

VC Z01X™ User Guide

Version V-2023.12-SP2, June 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

© 2024 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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 <https://www.synopsys.com/company/legal/trademarks-brands.html>.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

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.

www.synopsys.com

Contents

1.	Introduction	17
<hr/>		
2.	Setup	20
	Verifying Your System Configuration	20
	Obtaining a License	21
	VC Z01X License for Serial Flow	22
	Licensing Keys	22
	Setting Up Your Environment	23
<hr/>		
3.	Compilation	24
	Compiling a Design	24
	Overriding the Default Simulation Timescale	26
	Delay Modes	26
	Serial Mode	26
	Concurrent Fault Simulation Mode	27
	Resolving Race Conditions	28
	SDF	28
	SystemVerilog	28
	Using SystemVerilog DPIs	29
	Fault Simulation Options	29
	Enabling Port Faults	29
	Defining Cells for Port Fault Creation	29
	Compiler Directives	29
	`enable_portfaults/`disable_portfaults	30
	`begin_faultfree/`end_faultfree	30
	`begin_faultprevent / `end_faultprevent	30
<hr/>		
4.	Stimulus	31
	Testbench	31

Contents

Common External Stimulus Options	32
Compile Options	32
Runtime Options	32
Extended VCD (eVCD)	34
FSDB	34
External Forcelist	35
VC Z01X Compile Options	35
Runtime Options With simv	36
\$fs_verify Usage	36
5. Logic Simulation	38
6. Fault Simulation Concepts	40
Fault Simulation in the Safety Environment	40
Fault Simulation Mechanics	41
Setting up FDB	42
Fault Models	45
Using Fault Campaign Manager	45
Interpreting Fault Status	46
Detected Group	47
DD (Dropped Detected)	47
DE (Detected Error)	47
DF (Detected \$finish/\$stop)	47
Potential Group	47
PT (Potential Detect)	48
PD (Dropped Potential)	48
Oscillating Group	48
OZ (Oscillating, Zero)	48
Hyper Group	48
HA (Hyperactive)	48
HT (Hypertrophic)	49
Illegal Group	49
IA (Illegal Access)	49
IP (Illegal User PLI)	49
IF (Illegal File I/O)	50
IX (Impossible X-state)	50
Unselected Group	50

Contents

NI (Not Injected)	50
NO (Not Observed)	50
NC (Not Controlled)	51
NT (Not Tested)	51
Untestable Group	52
UU (Untestable Unused)	52
UT (Untestable Tied)	52
UB (Untestable Blocked)	53
Excluded Statuses	53
EN	53
Fault Status' Not Included in a Built-In Status Group	54
EN (User-defined exclude status)	54
User Defined Status	54
NA (Not Attempted)	54
ND (Not Detected)	54
-- (Collapsed)	54
Detecting Faults	57
Detect Status Promotion	58
User Controlled Fault Detection	59
Syntax	60
\$fs_compare (system function)	60
\$fs_observe (system function)	60
\$fs_detect (system function)	61
\$fs_drop_status (system task)	62
\$fs_set_status (system task)	63
\$fs_default_status (system task)	63
\$fs_get_status (system function)	63
Placing User-Controlled Detection Tasks	64
Scenario 1: A simple testbench contained in a single module	64
Scenario 2: A complex testbench spanning several files/modules	64
Comparing Differences From Non-Signal Based Expressions	65
Example in Signal Value Strobing	65
Fault Sampling	66
Fault Sampling Selection	66
Permanent faults	66
Transient faults	66
Consideration of Testable vs. Untestable Faults	67
Fault Sampling and Exclude Blocks	67
7. Modeling	68

Contents

Verilog and SystemVerilog Support	68
Considerations for UDP Models	68
Using Timing with Serial Fault Simulation	69
Fault Simulation of Behavioral Logic	69
Constructs to Avoid	69
Hierarchical References	70
Forcing	70
Forcing a Signal Before Fault Injection	70
Understanding the Difference Between \$force and \$deposit	70
\$stop()/\$finish()	71
PLI	71
Fault Propagation and Port Isolation	71
Transient Fault Behavior	72
Transient Hold	73
Transient SEU	73
8. Fault Models	74
Fault Model Concepts	74
Stuck-At Faults	75
VC Z01X Stuck-At Fault Types and Locations	75
Transient Faults	77
Suggested Methodology	78
Importing Transient Faults and Creating a Coverage Report	78
Importing Transient Faults and Creating a Test-Level Coverage Report With Integrated Dictionary Data	79
MFO Faults	81
Reporting	81
Static/ Dynamic Testability	81
SFF MFO Input/ Output Format	82
Transition Faults	82
Specifying Transition Faults in SFF	83
Specifying the Delay	83
Transition Fault Collapsing	84
Transition Fault Merging	84
Fault Descriptors	84
FLOP Fault Class	86

Contents

Using the FLOP Fault Class	86
Behavior With `enable_portfaults and `suppress_faults	87
Fault Collapsing	90
Fault Placement	93
9. Standard Fault Format	95
General SFF Considerations	95
Creating a Standard Fault Format File	96
Header	97
Version	98
Date	98
User	98
Tool Information	98
Test Information Section	99
Fault Status Definition	99
User-Defined Fault Status (UDFS)	100
Redefining Statuses	100
Defining the Default Fault Status for Simulation	101
Selecting Faults Statuses for Simulation	102
Defining Status Interactions	102
Creating User-Defined Fault Status Groups	106
Using Fault Status Groups	108
Coverage Section	108
User-Defined Equation Syntax	108
Strobe Section	109
SafetyMechanism Section	111
FailureMode Section	112
Constraint Section	115
Limitation	116
FaultGenerate Section	116
Generating Faults Using FaultGenerate	116
Generating Multiple Fault List	119
SFF Syntax and Fault Generation Using vc_fcc	120
Wildcard Support	121
Non-Escaped Name Examples	122
Escaped Name Examples	126
Exclude Block	127

Contents

Instance Based Exclusion	129
Sampling	130
Specifying Sampling Methodologies	131
Sampling for Permanent and Transient Faults, and Random Sampling ..	132
Activating or Deactivating Active Sampling Configurations	133
Viewing the Sampling Results	135
Example: Sampling Directive	135
Scale Factor Aware Sampling	136
FaultInfo Section	138
FaultList Section	139
General Fault Descriptions	139
Fault Descriptions	139
Stuck-At Fault List Format	141
Transient Toggle Fault List Format	142
Transient Hold Fault List Format	144
Multiple Fault Origin Format	145
Fault Timing and Cycle Information	146
Setting Active Cycle Timing	147
Injection Faults Based on Verilog Condition	148
Common Keywords/Blocks/Constructs for FaultList and FaultGenerate	148
Condition Information	148
Example of Using FaultList and FaultGenerate Together	149
Example Coverage Reports Using Standard Fault Format	150
Example Coverage Report	150
<hr/> 10. Fault Generation	152
Fault Generation Compiler Directives	152
Design Compilation Options	155
Fault Selection Using System Tasks	155
Suggested Methodology	156
\$fs_inject()	156
Delaying Injection Time	156
Causing Fault Injection to Occur at the Beginning of the Time Step ..	156
Understanding Fault Placement	157
Generating PORT Faults	157
Generating VARI Faults	165
Generating WIRE Faults	167

11. Fault Simulation	173
Preparing for Fault Simulation	174
Procedural Flow	174
Invoking and Controlling Fault Campaign Manager	175
Invoking Fault Campaign Manager	175
Help	176
Displaying Variable Settings	176
Renaming or Disabling the Fault Campaign Manager Log	176
Generating or Importing a Fault List	177
Generating the Fault List	177
Invoking the Fault Generator	177
Using Percentage Sampling	177
Using Confidence Interval Sampling	178
Using Systematic Sampling Selection Method	181
Creating the Project	182
Adding Tests	183
Defining a Test	183
General Stimulus Options	184
Enable Verification	184
Define Maximum Number of Mismatches	184
Using a Testbench Stimulus	185
Using \$readmemb/h and \$test\$plusargs to Provide Stimulus	185
Using \$readmemb/h to Provide Stimulus	185
Using Scripts	186
Listing a Test	186
Deleting a Test	186
Configuring and Running Testability	187
Invoking Testability	187
Importing Toggle Results	187
Reporting Unselected Faults	188
Configuring Fault Simulation	188
Controlling Basic Fault Simulation Options	189
Setting Fault Simulation Command-Line Options	189
Enabling Simulation Messages	189
Controlling Simulation Retry	190
Configuring Distributed Fault Simulation	190
Setting Distributed Simulation Queues	191
Setting a Queue's Tool Groups	191

Contents

Testing Grid Configuration	192
Viewing All Defined Host Information	192
Setting Fault Detection Options	192
Delaying Fault Injection	193
Controlling Oscillation Detection	193
Zero-Delay Event Threshold	193
Zero-Delay Loop Iteration Threshold	193
Controlling Hyperfault Detection	194
Hyperactive Faults	194
Controlling Assertion Checking and Illegal Verilog Faults'	194
Disabling Assertion Checking and Illegal Verilog Faults	194
Managing Memory in Fault Simulation	195
Setting the Maximum Faults Per Pass	195
Setting the Minimum Faults Per Task	195
Starting Fault Simulation	196
Reporting	196
Fault Coverage	197
Coverage Summary	198
Behavior Analysis and Debug	198
Dumping GM/ FM Waveforms	198
Quitting Fault Campaign Manager	198
Example VC_FCM Scripts	199
Sharing Results Between Failure Modes	200
<hr/>	
12. Working with Results	201
Obtaining Results of Logic Simulation	201
Generating Logic Simulation Results	201
Working With Results of a Fault Simulation	202
Simulation Logs	202
Fault Campaign Manager Statistics	203
Fault Simulation Log Statistics	203
Fault Coverage Report	204
Test Coverage and Fault Coverage Calculations	206
Interpreting Autogenerated Gate Names	207
Using vc_fcc to Merge Separate Fault Report Files	209
Hierarchical Fault Coverage Report	210
Fault Dictionary	210

Contents

\$fs_set_status in Fault Dictionary	212
Fault Campaign Manager Usage Statistics	212
Unselected Fault Report	213
Unselected Report	215
Comma Separated Value (CSV) Fault Coverage Summary	218
Working with Results: Multiple Fault List	221
Coverage Report	221
Dictionary Report	222
CSV Report	223
Limitations: Multiple Fault Lists	223
Verdi Fault Analysis	223
<hr/>	
13. Advanced Features	226
Defining and Reporting Custom Fault Attributes	226
Introduction	226
Specifying Custom Fault Attributes Using Key, Value Pairs	228
Error Messages	230
Examples	231
FCM Coverage Usage	231
Coverage Report With Fault Attributes	232
<hr/>	
14. VC Z01X Methodology	234
VC Z01X Fault Injection Procedural Guidelines	235
Compile	236
Strobe System Tasks	236
Fault Locations	238
Testbench Setup	240
Timing Checks	241
Logic Simulation	241
Fault Generation	241
User Defined Fault Status (UDFS)	242
Defining UDFS Interactions and Merging Fault Lists	243
Defining and Generating Faults	244
Excluding Faults From a Fault List	246
Defining Fault Timing for Transient Faults	247
Extracting Flops and Latches for Transient faults	248
Sampling	248
Coverage Calculations	251
Reading the SFF file in Fault Campaign Manager	252

Contents

Testability	252
Testability for stuck-at	252
Fault Simulation	253
Defining the Workloads for Simulation	253
Simulation	253
Coverage	254
Interpreting Fault Status	254
Generating a Fault Coverage Report	254
Collapsed and Uncollapsed Fault Lists	255
Coverage Summary	255
Unselected vs. Untestable	256
Example Coverage Reports	257
Example Flows	259
Concurrent Simulation Flow	259
Using Software Test Libraries	260
Resolving Hyperfaults	261
15. Appendix A: Fault Campaign Manager Commands	263
Command Executed on the vc_fcm UNIX Command Line	263
Commands Executed on the Fault Campaign Manager Command Line	265
add_attribute	265
add_metadata	266
add_user	267
coats	267
cp_fault_results	268
create_campaign	269
create_fc	269
create_project	270
create_tcs	270
create_testcases	270
dump	273
exclude_faults	274
fdb_connect	275
fdb_disconnect	275
fsim	276
get_config	278
getobj_attributes	282
getobj_campaigns	283

Contents

getobj_ces	284
getobj_cov_equations	284
getobj_detection_points	285
getobj_dps	286
getobj_failure_modes	286
getobj_fault_results	287
getobj_faults	289
getobj_fcs	290
getobj_fms	290
getobj_host_groups	290
getobj_host_infos	291
getobj_metadata	291
getobj_observation_points	292
getobj_ops	293
getobj_projects	293
get_obj_promotion_tables	294
getobj_pts	295
getobj_safety_mechanisms	295
getobj_sgs	296
getobj_sms	296
getobj_status_groups	296
getobj_statuses	297
getobj_tcs	298
getobj_testcases	298
getobj_users	299
obj_mgmt	300
remove_attribute	301
remove_campaign	302
remove_fault_results	302
remove_fc	304
remove_metadata	304
remove_project	305
remove_tcs	305
remove_testcases	305
remove_user	306
report	306
set_campaign	311
set_config	311
set_fc	318

Contents

set_format	318
set_submit_cmd	318
show_attributes	320
show_campaign_summary	321
show_campaigns	322
show_ces	323
show_cov_equations	323
show_detection_points	323
show_dps	324
show_failure_modes	324
show_fault_results	325
show_faults	327
show_fc_summary	327
show_fcs	327
show_fms	327
show_host_groups	328
show_host_infos	328
show_metadata	329
show_observation_points	329
show_ops	330
show_projects	330
show_promotion_tables	330
show_pts	331
show_safety_mechanisms	331
show_sgs	332
show_sms	332
show_status_groups	332
show_statuses	333
show_task_failures	334
show_tasks	335
show_tcs	336
show_testcases	336
show_users	336
start_server	337
stop_server	338
test_grid_config	338
unset_campaign	339
unset_fc	340
unset_format	340

