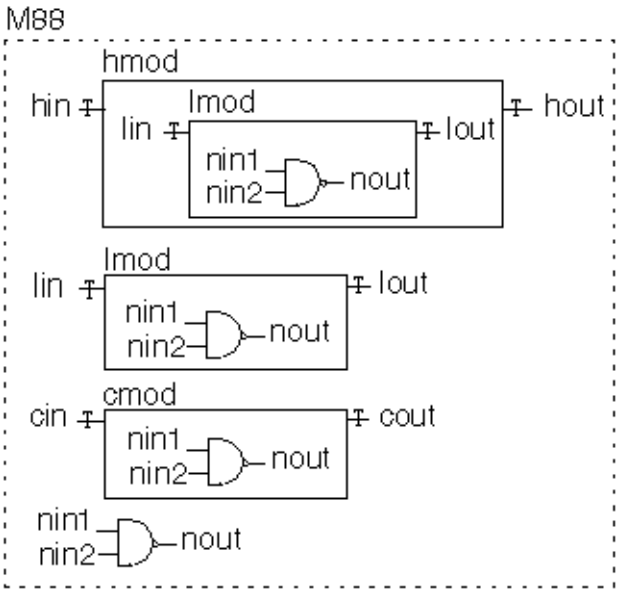
```
$ssi_iddq( "model SA port_IN port_OUT all_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hin
tbench.M88.hout
tbench.M88.hmod.lin
tbench.M88.hmod.lout
tbench.M88.lin
tbench.M88.lout
tbench.M88.cin
tbench.M88.cout
```

Figure 3 shows the resulting locations of seeds using this fault model.

Figure 3 Seed Locations: all_mods Stuck-At Fault Model



cell_mods

The `cell_mods` option chooses cells for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of every cell module inside tbench.M88?:

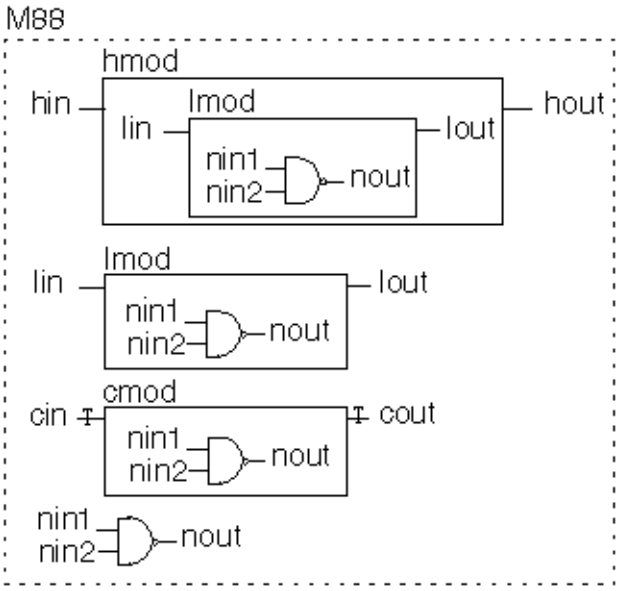
```
$ssi_iddq( "model SA port_IN port_OUT cell_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.cin
tbench.M88.cout
```

Figure 4 shows the resulting locations of seeds using this fault model.

Figure 4 Seed Locations: cell_mods Stuck-At Fault Model



leaf_mods

The `leaf_mods` option chooses leaf-level modules for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of every leaf module inside tbench.M88?:

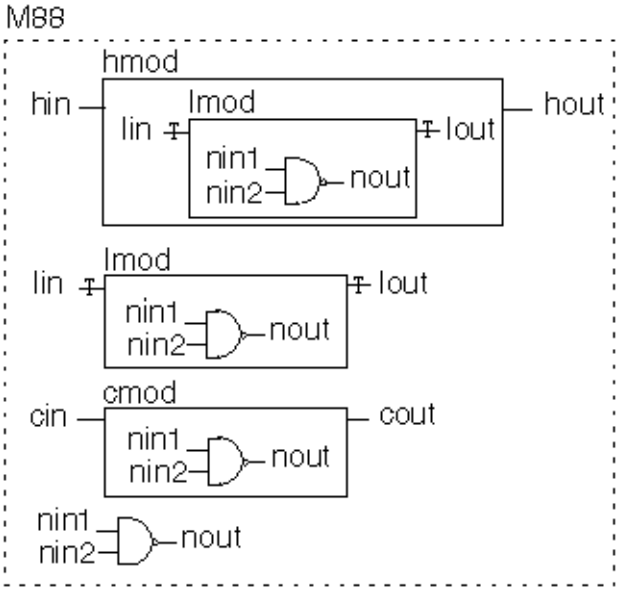
```
$ssi_iddq( "model SA port_IN port_OUT leaf_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lin
tbench.M88.hmod.lout
tbench.M88.lin
tbench.M88.lout
```

Figure 5 shows the resulting locations of seeds using this fault model.

Figure 5 Seed Locations: leaf_mods Stuck-At Fault Model



prims

The `prims` option chooses primitives for terminal stuck-at faults. Thus, the following two lines seed faults on the input terminal of every primitive inside tbench.M88?:

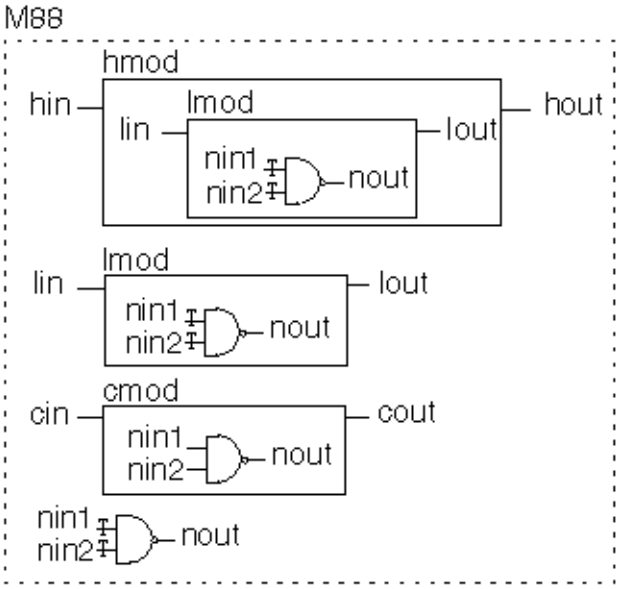
```
$ssi_iddq( "model SA term_IN prims" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lmod.nin1
tbench.M88.hmod.lmod.nin2
tbench.M88.lmod.nin1
tbench.M88.lmod.nin2
tbench.M88.nin1
tbench.M88.nin2
```

Figure 6 shows the resulting locations of seeds using this fault model.

Figure 6 Seed Locations: Primitive Input Stuck-At Fault Model



The following two lines seed faults on the output terminal of every primitive inside tbench.M88:

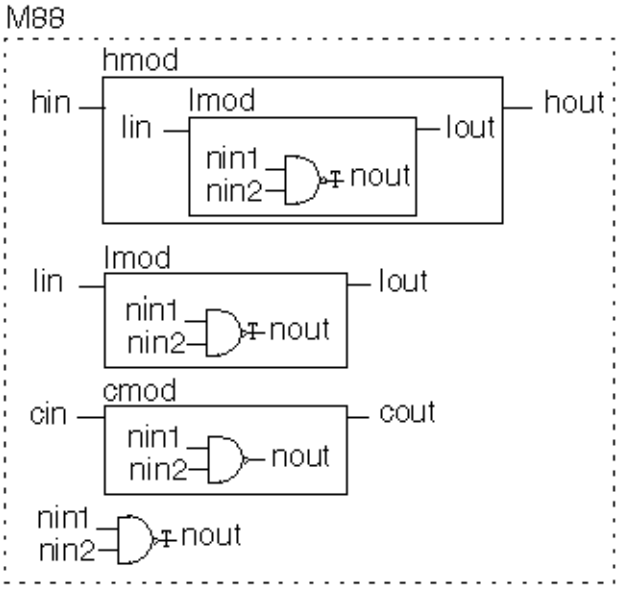
```
$ssi_iddq( "model SA term_OUT prims" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lmod.nout
tbench.M88.lmod.nout
tbench.M88.nout
```

Figure 7 shows the resulting locations of seeds using this fault model.

Figure 7 Seed Locations: Primitive Output Stuck-At Fault Model



seed_inside_cells

The `seed_inside_cells` option enables seeding of faults inside cells. Thus, the following two lines seed faults on the output terminal of every primitive inside tbench.M88?, including those inside cells:

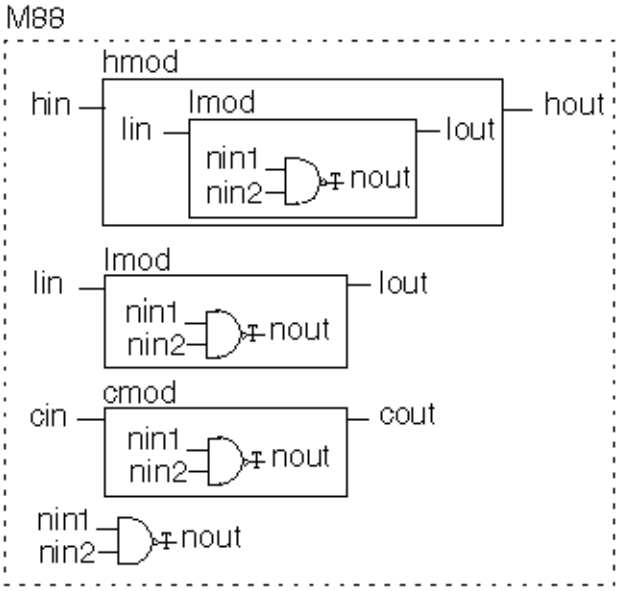
```
$ssi_iddq( "model SA term_OUT prims seed_inside_cells" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lmod.nout
tbench.M88.lmod.nout
tbench.M88.cmod.nout
tbench.M88.nout
```

Figure 8 shows the resulting locations of seeds using this fault model.

Figure 8 Primitive Output Seeding for seed_inside_cells



Bridging Faults

The `model B` command determines where the `seed B` command seeds bridge faults. Table 2 lists and describes the bridge placement options available for the `model B` command.

Table 2 Options for Bridging Fault Models

Bridge Placement Options

`cell_ports`	Enables bridging faults between adjacent ports of cells and between each input and output port of cells (if the cell has no more than two output ports)
`fet_terms`	Enables bridging faults between all pairs of terminals of FET switches
`gate_IN2IN`	Enables bridging faults between adjacent input terminals of non-FET primitives (including UDPs)
`gate_IN2OUT`	Enables bridging faults between all pairs of input and output terminals of non-FET primitives (including UDPs).
`vector`	Enables bridging faults between adjacent bits of expanded vectors

Seed Inside Cells Option

`seed_inside_cells`	Enables fault seeding inside cells

cell_ports

The `cell_ports` option seeds bridging faults between adjacent ports of each cell, and also between the cell inputs and outputs if the cell has no more than two output ports. Ports are considered adjacent when they appear next to each other in the module's port list definition. For example, consider the following module definition:

```
‘celldefine
module bsel( out, in1, in2, in3 );
output out;
input in1, in2, in3;
endmodule
‘endcelldefine
```

The following port pairs are considered adjacent:

```
out, in1
in1, in2
in2, in3
```

As a result, the `cell_ports` option seeds five bridging faults: three between pairs of adjacent ports and two more between the inputs and outputs. This is the bridging fault list:

```
out, in1
in1, in2
in2, in3
out, in2
out, in3
```

fet_terms

The `fet_terms` option seeds bridging faults between all pairs of terminals of each FET switch. This results in four bridging faults for a CMOS switch or three bridging faults for any other type of switch.

For example, consider this primitive:

```
nmos UF44( out, data, ctl );
```

The `term_fets` option seeds these three bridging faults:

```
out, data
out, ctl
data, ctl
```

gate_IN2IN

The `gate_IN2IN` option seeds bridging faults between adjacent input terminals of gates. Terminals are considered adjacent when they appear next to each other in the primitive's terminal list.

For example, consider the following primitive:

```
and U2033( out, in1, in2, in3 );
```

The `gate_IN2IN` option seeds the following two bridging faults:

```
in1, in2
in2, in3
```

gate_IN2OUT

The `gate_IN2OUT` option is like the `gate_IN2IN` option, except that it seeds bridging faults between inputs and outputs. For the previous example, the `gate_IN2OUT` option seeds the following three bridging faults:

```
out, in1
out, in2
out, in3
```

vector

The `vector` option seeds bridging faults between adjacent bits of a vector. Two bits are considered adjacent when they have an index within one unit of each other.

For example, consider the following vector:

```
wire [3:0] dvec;
```

The `vector` option seeds the following three bridging faults:

```
dvec[3], dvec[2]
dvec[2], dvec[1]
dvec[1], dvec[0]
```

seed_inside_cells

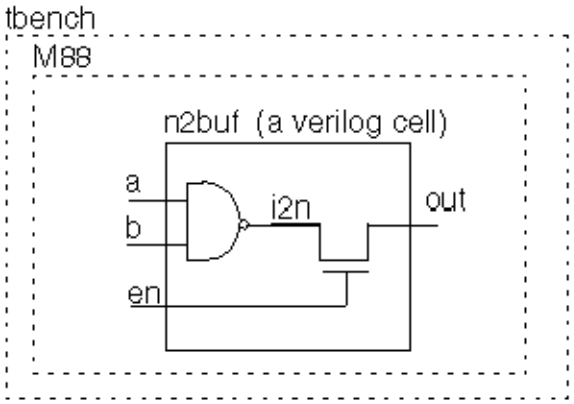
The `seed_inside_cells` option enables seeding of faults inside cells.

Assume that you have a circuit with a module tbench.M88 that contains an instance of the following cell:

```
‘celldefine
module n2buf( a, b, en, out);
input a, b, en;
output out;
nmos( out, n2out, en );
nand( a2out, a, b );
endmodule
‘endcelldefine
```

Figure 9 shows a circuit diagram for this cell.

Figure 9 Example Circuit for Bridging Faults



The following two lines seed bridging faults between cell ports and between FET-switch terminal pairs inside tbench.M88?:

```
$ssi_iddq( "model B cell_ports fet_terms" );
$ssi_iddq( "seed B tbench.M88" );
```

These commands seed five bridging faults between the ports of `n2buf`?:

```
a, b
b, en
a, out
b, out
en, out
```

By default, no faults are seeded inside of cells. Therefore, the internal net `i2n` is not considered for fault seeding. To include this internal node, use the `seed_inside_cells` option:

```
$ssi_iddq( "model B cell_ports fet_terms
   seed_inside_cells" );
$ssi_iddq( "seed B tbench.M88" );
```

In this case, the following additional bridging faults are seeded:

```
i2n, en
i2n, out
```

9

PowerFault Strobe Selection

After you run a Verilog/PowerFault simulation, you can use the PowerFault strobe selection tool, IDDQPro, to select a set of strobe times to obtain maximum fault coverage. IDDQPro uses the information in the IDDQ database produced by the Verilog/PowerFault simulation.

The following sections describe PowerFault strobe selection:

Overview of IDDQPro

IDDQPro is a strobe selection tool that operates on the IDDQ database produced by a Verilog/PowerFault simulation. IDDQPro selects a set of strobe times to maximize fault coverage for a given number of strobes.

When you run a Verilog/PowerFault simulation, the `output` command in the PowerFault Verilog module specifies the name of the IDDQ database. The database contains information on seeded faults and the faults detected at each qualified strobe time.

When you invoke IDDQPro, you specify the database name and the number of strobes you want to use. IDDQPro analyzes the database and finds a set of strobes that maximizes the number of faults detected.

You can run IDDQPro in batch mode or interactive mode.

- In batch mode, IDDQPro selects a set of strobes and reports the results.
- In interactive mode, IDDQPro displays a command prompt.

You can interactively enter commands to select strobes, display reports, and traverse the hierarchy of the design.

IDDQPro produces two report files: a strobe report (iddq.srpt) and a fault report (iddq.frpt).

- The strobe report shows the time value and cumulative fault coverage of each selected strobe point.
- The fault report lists the status of each seeded fault, either detected or undetected, for the complete set of selected strobes.

Each report file starts with a header that summarizes the report contents and tells you how to interpret the information provided.

After you use IDDQPro to select a set of strobes, it is a good idea to copy and save the strobe report file so that you will not need to generate it again. The strobe report can take a long time to generate. It is not as important to save the fault report file because you can quickly regenerate it, as long as you have the strobe report file.

Invoking IDDQPro

You invoke IDDQPro at an operating system prompt. The following sections describe the process for invoke IDDQPro:

- ipro Command Syntax
- Strobe Selection Options
- Report Configuration Options
- Log File and Interactive Options

ipro Command Syntax

The full Backus-Naur form (BNF) description of the command syntax for IDDQPro is as follows:

```
ipro options* iddq-database-name+
options ::=
-strb_lim max-strobes |
-cov_lim percent-cov |
-ign_ucov |
-strb_set file-name |
-strb_unset file-name |
-strb_all |
-prnt_fmt (tmax|verifault|zycad) |
-prnt_nofrpt |
-prnt_full |
-prnt_times |
-path_sep (.|/)  |
-log file-name |
-inter
```

The command consists of the keyword `ipro`?, followed by zero or more options, followed by one or more IDDQ database names. A typical command specifies a limit on the number of strobes with the `-strb_lim`option and specifies a single IDDQ database. For example:

```
ipro -strb_lim 5 iddq
```

This command invokes IDDQPro, specifies a maximum limit of five strobes, and specifies `iddq` as the name of the IDDQ database.

Here are some more examples of IDDQPro invocation commands:

```
ipro -strb_lim 5 iddqdb1 iddqdb2
ipro -strb_lim 8 /net/simserver/CCD/iddq
ipro -strb_lim 10 iddq
ipro -strb_lim 10 -cov_lim 0.95 iddq
ipro -strb_lim 10 -cov_lim 0.95 -prnt_fmt verifault iddq
```

Strobe Selection Options

You can control strobe selection by using the following `ipro` command options:

`-strb_lim` *max-strobes*
`-cov_lim` *percent-cov*
`-strb_set` *file-name*
`-strb_unset` *file-name*
`-strb_all`

If you do not use any options, IDDQPro selects strobes until it either uses up all the possible strobe points or reaches the absolute maximum coverage possible.

-strb_lim

The `-strb_lim` option specifies the maximum number of strobe points to select. The practical maximum number depends on the test equipment being used. Typically, only five to ten IDDQ strobes are allowed per test. IDDQPro attempts to obtain the best possible coverage, given the specified maximum number of strobes.

For example, to limit the number of selected strobes to ten, you would use a command such as the following:

```
ipro -strb_lim 10 iddq
```

-cov_lim

The `-cov_lim` option specifies the target fault coverage percentage. Strobe selection stops when fault coverage reaches or exceeds this limit. Coverage is expressed as a decimal fraction between 0.00 and 1.00. For example, to choose as many strobes as necessary to reach 80 percent fault coverage, you would use a command such as the following:

```
ipro -cov_lim 0.80 iddq
```

-strb_set

The `-strb_set` option causes IDDQPro to select the strobe times listed in a file. IDDQPro evaluates the effectiveness of the strobes listed in the file. If you have a set of strobe times you think are good for IDDQ testing, put them into a file, with one time value per line.

For example, to force the selection of strobes at times 29900 and 39900, put those two times into a file named `stimes` like this,

```
29900
39900
```

and then use a command such as the following:

```
ipro -strb_set stimes -strb_lim 8 /cad/sim/M88/iddq
```

As a result of this command, IDDQPro selects the two specified strobe times, plus six other strobe times that it selects with its regular coverage-maximizing algorithm. The usual strobe report, `iddq.srpt`?, includes all eight strobes. In addition, IDDQPro generates a separate strobe evaluation report called iddq.seval?, which shows the coverage obtained by just the two file-specified strobe times.

If you are using multiple testbenches, specify the testbench number before each strobe time. Testbench numbering starts at 1. For example, to select the strobes at times 299 and 1899 in the first testbench and time 399 in the second testbench, enter the following lines in the strobe time file:

```
tb=1 299
tb=1 1899
tb=2 399
```

To regenerate a fault report from a saved strobe report, use the `-strb_set` option and specify the name of the strobe report file. For example:

```
ipro -strb_set iddq.srpt -strb_lim 5 iddq
```

-strb_unset

The `-strb_unset` option prevents IDDQPro from selecting the strobe times listed in a file. If you have a set of strobe times that you do not want IDDQPro to use, put them into a file, with one time value per line. For example, if you want to prevent the strobes at times 59900 and 89900 from being selected, put those two times into a file named `bad_stimes` and then use a command such as the following:

```
ipro -strb_unset bad_stimes -strb_lim 8 /cad/sim/M88/iddq
```

As a result of this command, IDDQPro selects eight strobe times using its regular coverage-maximizing algorithm, but excluding the strobes at times 59900 and 89900. If you are using multiple testbenches, specify the testbench number before each strobe time as explained previously for the `-strb_set` option.

-strb_all

The `-strb_all` option causes IDDQPro to select all qualified strobe points, starting with the first strobe time, instead of using the coverage-maximizing algorithm. The strobe report and fault report show the coverage obtained by making an IDDQ measurement at every qualified strobe point.

Although it is usually impractical to make so many measurements, the `-strb_all` option is useful because it determines the maximum possible coverage that can be obtained from your testbench or testbenches. In addition, the resulting fault report identifies nets that never get toggled; they are reported as undetected.

Report Configuration Options

You can control the generation of the fault report by IDDQPro by using the following `ipro` command options:

`-prnt_fmt (tmax|verifault|zycad)`
`-prnt_nofrpt`
`-prnt_full`
`-prnt_times`
`-path_sep`
`-ign_ucov`

-prnt_fmt

The `-prnt_fmt` option specifies the format of the fault report produced by IDDQPro. The format choices are `tmax?`, `verifault?`, and `zycad?`. The default format is `tmax?`.

In the default format, the faults are reported as shown in the following example:

```
sa0 NO .testbench.fadder.co
sa1 DS .testbench.fadder.co
sa0 DS .testbench.fadder.sum
sa1 NO .testbench.fadder.sum
...
```

To generate a fault report in Zycad `.fog` format, use a command similar to the following:

```
ipro -prnt_fmt zycad -strb_lim 5 iddq
```

In the Zycad configuration, faults are reported as shown in the following example:

```
@testbench.fadder
        co 0 n U
        co 1 n D
        sum 0 n D
        sum 1 n U
       ...
```

To generate a fault report in Verifault format, use a command similar to the following:

```
ipro -prnt_fmt verifault -strb_lim 5 iddq
```

In the Verifault configuration, faults are reported as shown in the following example:

```
fault net sa0 testbench.fadder.co 'status=undetected';
fault net sa1 testbench.fadder.co 'status=detected';
fault net sa0 testbench.fadder.sum 'status=detected';
fault net sa1 testbench.fadder.sum 'status=undetected';
...
```

-prnt_nofrpt

Use the `-prnt_nofrpt` option to suppress generation of the fault report. Otherwise, by default, IDDQPro generates the iddq.frpt fault report every time the program is run in batch mode.

-prnt_full, -prnt_times, and -path_sep

The `-prnt_full`?, `-prnt_times`?, and `-path_sep` options control the generation of Zycad-format fault reports. These options do not affect on Verifault-format fault reports.

The `-prnt_full` option controls the reporting of hierarchical paths. By default, faults are divided into groups, with the cell name shown at the beginning of each group. Only the leaf-level net name is shown in each line.

Here is an example taken from a report in the default Zycad reporting format:

```
@tbench.M88
sio24 0 n D
sio24 1 n D
sio25 0 n D
sio25 1 n U
```

If you use the `-prnt_full` option, the full hierarchical paths are reported in each line, as shown in the following example:

```
tbench.M88.sio24 0 n D
tbench.M88.sio24 1 n D
tbench.M88.sio25 0 n D
tbench.M88.sio25 1 n U
```

The `-prnt_times` option causes the fault report to include the simulation time at which each fault was first detected. For example, with the `-prnt_times` options, the same faults as described in the preceding example are reported as follows:

```
tbench.M88.sio24 0 n 129900 D
tbench.M88.sio24 1 n 39900 D
tbench.M88.sio25 0 n 455990 D
tbench.M88.sio25 1 n U
```

The `-path_sep` option specifies the character for separating the components of a hierarchical path. The default character is a period (.) so that path names are compatible with Verilog. If you want Zycad-style path names, select the forward slash character (/) instead, as in the following example:

```
ipro -prnt_fmt zycad -prnt_full -path_sep / -strb_lim 5 iddq
```

Then the same faults described previously are reported as follows:

```
/tbench/M88/sio24 0 n D
/tbench/M88/sio24 1 n D
/tbench/M88/sio25 0 n D
/tbench/M88/sio25 1 n U
```

-ign_uncov

The `-ign_uncov` option prevents IDDQPro from using the “potential” status in the fault report. All faults are still listed, but faults that would normally be reported as potential are instead reported as undetected. This option also prevents IDDQPro from generating coverage statistics for uninitialized nodes in the strobe report. For information on uninitialized nodes, see “Faults Detected at Uninitialized Nodes”.

Log File and Interactive Options

The `-log` option lets you specify the name of the IDDQPro log file. The log file contains a copy of all messages displayed during the IDDQPro session. By default, the log file name is iddq.log.

By default, IDDQPro runs in batch mode. This means that IDDQPro reads the IDDQ database, selects the strobe times, produces the strobe report and fault report files, and returns you to the operating system prompt.

The `-inter` option lets you run IDDQPro in interactive mode. In this mode, IDDQPro displays a prompt. You interactively select strobes manually or automatically, request the reports that you want to see, and optionally browse through the hierarchy of the design.

The IDDQPro interactive commands are described in the next section, “Interactive Strobe Selection.”

Interactive Strobe Selection

To use IDDQPro in interactive mode, invoke it with the `-inter` option, as in the following example:

```
% ipro -inter iddq
```

When IDDQPro is started in interactive mode, it loads the results from the Verilog simulation and waits for you to enter a command. No strobes are selected and no reports are generated until you enter the commands to request these actions.

At the interactive command prompt, you can enter commands to select strobes, display reports, and traverse the hierarchy of the design. When you change to a lower-level module in the design hierarchy, the reports that you generate apply only to the current scope of the design.

Table 1 lists and briefly describes the interactive commands. The following sections provide detailed descriptions of these commands.

Table 1 IDDQPro Interactive Commands

Command	Description
cd	Changes the interactive scope to lower-level instance
desel	Prevents selection of specified strobe times
exec	Executes a list of interactive commands in a file
help	Displays a summary description of all commands or one command
ls	Displays a list of lower-level instances at the current level
prc	Prints a fault coverage report
prf	Prints a list of all seeded faults and their detection status
prs	Prints a list of all qualified strobes and their status
quit	Terminates IDDQPro
reset	Cancels all strobe selections and detected faults
sela	Selects strobes automatically using the coverage-maximizing algorithm
selall	Selects all strobes
selm	Selects one or more strobes manually, specified by time value

To run an interactive IDDQPro session, you typically use the following steps:

1. Select the strobes automatically or manually, or select all strobes (?`sela`?, `selm`?, or `selall`?).
2. If you want to analyze just a submodule of the design, change to hierarchical scope for that module (?`ls`?, `cd`?).
3. Print a strobe report, coverage report, and fault report (?`prs`?, `prc`?, `prf`?).
4. Repeat steps 1 through 3 to examine different sets of strobes or different parts of the design. Use the `reset` command to select an entirely new set of strobes.
5. Exit from IDDQPro (`?quit?`).

By default, the output of all interactive commands is sent to the terminal (stdout). The printing commands, especially `prf` and `prs?`, can produce very long reports. If you want to redirect the output of one of these commands to a file, use the `-out` option.

---

cd

`cd` *`module-instance`*

The `cd` command changes the current scope of the analysis to a specified module instance. You can use this command to produce different reports for different parts of the design. For example, to print separate fault reports for modules `top.M88.alu` and `top.M88.io`?, enter the following commands:

```
cd top.M88.alu
prf -out alu.frpt
cd top.M88.io
prf -out io.frpt
```

To get a listing of modules in the current hierarchical scope, use the `ls` command. To move up to the next higher level of hierarchy, use the following command:

```
cd ..
```

---

desel

`desel` *`strobe-times* selm-options*`*

`strobe-times` `::=` `[tb=`*`testbench-number`*`]` *`simulation-time`*

`selm-options` `::=` `-in` *`file-name`* | `-out` *`file-name`*

The `desel` (deselect) command prevents IDDQPro from selecting one or more specified strobe times when you later use the `sela` or `selall` command. The strobe times can be explicitly listed in the command line or read from an input file.

If the `desel` command specifies strobes that are currently selected, they are first deselected. The specified strobes are all made unselectable by subsequent invocations of the `sela` or `selall` command. However, they can still be selected manually with the `selm` command.

For example, the following command deselects the two strobes at 59900 and 89900 and prevents them from being selected automatically by a subsequent `sela` or `selall` command:

```
desel 59900 89900
```

If you are using multiple testbenches, you can deselect strobes from different testbenches. For example, the following command manually deselects strobes at time 799 and 1299 from testbench 1 and a strobe at time 399 from testbench 2:

```
desel tb=1 799 tb=1 1299 tb=2 399
```

---

exec

```
exec file-name
```

The `exec` command executes a list of interactive commands stored in a file.

---

help

```
help [command-name]
```

The `help` command displays help on a specified interactive command. If you do not specify a command name, the `help` command provides help on all interactive commands.

---

ls

```
ls
```

The `ls` command lists the lower-level instances contained in the current scope of the design. To change the hierarchical scope, use the `cd` command.

---

prc

```
prc [-out file-name]
```

The `prc` (print coverage) command displays a report on the fault coverage of instances in the current hierarchical scope. This report shows which blocks in your design have high coverage and which have low coverage.

This command reports statistics on seeded faults. Faults that were not seeded during the Verilog/PowerFault simulation (such as faults detected by a previous run) are not included in the fault coverage statistics.

---

prf

```
prf [-fmt (tmax|verifault|zycad)] [-full] [-times]
   [-out file-name]
```

The `prf` (print faults) command displays a report on the faults in the instances contained in the current hierarchical scope.

The output of this command is just like the default fault report file produced in batch mode, iddq.frpt, except that the `prf` command lists the status of faults beneath the current hierarchical scope, rather than all faults in the whole design.

The `prf` command complements the `prc` command. The `prc` command shows which blocks have low coverage, and the `prf` command shows which faults are causing the low coverage.

prs

```
prs [-out file-name]
```

The `prs` (print strobe) command displays the time value for every qualified IDDQ strobe. For each selected strobe, the number of incremental (additional new) faults detected by the strobe is also reported.

quit

```
quit
```

The `quit` command terminates IDDQPro.

reset

```
reset
```

The `reset` command clears the set of selected strobes and detected faults, allowing you to start over.

sela

```
sela sela-options*
sela-options ::=
   -cov_lim percent_cov |
   -strb_lim max_strobes |
   -out file-name
```

The `sela` (select automatic) command automatically selects strobes using a coverage-maximizing algorithm. This is the same selection algorithm IDDQPro uses in batch mode.

The `-cov_lim` and `-strb_lim` options work exactly like the command-line options described in "Strobe Selection Options".

The `-out` option redirects the output of the command to a specified file.

selm

```
selm strobe-times* selm-options*
strobe-times ::= [tb=testbench-number] simulation-time
selm-options ::= -in file-name | -out file-name
```

The `selm` (select manual) command lets you manually select strobes by specifying the strobe times. You can explicitly list the strobe times in the command line or read them from an input file using the `-in` option.

After you run this command, IDDQPro analyzes the strobe set and reports the results. To redirect the output to a file, use the `-out` option.

The `selm` and `sela` commands work together in an incremental fashion. Each time you use one of these commands, it adds the newly selected strobes to the list of previously selected strobes. This continues until the maximum possible coverage is achieved, after which no more

strobes can be selected. If the IDDQPro analysis determines that a manually selected strobe fails to detect any additional faults, the selection is automatically canceled.

For example, consider the following two commands:

```
selm 29900 39900
sela -strb_lim 6
```

The first command manually selects the two strobes at 29900 and 39900. The second command automatically selects six more strobes that complement the first two strobes and maximize the fault coverage.

To clear all strobe selections and start over, use the `reset` command.

If you are using multiple testbenches, you can select strobes from different testbenches. For example, the following command manually selects strobes at times 799 and 1299 in testbench 1 and the strobe at time 399 in testbench 2:

```
selm tb=1 799 tb=1 1299 tb=2 399
```

---

selall

`selall [-out` *`file-name`*`]`

The `selall` (select all) command automatically selects every qualified strobe, starting with the first strobe time and continuing until the maximum possible coverage is achieved or all qualified strobes are selected.