Contents

unset_submit_cmd	340
user_mgmt	341
vcf	342
verdi	345
write_fault_results	345
16. Appendix B: Command Syntax	347
Available Options	348
vc_fcc	348
vc_fdb_report	353
vcs	358
simv	359
Special Verilog directives, system tasks, and system functions	361
\$fs_inject (system task)	361
\$fs_verify (system task)	362
\$fs_verify_onevent (system task)	362
System Tasks for User-Controlled Fault Detection	363
\$fs_compare (system function)	363
\$fs_observe (system function)	364
\$fs_detect (system function)	365
\$fs_set_attribute (system task)	367
\$fs_add_attribute (system task)	367
\$fs_drop_status (system task)	367
\$fs_set_status (system task)	368
\$fs_default_status (system task)	368
\$fs_get_status (system function)	368
\$fs_strobe_onevent() (system task)	369
17. Appendix C: Language Support	370
Supported Language Constructs	370
Supported VCS Technologies	370
18. Appendix D: Using VC Z01X with Verdi	372
Generating Verdi KDB	372
Viewing eVCD Simulation Output in Verdi	373

Contents

Enabling FSDB Results	373
VC Z01X FSDB Options	373
Supported Verilog System Tasks	374
Generate Block Display	376
Fault Analysis and Debug	376
Using FDB Database	377
Dumping GM/ FM into the same FSDB	377
Supported Environment Variables and Simulation Command Line Options	378
Limitations	380
19. Appendix E: VC Z01X Command Map	381
Z01X to VC Z01X Command Map	381
Z01X to VC Z01X Fault Generation Command Map	386
Z01X to VC Z01X Simulation Command Map	388
Z01X to VC Z01X Reporting Command Map	395

1

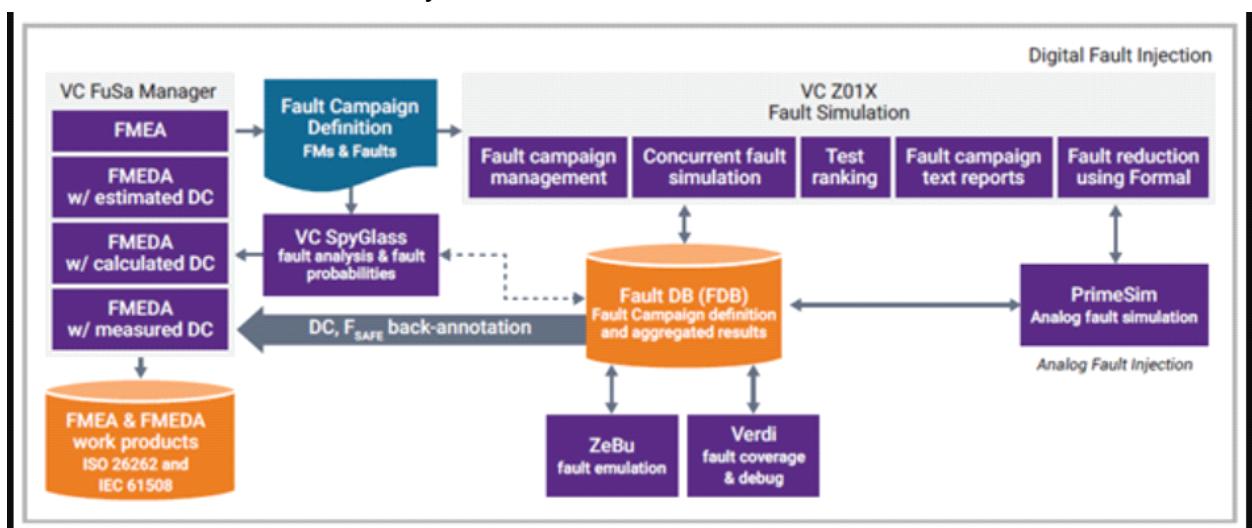
Introduction

About VC Z01X

VC Z01X is an integral part of the Unified Functional Safety Platform and fault eco-system. This native fault simulation is built on VCS and reuses functional verification setup for fault simulations. It is used to provide the fastest fault simulation as part of a comprehensive solution. The software can leverage shared information through a fault database to share results with other technologies such as VC Functional Safety Manager, VC Formal, and Verdi Fault Analysis.

It is a state of the art concurrent fault simulator used to deterministically measure fault coverage of a design and its related safety mechanisms. VC Z01X simulates and injects stuck-at and transient faults in gate-level, switch-level, RTL, and behavioral descriptions of designs. The tool provides a wide range of features including statistical sampling, stimulus support for SystemVerilog testbenches, eVCD files, and flexible fault detection through strobing system tasks.

Figure 1 Unified Functional Safety Platform



The remainder of the User Guide introduces the concepts and describes the details of fault simulation and describes guidelines on how to efficiently use VC Z01X to measure fault coverage.

Audience

This user guide assumes:

- You are familiar with modeling digital circuit with:
 - Verilog Language as described in IEEE 1364-1995, IEEE 1364-2001, IEEE 1364-2005, and IEEE 1800-2017.
 - System Verilog Language (With some exceptions) IEEE 1800-2005, IEEE 1800-2009, and IEEE 1076-2000.
 - VHDL as described in IEEE 1096-1993.
 - You are familiar with UNIX™ conventions, operations, and shell commands. This user's guide always provides commands for C shell; it may also provide commands for a Bourne or Korn shell if they are different from C shell.
 - You have the correct hardware software environment to run VC Z01X as described in the "Setup" chapter.
-

Known Safety-Critical Defects of VC Z01X

Synopsys maintains current information for reported defects through Jira issues. The VC Z01X team evaluates each reported issue for potential impact on functional safety. Known Safety-related defects are identified in the Release Notes section and updated for each release of the product.

When installing a new version of VC Z01X, you must assess, as part of the Safety analysis, the potential impact of the known Safety-related defects in the design and assure mitigation of any relevant Safety- related defects. The accompanying version's Release Notes has the information available for the following:

- Any changes that are related to Safety Use Model(s).
- List of known Safety-related defects in the new version of VC Z01X.

For more details on the safety-related defects in VC Z01X, see the following:

<https://solvnetplus.synopsys.com/s/article/VC-Z01X-Safety-Related-Issues>

Installation and Supported Platforms

Synopsys provides updates (including security patches) to computing environments (including operating systems) and ensures that they are backward compatible with the previous versions of the computing environment to test VC Z01X. The results of the testing done using such previous versions are applicable.

For more details on the installation, see *VC Z01X Installation Notes*.

Technical Support

Email:

support_center@synopsys.com

Contact:

<https://www.synopsys.com/support/global-support-centers.html>

2

Setup

This chapter explains the setup requirements for running VC Z01X. The tool is built on the Synopsys VCS Simulator and the configuration setup and licensing are similar to the VCS setup. This chapter includes the following information:

- [Verifying Your System Configuration](#)
- [Obtaining a License](#)
- [Setting Up Your Environment](#)

Verifying Your System Configuration

The syschk.sh script is used to check if your system and environment match the QSC (Qualified System Configurations) requirements for a given release of a Synopsys product. The QSC represents all system configurations maintained internally and tested by Synopsys.

To check whether the system you are on, meets the QSC requirements, enter:

```
% syschk.sh
```

When you come across any issue, run the script with tracing enabled to capture the output and contact Synopsys. To enable tracing, you can either uncomment the set -x line in the % syschk.sh file or enter the following command:

```
% sh -x syschk.sh >& syschk.log
```

Use `syschk.sh -v` to generate a more verbose output stream including the exact path for various binaries used by the script, and so on.

Example:

```
% syschk.sh -v
```

Note:

If you copy the syschk.sh script to another location before using it, you must also copy the syschk.dat data file to the same directory.

See the *Supported Platforms and Products* section of the VC Z01X Release Notes for the list of supported platforms and recommended C compiler and linker versions.

Obtaining a License

You must have a license to run VC Z01X. To obtain a license, contact your local Synopsys Sales Representative. The Sales Representative will need the hostid for your machine.

To start a new license, perform the following:

1. Verify that your license file is functioning correctly:

```
% lmcksum -c license_file_pathname
```

Running this licensing utility ensures that the license file is not corrupt. You should see an "OK" for every INCREMENT statement in the license file.

Note:

The snpslmd platform binaries and accompanying FlexLM utilities are shipped separately and are not included with this distribution. You can download these binaries as part of the Synopsys Common Licensing (SCL) kit from the Synopsys Web Site at: <http://www.synopsys.com/cgi-bin/ASP/sk/smartkeys.cgi>

2. Start the license server:

```
% lmgrd -c license_file_pathname -l logfile_pathname
```

3. Set the *LM_LICENSE_FILE* or *SNPSLMD_LICENSE_FILE* environment variable to point to the license file.

Example:

```
% setenv LM_LICENSE_FILE /u/edatools/vczoix/license.dat
```

or

```
% setenv SNPSLMD_LICENSE_FILE /u/edatools/vczoix/license.dat
```

Note:

You can use *SNPSLMD_LICENSE_FILE* environment variable to set licenses explicitly for Synopsys tools. If you set the *SNPSLMD_LICENSE_FILE* environment variable, then VC Z01X ignores the *LM_LICENSE_FILE* environment variable.

VC Z01X License for Serial Flow

The following are the approaches in serial fault simulation mode related to licensing:

- By default, VC Z01X serial fault simulation will check out One VC-ZOIX-FSIM-BASE key and shares the same license key for the remaining serial fault simulations in the run. In addition, VC Z01X also checks out One VCS runtime license key VCS Runtime_Net per each serial fault simulation.
- When the `-fsim=use_vczoix_license` runtime switch is added or used, VC Z01X will not check out any VCS runtime license keys and will only check out one VC-ZOIX-FSIM-BASE per each serial fault simulation job.

Licensing Keys

The following table outlines the licensing keys to be used at different stages:

Stage	License Keys
Analysis (vlog/vhdl)	No license checked out
Elaboration (vcs)	Checkout: VC-ZOIX-FSIM-BASE
Fault Generation (FCC)	Checkout: VC-ZOIX-UFC-BASE
Fault Campaign Manager run(vc_fcm)	Authorization: VC-ZOIX-UFC-BASE
Toggle simulation (tsim)	Checkout: VC-ZOIX-FSIM-BASE
Dynamic testability analysis (coats)	Checkout: VC-ZOIX-FSIM-BASE
Fault simulation(fsim)	Checkout: VC-ZOIX-FSIM-BASE
Report generation(report/vc_fdb_repo rt)	Checkout: VC-ZOIX-UFC-BASE

Setting Up Your Environment

To run VC Z01X, set the following environment variables:

- `$VCS_HOME` environment variable

Set the environment variable `$VCS_HOME` to the path where VC Z01X is installed as follows:

```
% setenv VCS_HOME installation_path
```

- `$PATH` environment variable

Set your UNIX PATH variable to `$VCS_HOME/bin` as follows:

```
% set path = ($VCS_HOME/bin $path)
```

OR

```
% setenv PATH $VCS_HOME/bin:$PATH
```

- `LM_LICENSE_FILE` or `SNPSLMD_LICENSE_FILE` environment variable:

Set the license variable `LM_LICENSE_FILE` or `SNPSLMD_LICENSE_FILE` to your license file as follows:

```
% setenv LM_LICENSE_FILE Location_to_the_license_file
```

OR

```
% setenv SNPSLMD_LICENSE_FILE /u/edatools/vcs/ license.dat
```

Note:

You can use the `SNPSLMD_LICENSE_FILE` environment variable to set licenses explicitly for Synopsys tools. If you set the `SNPSLMD_LICENSE_FILE` environment variable, then VCS ignores the `LM_LICENSE_FILE` environment variable.

For more information on environment variables, see the *VCS Environment Variables* section in the *VCS User Guide*.

3

Compilation

This chapter provides information on VC Z01X compilation. For more information on a list of VCS compile options, see VCS User Guide.

Compiling a Design

This section describes the flows available to compile a design.

For more information, see the following subsections:

Prerequisites

Before using these flows, ensure you have the following files:

- Verilog design source (.v) files, including the top-level module in the network being simulated.
- Verilog (.v) files containing descriptions of all library cells, including any user-defined primitives.
- VHDL (.vhdl) files containing descriptions of all modules being included.
- Verilog (.v) files containing descriptions of all supporting custom modules (PLLs, A2D converters, and so on).

Two-Step Flow

The `vcs` command runs the VC Z01X and C compilers, which creates the simulator executable used for either logic or fault simulation.

Apart from the [Prerequisites](#), ensure you have corrected all errors in the Verilog design description reported by the VC Z01X compiler.

To simulate a design using this flow, perform the following steps:

1. **Compilation:** In this step, the VC Z01X compiler is called by the `vcs` command to create the simulator executable, `simv`, and the design database in the same directory where the compiler is run.

To compile the circuit, specify the following command:

```
vcs -full64 -fsim [-f <filelist.f>] \
[-l <file>] \
[-v <file>] \
[-y <dir>] \
[<plusargs>]
```

2. **Simulation:** You can use `simv` to run the simulation. To perform this step, specify the following:

```
./simv <other options>
```

Three-Step Flow

The three-step flow leverages the capabilities of VCS to compile the design.

In addition to [Prerequisites](#), ensure that the library mappings are defined in the `synopsys_sim.setup` file and the specified physical library (directory) for the logical library exists. If the physical directory does not exist, VCS exits with an error message.

To simulate a design using this flow, perform the following steps:

1. **Analysis:** In this step, the `vlogan/vhdlan` utility checks the design for syntax errors, generates the intermediate files required for elaboration, and saves these files in the design or work library pointed to by your default logical library.