Although it is usually impractical to make so many measurements, the `-selall` command is useful because it determines the maximum possible coverage that can be obtained from your testbench or testbenches. If you use the `prf` command after the `selall` command, the resulting fault report identifies nets that never get toggled; they are reported as undetected.

Understanding the Strobe Report

A strobe report (iddq.srpt file) is generated when you run IDDQPro in batch mode and each time you select strobes in interactive mode. The following sections describe a strobe report:

Example Strobe Report

A strobe report lists the selected strobes in time order and shows the following information for each strobe:

- The simulation time
- The simulation cycle number
- The cumulative coverage achieved

- The cumulative number of faults detected
- The incremental (additional new) faults detected

The report gives you an idea of the effectiveness of each strobe. A large jump in coverage indicates a valuable strobe. A very small increase in coverage indicates a strobe with little value.

Here is an example of a strobe report:

```
# IDDQ-Test strobe report
# Date: day date time
#   Reached requested fault coverage.
#   Selected 6 strobes out of 988 qualified.
#   Fault Coverage (detected/seeded) = 90.3% (23082/25561)
# Timeunits 1.0ns
# Strobe: Time   Cycle  Cum-Cov  Cum-Detects Inc-Detects
         19990     2      48.3%    12346       12346
         329990    33     69.0%    17637       5291
         2109990  211     74.2%    18966       1329
         2129990  213     77.9%    19912       946
         2759990  276     85.7%    21906       1994
         2809990  281     90.3%    23082       1176
```

Fault Coverage Calculation

The fault coverage statistics in a strobe report include the following types of faults:

- Faults Detected by Previous Runs
- Undetected Faults Excluded From Simulation
- Faults Detected at Uninitialized Nodes

Faults Detected by Previous Runs

For example, the following report indicates that faults were detected by previous runs:

```
#   Reached requested fault coverage.
#   Selected 8 strobes out of 755 qualified.
#   Fault Coverage (detected/seeded) = 90.0% (90/100)
#   Faults detected by previous runs = 60
```

In this example, an existing fault list was read into the Verilog simulation with `read_tmax` or a similar command. That fault list had 60 faults that were already detected, either by an external tool such as Verifault or by a previous IDDQPro run. Therefore, the eight selected strobes only detected 30 more faults than the 60 that were already detected.

Undetected Faults Excluded From Simulation

The following report indicates that undetected faults were excluded from simulation:

```
#   Reached requested fault coverage.
#   Selected 4 strobes out of 2223 qualified.
```

```
#   Fault Coverage (detected/seeded) = 85.0% (170/200)
#   Undetected faults excluded from simulation = 20
```

The fault list read in by `read_tmax` or a similar command had 20 faults that were undetected but excluded. Perhaps the fault list covered the entire chip, but 20 faults were excluded from seeding at the I/O pads. The four selected strobes detected 170 faults and did not detect 30 faults. However, of the 30 undetected faults, only 10 were simulated by IDDQPro.

Faults Detected at Uninitialized Nodes

The following report indicates that faults were detected at uninitialized nodes:

```
#   Reached requested fault coverage.
#   Selected 5 strobes out of 2223 qualified.
#   Fault Coverage (detected/seeded) = 92.5% (370/400)
#   Faults detected at un-initialized nodes = 10
```

If an uninitialized node is driven to X (unknown rather than floating) during every selected vector, a strobe detects one stuck-at fault, either stuck-at-0 or stuck-at-1, because the node is driven to either 1 or 0. However, it is not known which type of fault is detected. The report indicates that 370 out of 400 faults were detected. Of the 370 detected faults, 10 have an unknown type, corresponding to the 10 nodes that were never initialized.

Adding More Strobes

After a Verilog/PowerFault simulation, you can use IDDQPro repeatedly to evaluate the effectiveness of different strobe combinations. It is not necessary to rerun the Verilog/PowerFault simulation each time.

You can use the strobes selected from an IDDQPro run as the initial strobe set for subsequent runs. For example, consider the following sequence of commands:

```
ipro -strb_lim 6 /cad/sim/M88/iddq
mv iddq.srpt stimes
ipro -strb_set stimes -strb_lim 8 /cad/sim/M88/iddq
```

The first command runs IDDQPro and selects six strobe points. The second command copies the strobe report file to a new file. The third command invokes IDDQPro again, using the strobe report from the first run as the initial strobe set, and selects two additional strobe points. After the second run, the strobe report file (iddq.srpt) contains eight strobe points, consisting of the six original strobes plus two new ones.

Deleting Low-Coverage Strobes

If you identify a strobe that provides very little additional coverage, you can delete it from the strobe report and run IDDQPro again to recalculate the coverage:

1. Run IDDQPro to select an initial set of strobes:

   ```
   ipro -strb_limit 8 iddq
   ```

2. Save the strobe report to a separate file:

   ```
   mv iddq.srpt stimes
   ```

3. Edit the new file and delete the strobe that provides the fewest incremental fault detections.
4. Run IDDQPro again, using the edited file for initial strobe selection:

   ```
   ipro -strb_limit 8 -strb_set stimes iddq
   ```

For best results, delete only one strobe at a time and run IDDQPro each time to recalculate the coverage. Coverage lost by deleting multiple strobes cannot be calculated by simple addition of the incremental coverage because of overlapping coverage.

Fault Report Formats

A fault report (iddq.frpt file) is generated when you run IDDQPro in batch mode and each time you use the `prf` command in interactive mode. The fault report lists all the seeded faults and their detection status.

You can choose the fault report format by using the `-prnt_fmt` option when you invoke IDDQPro. The format choices are the TetraMAX ATPG, Verifault, and Zycad formats. The default is TetraMAX ATPG.

The following sections describe the various fault report formats:

- TetraMAX Format
- Verifault Format
- Zycad Format
- Listing Seeded Faults

TetraMAX Fault Report Format

A fault report in TetraMAX format lists one fault descriptor per simulated fault. Each fault descriptor shows the type of fault, the fault status (DS=detected by simulation, NO=not observed), and the full net name (or two net names for a bridging fault).

Here is a section of a fault report in TetraMAX format:

```
sa0 DS tb.fadder.co
sa1 DS tb.fadder.co
sa0 DS tb.fadder.sum
sa1 DS tb.fadder.sum
```

The fault report shows five faults, all of which are detected by the selected strobes. All five faults involve nets in the `tb.fadder` module instance. The first four faults are stuck-at-0 and stuck-at-1 faults for the `co` and `sum` nets. The last fault is a bridge fault between the `x` and `ci` nets.

Verifault Fault Report Format

A fault report in Verifault format lists one fault descriptor per simulated fault. Each fault descriptor begins with the keyword `fault?`, followed by type of the fault, the full name of the net, and the fault status.

Here is a section of a fault report in Verifault format:

```
fault net sa0 tb.fadder.co 'status=detected';
fault net sa1 tb.fadder.co 'status=detected';
fault net sa0 tb.fadder.sum 'status=detected';
fault net sa1 tb.fadder.sum 'status=detected';
fault bridge wire tb.fadder.x tb.fadder.ci
'status=detected';
```

The fault report shows five faults, all of which are detected by the selected strobes. All five faults involve nets in the `tb.fadder` module instance. The first four faults are stuck-at-0 and stuck-at-1 faults for the `co` and `sum` nets. The last fault is a bridge fault between the `x` and `ci` nets.

Zycad Fault Report Format

In a fault report in Zycad format, there are three types of lines (other than comment lines): cell locations, stuck-at fault descriptors, and bridging fault descriptors.

A cell location line indicates the hierarchical scope for the following list of net names:

`@`*`module-instance`*

A stuck-at fault descriptor line indicates a net stuck-at 1 or stuck-at 0 fault:

`net-namestuck-value` `n (D|U) [`*`time-of-first-detect`*`]`

A bridging fault descriptor line indicates a bridging fault between two nets:

`net1-namenet2-name` `b (D|U) [`*`time-of-first-detect`*`]`

Here is a section of a fault report in the default Zycad format:

```
# IDDQ-Test fault report
#
# ALL fault origins
# Date: day date time
#
# path    net   short  type  result
# name    name  value       (D, U)
#
@tb.fadder
    co 0 n D
    co 1 n D
    sum 0 n D
```

```
sum 1 n D
x ci b D
```

The fault report shows five faults, all of which are detected by the selected strobes. All five faults involve nets in the `tb.fadder` module instance. The first four faults are stuck-at-0 and stuck-at-1 faults for the `co` and `sum` nets. The last fault is a bridge fault between the `x` and `ci` nets.

The report will look different from this example if you modify the default format using the `-prnt_full?`, `-prnt_times?`, or `-path_sep` option when you invoke IDDQPro. For details, see the descriptions of the `-prnt_full`, `-prnt_times`, and `-path_sep` options in "Invoking IDDQPro".

Listing Seeded Faults

The IDDQ database stores the faults seeded by the Verilog/PowerFault simulation in a compact binary format. Usually, you use IDDQPro to select strobes, calculate the fault coverage, and print a fault report that lists all the seeded faults along with their detection status. However, there might be times you want a list of the seeded faults without selecting strobes. For example, if there are no quiet strobe points to select, IDDQPro cannot generate the fault report.

To generate a list of seeded faults under these circumstances, start IDDQPro in interactive mode, and then use the `prf` command to generate a fault report, and redirect the output to a file:

```
ipro -inter iddq-database-name
prf -out iddq.frpt
quit
```

10

Using PowerFault Technology

The following sections provide information on using PowerFault simulation technology:

PowerFault Verification and Strobe Selection

You can use PowerFault simulation technology to perform the following IDDQ tasks:

- Verify TetraMAX IDDQ Patterns for Quiescence
- Select Strobes in TetraMAX Stuck-At Patterns
- Select Strobe Points in Externally Generated Patterns

Verifying TetraMAX IDDQ Patterns for Quiescence

When you use the TetraMAX IDDQ fault model, TetraMAX ATPG generates test patterns that have an IDDQ strobe in every pattern. When you write the patterns to a Verilog-format file, TetraMAX ATPG automatically includes the PowerFault tasks necessary for verifying quiescence at every strobe.

To verify TetraMAX IDDQ test patterns for quiescence, use the following procedure:

1. In TetraMAX ATPG, use the `write_patterns` command to write the generated test patterns in STIL format. For example, to write a pattern file called test.stil, you could use the following command:

   ```
   write_patterns test.stil -internal -format stil
   ```

2. Using MAX Testbench, create a Verilog testbench (for details, see "Using the stil2Verilog Command"). For example, to write a Verilog testbench called test.v you could use the following command:

   ```
   stil2Verilog test.stil test
   ```

3. If you want to specify the name of the leaky node report file, open the test pattern file in a text editor and search for all occurrences of the `status drivers leaky` command, and change the default file name to the name you want to use. This is the default command:

   ```
   // NOTE: Uncomment the following line to activate
   //  processing of IDDQ events
   // define tmax_iddq
   `$ssi_iddq("status drivers leaky top_level_name.leaky");
   ```

   Substitute your own file name as in the following example:

   ```
   `$ssi_iddq("status drivers leaky my_report.leaky");
   ```

   Save the edited test pattern file.

4. Run a Verilog/PowerFault simulation using the test pattern file.

The simulator produces a quiescence analysis report, which you can use to debug any leaky nodes found in the design.

Selecting Strobes in TetraMAX Stuck-At Patterns

Instead of generating test patterns specifically for IDDQ testing, you can use TetraMAX ATPG to generate ordinary stuck-at ATPG patterns and then use PowerFault simulation technology to choose the best strobe times from those patterns. To do this, you need to modify the Verilog testbench file to enable the simulator's IDDQ tasks.

This is the general procedure:

1. In TetraMAX ATPG, use the `write_patterns` command to write the generated test patterns in STIL format. For example, to write a pattern file called test.stil, you could use the following command:

   ```
   write_patterns test.stil -internal -format stil
   ```

2. Using MAX Testbench, create a Verilog testbench(for details, see "Using the stil2Verilog Command"). For example, to write a Verilog testbench called test.v you could use the following command:

   ```
   stil2Verilog test.stil test
   ```

3. Open the test pattern file in a text editor.
4. At the beginning of the file, find the following comment line:

   ```
   // `define tmax_iddq
   ```

   Remove the two forward slash characters to change the comment into a `` `define tmax_iddq `` statement. This enables the PowerFault tasks that TetraMAX ATPG has embedded in the testbench.

   Note: Instead of activating the `` `define tmax_iddq `` statement in the file, you can define `tmax_iddq` when you invoke the Verilog simulator. For example, when you invoke VCS, use the `+define+tmax_iddq=0+` option.
5. If you want to specify the name of the leaky node report file, search for all occurrences of the `status drivers leaky` command and change the default file name to the name you want to use. This is the default command:

   ```
   `$ssi_iddq("status drivers leaky top_level_name.leaky");
   ```

6. Save the edited test pattern file.
7. Run a Verilog simulation using the edited test pattern file.
8. Run the IDDQ Profiler.

When you run the Verilog/PowerFault simulation, the IDDQ system tasks evaluate each strobe time for fault coverage. When you run the IDDQ Profiler, it selects the best strobe times.

Selecting Strobe Points in Externally Generated Patterns

You can use PowerFault simulation technology to select strobes from testbenches generated by sources other than TetraMAX ATPG. The procedure depends on the testbench source:

- For test vectors generated by other ATPG tools, edit the testbench to add the PowerFault tasks.
- For functional (design verification) test vectors, edit the testbench to add the PowerFault tasks and determine timing for the tester vector. Use t-1, the last increment of time within a test cycle, for IDDQ strobes.
- For BIST (built-in self-test), control the clock with tester and determine timing for the tester vector. Use t-1 for IDDQ strobes.

To see how to edit the testbench to add PowerFault tasks, you can look at some Verilog testbenches generated by TetraMAX ATPG. For example, after the `initial begin` statement, you need to insert $ssi_iddq tasks to invoke the PowerFault commands:

```
initial begin
//Begin IddQTest initial block
  $ssi_iddq("dut adder_test.dut");
  $ssi_iddq("verb on");
  $ssi_iddq("seed SA adder_test.dut");
  $display("NOTE: Testbench is calling IDDQ PLIs.");
  $ssi_iddq("status drivers leaky LEAKY_FILE");
//End of IddQTest initial block
...
end
```

You also need to find the capture event and insert the PowerFault commands to evaluate a strobe at that point. For example:

```
event capture_CLK;
always @ capture_CLK begin
   ->forcePI_default_WFT;
   #140; ->measurePO_default_WFT;
   #110 PI[4]=1;
   #130 PI[4]=0;
//IddQTest strobe try
   begin
      $ssi_iddq("strobe_try");
      $ssi_iddq("status drivers leaky LEAKY_FILE");
   end
//IddQTest strobe try
end
```

Testbenches for IDDQ Testability

When you create a testbench outside of the TetraMAX ATPG environment, the following design principles can significantly improve IDDQ testability:

Separate the Testbench From the Device Under Test

For better IDDQ testability, maintain a clean separation of the testbench from the device under test (DUT). The Verilog DUT module should model only the structure and behavior of the chip. Put the chip-external drivers and pullups in the testbench. The testbench should also generate stimulus for the chip and verify the correctness of the chip's outputs.

Drive All Input Pins to 0 or 1

The mapping of testbench Xs to automated test equipment (ATE) drive signals is not well defined. The results depend on how the active load on the ATE is programmed. Because Xs can be mapped to VDD, VSS, or some intermediate voltage, such as (VDD-VSS)/2, avoid having your testbench drive Xs into the chip. PowerFault reports input pins driven to X as "possible float."

Try Strobes After Scan Chain Loading

To minimize simulation time and database size when you run a Verilog/PowerFault simulation, do not perform a `strobe_try` on every serialized scan load step. Instead, use `strobe_try` only after the entire scan chain is loaded.

If your simulation does a parallel scan load or you are using functional vectors, use `strobe_try` before the end of each cycle.

Include a CMOS Gate in the Testbench for Bidirectional Pins

If your chip has bidirectional I/O pins, place a CMOS gate inside the testbench to transmit the signal between the testbench driver and the I/O pad. For details, see "Use Pass Gates".

Model the Load Board

Take into account external connections to the DUT. When a chip is tested by ATE, it resides on a load board. The load board is a printed circuit board that provides the encapsulating environment in which the chip is tested. It can contain pullups/pulldowns, latches for three-state I/O pins, power/ground connections, and so on.

In general, your Verilog testbench should model the load board as accurately as possible. Any pullups/pulldowns/latches that would exist on the load board should be modeled in the testbench. In general, if a chip requires pullups to operate correctly in a real system, you can assume they are needed on the load board also.

Mark the I/O Pins

The top-level ports of each DUT module are assumed to be primary I/O ports and are given special treatment by PowerFault. If the testbench drives the DUT through other ports, use the `io` command to tell PowerFault about these ports. For information on the `io` command, see “io” in the " PowerFault PLI Tasks " section.

Minimize High-Current States

Try to minimize times when analog, RAM, and I/O cells are in current-draining states. Put them into standby mode when possible and write a complete set of test vectors for analog/RAM/IO standby mode.

Because IDDQ testing can be performed when the circuit is in a low-current state, try to minimize the number of vectors that put the circuit into high-current states. For maximum coverage, you might need to repeat the vectors that are normally applied during high-current states. For example, if your I/O pads have active pullups during some vectors, you can apply those same vectors again when the pullups are disabled, so that IDDQ testing can be performed on those vectors.

Maximize Circuit Activity

Try to toggle each node during low-current states. Some easy methods for achieving high circuit activity include:

- Shift alternating 0/1 patterns into scan registers.
- Apply alternating 0/1 patterns to data and address lines.

Combining Multiple Verilog Simulations

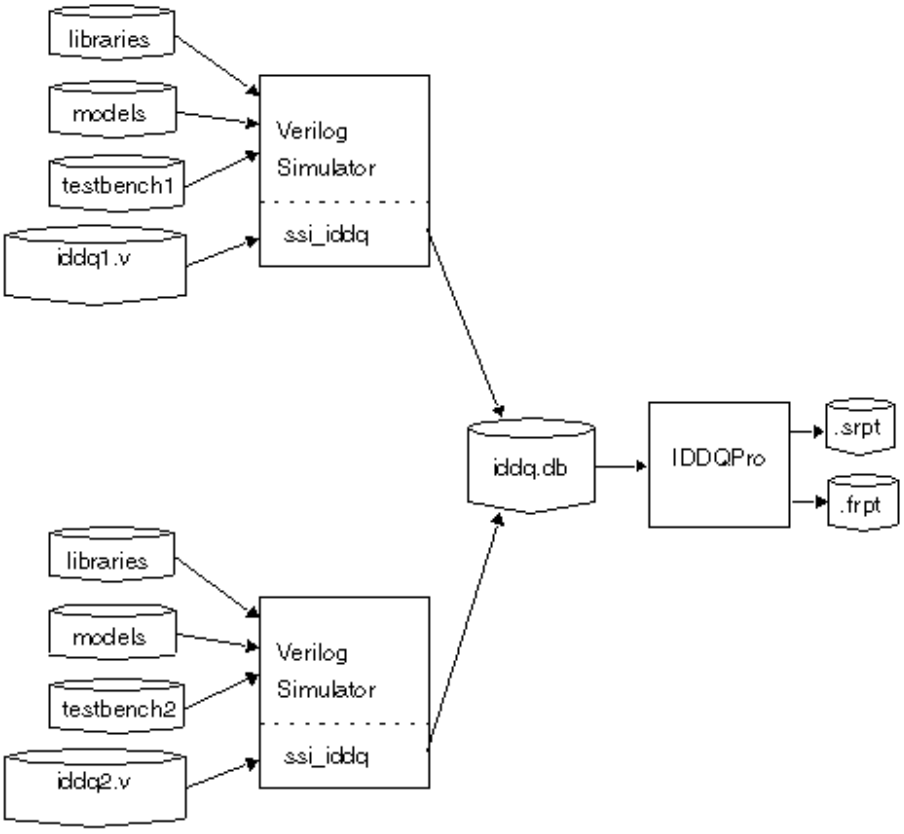
If you use different Verilog simulation runs to test different portions of a device or to drive a device into different states, you can use PowerFault technology to choose a set of strobe times for maximum fault coverage over all the resulting testbenches. For example, if there are 30

testbenches and your tester time budget allows only five IDDQ strobes, the five selected strobes ought to provide the best coverage out of all possible strobes in all 30 testbenches.

Note: If you want to improve coverage efficiency within a single testbench, see "Deleting Low-Coverage Strobes."

To combine multiple simulation results, you can merge the IDDQ information from each successive Verilog/PowerFault simulation into a single database. Then you can apply the IDDQ Profiler to that single database. This process is illustrated in Figure 1.

Figure 1 Using Multiple Testbenches



The following procedure is an example of a strobe selection session using two testbenches and a budget of five IDDQ strobes. The PowerFault PLI tasks for `testbench1` and `testbench2` are in files named `iddq1.v` and `iddq2.v?`, respectively.

1. In iddq1.v and iddq2.v, seed the entire set of faults, using either the `seed` command or `read` commands. For example:

```
$ssi_iddq( "seed SA iddq1.v" );

$ssi_iddq( "seed SA iddq2.v" );
```

2. In iddq1.v, use the `output create` command to save the simulation results to an IDDQ database named iddq.db?:

```
$ssi_iddq( "output create label=run1 iddq.db" );
```

3. In iddq2.v, use the `output append` command to append the simulation results to the database you created in Step 2:

   ```
   $ssi_iddq( "output append label=run2 iddq.db" );
   ```

4. Run a Verilog/PowerFault simulation using testbench1.v and iddq1.v?.
5. Run a Verilog/PowerFault simulation using testbench2 and iddq2.v?.
6. Run the IDDQ Profiler to select five good strobe points from the iddq.db database:

   ```
   ipro -strb_lim 5 iddq
   ```

A strobe report for multiple testbenches shows both the testbench number and simulation time within the respective testbench for each selected strobe. Testbench names and labels are listed in the header of the strobe report. Testbenches are numbered in sequence, starting with 1.

When you use multiple testbenches, the fault report files show only the comment lines from the first testbench. PowerFault does not try to merge the comment lines from the fault list in the second and subsequent testbenches with those in the first testbench.

Improving Fault Coverage

PowerFault does not require additional design-for-test (DFT) circuitry or modifications to your testbench, models, or libraries. It does not require configuration files, and it runs on any Verilog chip design.

If PowerFault is unable to find enough qualified strobes to provide satisfactory fault coverage, you might be able to find more qualified strobes by using the techniques described in the following sections:

- Determine Why the Chip Is Leaky
- Evaluate Solutions

Determine Why the Chip Is Leaky

The first step is to run the Verilog/PowerFault simulation to determine why the chip is leaky at strobe times. At each strobe try, PowerFault examines your chip for leaky states. If it finds any leaky states, it disqualifies the strobe point.

To check the leaky states for each strobe point, use the `status` command after the `strobe_try`?, as in the following example:

```
always begin
  fork
  # CLOCK_PERIOD;
  # (CLOCK_PERIOD -1)
     begin
     $ssi_iddq( "strobe_try" );
     $ssi_iddq( "status drivers leaky bad_nodes" );
    end
 join
```

```
   end
```

This example creates a file called bad_nodes that describes each leaky state at each strobe point. For example:

```
Time 3999
top.dut.vee[0] is leaky: Re: float
  HiZ <- top.dut.veePad0.out
top.dut.DIO[1] is leaky: Re: fight
  St0 <- top.dut.dpad1_cld
  St1 <- top.dut.dpad1_snd
  StX <- resolved value
```

For each `status` command, the simulator reports the simulation time and a list of leaky nodes. In the report, the full path name of each net is followed by a reason (such as `Re: float`?) and a list of drivers and their contribution to the net value. For example, in the preceding example, `top.dut.vee[0]` is floating because its lone driver (?`top.dut.veePad0`?) is in the high-impedance state.

For a complete description of the output of the `status` command, see "Status Command Output". For more information on leaky states, see "Leaky State Commands."

Evaluate Solutions

After you identify and understand the leaky states, you need to decide how to eliminate or ignore them so that you can change unqualified strobes into qualified ones. Use any of the following methods:

- Use the allow Command
- Configure the Verilog Testbench
- Configure the Verilog Models

Use the allow Command

The `allow` command can make PowerFault ignore leaky states that you know are not present in the real chip. For example, incomplete Verilog models can cause misleading leaky states that prevent PowerFault from qualifying strobe points. For more information, see "Leaky State Commands."

Configure the Verilog Testbench

In some cases, you can fix leaky states by modifying the Verilog testbench, as described in the following sections:

- Drive All Input Pins to 0 or 1
- Use Pass Gates
- Model the Load Board
- Mark the I/O Pins

Drive All Input Pins to 0 or 1

Make sure the testbench initializes all primary inputs. If your testbench drives Xs into the primary input pins of the device under test (DUT), PowerFault disqualifies the vector and flags those pins

as “possible float.” PowerFault takes the conservative position that Xs driven by the testbench might translate to the automated test equipment (ATE) turning off the drive signal and allowing the input pin to float.

If your ATE replaces Xs with a default drive value (either VDD or VSS), then driving Xs should be allowed. In that case, use the `allow float` command on all your input pins, as in the following example:

```
$ssi_iddq( "allow float testbench.chip.RE" );
$ssi_iddq( "allow float testbench.chip.ABUS[0]" );
$ssi_iddq( "allow float testbench.chip.ABUS[1]" );
```

Use Pass Gates

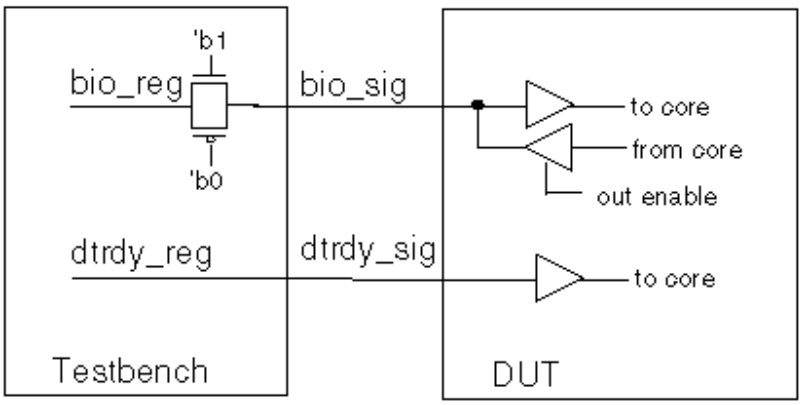
If your chip has bidirectional I/O pins, place a CMOS gate inside the testbench to transmit the signal between the testbench driver and the I/O pad.

The following code shows how two registers in the testbench are connected to signals that feed the DUT pins:

```
reg bio_reg, dtrdy_reg; // registers to hold stimulus
  // drive bidirectional "bio" signal through pass gate
wire bio_tmp = bio_reg;
cmos( bio_sig, bio_tmp,'b1,'b0);

  // drive input signal directly
wire dtrdy_sig = dtrdy_reg;

  // hookup signals to dut
dut dut( bio_sig, dtrdy_sig, ... );
```

Notice how the input signal `dtrdy_sig` is driven directly by the `dtrdy_reg` register, but the bidirectional signal `bio_sig` is driven through the `cmos` primitive, as shown in Figure 2.

Figure 2 Pass Transistor Between the Testbench and DUT



Model the Load Board

When a chip is tested by ATE, it resides on a load board. The load board is a printed circuit board that provides the encapsulating environment in which the chip is tested. It can contain pullups/pulldowns, latches for three-state I/O pins, power and ground connections, and so on.

In general, your Verilog testbench should model the load board as accurately as possible. Any pullups, pulldowns, and latches that would exist on the load board should be modeled in the testbench. In general, if a chip needs pullups to operate correctly in a real system, you can assume they are needed on the load board also.

Mark the I/O Pins

The top-level ports of a DUT module are assumed to be primary I/O ports and are given special treatment by PowerFault. If the testbench drives the DUT through other ports, use the `io` command to tell PowerFault about these ports. For information on the `io` command, see "PowerFault PLI Tasks."

Configure the Verilog Models

In general, the more your chip is modeled at a structural level (using gates, switches, and wires), the better for IDDQ testing. If your cells model logic behaviorally rather than with built-in Verilog primitives and user-defined primitives (UDPs), PowerFault might find fewer qualified strobe points. For details, see the following sections:

- Drive All Buses Possible
- Gate Buses That Cannot Be Driven
- Use Keeper Latches
- Enable Only One Driver
- Avoid Active Pullups and Pulldowns
- Avoid Bidirectional Switch Primitives

Drive All Buses Possible

Because floating buses can disqualify strobe points, try to always drive internal buses. Either configure the control logic to always enable one driver for the bus, or use keeper latches (holders).

For example, here is a bus that has two drivers that are fully multiplexed:

```
bufif1 ( addr0, X[0], sel );       // driver 1
bufif1 ( addr0, Y[0], sel_bar ); // driver 2
not ( sel_bar, sel );             // inverter
```

Gate Buses That Cannot Be Driven

If driving the bus is not always possible or desirable, gate the bus so that when it does float, the effect is blocked. For example, here is a bus that has two drivers and one load:

```
bufif1 ( addr0, X[0], x_en );                 // driver 1
bufif1 ( addr0, Y[0], y_en );                 // driver 2
or ( x_or_y_en, x_en, y_en );          // qualifier
and ( addr0_qualified, addr0, x_or_y_en );  // load
```

The bus value is blocked at the load (AND gate) when neither driver is active. If you want to use OR gates to block floating buses, use the `statedep_float` command. For more information on this command, see “statedep_float” in the "PowerFault PLI Tasks" section. For more information on blocking floating buses, see “State-Dependent Floating Nodes”.

Use Keeper Latches

If a bus cannot always be driven or gated, consider using keeper latches (also called “keepers”). A keeper retains the last value driven onto the bus. It has a weaker drive strength than normal bus drivers so that it can be overdriven.

Keepers should be modeled structurally. For example, here is a bus that has two drivers and one keeper:

```
bufif1 ( addr0, X[0], x_en );              // driver 1
bufif1 ( addr0, Y[0], y_en );              // driver 2
buf (pull0,pull1) ( addr0, addr0 );  // keeper
```

Avoid modeling keepers behaviorally or with continuous assignments:

```
wire (pull0,pull1) addr0 = addr0; // AVOID THIS
```

Use only strength-restoring gates such as `buf` for modeling keepers. Avoid using switch primitives (?`nmos`?, `pmos`?, `cmos`?) for modeling keepers:

```
rnmos( addr0, addr0, 'b1 ); // AVOID THIS
```

Enable Only One Driver

Because bus contention disqualifies strobe points, initialize all control logic (enabling lines) for bus drivers. Furthermore, if possible, configure the control logic to enable only one driver for the bus at a time.

Avoid Active Pullups and Pulldowns

Active pullups and pulldowns can also disqualify strobe points, so use keeper latches on three-state buses rather than pullups or pulldowns. PowerFault treats each of the following elements as a pullup or pulldown:

- `pullup` and `pulldown` primitives
- `tri1` and `tri0` nets
- `wand` and `wor` nets

Conflicting values on “wired AND” nets are reported as active pullups, and conflicting values on “wired OR” nets are reported as active pulldowns.

When you must use pullups or pulldowns, model them structurally like this:

```
wire n26;
pullup( n26 );
     OR
tri1 n26;
```

Avoid modeling pullups and pulldowns behaviorally or with continuous assignments, as in the following example:

```
wire (highz0,pull1) n26 = n26;   // AVOID THIS
```

Avoid Bidirectional Switch Primitives

Avoid using the `rtran?`, `rtranif1?`, and `rtranif0` primitives. If possible, replace them with `nmos?`, `pmos?`, or `cmos` primitives.

Floating Nodes and Drive Contention

PowerFault recognizes certain types of floating nodes and drive contention, and reports them according to their classification. The following sections describe floating nodes and drive contention:

- Floating Node Recognition
- Drive Contention Recognition

Floating Node Recognition

The following sections describe floating node recognition:

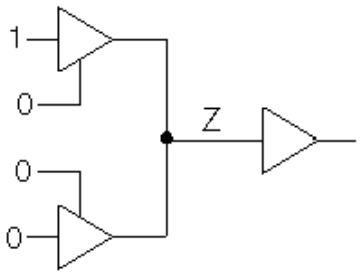
- Leaky Floating Nodes
- Floating Nodes Ignored by PowerFault
- State-Dependent Floating Nodes
- Configuring Floating Node Checks
- Floating Node Reports
- Nonfloating Nodes

Leaky Floating Nodes

PowerFault identifies thefollowing types of floating nodes as leaky:

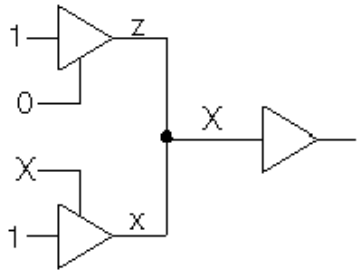
- **True floating node** — This is a node at Z, which does not have any active drivers, as shown in Figure 1.