To perform this step, specify the following:

For VHDL: `$VCS_HOME/bin/vhdlan <VHDL-source-files> <other-options>`

For more information on the `vlogan` utility, see the [VCS User Guide](#).

Note:

Use the `vlogan` script located in the bin directory.

2. **Elaboration:** In this step, using the intermediate files generated during the Analysis step, the `vcs` command builds the instance hierarchy and generates a binary executable called `simv`. This binary executable is later used for simulation.

To perform this step, specify the following:

```
$VCS_HOME/bin/vcs -full64 -fsim <top-module-names-or-config> <other-options>
```

3. **Simulation:** You can use *simv* to run the simulation. To perform this step, specify the following:

```
/simv <other options>
```

Overriding the Default Simulation Timescale

There are several command-line options to set or modify a module's timescale unit or precision. For all of these options the unit is 1, 10 or 100 and the precision is s, ms, us, ns, ps or fs. It is often necessary to set or override the simulation timescale when the testbench is replaced with external stimulus such as eVCD or FSDB.

Table 1 Simulation Timescale Switch

Switch	Description
-override_timescale=<unit>/<precision>	Set the timescale for all modules. This option overrides all `timescale directives in the Verilog source code. Use of this option carefully because unconditionally overriding the `timescale directives in the Verilog source code may cause improper scaling of delays or loss of precision causing a simulation to fail.

Note:

An error is reported if the first module, based on the order of compilation, does not have a timescale defined.

Delay Modes

Most library models contain path delays and path delay conditions because they are intended to be used for timing accurate logic simulation. This is normally not necessary for fault simulation and can consume significant amounts of memory and simulation time. VC Z01X supports path delay and SDF annotated simulation only in serial mode.

Serial Mode

VC Z01X enables a serial mode fault simulation. There are 2 flows of serial mode.

- For timing fault simulations with SDF, compile the design with VC Z01X by adding `-fsim=serial_flow`.
- For non-concurrent friendly faults that get dropped with *Illegal Group* fault status, run the serial mode without any additional option on VC Z01X compile.

Enabling serial mode at compile time has the advantage that many of the simulation optimizations that are disabled when compiling for concurrent fault simulation can remain enabled when compiling for serial mode fault simulation.

VC Z01X does not support compilation for serial mode simulation and enabling concurrent mode at runtime.

Concurrent Fault Simulation Mode

VC Z01X is the VCS simulation engine. The reference simulation (referred to as the Good Machine or GM) simulates the same as VCS.

The concurrent fault simulation engine is an extension of VCS. There are some limitations in language constructs that can be simulated in the concurrent simulation engine (referred to as the Faulty Machine or FM). Faults that affect these constructs are dropped from concurrent simulation as IA status and can be resolved in serial simulation mode.

The following are the limitations of concurrent fault simulation:

- Mixed VHDL design with SDF timing
- Faults that propagate to PLI and DPI
- SystemVerilog constructs:
 - Class objects
 - Dynamic arrays
 - Smart queues
 - Mailboxes
 - Synchronization primitives
 - Dynamic waits
 - Ref ports
 - File open/write operations

Note:

VC Z01X has the ability to simulate SystemVerilog class objects In concurrent fault simulation mode using `-fsim=class`.

Resolving Race Conditions

VC Z01X supports the VCS compile time option to automatically resolve clock and data signal race conditions. This switch might help simulations with race conditions to pass where event order differences (previously) caused the simulation to fail.

Table 2 Race Condition Option

Switch	Description
-deraceclockdata	Attempts to resolve race conditions by evaluating always blocks where the clock and data change at the same time to store the stable value of the data.

SDF

SDF simulation is only supported in serial mode simulation with `-fsim=serial_flow` added at compile.

SystemVerilog

By default, VC Z01X compiles according to the IEEE 1364-2001 standard. Files with .sv extension are automatically compiled according to IEEE 1800-2005 and 1800-2009 SystemVerilog standards.

Table 3 SystemVerilog Options

Switch	Description
-sverilog	Specifies that all files should be compiled using SystemVerilog 1800 keywords.
+systemverilogext+<extension>	Sets the language to SystemVerilog for those files with the specified extension(s). For example, <code>+systemverilogext+.sh</code> compiles all files with an.sh extension using SystemVerilog 1800 keywords. (Note that files with an .sv extension are automatically compiled using SystemVerilog keywords.)

This section describes the following:

[Using SystemVerilog APIs](#)

Using SystemVerilog DPLs

VC Z01X supports SystemVerilog DPI (direct programming interface) import statements (Verilog to C functions) and export calls (Verilog task calls from C/C++).

Link user DPI libraries by adding them to the `vcs` compile command line:

```
vcs -fsim -sverilog mydpi.so dpitest.v
```

Fault Simulation Options

This section describes the following:

- [Enabling Port Faults](#)
- [Defining Cells for Port Fault Creation](#)

Enabling Port Faults

Use the compile command line option `-fsim=portfaults` to enable port faults for the entire design. Synopsys recommends using this option only for RTL designs.

Defining Cells for Port Fault Creation

Use the compile command line options `-fsim=suppress+cell` and `+nolibcell` to define cells at the gate level as locations to create port faults. The `-fsim=suppress+cell` option automatically enables port faults and suppresses faults for all cells in the design. This includes any module or cell compiled with the `-v` or `-y` option in addition to any cells with ``celldefine`/`nocelldefine`` compiler directives.

Synopsys recommends including the `+nolibcell` option when using `-fsim=suppress+cell` to consider only cells within a ``celldefine`/`endcelldefine`` block as library cells.

Compiler Directives

This section describes the following subsections:

- ``enable_portfaults`/`disable_portfaults`
- ``begin_faultfree`/`end_faultfree`
- ``begin_faultprevent` / `end_faultprevent`

`enable_portfaults`/`disable_portfaults`

For more information on `enable_portfaults` and `disable_portfaults`, see [Fault Generation](#).

`begin_faultfree`/`end_faultfree`

For more information on `begin_faultfree` and `end_faultfree`, see [Fault Generation](#).

`begin_faultprevent` / `end_faultprevent`

For more information on `begin_faultprevent` and `end_faultprevent`, see [Fault Generation](#).

4

Stimulus

VC Z01X supports Testbench, eVCD, FSDB, and Force files to stimulate the design. This chapter outlines the basic functionality of each of the stimulus type and includes the following sections:

- [Testbench](#)
 - [Common External Stimulus Options](#)
 - [Extended VCD \(eVCD\)](#)
 - [FSDB](#)
 - [External Forcelist](#)
 - [\\$fs_verify Usage](#)
-

Testbench

VC Z01X supports Verilog and SystemVerilog (including UVM) testbenches as stimulus for logic and fault simulation. When running a testbench flow, we need to clearly mark the fault-free region versus the DUT where concurrency will run. The `-fsim=dut:<hierarchical_path>` compile option is highly recommended for performance reasons, as it will limit the areas of concurrency divergence to the DUT area only.

Note:

VC Z01X does not support the divergence of non-synthesizable code, but without the `-fsim=dut:<hierarchical_path>` option, may diverge synthesizable testbench constructs.

In situations where the time scale of the stimulus file is smaller than the precision of the design, VC Z01X will use the precision of the design instead. For example, if a testbench file uses a scaling of 1ns and the design's precision is at 10ms, stimulus will use the scale of 10ms. This may cause unexpected simulation behaviors.

Common External Stimulus Options

Extended VCD (eVCD) and FSDB are considered primary external stimulus formats. Most of the options necessary to use eVCD and FSDB stimulus options are similar between formats.

Compile Options

Options	Description
<code>-stim=module:<string></code>	Causes the compile to prepare the specified dut for stimulus capability. Required for external primary stimulus.

Runtime Options

Options	Description
<code>-stim=type:<string></code>	Defines the type of external stimulus being used. Possible values are <code>evcd</code> , <code>fsdb</code> , <code>stil</code> , or <code>wgl</code> . Default value is <code>evcd</code> .
<code>-stim=file:<filename></code>	Specifies the path of external stimulus file to be used.
<code>-stim=inst:<string></code>	Provides the hierarchical path to the instance where external stimulus must be applied. By default, it is assumed the hierarchy of the stimulus and Verilog design match.
<code>-stim=clk:<string>[+<string>]</code>	Lists signals in the design that should be treated as clock. To avoid race conditions between clocks and data, clock signals are driven in the non-blocking assign region.
<code>-stim=verify</code>	Enables the stimulus verification mode. This mode verifies signals whenever system tasks such as <code>\$fs_strobe</code> , <code>\$fs_compare</code> , or <code>\$fs_verify</code> are called by logic simulation. When verifying signals, the mode will compare the end-of-timestep values of any listed signal in such system task against the stimulus value in that timestep. If there is a difference, a mismatch warning will be reported. The purpose of this mode is to verify that relevant signals are correctly simulated at important times, such as <code>\$fs_strobe</code> calls.

-stim=verify_all	Enables the stimulus dut verification mode. This mode is similar to -stim=verify. However, it verifies the output ports of the stimulus dut instead of any other signals.
-stim=verify_mismatch_limit:<num>	Specifies the maximum number of mismatches between expected and stimulated values before the simulation is terminated. The default abort threshold is 200 mismatches.
-stim=verify_tolerance:<time>	When verifying simulation mismatch, this option causes VC Z01X to compare the range of stimulus values before and after the current simulation time- <time> and ends at the current simulation time + <time>. This option may be used in situations where stimulus is off by a cycle with the original simulation, to avoid false verify mismatches.
-fsim=fault+stats	Includes per-fault statistics messages in fault simulation. (Information included: fid, current fault status, fm event count, gm event count, and the fault description.) By default, the message is printed once at the end of simulation. However, it can be enabled for every progress message with an optional mode. Example Message - FSIM Fault Statistics:FID Status FM Events GM Events Description37059 DD 59606 20104559 {DD 1 {PORT "test.a0.u1.u0.u3.U173.A"}}37060 HA 52421 20104559 {HA 1 {PORT "test.a0.u1.u0.u3.U173.Z"}}
-fsim=fault+progress+<time>	Reports the current state of fault simulation at incremental times. The message includes: the current date and time, the current time of the verilog simulation, the CPU usage time, the elapsed time, the peak memory usage, the total, injected and completed fault counts, and the total fault statuses of all faults that completed simulation. An optional time value -fsim=fault+progress+<time> is added to control as to when the progress is reported. By default, the progress message reports every 1200 seconds (20 minutes). Example Message: FSIM Progress: 2024/03/06 19:31:44 - Sim Time: 106112107200ps - CPU Time: 1498s - Elapsed Time: 1500s - Peak Memory: 441MB - Total/Injected/Completed Faults: 2046/2046/261 - Completed Results: HA:257 OD:4

-fsim=fault+monitor+drop	Prints messages related to faults whenever a fault is getting dropped, when specified with the <code>create_testcases</code> command.
-fsim=fault+monitor+strobe	Prints a diagnostic message when a fault is being strobed. Check the <code>out.log</code> for the <code>fsim</code> task. Example Message: FSIM Strobe: FID34 - Time: 8500ps - Fault Description: NN 1 {PORT "test.risc1.alu1.accum[5]"} Strobe Signal: test.risc1.alu_out - GM: 00000000 - FM:1..

Extended VCD (eVCD)

eVCD is a supported stimulus format for logic and fault simulation. VC Z01X supports eVCD as defined by the IEEE 1364 -2009 Verilog standard.

Extended VCD (eVCD)

Use the `$dumpports` Verilog system task to obtain values for the port signals (input/output/inout) to create an eVCD file that can be used as stimulus instead of the testbench. Some guidelines for creating appropriate eVCD files are:

- Only output the DUT instance, for example:

```
$dumpports(testbench.dut);
```

- Do not dump additional hierarchy.
- Only input port values for the DUT are used to drive stimulus. Values in hierarchies above or below the DUT are ignored. See [Common External Stimulus Options](#) for commands used to specify stimulus options to the simulator.

Note:

In situations where the time scale of the stimulus file is smaller than the precision of the design. VC Z01X will use the precision of the design instead. For example, if an eVCD file uses a scaling of 1ns and the design's precision is at 10ms, stimulus will use the scale of 10ms. This may cause unexpected simulation behaviors.

FSDB

Fast Signal Database (FSDB) is supported as external stimulus for logic and fault simulation. Usage is similar to eVCD. See [Common External Stimulus Options](#) for

commands that are used to specify stimulus options to the simulator. Some guidelines for creating appropriate FSDB stimulus files are:

- Only output the DUT instance, for example:

```
$fsdbDumpvars(1, testbench.dut);
```
- Do not dump additional hierarchy that will not be referenced as stimulus during simulation.
- You can generate more than one hierarchy but should only do so if you will use it as stimulus during simulation.

Note:

There is nothing analogous to extended VCD for FSDB. Extended VCD supports bidirectional ports. The FSDB stimulus treats inout ports as inputs and issues a warning. Depending upon the design, this may not have the desired results.

In situations where the time scale of the stimulus file is smaller than the precision of the design. VC Z01X will use the precision of the design instead. For example, if an FSDB file uses a scaling of 1ns and the design's precision is at 10ms, stimulus will use the scale of 10ms. This may cause unexpected simulation behaviors.

Accommodating Bidirectional Ports in FSDB

Unlike eVCD, FSDB lacks the ability to model bidirectional ports for both input and output.

External Forcelist

Forcelist is considered as a secondary stimulus that can be used along with the primary stimulus. Forcelist can be enabled using the following options:

- [VC Z01X Compile Options](#)
- [Runtime Options With simv](#)

VC Z01X Compile Options

The following compile time option is required for using forcelist:

Option	Description
-stim=forcelist	Enables the usage of <code>forcelist</code> as stimulus at runtime.