  Figure 1 True Floating Node Example



- **Possibly floating node** — This is a node at X that might not have an active driver, as shown in Figure 2, or an undriven capacitive node. A capacitive node is a Verilog net with small, medium, or large strength.

Figure 2 Possibly Floating Node Example

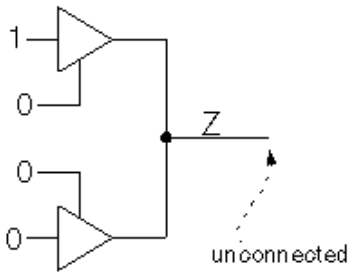


Floating Nodes Ignored by PowerFault

PowerFault ignores (does not report) these types of floating nodes:

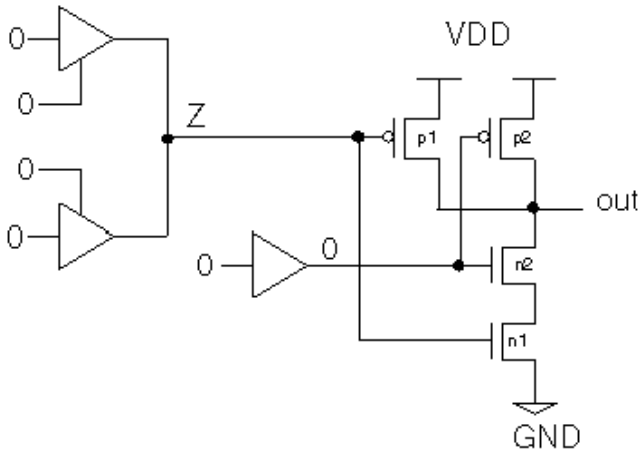
- **Floating node without a load** — This is a node that does not drive anything, as shown in Figure 3.

  Figure 3 Floating Node Without Load Example



- **State-dependent floating node** — This is a node that can be allowed to float because its effects are blocked by the states of other inputs, as shown in Figure 4.

  Figure 4 Blocked Floating Node Example



State-Dependent Floating Nodes

For AND, NAND, and NOR gates, the IDDQ effect of a floating input can be blocked by the other inputs. For example, if one input to a two-input NAND gate is floating but the other input is 0, the floating input is blocked so that it cannot cause a leakage current.

In Figure 4, the 0 input turns off transistor n2, so there is no conducting path from VDD to VSS through transistors p1 and n1. If the 0 input was 1 instead, PowerFault would identify the floating input as leaky.

By default, all inputs of 2-input and 3-input AND/NAND gates and 2-input NOR gates are treated as state-dependent floating nodes. By default, gates with more inputs and other types of gates are not allowed to have floating inputs. You can change the input limit for the AND, NAND, and NOR gates by using the `statedep_float` command. For more information, see "statedep_float" in "PowerFault PLI Tasks."

Configuring Floating Node Checks

Using the `allow` and `disallow` commands, you can configure how floating nodes are recognized. The `allow` command lets you do the following:

- Allow a particular node to float
- Allow all nodes to float
- Allow possible floating nodes (true floating nodes are still disallowed)

The `disallow` command lets you do the following:

- Disallow a Z on a particular node
- Disallow Zs on all nodes

For a complete description of the `allow` and `disallow` commands, see "PowerFault PLI Tasks."

Floating Node Reports

The `status leaky` command reports a list of floating nodes and nodes with drive contention. In order to save space, it reports only the floating node at the first strobe where the node is leaky. To get a report on all floating nodes (including those previously reported), use the `all_leaky` option with the `status` command. For example:

```
$ssi_iddq( "status drivers all_leaky bad_nodes" );
```

Nonfloating Nodes

To get a list of leaky nodes, use the following command:

```
$ssi_iddq( "status leaky" );
```

To get a list of nonleaky nodes, use the following command:

```
$ssi_iddq( "status nonleaky" );
```

This command reports a list of nodes that are not floating and do not have drive contention, together with the reason that each node was found to be nonleaky. This information can be useful when you think a node should be reported as floating, but it is not.

Drive Contention Recognition

PowerFault identifies the following types of drive contention:

- **Pullups and pulldowns** — For example, see the active pullup in Figure 5.
- **Contention between multiple bus drivers** — For example, see the true drive fight in Figure 6.

Figure 5 Active Pullup

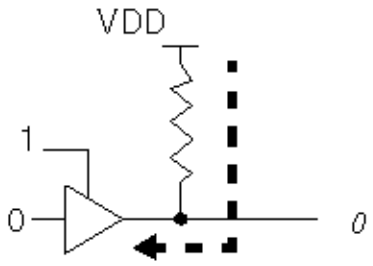
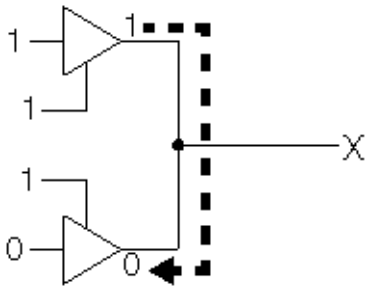
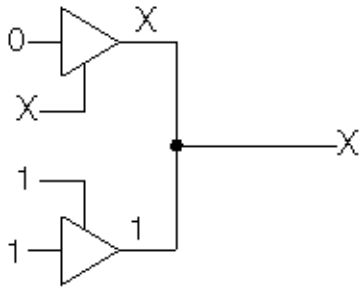


Figure 6 True Drive Fight



PowerFault makes a distinction between true and possible drive contention. A true fight occurs when a net has both a 0 (VSS) driver and a 1 (VDD) driver. A possible fight occurs when one or more drivers are at X on a bus with multiple drivers, as shown in Figure 7.

Figure 7 Possible Drive Fight



PowerFault also warns about unusual connections that indicate static leakage. The first time you execute the `status` command, it writes warning messages to the simulation log file about the following conditions:

- A node connected to both VSS (supply0) and VDD (supply1)
- A node connected to both a pullup and a pulldown

Status Command Output

The output of the `status` command can help you determine the cause of floating nodes and drive contention. Eliminating or reducing these types of leaky states not only makes your design more IDDQ-testable, it can also reduce the device power consumption.

The following sections describe the `status` command output:

- Overview
- Leaky Reasons
- Nonleaky Reasons
- Driver Information

Status Command Overview

The status command is executed during the Verilog/PowerFault simulation. It reports the nodes found to be leaky or nonleaky. For information on the command syntax, see “status” in "PowerFault PLI Tasks ".

The status of each node is reported in this format:

net-instance-name `is (leaky|non-leaky). Re:` *reason*

The instance name of each net is followed by a reason that explains why the node was found to be leaky or nonleaky. For example:

```
top.dut.TBIN is leaky: Re: float
top.dut.DIO is leaky: Re: possible float
```

The status command distinguishes between true and possible leaks. Possible leaks arise when nodes and drivers have unknown values (X). In the preceding example, `top.dut.TBIN` is truly floating (Z), whereas `top.dut.DIO` is possibly floating.

By default, the `status leaky` command reports only the first occurrence of a leaky node. When there are leaky nodes at a strobe, and all these leaky nodes have been reported at previous strobe times, the command prints the message “All reported.”

Leaky Reasons

The status command determines that a node is leaky for either a standard or user-defined reason. A standard reason is reported when the node is leaky due to a built-in quiescence check,

such as fight, float, pullup, or pulldown. A user-defined reason is reported when the node violates a condition specified by the `disallow` command.

Table 1 lists the standard leaky reasons and Table 2 lists the user-defined leaky reasons.

Table 1 Standard Leaky Reasons

Reason	Description
Fight	A drive fight between two or more drivers of equal strength. One driver is at 0 and another is at 1.
Pullup	An active pullup. A net with a pullup is being driven to 0. Any time a stronger driver at 0 is overriding a weaker driver at 1, the net is flagged as having an active pullup.
Pulldown	An active pulldown. A net with a pulldown is being driven to 1. Any time a stronger driver at 1 is overriding a weaker driver at 0, the net is flagged as having an active pulldown.
Float	A floating input node; an input node that is undriven (Z).
Possible Fight	A possible drive fight. One driver at X might be fighting with another driver (see Figure 7 in "Floating Nodes and Drive Contention").
Possible Pullup	A possible active pullup. A net with a pullup is being driven by an X. Any time a stronger driver at X is overriding a weaker driver at 1, the net is flagged as having a possible pullup.
Possible Pulldown	A possible active pulldown. A net with a pulldown is being driven by an X. Any time a stronger driver at X is overriding a weaker driver at 0, the net is flagged as having a possible pulldown.
Possible Float	A possible floating input node. The node is at X, but might have no active drivers (see Figure 2 in "Floating Nodes and Drive Contention").

Table 2 User-Defined Leaky Reasons

Reason	Description
Disallowed 0	A `disallow` command flags the net's present state (0) as leaky.

Reason	Description
Disallowed 1	A `disallow` command flags the net's present state (1) as leaky.
Disallowed X	A `disallow` command flags the net's present state (X) as leaky.
Disallowed Z	A `disallow` command flags the net's present state (Z) as leaky.
Disallow all Xs	A `disallow X` command flags the net's state (X) as leaky.
Disallow all Zs	A `disallow Z` command flags the net's present state (Z) as leaky.
Disallow all Caps	A `disallow Caps` command flags the net's present capacitive state as leaky.
Disallowed 0	A `disallow` command flags the net's present state (0) as leaky.
Disallowed 1	A `disallow` command flags the net's present state (1) as leaky.

A user-defined leaky reason appears when a node has a state specifically disallowed by a `disallow` command. For example:

```
$ssi_iddq( "disallow top.dut.SDD == 0" );
$ssi_iddq( "disallow Z" );
```

These two `disallow` commands produce a report like the following:

```
top.dut.SDD is leaky: Re: disallowed 0
top.dut.BIO is leaky: Re: disallow all Zs
```

In this example, `top.dut.SDD` is 0, which is disallowed by the first `disallow` command; and `top.dut.BIO` is Z, which is disallowed by the second `disallow` command.

Nonleaky Reasons

Table 3 lists the standard nonleaky reasons and Table 4 lists the user-defined nonleaky reasons.

Table 3 Standard Nonleaky Reasons

Reason	Description
0 or 1	The node is a quiet 0 or 1.

Reason	Description
Z no loads	The node is floating, but not connected to any inputs.
Z blocked	The node is floating, but is blocked (see Figure 4 in "Floating Nodes and Drive Contention").
X no contention	The node is driven to X (it is not floating) and has no contention; it is probably uninitialized.
Possible float no loads	The node is X and might be floating, but is not connected to any inputs.
Possible float blocked	The node is X and might be floating, but is blocked.

Table 4 User-Defined Nonleaky Reasons

Reason	Description
Allowable float	The node is (or possibly is) floating, but an allow command permits it.
Allowable fight	The node has (or possibly has) drive contention, but an `allow` command allows it.
Allow all fights	The node has (or possibly has) drive contention, but an `allow` command allows all contention.
Allow poss fights	The node possibly has drive contention, but an `allow` command allows possible contention.
Allow all floats	The node is (or possibly is) floating, but an `allow` command allows all floats.
Allow poss floats	The node is possibly floating, but an `allow` command allows all possible floats.

A user-defined nonleaky reason appears when a node has a state specifically allowed by an `allow` command. For example:

```
$ssi_iddq( "allow fight top.dut.PL" );
$ssi_iddq( "allow all float" );
```

These two `allow` commands can produce a report like the following:

```
top.dut.PL is non-leaky: Re: allowable fight
top.dut.BIO is non-leaky: Re: allow all floats
```

Driver Information

To determine why a net is floating or has drive contention, its drivers must be examined. Simulation debuggers and even some system tasks (such as the `$showvar` task in the Verilog simulator) can perform this examination. You can also use the `drivers` option of the status command, but this option generates only gate-level driver information.

The `drivers` option causes the status command to print the contribution of each driver. For example:

```
$ssi_iddq( "status drivers leaky bad_nodes" );
can produce output like:
top.dut.mmu.DIO is leaky: Re: fight
  St0 <- top.dut.mmu.UT344
  St1 <- top.dut.mmu.UT366
  StX <- resolved value
top.dut.mmu.TDATA is leaky: Re: float
  HiZ <- top.dut.mmu.UT455
  HiZ <- top.dut.mmu.UT456
```

In this example, `top.dut.mmu.DIO` has a drive fight. One driver is at strong 0 (?`St0`?) and the other at strong 1 (?`St1`?). The contributing value of each driver is printed in Verilog strength/value format, as described in section 7.10 of the IEEE 1364 Verilog LRM.

The same status command without the `drivers` option produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
top.dut.mmu.TDATA is leaky: Re: float
```

Driver Information

To determine why a net is floating or has drive contention, its drivers must be examined. Simulation debuggers and even some system tasks (such as the `$showvar` task in the Verilog simulator) can perform this examination. You can also use the `drivers` option of the status command, but this option generates only gate-level driver information.

The `drivers` option causes the status command to print the contribution of each driver. For example:

```
$ssi_iddq( "status drivers leaky bad_nodes" );
can produce output like:
top.dut.mmu.DIO is leaky: Re: fight
  St0 <- top.dut.mmu.UT344
  St1 <- top.dut.mmu.UT366
  StX <- resolved value
top.dut.mmu.TDATA is leaky: Re: float
  HiZ <- top.dut.mmu.UT455
  HiZ <- top.dut.mmu.UT456
```

In this example, `top.dut.mmu.DIO` has a drive fight. One driver is at strong 0 (?`St0`?) and the other at strong 1 (?`St1`?). The contributing value of each driver is printed in Verilog strength/value format, as described in section 7.10 of the IEEE 1364 Verilog LRM.

The same status command without the `drivers` option produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
top.dut.mmu.TDATA is leaky: Re: float
```

Behavioral and External Models

PowerFault examines the structure of your Verilog HDL model to determine whether the chip is quiescent. PowerFault looks for bus contention, floating inputs, active pullups, and other current-drawing states.

If you use behavioral models or external models (like LMC, LAI, or VHDL cosimulated models) to simulate subblocks of the chip, PowerFault cannot to determine when those subblocks are quiescent. As a result, it might select strobe points that are inappropriate for IDDQ testing. To prevent this from happening, use the `disallow` command.

The following sections describe the `disallow` command in more detail:

- Disallowing Specific States
- Disallowing Global States

Disallowing Specific States

The`disallow` command is a flexible command that lets you describe the leaky states for all instances of a behavioral or external model. One or more commands can describe which input, output, or internal states correspond to nonquiescence.

For example, the three following `disallow` commands describe when instances of the BRAM and DAC entities are leaky:

```
$ssi_iddq( "disallow BRAM (REFRESH == 1 && ENABLE == 0)" );
$ssi_iddq( "disallow BRAM (WRITE_EN == 1 || READ_EN == 1)" );
$ssi_iddq( "disallow DAC (port.0 != 0 && port.1 != 0)" );
```

Disallowing Global States

You can use the `disallow` command to disallow all nets in the Verilog simulation from having a particular value. This is useful if the libraries contain behavioral gate models. For example, if the three-state buffers are not modeled with Verilog primitives or UDPs, then PowerFault might not be able to detect bus contention.

Here is an example of a three-state buffer modeled behaviorally:

```
module BUF0 (out, data, control);
output out;
input data, control;
wire out = ( control == 0 ) ? data : 'bZ;
endmodule
```

To prevent bus contention during an IDDQ strobe, you can disallow all Xs with this command:

```
$ssi_iddq( "disallow X" );
```

If disallowing all Xs is too pessimistic, you can use a specific `disallow` command for each three-state buffer entity. For example, if you have two types of three-state buffers, BUF0 and BUF1, use the following commands:

```
$ssi_iddq( "disallow BUF0 ( out == X )" );
$ssi_iddq( "disallow BUF1 ( out == X )" );
```

If the libraries contain behavioral gate models, PowerFault might not be able to detect floating buses (buses with all drivers turned off). To prevent floating buses during an IDDQ strobe, you can disallow all Zs with this command:

```
$ssi_iddq( "disallow Z" );
```

If disallowing all Zs is too pessimistic, you can use a `disallow` command for each three-state buffer entity. For example, you could use the following commands:

```
$ssi_iddq( "disallow BUF0 ( out == Z )" );
$ssi_iddq( "disallow BUF1 ( out == Z )" );
```

For more information on the `disallow` command, see "Leaky State Commands."

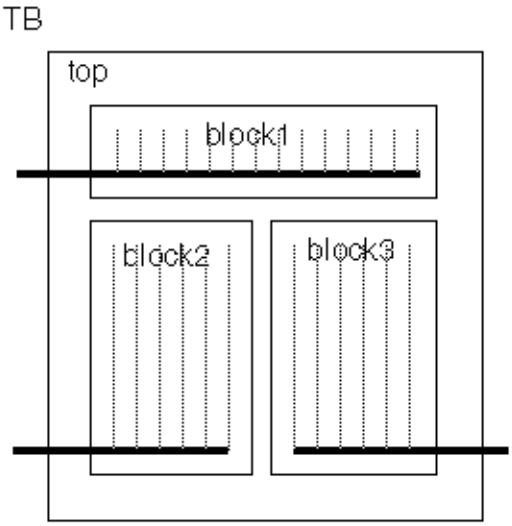
Multiple Power Rails

This section describes how to apply PowerFault to a chip with multiple power rails, where each power rail feeds a separate logic block on the chip. The overall strategy is as follows:

1. Determine the number of IDDQ test points for each block.
2. For one block, run a Verilog/PowerFault simulation, seeding only the faults in that block; and use IDDQPro to select strobes for the block.
3. Repeat step 2 for each block in the design. Exclude any strobes that have already been selected for previous blocks.
4. To determine the fault coverage for each block using the full set of strobes, run IDDQPro separately on each database, manually selecting all strobes selected in steps 2 and 3.

Here is an example. Suppose you have a chip with three power rails, as shown in Figure 1.

Figure 1 Chip With Three Power Rails



Step 1

Select two IDDQ strobes for each block.

Step 2

Run a Verilog simulation, seeding faults only in block1. The Verilog simulation produces a database named db1 (see Figure 2). You then use IDDQPro to automatically select two strobes from the database and save the strobe report in accum.strobes (see Figure 3):

```
ipro -strb_lim 2 -prnt_nofrpt db1
mv iddq.srpt accum.strobes
```

Figure 2 Create a Database for Block 1

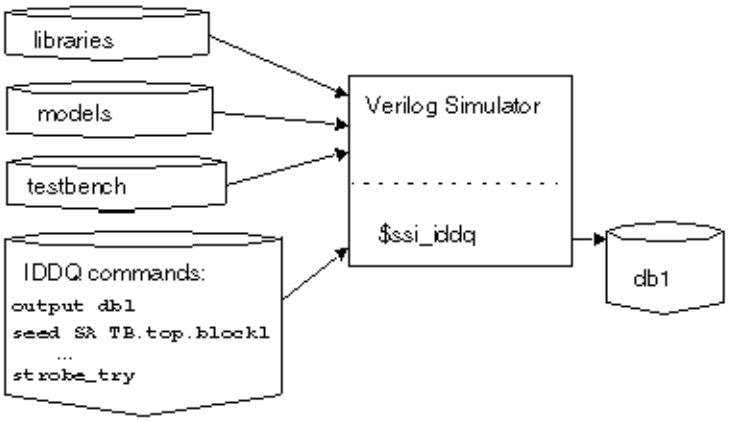
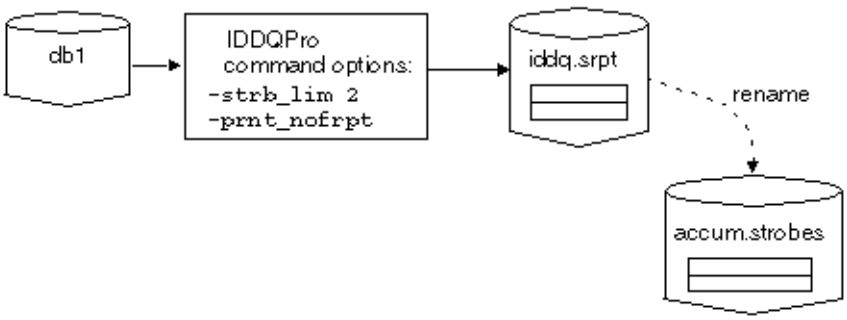


Figure 3 Select Two Strobes for Block 1



Step 3

Run the next Verilog simulation, this one seeding faults only in block2. The Verilog simulation produces a database named db2?. You then use IDDQPro to automatically select two strobes from db2 and append the two strobes to accum.strobes (see Figure 4 and Figure 5):

```
ipro -strb_lim 2 -strb_unset accum.strobes -prnt_nofrpt db2

cat iddq.srpt >> accum.strobes
```

Figure 4 Create a Database for Block 2

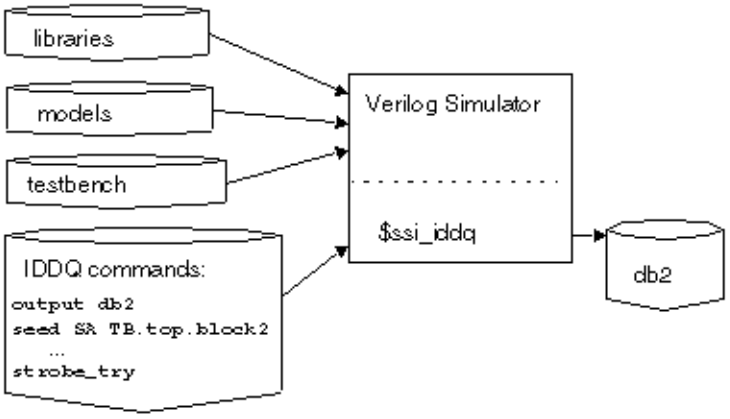
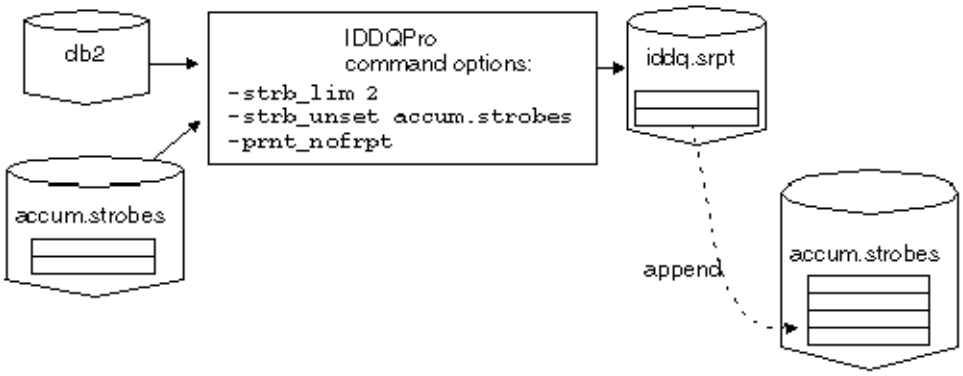


Figure 5 Select Two Strobes for Block 2



To complete step 3, you run the last Verilog simulation, this one seeding faults only in block3. The Verilog simulation produces a database named db3. You then use IDDQPro to automatically select two strobes from db3 and append the two strobes to accum.strobes?:

```
ipro -strb_lim 2 -strb_unset accum.strobes -prnt_nofrpt db3
cat iddq.srpt >> accum.strobes
```

The accum.strobes file now has six strobes (two for each block). The strobes you selected for any one block might be qualified for the other two blocks, so in step 4 you will try to select all six strobes.

Step 4

To begin step 4, you run IDDQPro to manually select six strobes from db1?. You select the strobes stored in accum.strobes and save the resulting strobe and fault reports:

```
ipro -strb_lim 6 -strb_set accum.strobes db1
mv iddq.srpt iddq.srpt1
mv iddq.frpt iddq.frpt1
```

Continuing step 4, you run IDDQPro to manually select six strobes from db2?. You select the strobes stored in accum.strobes and save the resulting strobe and fault reports:

```
ipro -strb_lim 6 -strb_set accum.strobes db2
mv iddq.srpt iddq.srpt2
mv iddq.frpt iddq.frpt2
```

To finish step 4, you repeat the same procedure using db3?:

```
ipro -strb_lim 6 -strb_set accum.strobes db3
mv iddq.srpt iddq.srpt3
mv iddq.frpt iddq.frpt3
```

Conclusion

After step 4 is complete, you have selected a total of six strobes (two for each block). The three individual strobe reports describe the fault coverage of the six strobes for each of the three blocks. The three individual fault reports describe the detected faults for each of the three blocks.

Testing I/O and Core Logic Separately

PowerFault looks at the chip as a whole. By default, everything in the DUT module, including I/O pads, must be quiescent to qualify a strobe point for IDDQ testing.

If the I/O pads and core logic have separate power rails, you can probably increase fault coverage by testing the core logic separately. This is because you can test the core at times when the I/O pads are leaky, assuming that you are able to measure IDDQ just for the core logic (ignoring the current drawn by the I/O pads).

To qualify strobes just for the core logic, use the `allow` command to ignore floating I/O pins and drive contention at I/O pins. This command makes PowerFault ignore all leaky states at the I/O pads. Also use the `exclude` command to prevent faults from being seeded inside the I/O pads.

Here is an example:

```
$ssi_iddq( "allow float top.dut.clk33_pad" );
$ssi_iddq( "allow fight top.dut.clk33_pad" );
```

```
$ssi_iddq( "exclude top.dut.clk33_pad" );
$ssi_iddq( "allow float top.dut.dto_pad" );
$ssi_iddq( "allow fight top.dut.dto_pad" );
$ssi_iddq( "exclude top.dut.dto_pad" );
```

11

Strobe Selection Tutorial

After you install the Synopsys IDDQ option to TetraMAX ATPG, you can do the Strobe Selection Tutorial to test the installation and to get an introduction to PowerFault strobe selection.

Note: This tutorial is intended to be a brief demonstration, not a comprehensive training session.

The following sections guide you through the Strobe Selection Tutorial:

Simulation and Strobe Selection

The `$IDDQ_HOME/samples` directory contains some examples of designs and scripts to demonstrate PowerFault capabilities. One example is a simple one-bit full adder. In the following set of tutorial procedures, you will run a script that simulates the testbench and selects a set of IDDQ strobe times in the testbench:

- Examine the Verilog File
- Run the doit Script
- Examine the Output Files

Examine the Verilog File

The following steps show you how to examine the Verilog design file.

1. Change to the directory `$IDDQ_HOME/samples/fadder`?.
2. Look for two files in the directory: the `doit` script and the `fadder.v` Verilog file.
3. Using any text editor, view the contents of the `fadder.v` file.

The `fadder.v` Verilog file contains three modules: `testbench`?, `iddqtest`?, and `fadder`?.

The `testbench` module is the testbench for the full adder. It tests every possible input pattern, from b000 through b111, and prints out the port values at one time unit before the end of each cycle.

The `iddqtest` module invokes the PLI tasks for IDDQ analysis. It contains the following $ssi_iddq commands:

```
$ssi_iddq( "dut testbench.fadder" );
$ssi_iddq( "seed SA testbench.fadder" );
// strobe 1 time unit before end of cycle
forever begin
    # (testbench.CYCLE - 1)
        $ssi_iddq( "strobe_try" );
    # 1;
end
```

The first command defines the device under test to be `testbench.fadder`?. The second one seeds stuck-at faults throughout the entire device. The third one performs IDDQ strobe evaluation one time unit before the end of each cycle.

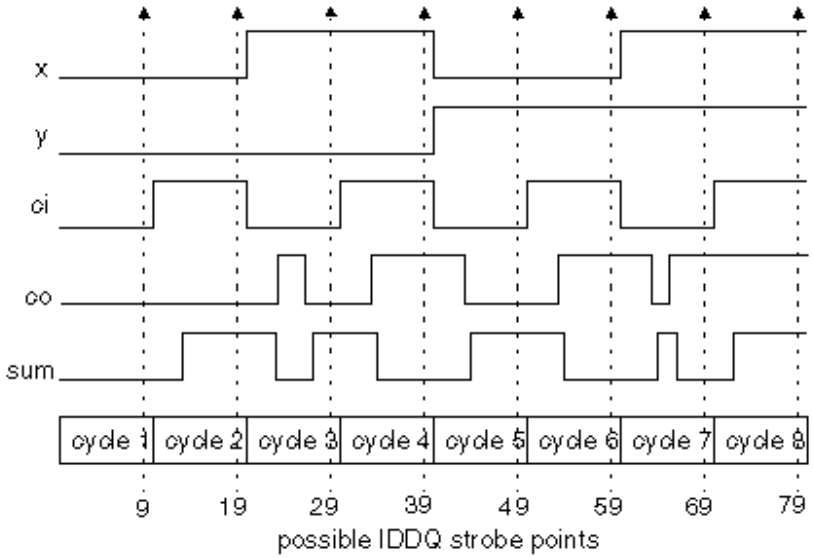
The `fadder` module is a gate-level description of the device under test, a single-bit full adder implemented with NOR gates. Each gate has a unit delay. Given two input bits (x and y) and a carry-in bit (ci), the full adder computes the sum bit and the carry-out (co) bit. The model implements the following Boolean equations:

co = (x & y) | (x & ci) | (y & ci)

sum = x ^ y ^ ci

Figure 1 shows the stimulus, response, and IDDQ strobe points for the full adder simulation.

Figure 1 Full Adder Simulation Strobe Points



Run the doit Script

The following steps show you how to run the `doit` script, which runs the Verilog/Powerfault simulation and IDDQ Profiler.

1. Using any text editor, view the contents of the `doit` (do it) file. This is a script that creates a directory for the simulator output, invokes the Verilog simulator (with IDDQ PLI tasks), and runs the IDDQ Profiler to select the strobe times.
2. If necessary, edit the file to work with your system configuration. For example, if your simulator is invoked by a command other than `vcs` or `Verilog`?, modify the line that invokes the simulator.
3. Run the script.

The script runs the Verilog simulation, which produces the following results:

```
time co sum {x,y,ci}
   9  0   0   000
  19  0   1   001
  29  0   1   010
  39  1   0   011
  49  0   1   100
  59  1   0   101
  69  1   0   110
  79  1   1   111
```

The `$ssi_iddq` tasks produce the following summary report:

```
IDDQ-Test
Strobes (qualified/tested) =  8/8
Faults seeded (stuck-ats/bridges) = 32/0
Created IDDQ database: iddq
```

This report tells you that eight strobes were tested, and all eight were found to be quiescent.

The script then invokes the IDDQ Profiler, which selects some of the eight quiescent strobes. It generates two files: a strobe report named `iddq.srpt` and a fault report named `iddq.frpt`?. The script then tells you the path to the output files.

```
Loading seeds
Beginning strobe selection
...
Strobe selection complete
Strobe report is printed to iddq.srpt
Fault report is printed to iddq.frpt
```

Examine the Output Files

The following steps show you how to examine the report files.

1. Go to the directory containing the `fadder` output files. Find the subdirectory called `iddq`?, which contains the IDDQ database generated by the `$ssi_iddq` PLI tasks, and the two IDDQ Profiler output files, `iddq.srpt` and `iddq.frpt`?.
2. Examine the contents of the strobe report file, `iddq.srpt`?. You should see the following report:

   ```
   # Date: day/date/time
   #  Reached requested fault coverage.
   #  Selected 3 strobes out of 8 qualified.
   #  Fault Coverage (detected/seeded) = 100.0% (32/32)
   # Timeunits: 1.0ns
   #
   # Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
             9   1       50.0%  16           16
            39   4       84.4%  27           11
            49   5      100.0%  32           5
   ```

   The report shows the requested level of fault coverage, 100 percent, was achieved by three strobes. A table shows the time values and cycle numbers of the selected strobes, the cumulative fault coverage achieved by each successive strobe, the cumulative number of faults detected with each successive strobe, and the incremental (additional) faults detected with each successive strobe.
3. Examine the contents of the fault report file, `iddq.frpt`?. The report shows the list of faults and the test result for each fault:

   ```
   sa0 DS .testbench.fadder.co
   sa1 DS .testbench.fadder.co
   sa0 DS .testbench.fadder.sum
   sa1 DS .testbench.fadder.sum
   sa0 DS .testbench.fadder.x
   sa1 DS .testbench.fadder.x
   sa0 DS .testbench.fadder.y
   sa1 DS .testbench.fadder.y
   sa0 DS .testbench.fadder.ci
   ```

```
sa1 DS .testbench.fadder.ci
sa0 DS .testbench.fadder.u12_out
sa1 DS .testbench.fadder.u12_out
sa0 DS .testbench.fadder.u10_out
sa1 DS .testbench.fadder.u10_out

...

sa0 DS .testbench.fadder._x
sa1 DS .testbench.fadder._x
sa0 DS .testbench.fadder._y
sa1 DS .testbench.fadder._y
```

The test result for each fault is either DS (detected by simulation) or NO (not observed). In this case, all faults were detected. Each fault is identified by fault type (sa0 = stuck-at-0, sa1 = stuck-at-1) and the hierarchical net name.