Runtime Options With simv

To enable forcelist file at runtime, the following options can be added:

Option	Description
-stim=forcelist_file:<filename>	Path to the forcelist stimulus file.
-stim=forcelist_dut:<string> +<string>	Determines the hierarchical paths for the design's dut and forcelist's dut.
-stim=forcelist_mode:<string>	Describes the force types that are applied. Possible values are language, external, or all. By default, the mode is set to language forces.

Note:

In situations where the time scale of the stimulus file is smaller than the precision of the design. VC Z01X will use the precision of the design instead. For example, if a forcelist uses a scaling of 1ns and the design's precision is at 10ms, stimulus will use the scale of 10ms. This may cause unexpected simulation behaviors.

\$fs_verify Usage

VC Z01X provides a method to verify that simulation values match expected values. Automatic verification is enabled through `-stim=verify` or `-stim=verify_all`. These command line options work in conjunction with system tasks for fault strobing and with the verification system tasks.

Syntax:

- `$fs_verify (<signal1>[,<signal>]);`
- `$fs_verify_onevent(<signal1>[,>signal>]);`

Notes:

- Within the `-stim=module:<dut module name>` option, each specified signal should be a port of the eVCD top module as specified through the system task. Other signals are ignored.
- Bit / part-select on a vector signal is converted into the complete base signal with a warning.

- Whenever any signal in the specified signal list of a `$fs_verify` system task changes in simulation, all signals in the signal list of the call are verified with corresponding stimulus values at the end of the timestamp.
- `$fs_verify_onevent` is persistent, which means once it is executed in simulation, it remains active throughout the rest of the simulation.

5

Logic Simulation

After compiling the Verilog design successfully, you can perform logic simulation, which serves as the basis to any fault simulation. There may be issues in modeling or stimulus, and these are much easier to resolve in a logic simulation than in a fault simulation.

Synopsys recommends you to validate logic simulation after compiling with fault simulation enabled (by compiling with the `-fsim` switch). Whenever design files, test bench files, and/or stimulus inputs change, Synopsys recommends that you re-run this step.

For more details on runtime switches and configuration, see *VCS User Guide*.

If you have performed VC Z01X logic simulation on the current design and are confident of its quality, you can skip this step and go to the following chapters.

Note:

For more information on fault simulation, see [Fault Simulation](#) section.

The following is an example testcase with fsdb as stimulus:

Example Usage:

Syntax:

```
vcs -stim=module:mid dut.v -fsim -debug_access -kdb -lca -l comp.log
simv -stim=type:fsdb -stim=inst:top.m -stim=file:ref.fsdb -l simv.log
```

Output:

```
0 ZN = 1
10 ZN = 0
20 ZN = 1
30 ZN = 0
40 ZN = 1
50 ZN = 0
60 ZN = 1
70 ZN = 0
80 ZN = 1
90 ZN = 0
100 ZN = 1
110 ZN = 0
120 ZN = 1
130 ZN = 0
140 ZN = 1
150 ZN = 0
```

```
160 ZN = 1
170 ZN = 0
180 ZN = 1
200 ZN = 1
Info: FSDB stimulus completed with 0 mismatches.
190 ZN = 0
VCS Simulation Report
Time: 201 ns
```

6

Fault Simulation Concepts

This chapter describes the basic concepts of fault simulation and how fault simulation is used as part of the test strategy. It includes the following:

- [Fault Simulation in the Safety Environment](#)
- [Fault Simulation Mechanics](#)
- [Setting up FDB](#)
- [Fault Models](#)
- [Using Fault Campaign Manager](#)
- [Interpreting Fault Status](#)
- [Detecting Faults](#)
- [Detect Status Promotion](#)
- [User Controlled Fault Detection](#)
- [Fault Sampling](#)

Fault Simulation in the Safety Environment

Model based fault injection can be used to quantify the number of faults within a design that:

- May cause the failure of a device such that it violates a safety goal.
- The ability of the safety mechanism(s) to correct, react, or signal the violation of a safety goal.

Faults are injected and the results of the simulation is specific to the workload provided. This workload can consist of application code, software test libraries, diagnostics, or even functional test patterns.

Failures during the operation of the device can range from simple shorts to complex behavior. Different fault models closely reflect the different defects. VC Z01X offers

several models as identified in the ISO-26262:2011 standard (see Fault Models for more information).

The immediate result of a fault injection is a report of the number and percentage of faults that are detected by a workload or a set of workloads. In a verification environment, fault simulation identifies areas of the design that are not yet tested and need test improvements. You can use this information to improve the quality of both the workloads and the safety mechanisms.

Workload validation helps you:

- Determine the quality of the workloads by quantifying the faults detected by each test.
- Determine areas of the design that require additional or modified workloads.
- Find tests that improve the quality of the test suite.

Fault simulation, in some instances, can detect areas of low testability due to design flaws. If functional tests do not cause observable behavior in a portion of the chip during fault simulation, there may be a design problem in that part of the chip.

Fault injection tests have an additional requirement over design verification tests. For fault simulation, you must define where and when the simulator should observe the fault. This includes observing where the fault may cause a device failure and when and where the safety mechanism(s) may detect the failure.

Fault Simulation Mechanics

Fault simulation verifies the completeness of test sets for data and control paths within a chip. It works by inserting hypothetical faults into the chip design and running the tests against the faulty chip. The results are then compared against the unfaulted network.

Fundamental to the operation of VC Z01X fault simulation are the concepts of good machine and faulty machine.

- The good machine (GM) defines the operation of the circuit in the absence of any faulty behavior. The good machine is a representation of expected behavior.
- The faulty machine(s) (FM) defines the behavior that differs from the good machine behavior. A faulty machine is a representation of the circuit if a defect occurs. The repercussions of a defect throughout the circuit are fault effects.

VC Z01X provides two algorithms for fault simulation.

The concurrent fault simulation algorithm that VC Z01X uses means that it can simulate the good machine with large numbers of faulty machines at the same time. This is accomplished by tracking divergences (differences) between the faulty machines and the good machine.

The concurrent fault simulation algorithm assumes that any given fault effect has a small, localized effect on the total simulation. By running many faults concurrently, the time required to run the faults is significantly reduced.

The results of the simulation of the good machine and the faulty machine are compared at the observable points of the device and in the safety mechanism(s). If the faulty machine behavior differs from the good machine at circuit observation points, the faults at those points can be detected.

VC Z01X also provides a serial fault simulation algorithm. When fault simulation runs in this mode, a single fault is injected into the simulation. In this mode, faults can be resolved that may not be friendly to the concurrent algorithm. These include faults that affect the PLI, some System Verilog constructs, and other objects that are not implemented in the concurrent algorithm. In serial mode, the fault values are compared to the stored FSDB values that are recorded during the testability phase of the fault campaign.

Setting up FDB

All fault related data is managed in a Fault Database (FDB). The FDB is organized by FMEDA projects and each project holds fault campaigns, which are executed to measure the results for the Failure Modes. A fault campaign represents the full definition on what to execute and the results of the fault qualification. The FDB provides a common understanding of faults and faults status.

The following are the two modes available to run the FDB server:

- Explicit start and stop of the server (recommended when multiple users share the same FDB).
- Automated start and stop of the server as needed (makes the server transparent to the user).

FDB has a persistent storage on disk while the server provides the data access. The server process can be stopped and restarted without any data loss. This can be on a different machine. By default, the user who creates the FDB has access to the FDB. Other users can be added with the `user_mgmt` command in FCM.

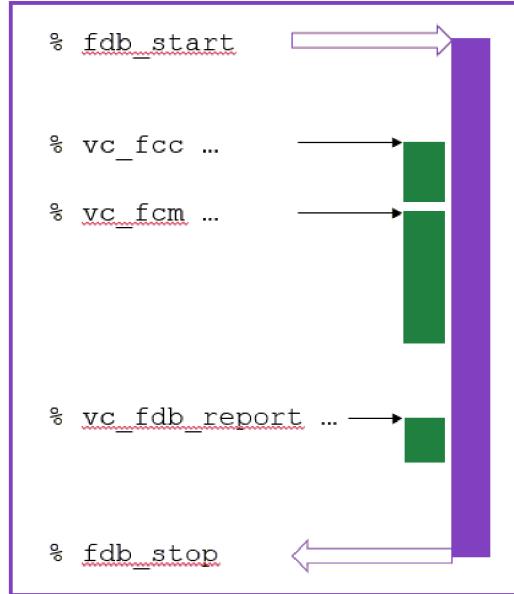
Explicitly start/stop an FDB Server

A new FDB needs to be initialized before a server can be started. This step creates the database structure on disk. The data directory storage path needs to be an empty directory when initializing the FDB. After it is initialized, the server can be started. After the server runs, a project needs to be created. Fault campaigns are stored in a project. When the server is no longer used, it can be stopped.

This method is used to manually control the start and stop of an FDB server (keep the server running as desired).

Explicit FDB Server start/stop

 Program lifetime
 FDB Server lifetime



The following commands are used to explicitly start/ stop an FDB server.

```
fdb_start
[-f|-fdb_path <path>]
[-p|-fdb_project <project>]
[-a|-auto_stop [timeout_seconds]]
[-v|-verbose] [-monitor_period <seconds>]
[-kill_timeout <seconds>]
[-server_mode <daemon | background>]
[-h|-help]
```

```
fdb_stop
[-f|-fdb_path <path>]
[-stop | -kill | -kill-9 | -s|-status]
[-dont_wait]
[-verbose]
[-h|-help]
```

Example 1:

```
% fdb_start
% vc_fcc
% vc_fcm
% fdb_stop
```

Example 2:

```
% fdb_start -fdb_path my_fdb
% vc_fcc -fdb_path my_fdb
% vc_fcm -fdb_path my_fdb
% fdb_stop -fdb_path my_fdb
```

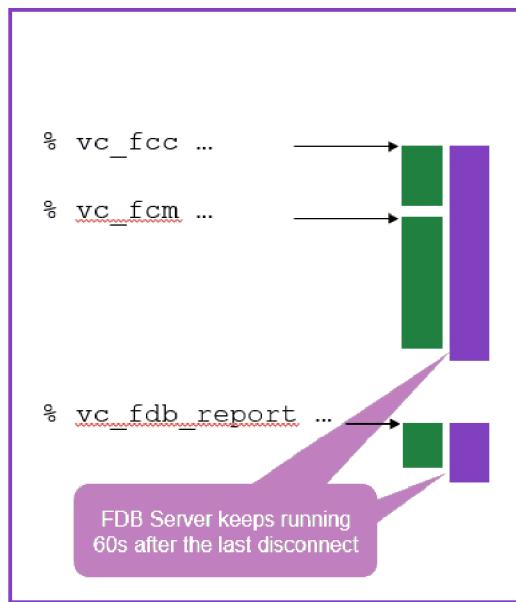
Automatic start/stop an FDB Server

This method automatically initiates the DB, starts the server, creates the default project as needed or just connects to a running server. The FDB server always runs detached from the client who started it.

- Automatically started FDB server is stopped after a minute the last client is disconnected.
- `fdb_stop` - Allows to terminate FDB server on a remote machine.

Automatic FDB Server handling

 Program lifetime
 FDB Server lifetime



Example 1:

```
% vc_fcc -full64 -sff faults.sff -campaign or1200
% vc_fcm -tcl_script fcm.tcl -fc or1200 -connect
```

Example 2:

```
% vc_fcc -full64 -sff faults.sff -campaign or1200 -fdb_path /x/y/or_fdb
% vc_fcm -tcl_script fcm.tcl -fc or1200 -connect -fdb_path /x/y/or_fdb
```

Fault Models

A good fault model mimics real-life defects sufficiently such that when faults are detected by a safety mechanism in simulation, the safety mechanism also detects the defect in the physical device. Defects on the device can range from simple shorts to complex behavior. Different fault models mimic different defects, and VC Z01X offers several models (see [Fault Models](#) for more information) as defined in the ISO-26262 standard. VC Z01X has implemented the following fault models:

- Stuck-At Faults, which are s@0 and s@1 faults on primitive input and output terminals, module ports and nets, and arrays.
 - Transient Faults, which are faults that occur once and then disappear. Transient faults can be placed on outputs of sequential elements and in memories
-

Using Fault Campaign Manager

VC Z01X's Fault Campaign Manager controls and synchronizes the entire fault simulation process, from creating the fault list through obtaining fault coverage reports. It augments designer productivity by automatically handling very large test suites. See [Fault Simulation](#) for information on using Fault Campaign Manager.

The following is a general fault simulation flow using Fault Campaign Manager:

1. **Compile and logic simulate the design with VC Z01X before launching Fault Campaign Manager.** This verifies that the design was correctly compiled, and also generates the simulation database (simv.daidir) and simulation executable (simv) that Fault Campaign Manager requires.
2. **Generate the fault list.** You perform a one-time operation to set fault creation or import options. See the [Fault Generation](#) chapter for more information on Verilog tasks required to generate faults. The faults are written to the fault database.
3. **Add tests.** You will specify a list of tests to be used for fault simulation. Adding tests includes listing any necessary runtime command line arguments for logic (toggle) or fault simulation. See "Adding Tests".
4. **Configure and initiate fault simulation.** You select tests to simulate and the fault simulation parameters that will control the simulations. Supporting processes such as toggle simulation and testability analysis are automatically invoked when you launch fault simulation; or you may invoke them separately for purposes of reporting and/or test selection. Fault Campaign Manager simulates the tests you specify that have not been previously simulated. Toggle and testability analysis results are written to the fault database.

5. **At the start of simulation, all faults are considered Not Attempted (NA).** As simulation progresses, statuses are updated according to the status detection table. See “Detect Status Promotion”.
6. Fault Campaign Manager performs the following simulation tasks.
 - **Runs toggle simulation:** This simulation is only performed once per test, unless the test subsequently changes. It is not run on a test unless you select that test for fault simulation.
 - **Performs testability analysis:** Relying on data gathered during toggle simulation, Fault Campaign Manager determines which test will most effectively detect the greatest number of faults. You can set your validation strategy by specifying the test characteristics that are most important to you. Testability analysis is run repeatedly on a test until it is chosen for fault simulation.
 - **Continues the simulation/testability analysis cycle:** This cycle is repeated until the selected tests are exhausted or according to parameters you set.
7. **View simulation results in report form:** Fault Campaign Manager offers several useful reporting formats, which are customizable to show the information that is most relevant to you. Data integrity is critical to the decisions taken by Fault Campaign Manager, and consistency checks ensure that data presented is up-to-date.

Interpreting Fault Status

Each fault is assigned a status according to its behavior during fault simulation. This status is updated in the fault database and is referenced in numerous places throughout Fault Campaign Manager. The status of faults is shortened to two characters.

Functionally similar faults belong to built-in status groups. By using the built-in status group, you can reference all faults that belong to that group. Each group is identified by a unique two-character abbreviation, a group name, and the contained statuses. Therefore, you can reference each built-in status group by the two-character abbreviation.

The following descriptions define the built-in fault groups, the fault status descriptions and usage considerations. See [Table 4](#) for the fault statuses categorized into built-in status groups.

For information on defining fault status groups, see [Creating User-Defined Fault Status Groups](#).

See the following subsections:

- [“Detected Group”](#)
- [Potential Group](#)

- Oscillating Group
 - Hyper Group
 - Illegal Group
 - Unselected Group
 - Untestable Group
 - Excluded Statuses
 - Fault Status' Not Included in a Built-In Status Group
-

Detected Group

The built-in detected status group contains all fault status that are detected because a difference in behavior was observed between the GM and the FM. For a fault that qualifies as a detect, the faulty machine must be in one known state and the good machine in a different known state at the locations and times where the circuit or safety mechanism was observed.

DD (Dropped Detected)

Indicates that the fault was detected during simulation. A dropped detected fault has reached the threshold where it is removed from the simulation. This threshold is set to 1 by default and may be modified with the following setting:

```
-fsim=maxhard:<num> (runtime switch)
```

DE (Detected Error)

A detected error fault indicates that a faulty machine has caused a Verilog assertion to fire, resulting in a call to \$fatal or \$error. (The default behavior for an assert is to call \$error.) If the user disables this fault status, the simulation continues as if the assert had not been executed.

DF (Detected \$finish/\$stop)

A detected \$finish/ \$stop fault indicates that a faulty machine has encountered a \$stop or \$finish that was not executed in the good machine. If you disable this fault status, the simulation continues as if the system task had not been executed.

Potential Group

The potential group consists of faults that cannot be classified with certainty as detected. For a fault that qualifies as a possible detect, the good machine is in a known state (0 or 1)

and the faulty machine is unknown (X). The good machine must be in a known state for a detect to occur.

For example, if a clock on a flip-flop is faulted, the FM might never clock the flip-flop output to a known state, which can cause a potential detect.

PT (Potential Detect)

Indicates the fault was a potential detect at least once but not the specified maximum number of times. The threshold `-fsim=maxpot:<num>` (runtime switch) is used to set the number of times a fault is detected before it is promoted to a PD (Dropped Potential) status.

PD (Dropped Potential)

Indicates the fault was a potential detected the specified maximum number of times. See [Setting Fault Detection Options](#) for more information on the variable `-fsim=maxpot:<num>`.

Oscillating Group

VC Z01X detects zero delay oscillating faults. By default, VC Z01X checks at each strobe for oscillating faults.

OZ (Oscillating, Zero)

A zero delay oscillating fault is caused by a feedback path without any delay. To be classified as an oscillating fault, a fault must not allow the simulation to advance time. Zero delay oscillating faults are detected by counting the number of events. VC Z01X marks, then drops oscillating faults from the simulation.

Hyper Group

The concurrent fault simulation algorithm is very compute and memory intensive, so it is sometimes necessary to remove faults from the simulation that consume an excessive amount of memory or simulation time. These faults are called hyperfaults. For more information on controlling hyperfaults, [Controlling Hyperfault Detection](#) section. Hyperfaults are typically a very small number of faults that require more system resources than other faults. By dropping these faults from the simulation, the fault simulation can progress much more quickly.

HA (Hyperactive)

A hyperactive fault affects a small portion of the design but has many faulty machine events. A fault is considered hyperactive if the ratio of good machine events to faulty machine events exceeds a threshold.

HT (Hypertrophic)

A hypertrophic fault causes a large portion of the design to be diverged consuming a large amount of memory.

Illegal Group

An illegal access fault occurs when a fault propagates into a Verilog or C (PLI) construct that is unsupported for inclusion into a diverged FM. An illegal fault can occur if a fault location must be moved to a design location that cannot be simulated. The faults cannot be simulated and are dropped from simulation. Faults in the illegal group can often be resolved by running them with serial fault simulation.

The following items result in IA fault status during concurrent mode:

- Class objects
 - Default setup disables class support with concurrent mode. You can recompile with `-fsim=class` to enable concurrent mode with class objects.
- Dynamic arrays
- Smart queues
- Mailboxes
- Synchronization primitives
- Dynamic waits
- Ref ports
- File open/write operations
- Any user PLI, DPI

IA (Illegal Access)

An illegal access fault is the result of a fault, which was diverges a location that is not supported for concurrent fault simulation. Illegal access faults can be resolved using serial fault simulation.

IP (Illegal User PLI)

An illegal user PLI fault occurs anytime when PLI is called in a faulty machine. VCS cannot diverge user PLI code.

IF (Illegal File I/O)

An illegal file I/O fault occurs anytime a faulty machine calls one of the following functions/tasks:

\$fopen	\$fwriteb	\$fstrobeh	\$fseek	\$fdisplayh	\$fmonitro
<code>\$fgetc</code>	<code>\$fwriteh</code>	<code>\$fstrobeo</code>	<code>\$rewind</code>	<code>\$fdisplayo</code>	<code>\$fflush</code>
<code>\$ungetc</code>	<code>\$fwriteo</code>	<code>\$fmonitor</code>	<code>\$fread</code>	<code>\$fclose</code>	<code>\$log</code>
<code>\$fgets</code>	<code>\$fdisplay</code>	<code>\$fmonitorb</code>	<code>\$fscanf</code>	<code>\$fstrobe</code>	<code>\$writememb</code>
<code>\$ftell</code>	<code>\$fdisplayb</code>	<code>\$fmonitorh</code>	<code>\$fwrite</code>	<code>\$fstrobeb</code>	<code>\$writememh</code>

If the user disables this fault status, a faulty machine call to any of the listed functions/tasks is ignored. A value of zero is returned for the following functions:

<code>\$fopen</code>	<code>\$fseek</code>	<code>\$fgets</code>
<code>\$fgetc</code>	<code>\$rewind</code>	<code>\$fscanf</code>
<code>\$ungetc</code>	<code>\$fread</code>	<code>\$ftell</code>

IX (Impossible X-state)

In impossible x-state indicates a transient fault had all associated fault injections attempted on sequential UDP origins with value `x`.

Unselected Group

The Unselected Group of faults consists of faults that were determined to be unable to be detected with the set of workloads provided for a design. These faults are categorized by testability as they are measured based on their activity in a simulation and by their connectivity to visible strobe locations.

NI (Not Injected)

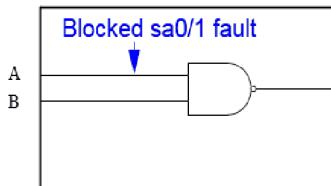
Indicates a fault had all associated fault injection times set to be attempted after the end of logic simulation. This fault status is used for transient faults and for faults where the proper capabilities in the simulator have not been enabled to inject the fault.

NO (Not Observed)

Indicates the fault cannot be observed because the workload(s) blocked the activity of that fault location from propagating to an observation point. Such faults are created

when a dominant signal to a gate never toggles during toggle simulation and stays 0 or 1 (depending on the gate) the whole simulation. 0-X-0 or 1-X-1 are not considered toggles. Running Fault Campaign Manager with testability enabled is a requirement to generate this type of fault. For array testability, an NO corresponds to an array word which was never read.

Figure 2 NO (Not Observed)

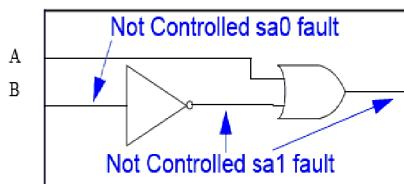


If B (input) stays 0 or X the entire toggle simulation, then the fault will be NO

NC (Not Controlled)

Indicates the fault cannot be controlled. Faults selected by Fault Campaign Manager that are not controllable will be classified NC. If a signal (input to the fault) does not toggle during toggle simulation, then the fault with the same value is not a controlled fault. For example, if the signal stays 0 for the entire simulation, then SA0 is NC, but SA1 is not. 0-X-0 or 1-X-1 are not considered toggles. Running Fault Campaign Manager with testability enabled is a requirement to generate this type of fault. For array testability, an NC corresponds to an array word which was never written.

Figure 3 NC (Not Controlled)



If B (input) stays 0 or X the entire toggle simulation, then the fault will be NC

NT (Not Tested)

Indicates the fault was a combination of NO or NC across more than one test.

Untestable Group

Faults in the Untestable Group have been analyzed and determined to be structurally undetectable. These faults differ from the Unselected Group because they are not dependent on the workload(s) simulated. Untestable faults cannot be detected by any workload and require changes to the design or to the strobing information to become testable.

By default, Untestable faults are omitted from coverage reports and are not included in coverage statistics.

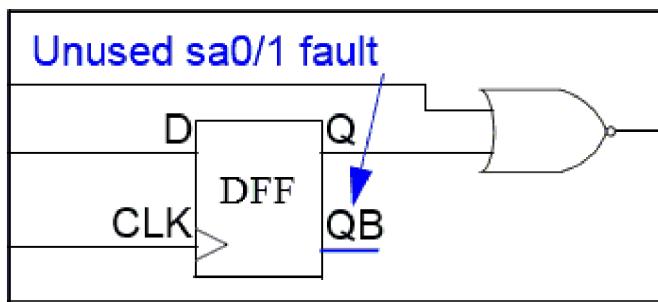
UU (Untestable Unused)

Indicates the fault cannot be detected because it is unused. A UU fault is set from one of two situations in the structure of the netlist.

1. The fault is on a location, typically an output, that is unconnected.
2. The fault is on a location that is connected to a top-level design output or bidirectional port that is never observed or strobed.

When generating faults, include all observable locations whether they are strobed or not. When a port is not included in the strobe list the faults are marked UU and are not counted as undetected faults. This leads to optimistic results.

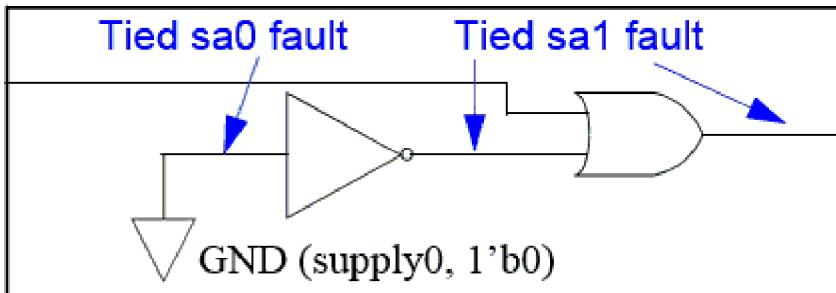
Figure 4 UU (Untestable Unused)



UT (Untestable Tied)

An untestable tied faults happen with a connection to a supply net or constant. Supply net stuck-at faults are s@0 on a supply0 net and s@1 on a supply1 net. These faults cannot be detected because they are the same value as the supply net. A UT fault can also be UB or UU (only UT is displayed).

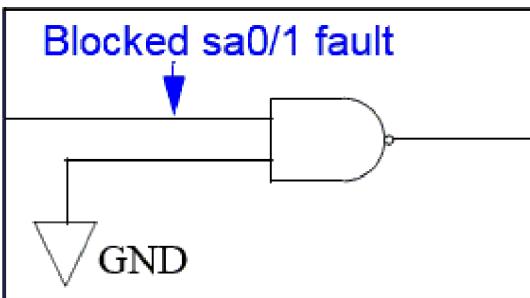
Figure 5 UT (Untestable Tied)



UB (Untestable Blocked)

An untestable blocked fault happens when it is blocked by another signal connected to a supply net or a constant. UB fault can also be UU (only UB is displayed).

Figure 6 UB (Untestable Blocked)



Excluded Statuses

EN

Indicates the fault was marked as excluded using the Verdi Fault Analysis exclude status function. From Verdi Fault Analysis , users can create up to 10 fault statuses to manually exclude faults. New fault classes are given the status EN, where N is a number 0-9. As EN is a way to denote E0-E9, it is not a status or group itself, so it cannot be redefined or be part of a user-defined status group. See the *Verdi Fault Analysis User Guide*.

As EN is just a way to denote E0-E9, it is not a status or group itself, so it cannot be redefined or be part of a user defined status group.

Fault Status' Not Included in a Built-In Status Group

EN (User-defined exclude status)

Indicates the fault was marked as excluded using the Verdi Fault Analysis exclude status function. From Verdi Fault Analysis, you can create up to 10 fault statuses to manually exclude faults. New fault classes are given the status EN, where N is a number 0-9. See the *Verdi Fault Analysis User Guide*.

User Defined Status

When using the Standard Fault format, users can create custom statuses and define their interactions and weights in the coverage calculations. See [Fault Status Definition](#) section.

NA (Not Attempted)

Indicates the fault is not yet attempted by any simulation. NA faults change to another status once simulated. If NA faults remain after simulation, there may be a simulation problem.