Interactive Strobe Selection

In the previous steps of this tutorial, you used the IDDQ Profiler in batch mode, which is the default operating mode. In this mode, the IDDQ Profiler selects a set of strobes and attempts to obtain the requested fault coverage with the fewest possible strobes.

You can also use the IDDQ Profiler in interactive mode to perform strobe and fault coverage analysis. In a typical interactive session, you select a set of strobes, print a strobe report and a fault coverage report for that set of strobes, and then repeat this process for different sets of strobes. You can examine the status of all faults or just the faults within a specified hierarchical scope.

The following sections guide you through the interactive strobe selection portion of this tutorial:

- Select Strobes Automatically
- Select All Strobes
- Select Strobes Manually
- Cumulative Fault Selection

Select Strobes Automatically

The following steps show you how to use the IDDQ Profiler to automatically select the strobes in a single step:

1. In the directory containing the fadder output files, execute the following command:

   ```
   % ipro -inter iddq
   ```

   The `ipro -inter` command invokes the IDDQ Profiler in interactive mode, and the `iddq` argument specifies the name of the IDDQ database to use for the interactive session.

2. At the IDDQ Profiler prompt (>), enter the “select automatic” command:

   ```
   > sela
   ```

   This command invokes the same strobe selection algorithm used in batch mode. The IDDQ Profiler responds as follows:

   ```
   ...
   #  Reached requested fault coverage.
   #  Selected 3 strobes out of 8 qualified.
   #  Fault Coverage (detected/seeded) = 100.0% (32/32)
   # Timeunits: 1.0ns
   #
   # Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
             9   1       50.0%  16           16
            39   4       84.4%  27           11
            49   5      100.0%  32           5
   ```

   The list of selected strobes is the same as in batch mode.

3. Enter the “print coverage” command:

   ```
   > prc
   ```

   The IDDQ Profiler responds as follows:

   ```
   Fault coverage for top modules

   Instance NumDet NumFaults %Coverage (stuck-at bridge)

   testbench 32 32 100.0%   (32/32 0/0)
   ```

   For the current set of selected strobes, 32 out of 32 faults are detected, and coverage is 100 percent.

4. Enter the “print faults” command:

   ```
   > prf
   ```

   The IDDQ Profiler produces the same fault report that you saw earlier in the `iddq.frpt` file:

   ```
   sa0 DS .testbench.fadder.co
   sa1 DS .testbench.fadder.co
   ...
   sa0 DS .testbench.fadder._y
   sa1 DS .testbench.fadder._y
   ```

5. Enter the “reset” command:

   ```
   > reset
   ```

   This command clears the set of selected strobes and detected faults.

Select All Strobes

The following steps show you how to manually select all possible strobes.

1. Enter the “select all” command:

   > **selall**

   The IDDQ Profiler responds as follows:

```
#  Selected all qualified strobes.
#  Selected 5 strobes out of 8 qualified.
#  Fault Coverage (detected/seeded) = 100.0% (32/32)
# Timeunits: 1.0ns
#
# Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
           9   1       50.0%  16           16
          19   2       62.5%  20           4
          29   3       84.4%  27           7
          39   4       90.6%  29           2
          49   5      100.0%  32           3
```

   All qualified strobes were selected in sequence, starting with the first strobe at time=9, until the target coverage of 100 percent was achieved. Five strobes were required, rather than the three selected by the `sela` (select automatic) command.

2. Reset the strobe selection and detected faults:

   > **reset**

Select Strobes Manually

The following steps show you how to select strobes manually.

1. Enter the following “select manual” command to manually select a single strobe at time=39:

   > **selm 39**

   The IDDQ Profiler responds as follows:

```
#  Selected 1 strobes out of 8 qualified.
#  Fault Coverage (detected/seeded) = 50.0% (16/32)
# Timeunits: 1.0ns
#
# Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
          39   4       50.0%  16           16
```

   This single strobe detected 16 faults, providing coverage of 50 percent.

2. To find out which faults have not yet been detected, enter the “print faults” command:

   > **prf**

   You should see the following response:

```
sa0 DS .testbench.fadder.co
sa1 NO .testbench.fadder.co
sa0 NO .testbench.fadder.sum
sa1 DS .testbench.fadder.sum
```

```
...
sa0 DS .testbench.fadder._x
sa1 NO .testbench.fadder._x
sa0 NO .testbench.fadder._y vsa1 DS .testbench.fadder._y
```

The second column shows `DS` for "detected by simulation" or `NO` for "not observed."

3. Enter the following command to see a list of modules:

   > **ls**

   The IDDQ Profiler responds as follows:

   ```
   ls
   testbench
   ```

   This simple model has only one level of hierarchy. In a multilevel hierarchical model, you can change the scope of the design view by using the `ls?`, `cd` *`module_name`*`?`, and `cd ..` commands. When you use the `prf` command, only the faults residing within the current scope (in the current module and below) are reported. Similarly, a coverage report generated by the `prc` command applies only to the current scope.

4. Enter the following command to manually select another strobe at time=49:

   > **selm 49**

   The IDDQ Profiler responds as follows:

   ```
   #  Selected 2 strobes out of 8 qualified.
   #  Fault Coverage (detected/seeded) = 87.5% (28/32)
   # Timeunits: 1.0ns
   #
   # Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
             39   4       50.0%  16           16
             49   5       87.5%  28           12
   ```

   The IDDQ Profiler adds each successive strobe selection to the previous selection set. The report shows the cumulative coverage and cumulative defects detected by each successive strobe.

5. Look at the fault list:

   > **prf**

6. Reset the strobe selection and detected faults:

   > **reset**

Cumulative Fault Selection

The following steps show you how to combine manual and automatic selection techniques:

1. Manually select the two strobes at time=19 and time=29:

   > **selm 19 29**

   The IDDQ Profiler responds as follows:

   ```
   #  Selected 2 strobes out of 8 qualified.
   ```

```
#  Fault Coverage (detected/seeded) = 78.1% (25/32)
# Timeunits: 1.0ns
#
# Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
          19   2       50.0%  16           16
          29   3       78.1%  25           9
```

2. Enter the “select automatic” command:

   > **sela**

   The IDDQ Profiler responds as follows:

```
#  Reached requested fault coverage.
#  Selected 4 strobes out of 8 qualified.
#  Fault Coverage (detected/seeded) = 100.0% (32/32)
# Timeunits: 1.0ns
#  v# Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
v          19   2       50.0%  16           16
          29   3       78.1%  25           9
          59   6       96.9%  31           6
          69   7      100.0%  32           1
```

   The `sela` command keeps the existing selected strobes and applies the automatic selection algorithm to the remaining undetected faults. In this case, four strobes were required to achieve 100 percent coverage.

3. Reset the strobe selection and detected faults:

   > **reset**

4. Continue to experiment with the commands you have learned. For help on command syntax, use the `help` command:

   > **help**

   or

   > **help *command_name***

5. When you are done, exit with the `quit` command:

   > **quit**

12

Interfaces to Fault Simulators

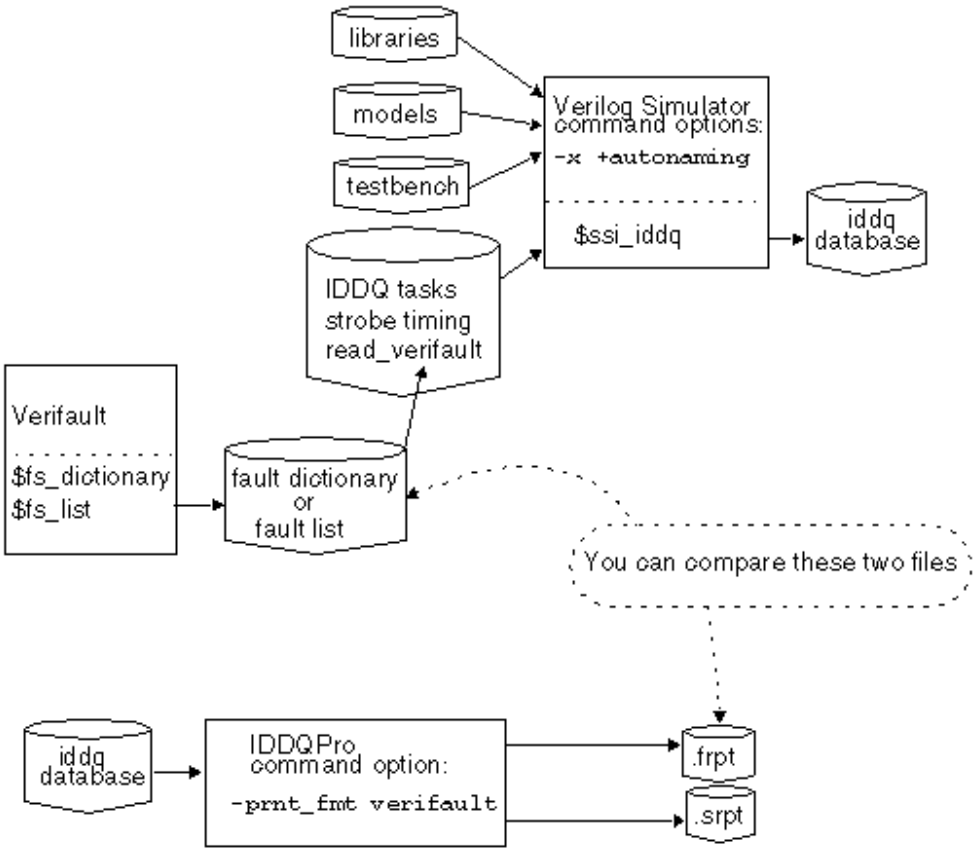
PowerFault is compatible with the Verifault and Zycad fault simulators. You can read the fault lists generated by these tools into PowerFault.

The following sections describe the interfaces to these fault simulators:

Verifault Interface

You can seed a design with faults taken from a Verifault fault list. Figure 1 shows the data flow for this type of fault seeding.

Figure 1 Data Flow for Verifault Interface



To seed faults from Verifault fault dictionaries and fault lists, use the `read_verifault` command in the Verilog/PowerFault simulation, as described in “read_verifault” in the " PowerFault PLI Tasks" section. By default, PowerFault remembers all the comment lines and unseeded faults in the Verifault file, so that when it produces the final fault report, you can easily compare the report to the original file.

When you use the `read_verifault` command to seed fault descriptors generated by Verifault, and your simulator is Verilog-XL, use the `-x` and `+autonaming` options when you start the simulation:

```
Verilog -x +autonaming iddq.v ...
```

Otherwise, the `read_verifault` command might not be able to find the nets and terminals referenced by your fault descriptors.

By default, the `read_verifault` command seeds both prime and nonprime faults. When you run IDDQPro after the Verilog simulation to select strobes and print fault reports, all fault coverage statistics produced by IDDQPro include nonprime faults. If you want to see statistics for only prime faults, seed only those faults. For example, you can create a fault list with just prime faults and use that list with the `read_verifault` command.

By default, IDDQPro generates fault reports in TetraMAX format. To print a fault report in Verifault format, use the `-prnt_fmt verifault` option:

```
ipro -prnt_fmt verifault -strb_lim 5 iddq-database-name
```

When you use multiple testbenches, the fault report files show only the comment lines from the first testbench. PowerFault does not try to merge the comment lines from the fault list in the second and subsequent testbenches with those in the first testbench.

If you mix fault seeds from other formats, like using the `read_zycad` command to seed faults from a Zycad .fog file, the Zycad faults detected in previous iterations are counted in the coverage statistics but are not printed in the fault report.

Zycad Interface

To seed faults from Zycad .fog files, use the `read_zycad` command in the Verilog/PowerFault simulation, as described in “read_zycad” in the "PowerFault PLI Tasks " section. By default, PowerFault does the following:

- Remembers all the comment lines and unseeded faults in the .fog file, so that when it produces the final report, you can easily compare the report to the original file.
- Generates fault reports in TetraMAX format, which are not easily compared with Zycad files. To print a fault report in Zycad format, use the `-prnt_fmt zycad` option:

  ```
  ipro -prnt_fmt zycad -strb_lim 5 iddq-database-name
  ```

- Prints out the fault report using a period (.) as the path separator for hierarchical names. You might want to print the fault report with a forward slash character as the path separator so that the report can be more easily compared to the original .fog file. To do so, use the `-path_sep /` option:

  ```
  ipro -prnt_fmt zycad -path_sep / -strb_lim 5
     iddq-database-name
  ```

When you use multiple testbenches, the fault report files show only the comment lines from the first testbench. PowerFault does not try to merge the comment lines from the fault list in the second and subsequent testbenches with those in the first testbench.

If you mix fault seeds from other formats, like using the `read_verifault` command to seed faults from a Verifault file, the Verifault faults detected in previous iterations are counted in the coverage statistics but are not printed in the fault report.

13

Iterative Simulation

You can run PowerFault iteratively, using each successive testbench to reduce the number of undetected faults. This feature is supplied only for backward compatibility with earlier versions of PowerFault. In general, you get better results by using the multiple testbench methodology explained in Combining Multiple Verilog Simulations.

In the following example, you have two testbenches and you want to choose five strobes from each testbench. All of the PowerFault tasks have been put into one file named ssi.v?.

This is the procedure to perform simulations iteratively:

1. In ssi.v?, seed the entire set of faults, using either the `seed` command or the `read` commands.
2. Run the Verilog simulation with the first testbench:

   ```
   vcs +acc+2 -R -P $IDDQ_HOME/lib/iddq_vcs.tab
      testbench1.v ssi.v ... $IDDQ_HOME/lib/libiddq_vcs.a
   or
   Verilog testbench1.v ssi.v ...
   ```

3. Run IDDQPro to select five strobe points:

   ```
   ipro -strb_lim 5 ...
   ```

4. Save the fault report and strobe report:

   ```
   mv iddq.srpt run1.srpt
   mv iddq.frpt run1.frpt
   ```

5. Edit and change ssi.v so that it seeds only the undetected faults in run1.frpt?:

```
...
$ssi_iddq( "read_tmax run1.frpt" );
...
```

6. Run the Verilog simulation again, using the second testbench:

```
vcs +acc+2 -R -P $IDDQ_HOME/lib/iddq_vcs.tab
   testbench2.v ssi.v ... $IDDQ_HOME/lib/libiddq_vcs.a
or
Verilog testbench2.v ssi.v ...
```

7. Run IDDQPro again to select five strobe points from the second testbench:

```
ipro -strb_lim 5 ...
```

8. Save the fault report and strobe report:

```
mv iddq.srpt run2.srpt
mv iddq.frpt run2.frpt
```

After completion of these steps, run1.srpt contains five strobe points for the first testbench and run2.srpt contains five strobe points for the second testbench.

If you have more than two testbenches, repeat steps 5 through 8 for each testbench, substituting the appropriate file names each time.

A

Simulation Debug Using MAX Testbench and Verdi

Verdi is an advanced automated open platform for debugging designs. It offers a full-featured waveform viewer and enables you to quickly process and debug simulation data.

When you use combine MAX Testbench, VCS, and Verdi for simulation debug, you can perform a variety of tasks, including displaying the current pattern number, the cycle count, and the active STIL statement, and adding input and output signals.