ND (Not Detected)

Indicates the faulty machine was never different from the good machine at the locations and times where the circuit or safety mechanisms were observed. When the good machine is an unknown value and the faulty machine is known, this is also marked "not detected". Faults selected by Fault Campaign Manager will be classified as ND, if the fault is not reclassified into a detected/potential/hyperfault/and so on, it stays ND.

-- (Collapsed)

Indicates the fault was not simulated because it has an equivalent prime (source) fault. The status of this fault matches the status of the prime fault.

Table 4 Default Fault Group and Fault Statuses

Built-In Status Group	Fault Status	Description
DG (Detected)	DD (Dropped Detected)	Indicates the fault was detected the specified maximum number of times.
DG (Detected)	DE (Detected Error)	A detected error fault indicates a faulty machine has caused a Verilog assertion to fire and resulted in a call to \$fatal or \$error.

Table 4 Default Fault Group and Fault Statuses (Continued)

Built-In Status Group	Fault Status	Description
DG (Detected)	DF (Detected \$finish/\$stop)	A fault that is dropped when the FM encounters a \$stop or \$finish that is not executed in the GM.
PG (Potential)	PT (Potential Detect)	Indicates the fault cannot be classified with certainty as detected, but less than the specified maximum number of times.
PG (Potential)	PD (Dropped Potential)	Indicates the fault was possibly detected the specified maximum number of times.
OG (Oscillating)	OZ (Oscillating, Zero)	Indicates the faulty machine caused the simulation to oscillate with a zero-delay oscillation.
HG (Hyper)	HA (Hyperactive)	A fault is hyperactive if the ratio of GM events to FM events exceeds a threshold.
HG (Hyper)	HT (Hypertrophic)	Indicates a large number of divergences from the good machine.
IG (Illegal)	IA (Illegal Access)	An illegal access fault results from a class reference in a faulty machine. Can be a result of a fault, which was attempted to be moved to a new location that was not set up for fault simulation.
IG (Illegal)	IF (Illegal File I/O)	An IF fault occurs anytime a faulty machine calls a file I/O.
IG (Illegal)	IX (Impossible x-state)	Indicates a transient fault that had all associated fault injections attempted on sequential cell when the output value of that cell was 'x'.

Table 4 Default Fault Group and Fault Statuses (Continued)

Built-In Status Group	Fault Status	Description
Unselected (NG)	NI (Not Injected)	Indicates a fault had all associated fault injection times set to be attempted after the end of logic simulation or for faults where the proper capabilities in the simulator have not been enabled to inject the fault.
Unselected (NG)	NO (Not Observed)	Indicates the fault cannot be observed because the workload(s) blocked the activity of that fault location from propagating to an observation point.
Unselected (NG)	NC (Not Controlled)	Indicates the fault cannot be controlled with the executed workload(s).
Unselected (NG)	NT (Not Tested)	Indicates the fault was a combination of NO or NC across more than one test.
Untestable (UG)	UU (Untestable Unused)	Indicates the fault cannot be detected because it is unused.
Untestable (UG)	UT (Untestable Tied)	Indicates the fault cannot be detected because of connection to a supply net or a constant.
Untestable (UG)	UB (Untestable Blocked)	Indicates the fault cannot be detected because it is blocked from a signal connected to a supply net or a constant.
Untestable (UG)	UR (Untestable Redundant)	Indicates the fault cannot be detected because it is masked by another fault.
Untestable (UG)	UI (Untestable Imported)	Indicates the fault cannot be detected for unknown reasons. UI faults are only assigned when importing a fault list from an older version of VC Z01X.

Table 4 Default Fault Group and Fault Statuses (Continued)

Built-In Status Group	Fault Status	Description
Untestable (UG)	UO (Untestable Unobservable)	Indicates the fault cannot be detected. UO faults are only assigned when importing a fault list.
Not included in a built-in status group	EN (User defined exclude status)	Indicates the fault was marked as excluded using the Verdi Fault Analysis exclude status function.
Not included in a built-in status group	User defined status	The fault status was defined by the user through the Standard Fault format.
Not included in a built-in status group	NA (Not Attempted)	Indicates the fault is not yet attempted by simulation.
Not included in a built-in status group	ND (Not Detected)	Indicates the faulty machine was never different from the good machine at the locations and times where the circuit or safety mechanism(s) were observed.
Not included in a built-in status group	-- (Collapsed)	Indicates the fault was not simulated because it has an equivalent prime (source) fault.

Detecting Faults

VC Z01X uses the following table to classify a fault detect at a strobe. During a strobe operation, VC Z01X compares the logic value of the good machine with that of a faulty machine at the defined locations. A Z-0-1-X table-based comparison of each faulty machine is done against the good machine. This lookup table performs a standard fault simulation comparison, where a definite detect is by default a 0-1 or 1-0 good-to-faulty compare, and possible detect is a 0-(X or Z) or 1-(X or Z) compare.

The following table shows the default fault detection criteria.

Table 5 Default Fault Detection Criteria

FM		0	1	Z	X
GM	0	ND	DD	PD	PD
	1	DD	ND	PD	PD
	Z	ND	ND	ND	ND
	X	ND	ND	ND	ND

Detect Status Promotion

At the start of fault simulation all faults are considered to be not attempted(NA) even if they were already a different status such as hyperactive (HA). At the end of fault simulation, the simulated status is merged with the pre-simulation status, as shown in the figure. Note that hyper (HA, HT) and oscillating (OZ) fault statuses have priority over a potential detected fault (PT). This is to avoid repeated simulation of hyper and oscillating faults which are take significantly more runtime and memory to simulate. It is possible for potential fault coverage to decrease between tests due to hyper and oscillating faults.

When using Standard Fault format, detect status promotion can be defined using the *Promotion* sub block. See [Fault Status Definition](#) section in the [Standard Fault Format](#).

Figure 7 Detect Status Promotion

		New Status																										
		NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UR	UI	UO	EN
Old Status	NA	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UR	UI	UO	EN
	ND	ND	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	ND	ND	NI	ND	UU	ND	ND	ND	ND	EN	
DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	DD	EN	
DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	DT	EN	
DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	DE	EN	
DF	DF	DF	DD	DT	DE	DF	EN																					
PD	PD	PD	DD	DT	DE	DF	PD	PD	PT	OZ	HA	HT	PD	EN														
PT	PT	PT	DD	DT	DE	DF	PD	PT	OZ	HA	HT	PT	EN															
OZ	OZ	OZ	DD	DT	DE	DF	PD	OZ	OZ	HA	HT	OZ	EN															
HA	HA	HA	HA	DD	DT	DE	DF	PD	HA	EN																		
HT	HT	HT	HT	DD	DT	DE	DF	PD	HT	EN																		
IA	IA	IA	IA	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	EN														
IP	IP	IP	IP	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	EN													
IH	IH	IH	IH	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	EN												
IF	IF	IF	IF	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IF	EN												
IX	IX	IX	IX	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	EN										
NC	NC	NC	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NT	NT	NC	NC	NC	NC	NC	EN		
NO	NO	NO	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NT	NO	NT	NT	NO	NO	NO	NO	NO	EN	
NI	NI	NI	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NT	NT	NI	NT	NI	NI	NI	NI	NI	EN	
NT	NT	NT	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NT	EN									
UU	NA	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UU	UU	UU	UU	EN	
UT	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UT	UT	UT	EN	
UB	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UB	UB	UB	UB	UB	EN	
UR	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UT	UR	UR	EN	
UI	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UT	UI	UO	EN	
UO	NA	ND	DD	DT	DE	DF	PD	PT	OZ	HA	HT	IA	IP	IH	IF	IX	NC	NO	NI	NT	UU	UT	UB	UT	UO	UO	EN	
EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	EN	

Legend

- Detectable Faults
- Potential Faults
- Hyper Faults
- Oscillating Faults
- Undetected Faults
- Illegal Access Faults

User Controlled Fault Detection

A set of system tasks provides flexibility in fault detection. Use these system tasks to determine what constitutes a detect and to set the status of a fault based on more than just good machine/faulty machine signal value comparisons.

Capabilities:

- Allows you to determine fault detection criteria.
- Do not block the module containing the system task from diverging.
- Allows you to specify whether to drop or continue to simulate a fault.

User controlled fault status definitions provide greater flexibility when using fault detection system tasks. Using Standard Fault Format, you can modify the behavior of non-core fault statuses, define new statuses, and modify coverage calculations to include user-defined statuses and modify status weighting. See “Fault Status Definition” section in the “Standard Fault Format” chapter.

See the following subsections:

- [Syntax](#)
 - [Placing User-Controlled Detection Tasks](#)
 - [Comparing Differences From Non-Signal Based Expressions](#)
 - [Example in Signal Value Strobing](#)
-

Syntax

\$fs_compare (system function)

Causes the simulator to compare the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs_compare()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs_compare()` returns 1.
- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_compare()` returns 2.
- If both the second and third condition are met in one `$fs_compare()` signal list, the function returns 1.

This function is only effective in the faulty machine. If this function is called from within the good machine, the return value is 0.

Following a good machine call to `$fs_compare`, several fault simulation checks occur at the end of the time step:

- Oscillation fault checking: For more information, see *Controlling Oscillation Detection*.
- Hyperfault checking: For more information, see *Controlling Hyperfault Detection*.

Listed signals are treated as strobe points during fault generation.

Syntax:

```
int compare;
...
compare = $fs_compare(sig1,sig2,sig3);
```

\$fs_observe (system function)

The `$fs_observe` system function does not require signal arguments like `$fs_compare`.

`$fs_observe()` links to observation points (signals) defined through SFF.

Causes the simulator to observe the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs.observe()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs.observe()` returns 1.
- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs.observe()` returns 2.
- If both the second (that is, return 1) and third conditions (return 2) are met in one `$fs.observe()` signal list, the function returns 1.

The `$fs.observe` function is only effective when called from the faulty machine (FM). If this function is called from within the good machine (GM), the return value is 0.

The FailureMode associated with the FM determines the observation points when making a call to the `$fs.observe` function.

Syntax:

```
$fs.observe();
```

Example:

```
Input SFF:
FailureMode fml {
    Observe {
        "tb.dut.sig1"
        "tb.dut.sig2"
    }
    SafetyMechanisms(SM1)
}
SafetyMechanism SM1 {
    Detect { "tb.dut.sig3" }
}

FaultList fml {
    NA 0 { CELL "tb.dut.cell1.Z" }
    NA 0 { CELL "tb.dut.cell2.Z" }
}

Strobe:
always @(negedge clock)
    if ($fs.observe() == 1)
        $fs_set_status("ON");
Simulation:
0 CELL tb.dut.cell1.Z - Set to ON through the $fs.observe block
0 CELL tb.dut.cell2.Z - Set to OD through the $fs_detect block
always @(sm_detect_sig)
    if ($fs_detect() == 1)
        $fs_drop_status("OD");
```

\$fs_detect (system function)

The `$fs.detect` system function does not require signal arguments like `$fs.compare`.

`$fs.detect()` links to detection points (signals) defined through SFF.

Causes the simulator to detect the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs.detect()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs.detect()` returns 1.

- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_detect()` returns 2.
- If both the second (that is, return 1) and third conditions (return 2) are met in one `$fs_detect()` signal list, the function returns 1.

The `$fs_detect` function is only effective when called from the faulty machine (FM). If this function is called from within the good machine (GM), the return value is 0.

The FailureMode associated with the FM determines the detection points when making a call to the `$fs_detect` function.

Syntax:

```
$fs_detect();
```

Example:

```
Input SFF:
FailureMode fm1 {
    Observe {
        "tb.dut.sig1"
        "tb.dut.sig2"
    }
    SafetyMechanisms(SM1)
}
SafetyMechanism SM1 {
    Detect { "tb.dut.sig3" }
}

FaultList fm1 {
    NA 0 { CELL "tb.dut.cell1.Z" }
    NA 0 { CELL "tb.dut.cell2.Z" }
}

Strobe:
always @(negedge clock)
    if ($fs_observe() == 1)
        $fs_set_status("ON");

always @(sm_detect_sig)
    if ($fs_detect() == 1)
        $fs_drop_status("OD");

Simulation:
0 CELL tb.dut.cell1.Z - Set to ON through the $fs_observe block
0 CELL tb.dut.cell2.Z - Set to OD through the $fs_detect block
```

\$fs_drop_status (system task)

This system task is added to the testbench or safety mechanisms at the points where the testbench is checking the results of a simulation. It causes the simulator to assign the specified status to a fault and drop the fault from further simulation.

The status argument is a string constant containing a two-character VC Z0IX status code. (See “Interpreting Fault Status”.) The `signal_list` argument is an optional list of sense points. These signals are treated as strobe points during fault generation. A signal list is required for consideration of the faults by testability and Fault Campaign Manager.

This task is only allowed in the faulty machine. If the task is called from the good machine, a warning message is reported, as shown in the example message below:

```
Warning-[FSIM-GM-CALL] Ignoring good machine task call
./src/strobe.sv, 20
$fs_drop_status cannot be called by the good machine.
```

Syntax

```
$fs_drop_status(<"status">[, <signal_list>]);
```

\$fs_set_status (system task)

This system task is added to the testbench at the points where the testbench is checking the results of a simulation. It causes the simulator to assign a status to a fault and continue to simulate the fault.

The status argument is a string constant containing a two-character VC Z0IX status code. (See “Interpreting Fault Status”.) The `signal_list` argument is an optional list of sense points. These signals are treated as strobe points during fault generation. A signal list is required for consideration of the faults by testability and Fault Campaign Manager.

This task is only allowed in the faulty machine. If the task is called from the good machine, a warning message is reported, as shown in the example message below:

```
Warning-[FSIM-GM-CALL] Ignoring good machine task call
./src/strobe.sv, 20
$fs_set_status cannot be called by the good machine.
```

Syntax

```
$fs_set_status(<"status">[, <signal_list>]);
```

\$fs_default_status (system task)

Causes the simulator to assign a status to all faults for which a status was not set by `$fs_drop_status()` or `$fs_set_status()`. This is typically the undetected status. The status argument is a string constant containing a two-character VC Z0IX status code. (See “Interpreting Fault Status”.) This task is only effective in the good machine.

If no default status is set using `$fs_default_status()`, the default status is ND.

Syntax

```
$fs_default_status("status");
```

\$fs_get_status (system function)

Causes the simulator to get the status of a fault. A string constant containing a two-character VC Z0IX-specific status code is returned. This function is only effective in the faulty machine. If this function is called from within the good machine, an empty string is returned.

Syntax

```
#100 status = $fs_get_status();
```

The system tasks that update fault statuses (`$fs_set_status`, `$fs_drop_status`, `$fs_default_status`, `$fs_set_status_onevent`, `$fs_drop_status_onevent`) cannot use reserved fault statuses in their task calls. The testability statuses (NS, NI, NO, NC, NT, UO, UI, UR, UT, UB, UU, UC), NA, or any redefinition thereof are not allowed. VC

FCC will report a warning as follows when a task call is made with one of those reserved fault statuses:

Example Message:

```
Error-[FCC-INVALID-STATUS-SYS-CALL] Invalid status used in update status
system call.
./src/strobe.sv, 37
'NI' is an invalid status for system call '$fs_set_status()'.
```

Ensure that the status is defined and is not a testability status, NA, or a redefinition thereof.

Placing User-Controlled Detection Tasks

These tasks must be placed in the testbench where there is a pass/fail condition. A testbench may have multiple locations where this is determined, and all locations must have `$fs_set_status()` or `$fs_drop_status()` added.

The tasks must be executed in the FM. However, it is sometimes difficult to know how a FM gets created. There needs to be a connection between the DUT and the testbench for it to diverge using the signals necessary to determine the pass/fail condition. It is up to the user to locate the pass/fail locations and determine the necessary net connections.

Note that executing `$fs_set_status()` or `$fs_drop_status()` based on simulation time on the assumption that it will affect all faults does not have the desired effect, since the testbench may not be diverged for all FMs. A call based on time will affect only those FMs that have diverged the testbench.

Using a task/function variable as an argument to a fault strobe system task is detected as an error. A function and its argument values only exist while the function is executing. You should only compare signals that are persistent within the module. See “Example in Signal Value Strobing” for an example of comparing synchronized signals.

Scenario 1: A simple testbench contained in a single module

In this case, any fault that reaches a DUT output causes the testbench to diverge. The `$fs_set_status()` or `$fs_drop_status()` tasks can be used freely within this simple testbench since all FMs that propagate to a DUT output will diverge the testbench.

Scenario 2: A complex testbench spanning several files/modules

In this case the user must be sure the net(s) within the DUT that are necessary to indicate pass/fail are connected to the module containing the `$fs_set_status()` / `$fs_drop_status()` tasks. This connection can be made by nets that pass from the DUT into the testbench through a port or a hierarchical reference.

In either scenario do not use `begin_faultfree/`end_faultfree. Use of `begin_faultfree/`end_faultfree prevents FMs to diverge the testbench.

Comparing Differences From Non-Signal Based Expressions

The valid signal list arguments for \$fs_compare, \$fs_drop_status, and \$fs_set_status can consist of packed variables, bit-selects of packed variables, packed/unpacked nets, bit-selects of packed/unpacked nets, or elements of unpacked nets. Any other expression is not allowed and results in compilation errors.

To compare the results of non-signal based expression types, first assign the value to a signal that can be correctly passed to these routines. Declare the signal with the correct bit-width of the resulting expression value. You should have a unique signal for each expression that is compared this way for identification within fault dictionaries.

Example:

```
reg [<size of expr>:1] r;
...
initial begin
...
r = <expr>;
  cmp = $fs_compare(r);
    if (cmp)
      $fs_drop_status(<status>, r, <signals in expr>);
...

```

Where,

- <expr> can be anything legal on the right hand side of a procedural assignment.
- <size of expr> is the bit-width of the resulting value of <expr>.
- <signals in expr> is an option to list all variables used in <expr> for reporting in a fault dictionary.

Example in Signal Value Strobing

The following example compares three signals. If the result of the \$fs_compare is 1 (the typical case of detection of an unsafe fault), the faults are set to DD and dropped from simulation. If the result of the \$fs_compare is 2, the faults are set to PT and continue simulating.

```
int compare;
always @(valid)
begin
  compare = $fs_compare(sig1,sig2,sig3);
  if (1 == compare)
    $fs_drop_status("DD");

```

```
else if (2 == compare)
$fs_set_status("PD");
end
```

Fault Sampling

Fault sampling is a technique used to reduce the number of faults simulated, while achieving a result that is close to the coverage result if all faults are simulated. VC Z01X provides three methods for calculating the sample size. Percentage based sampling calculates the number of faults based on the percentage given by the user. Fixed number sampling enables the user to directly specify the sample size to be generated. Confidence Interval sampling uses mathematical formulas to determine the required number of faults based on a margin of error (confidence interval) and the confidence level given by the user.

Fault Sampling Selection

Permanent faults

Permanent faults are selected by creating all faults in the design first. VC Z01X calculates the frequency of fault to be selected. For example, if 10 faults are to be selected out of 100 possible faults, the frequency to select a fault is 10. Next, VC Z01X selects a random starting point in the fault list to begin selecting faults. In the earlier example, if the random position selected is 72, VC Z01X will proceed to select fault 72, 82, 92, 2, 12, and so on.

Transient faults

Transient faults are sampled as fault generation occurs. This process differs from sampling permanent faults, where faults are first created and then the sampling is performed. VC Z01X saves runtime and memory when sampling fault universes that can be very large by using this tactic to sample transient faults and by only creating the data structures for faults that are chosen as part of the sample.

When creating sampling information for transient faults, VC Z01X calculates the size of the fault universe by identifying the number of physical locations in the design that are requested by the user for faults. For each location, the number of individual faults is determined from the number of cycles specified in the SFF input.

Example:

```
FaultGenerate {
Timing ("clock1", CycleTime 5ps, Offset 1ps)
UseTiming("clock1")
NA ~ (20:100) {FLOP "test.dut.blkA.**"}
}
```

If the example above identifies 100 flops to be faulted, each flop will have 80 possible faults injected based on the cycles specified, resulting in 8,000 faults. This becomes the basis for fault sampling. Faults will be sampled from this set across all physical locations and cycle times.

Consideration of Testable vs. Untestable Faults

When selecting faults to fulfill the sample number, only testable faults are considered by default. Faults in the Untestable Group (UG) are not part of the sample.

Fault Sampling and Exclude Blocks

The *Exclude* blocks do not affect the generated fault sample. Statements in the SFF file are executed sequentially and exclude blocks are considered after the fault sample is created. When using both sampling and exclude blocks, the fault sample size may appear to be different than what you expect because the sample is created first and faults are excluded from the specified hierarchies if they exist in the sampled fault list.

7

Modeling

Fault simulation is memory and compute intensive, so it is beneficial to have a library designed with fault simulation in mind. Modeling can mean the difference between a fault simulation running for hours versus weeks.

This chapter describes how different types of modeling affect fault simulation. It includes the following:

- [Verilog and SystemVerilog Support](#)
 - [Using Timing with Serial Fault Simulation](#)
 - [Fault Simulation of Behavioral Logic](#)
 - [Constructs to Avoid](#)
 - [Fault Propagation and Port Isolation](#)
 - [Transient Fault Behavior](#)
-

Verilog and SystemVerilog Support

VC Z01X is a complete simulator that supports the Verilog and SystemVerilog language and features as described in the Verilog IEEE 1364-1995, 1364-2001, 1364-2005 standards and the SystemVerilog IEEE 1800- 2005 and 1800-2009 standards.

VC Z01X supports full timing in serial fault simulation, and timing checks for logic simulation.

SystemVerilog supported constructs are recommended for use in the testbench only and should not be used in areas of the design where faults will propagate.

See [Extended VCD \(eVCD\)](#)for more information.

See [Appendix C: Language Support](#) for a list of Verilog constructs that are not supported.

Considerations for UDP Models

UDP models for fault simulation should be written for both correct logic behavior and unexpected inputs/transitions. Unexpected, often missing, inputs/transitions may be

encountered when faults are present. A missing table entry will result in an unknown value on the output of UDPs and can cause potential faults (PD/PT). For example, level sensitive inputs should ignore edge transitions.

Using Timing with Serial Fault Simulation

VC Z01X can fault simulate using full timing (gate delays, path delays, negative timing checks) including SDF back-annotation. However, in general the more timing detail used the slower the fault simulation will run. If possible, fault simulation should be done with unit delay on sequential cells and zero delay on combinational cells. This is sometimes referred to as functional simulation.

Although a library constructed for functional simulation is preferred, VC Z01X provides several compile time options to control delay modeling.

Fault Simulation of Behavioral Logic

VC Z01X fully supports fault simulation of designs that contain behavioral and RTL (Register Transfer Level) constructs. VC Z01X is optimized for gate level simulation, and Synopsys recommends using gate level representations of models whenever possible.

There is a host memory cost for having significant behavioral devices in a fault simulation. Each of these constructs may require a large amount of memory if the device being modeled is large. The size of a construct may limit the number of faults that can be run on a given fault simulation pass.

Constructs to Avoid

Certain Verilog constructs are not supported in fault simulation. These constructs are described in detail as follows:

See the following subsections:

- [Hierarchical References](#)
- [Forcing](#)
- [\\$stop\(\)/\\$finish\(\)](#)
- [PLI](#)

Hierarchical References

Hierarchical references from a testbench are allowed during fault simulation, primarily those used in data displays and hierarchical forces to initialize a known state in certain models in the circuit. Hierarchical references internal to the device under test are not recommended for fault simulation because they bypass the testbench-DUT boundary for stimulus application or reading values.

Forcing

Verilog force and deposit statements are supported for both logic and fault simulation but are not recommended for use in fault simulation. Force and deposit do not override fault origins, but they can override propagation effects. Resulting fault simulation coverage can be inconsistent. If a force/deposit is applied while fault effects are present, a warning occurs

When force is imposed on a signal in fault simulation, behavior is as follows:

- **Good machine source:** If the good machine is the source of the force, both the good machine signal and any diverged faulty machine signals are forced (unless the faulty machine is diverged at the source of the force statement).
- **Faulty machine source:** If a faulty machine is the source of a force, only the faulty machine signal is affected.

When a forced signal is released, it changes to its unforced value on both the good machine and faulty machines. The exception is if the faulty machine is diverged at the source of the release statement. Fault effects that propagate to a forced location are blocked during fault simulation and cause faults to not be detectable.

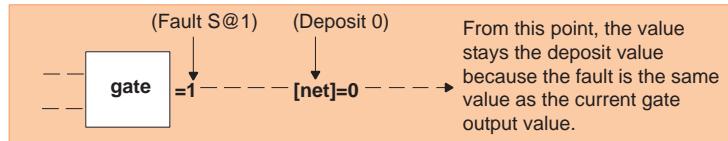
Forcing a Signal Before Fault Injection

There are some conditions where it is necessary to force a signal to a known value. For example, it may be necessary to reset all the latches to a known value or to provide stimulus for a parallel load scan pattern. If so, Synopsys recommends using a force command ahead of fault injection, at the start of simulation. In this case, fault simulation behaves as expected.

Understanding the Difference Between \$force and \$deposit

Synopsys does not recommend using deposit to force a value during fault simulation. Use force instead. A problem can occur if deposit is used to set a net value before faults are injected. If a gate is already driving a value 1, and the deposit is applied to the net with a value 0, the fault origin S@1 is placed on the gate output, but the gate is already driving a 1 value. The net is a separate object in the simulator and maintains its own value. This

situation is not remedied by delaying fault injection until after the deposit. This is an event driven simulation behavior that is specific to using `$deposit`.



A force will operate differently in this situation because it can be released of the force value. The release can be applied to the forced net, and it automatically changes the net to the value of the driving gate because it creates a value change event and propagates the fault effect.

\$stop()/\$finish()

Often `$stop()`/`$finish()` is used to stop a logic simulation on an error condition. If a FM executes `$stop()`/`$finish()`, the fault is marked as detected `$finish/$stop` (DF) and dropped from simulation.

PLI

PLI calls should not be used within the design under test. The PLI is not designed to be used with concurrent fault simulation because there is no way to diverge the PLI memory.

Consider a module which contains a PLI call used to model a memory. When a fault causes this module to diverge, the simulator makes a copy of the diverged module as the fault machine (FM). This copy simulates in parallel with the fault free copy (GM). However, there is only one copy of memory allocated by the PLI. When a memory write occurs it is not possible to tell if the write occurred in the GM or FM. So if the write occurred in the FM and the GM were to subsequently read this address the simulation results would be incorrect.