The following topics describe the process for setting up and running Verdi with MAX Testbench and VCS:

Setting the Environment

To set up the install path for Verdi, specify the following settings:

```
setenv NOVAS_HOME path_to_verdi_installation
set path = ($NOVAS_HOME/bin $path)
```

To set up the license file for Verdi, use one of the following environment variables:

```
setenv NOVAS_LICENSE_FILE license_file:$NOVAS_LICENSE_FILE
setenv SPS_LICENSE_FILE license_file:$SPS_LICENSE_FILE
setenv LM_LICENSE_FILE license_file:$LM_LICENSE_FILE
```

The license search priority is as follows:

1. SPS_LICENSE_FILE
2. NOVAS_LICENSE_FILE
3. LM_LICENSE_FILE

To set up the install path and license file for VCS, specify the following:

```
setenv VCS_HOME path_to_vcs_installation
set path=($VCS_HOME/bin $path)
setenv SNPSLMD_LICENSE_FILE license_file:$SNPSLMD_LICENSE_FILE
```

Preparing MAX Testbench

To prepare MAX Testbench to run with VCS and Verdi, you need to add a series of FSDB dump tasks to the testbench file. Some of the common FSDB dump tasks include:

- `$fsdbDumpfile` – Specifies the filename for the FSDB database.
- `$fsdbDumpvars` – Dumps signal value changes of specified instances and depth. To use this command, specify the FSDB file name. The default file name is novas.fsdb. You can specify several different FSDB file names in each fsdbDumpvars command
- `$fsdbDumpvarsByFile` - Uses a text file to select which scopes and signals to dump to the FSDB file. The contents of the file can be modified for each simulation without recompiling the simulation database.

The following example sets the `$fsdbDumpfile` and `$fsdbDumpvarsByFile` tasks in the MAX Testbench file (make sure you insert the `'ifdef WAVES` statement just before the `'ifdef tmax_vcde` statement):

```
`ifdef WAVES
    $fsdbDumpfile("../patterns/(YourPatternFileName).fsdb");
    $fsdbDumpvars(0);
`endif
```

For complete information on all FSDBdump tasks, refer to the following document:

```
$NOVAS_HOME/doc/linking_dumping.pdf
```

Linking Novas Object Files to the Simulation Executable

When you compile the VCS executable, you need to add a pointer to the Novas object files. You can do this using either of the following methods:

- Use the `-fsdb` option to automatically point to the novas.tab and pli.a files
  ```
  % vcs [design_files] [other_desired_vcs_options] -fsdb
  ```
- Use the `-P` option to point to the novas.tab and pli.a files provided by Verdi, as shown in the following example:
  ```
  % vcs -debug_pp \
  -P $NOVAS_HOME/share/PLI/VCS/$PLATFORM/novas.tab \
  $NOVAS_HOME/share/PLI/VCS/$PLATFORM/pli.a \
  +vcsd +vpi +memcbk [design_files] [other_desired_vcs_options]
  ```

For interactive mode, you need to add the `-debug_all` option. If you need to include model-driven architecture signals (MDAs) or SystemVerilog assertions (SVAs), use the `-debug_pp` option or the `+vcsd+vpi+memcbk` option.

Running VCS and Dumping an FSDB File

The following example shows how to use VCS to compile a simulation executable with links to Novas object files, run the simulation, and dump an FSDB file:

```
LIB_FILES=" -v ../design/class.v" DEFINES="+define+WAVES" DEBUG_
OPTIONS="-debug_pp -P $NOVAS_HOME/share/PLI/VCS/LINUX64/novas.tab
$NOVAS_HOME/share/PLI/VCS/LINUX64/pli.a"

OPTIONS="-full64 +tetramax +delay_mode_zero +notimingcheck
+nospecify ${DEBUG_OPTIONS}" NETLIST_FILES="../design/snps_micro_
dftmax_net.v.sa1" TBENCH_FILE="../patterns/pats.v" SIMULATOR="vcs"

${SIMULATOR} -R ${DEFINES} ${OPTIONS} ${TBENCH_FILE} ${NETLIST_
FILES} ${LIB_FILES} -l parallel_sim_verdiwv.log
```

Running Verdi

The following example shows how to set up and run Verdi:

```
LIB_FILES=" -v ../design/class.v"
DEFINES=""
```

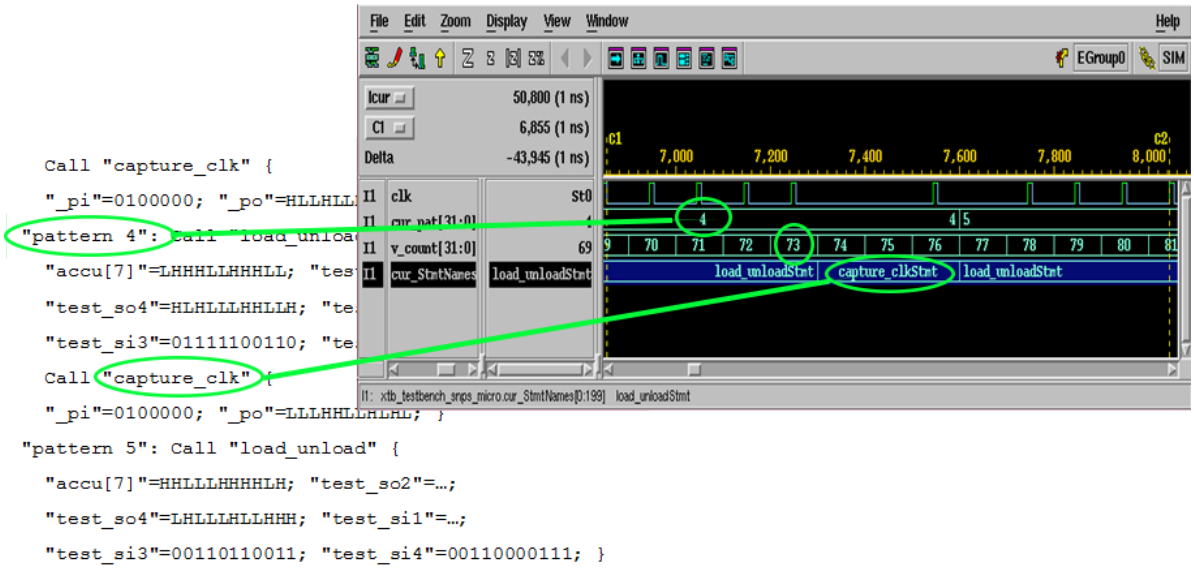
```
ANALYZE_OPTIONS=""
GUI_OPTIONS="-top snps_micro_test -ssf ../patterns/pats.fsdb"
NETLIST_FILES="../design/snps_micro_dftmax_net.v.sa1" TBENCH_
FILE="../patterns/pats.v"
ANALYZER="vericom"
GUI="verdi"
${ANALYZER} ${DEFINES} ${ANALYZE_OPTIONS} ${TBENCH_FILE}
${NETLIST_FILES} ${LIB_FILES}
```

The following topics describe several scenarios for using Verdi for debugging:

- Debugging MAX Testbench and VCS
- Changing Radix to ASCII
- Displaying the Current Pattern Number
- Displaying the Vector Count
- Using Search in the Signal List

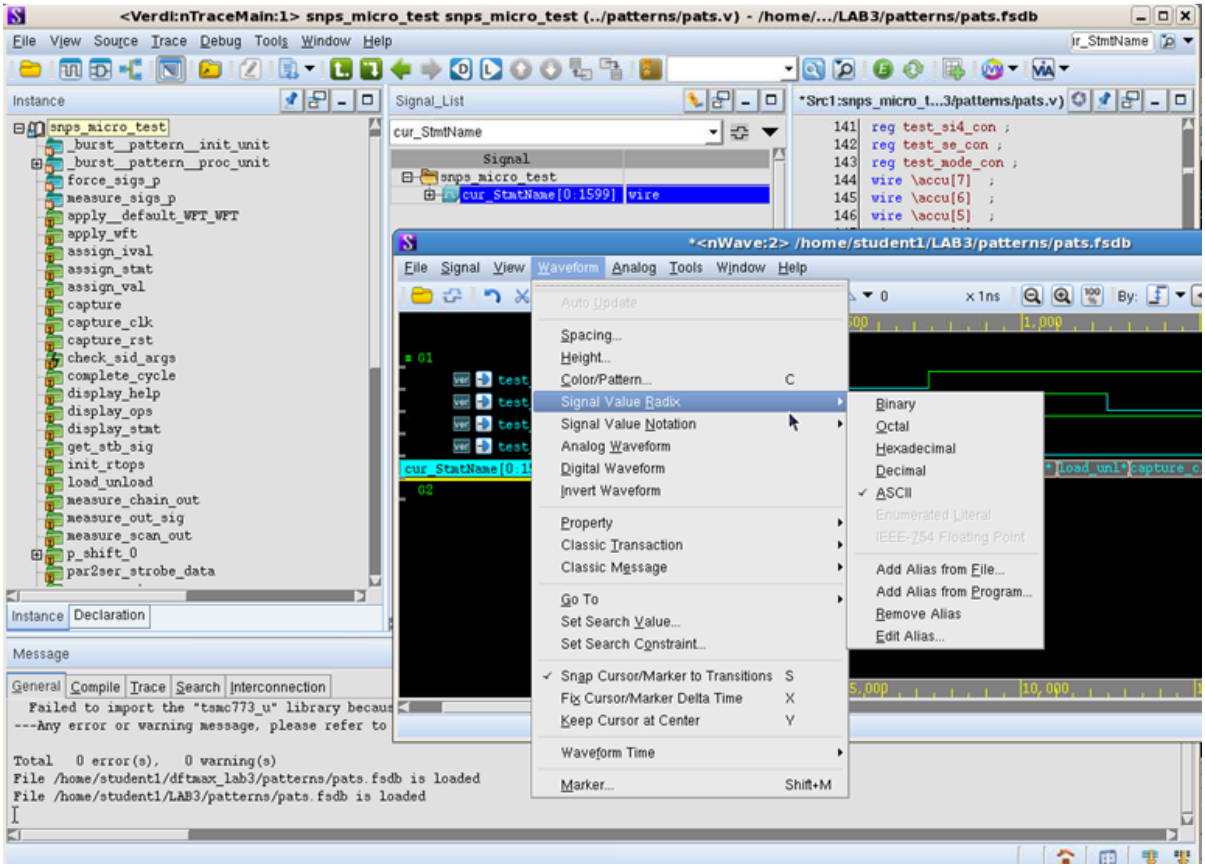
Debugging MAX Testbench and VCS

The following figure shows an example of how to use Verdi to debug MAX Testbench and VCS.



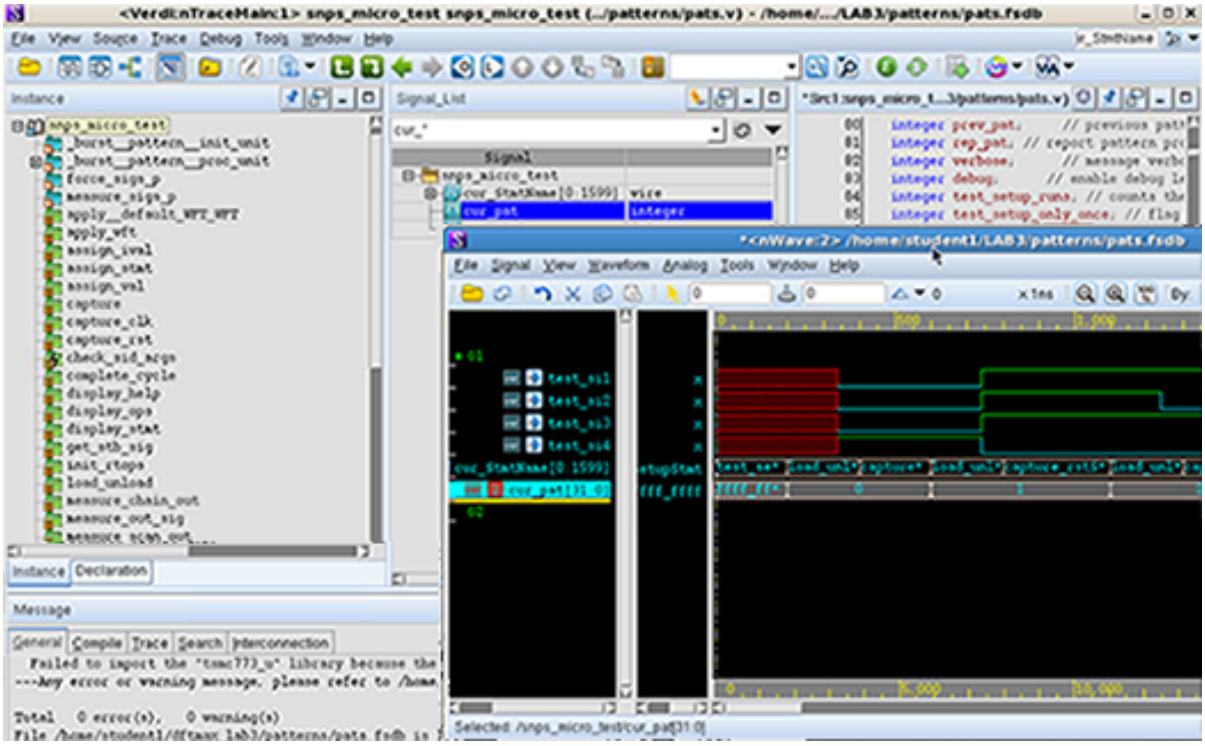
Changing Radix to ASCII

The following example shows how to change Radix-formatted signal values to an ASCII format:



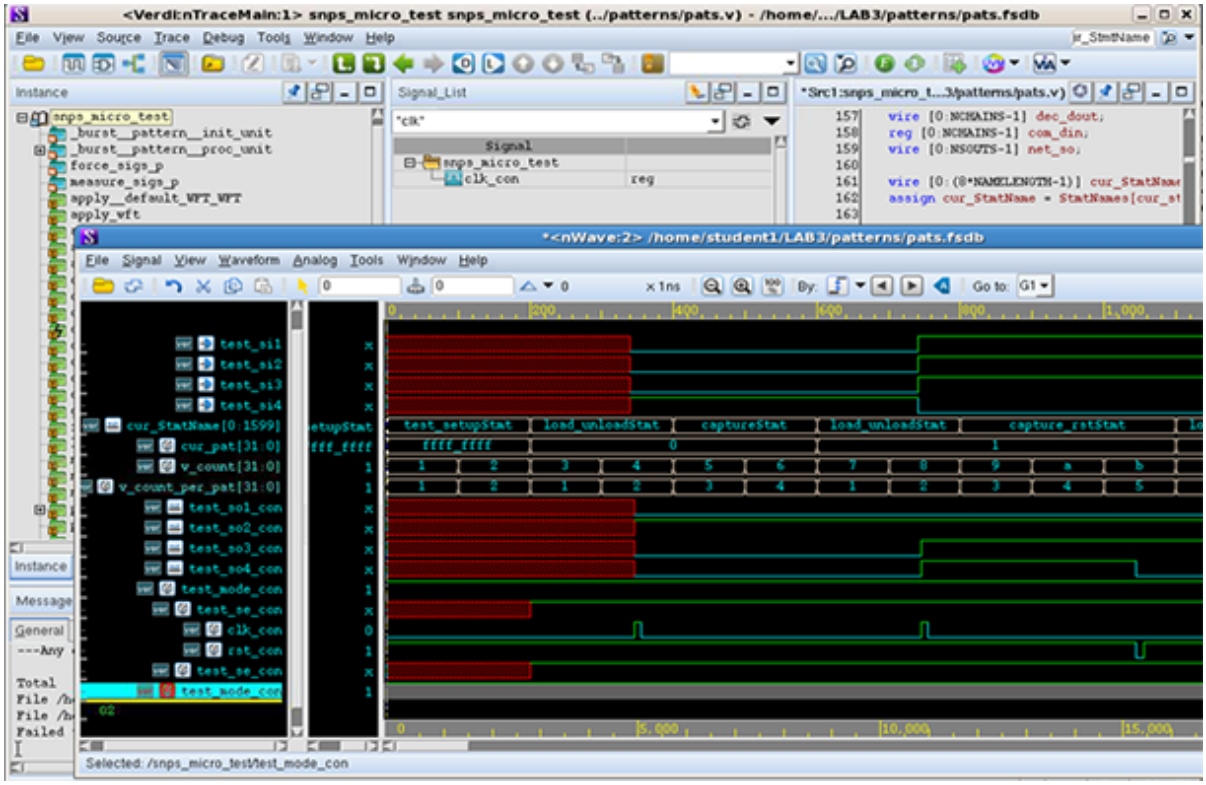
Displaying the Current Pattern Number

The following example shows how to display the current pattern number.



Displaying the Vector Count

The following example shows how to display the vector count.



Using Search in the Signal List

The following example shows how to add input and output signals by searching the signal list.