Fault Propagation and Port Isolation

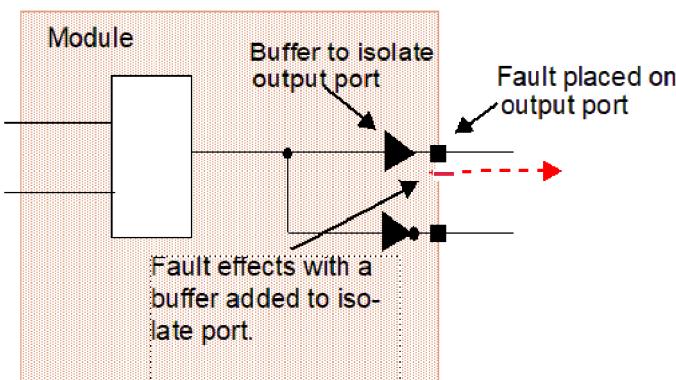
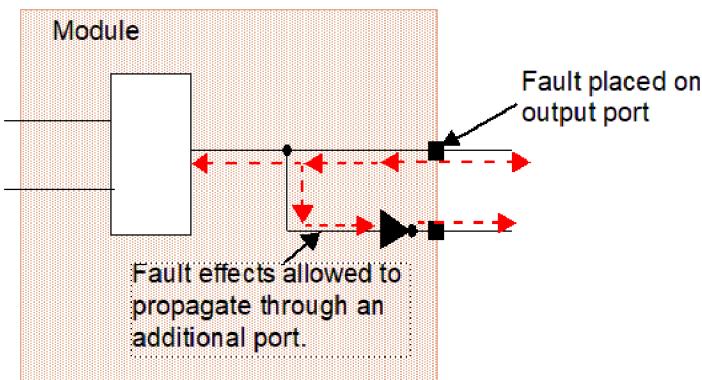
Synopsys recommends that cell models be defined to create ports that are isolated. Each port should be modeled internally such that it isolates the fault effects from undesired propagation paths. An input port should not have any internal drivers (only internal fanouts) and an output port should only have internal drivers (no internal fanouts). VC Z01X protects against back propagation of port faults by placing the fault on the loads of the port instead of the driver.

The cell should ideally be remodeled in a manner that isolates the port, similar to what is shown in the diagrams below. The remodeling will not affect the fault results, but may help eliminate any confusion about how the fault effects propagate in the design. Remodeling

your cells to establish port isolation will also assist in using Verilog force statements to replicate the behavior of stuck-at faults.

See the following figure. This example shows the effects of a "buf" placed between the port and its driver to isolate the port from the internal fanout. VC Z01X models the output fault in the manner of the second diagram below to effectively isolate the port from back propagation of values.

Fault Propagation and Port Isolation



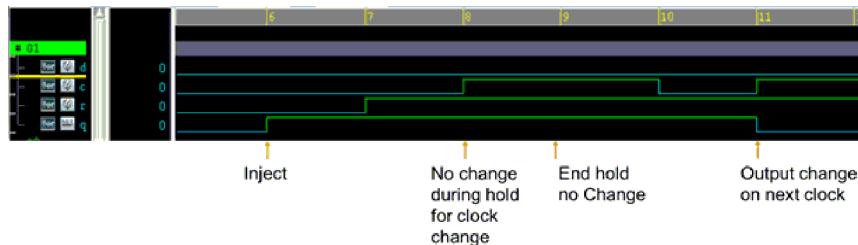
Transient Fault Behavior

The transient faults behavior is based on Verilog force/release for a hold fault and a \$deposit for the SEU. When the target is a storage element like an RTL register or sequential UDP, then the value of the target will change when injected and when a new assignment is made. The force is used to model the transient hold value for one or more clock cycles. Upon release, the storage element maintains the current value until the next assignment is driven by an input change. In a similar way a\$deposit is used for a single

cycle SEU; the storage element maintains the deposit value until the next assignment is driven by an input change.

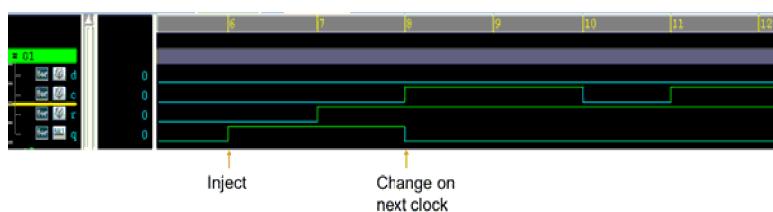
Transient Hold

Example: NA ~ ([6^9]) {PORT "test.d1.q"}



Transient SEU

Example: NA ~ (6) {PORT "test.d1.q"}



8

Fault Models

This chapter describes the fault models available for running fault simulation with VC Z01X. It includes the following:

- [Fault Model Concepts](#)
 - [Stuck-At Faults](#)
 - [Transient Faults](#)
 - [MFO Faults](#)
 - [Transition Faults](#)
 - [Fault Descriptors](#)
 - [Fault Collapsing](#)
 - [Fault Placement](#)
-

Fault Model Concepts

A good fault model mimics real-life defects sufficiently such that when faults are detected by a test in simulation, the test will also detect defective parts in production. Real-life defects are caused by numerous environmental and aging factors or manufacturing variations, and can include shorts to power, shorts to ground, open transistors, shorted transistors, bridged interconnects and open interconnects. Defects like these cause a variety of unintended physical and simulated behavior. Some defects cause the combinational logic to behave as latches; some cause the timing of the circuit to change; some cause oscillations; some stop logic from changing state.

You specify fault models when faults are generated. Simulation results vary significantly based on the fault model used. You instruct VC Z01X which fault model(s) to use that will best measure the quality of your test.

VC Z01X supports the following functional safety fault models in addition to the common manufacturing faults (stuck-at):

- **Stuck-At Faults**, which are s@0 and s@1 faults on primitive input and output terminals, module ports and nets, and arrays.
- **Transient Faults**, which are faults that occur once and then disappear.

Stuck-At Faults

The most prevalent fault models are the stuck-at-0 and the stuck-at-1. The stuck-at fault model holds an input or output of a gate to a zero or one state throughout the simulation. Single stuck-at faults are created with the VC Z01X fault campaign compiler tool. Multipoint stuck-at modeling is offered by VC Z01X through the Standard Fault Format.

VC Z01X Stuck-At Fault Types and Locations

There are six types of stuck-at faults that can be modeled:

- Input faults: Cause a given input of a Verilog primitive, UDP, or module to be stuck at a level.
- Output faults: Cause the output of a Verilog primitive, UDP, or module to be stuck at a level.
- Net faults: Cause net to be stuck at a level. Automatically enabled by `-fsim` switch.
- Var: Place faults on variables of type reg, logic, bit, and byte. Variables used only in procedural statements, which are generally statements within initial and always blocks. Variable faults are automatically enabled with `-fsim` switch during compile.
- Array: Place faults on bits of reg, logic, bit and byte array types.

In each case, the fault may be tied to power (“stuck at one” or s@1) or tied to ground (“stuck at zero” or s@0).

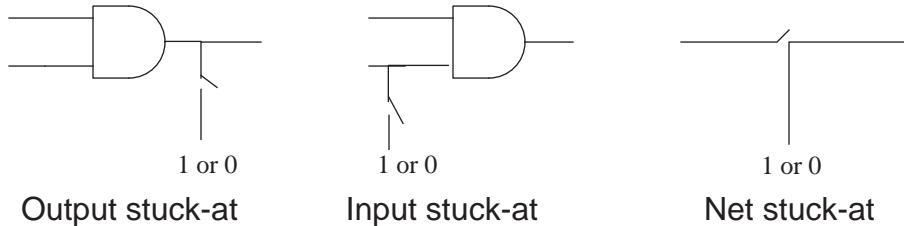
WIRE and VARI faults can be generated with only `-fsim` compile switch. Port faults generation requires `-fsim=portfaults` compile switch. The following figure is a representation of these stuck-at fault types.

Note:

VC Z01X does not support Stuck-at fault on continuous assignment statement. Hence it reports a warning during `vc_fcc` and `vc_fcc` does not generate any fault for “ASGN” fault class. Example `Warning: Warning- [FCC-UNSUPPORTED-FAULT-CLASS] Unsupported fault class ./COLLATERAL/`

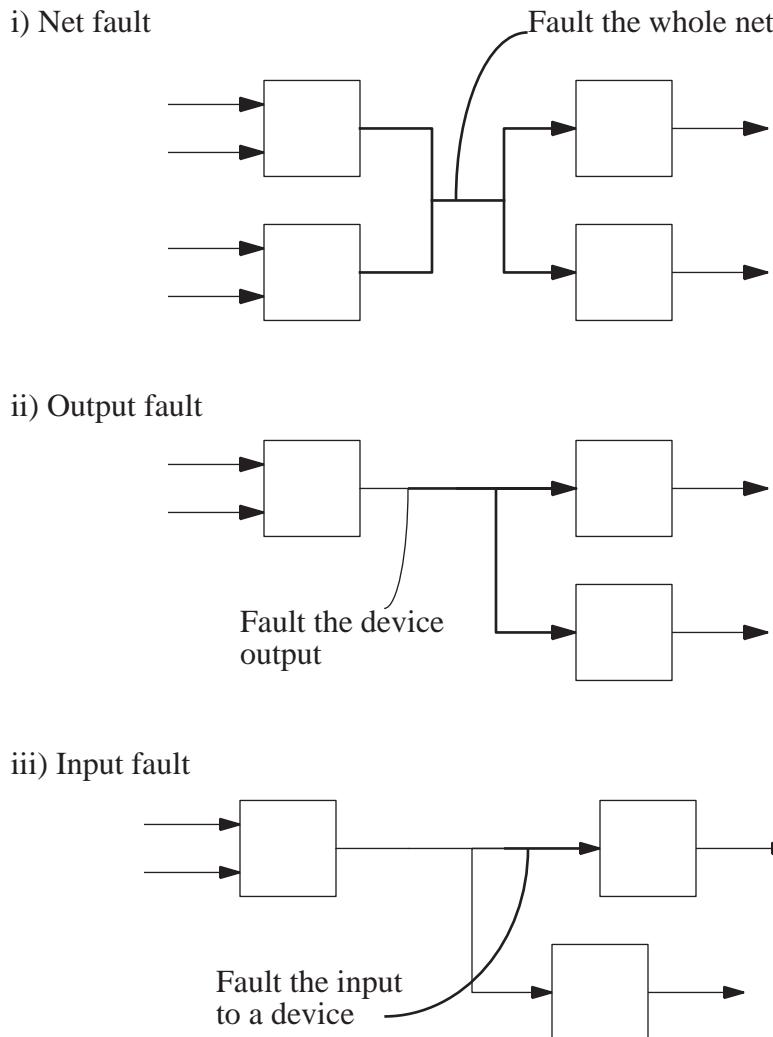
SFF/faults.sff, 3 Fault class 'ASGN' is not supported and will be ignored.

Figure 8 Types of Stuck-At Faults



See “Stuck at Fault Locations” figure to see where faults are placed for each stuck-at fault type.

Figure 9 Stuck-At Fault Locations



Transient Faults

Transient faults are faults that occur once and then disappear. They may appear at random times and do not result in permanent hardware damage. Transient faults, or soft errors, as they are sometimes known, are caused by interference such as external radiation strikes or power-supply spikes. They are commonly used in Functional Safety (ISO-26262/IEC-61508) applications to verify that designs can detect and mitigate soft errors. In security environments they can represent attacks aimed at breaking the security measures of the hardware.

VC Z01X transient fault model includes the following fault type:

- **Transient toggle:** Used for both function safety and security applications. It injects a fault location bit-flip at specified cycle times by inverting the GM value of the fault location.

The fault database must be created using VC Z01X's Standard format. See [Standard Fault Format](#).

Notes:

- Transient fault placement is restricted to outputs only. When generating a transient fault on an input, an error is reported. The generation of transient faults on wild-carded locations automatically limits fault placement to outputs.
- Transient fault origins are injected at the end of the time cycle. If a transient fault origin is released following the duration of a hold range, the release will also occur at the end of the time cycle. If a fault contains simultaneous injections and releases, all releases will occur prior to any injections.
- Transient faults allow for multiple fault origins and the possibility of conflicting fault effects.
- Transient toggle fault locations that remain in an unknown X state for the duration of the testability evaluation window are Not Controlled.
- For transient-hold faults placed on sequential output logic, the effect of the fault release is not realized until the next input change. For transient-hold faults placed on combinational output logic, the effect of the fault release is realized immediate at the release time, taking into account the current input state, feedback, and delayed events scheduled for release time on the device.

Suggested Methodology

Importing Transient Faults and Creating a Coverage Report

1. Create a fault definitions file using the Standard format `fdef` specifications. See [Transient Zero and Transient One Fault List Format](#) and [Transient Toggle Fault List Format](#).
2. Import the fault list file with `vc_fcc` and run fault simulation with fault campaign manager. The `vc_fcc` tool can be used independently or can be called from the Fault Campaign Manager. To create a Coverage report with coverage and definition data for all signals, regardless of whether there was a difference between the GM and FM, see [Fault Descriptors](#).

```
create_campaign -args "--full64 -daidir simv.daidir -sff standard.txt
-dut_path
```

```
<dut hierarchy path> -campaign fc1"
create_testcases -name test1 -exec "./simv" -args "+test1"
create_testcases -name test2 -exec "./simv" -args "+test2"
report -report vczoix_.sff -overwrite -showfaultid
```

Where:

- `create_campaign` command will call `vc_fcc` and generate fault campaign. For existing fault campaign, `set_campaign` should be used to load existing fault campaign.

```
set_campaign -campaign fc1
```

3. Review the Coverage Report

The initial statuses in the Standard fault definition file are updated with the statuses after simulation.

Also see:

- Example Coverage Report (No Dictionary)
- Transient Zero and Transient One Fault List Format
- Transient Toggle Fault List Format

Notes:

- An error is issued for invalid cycle sequences in transient faults. If a cycle resolves to an earlier time value than a preceding cycle in the transient fault sequence, the order is considered to be invalid. An error message is output, and the fault is not created.
- If a transient fault PORT location does not back trace to any sequential element (UDP, register, etc.) a warning message will be output, and the fault will not be generated.

Importing Transient Faults and Creating a Test-Level Coverage Report With Integrated Dictionary Data

1. Create a fault definitions file using the Standard format `fdef` specifications. See *Transient Zero and Transient One Fault List Format* and *Transient Toggle Fault List Format*.
2. Import the fault list file with `vc_fcc` and run fault simulation using Fault Campaign Manager. The following is a sample script for simulating transient faults and creating a Coverage report with coverage and definition data for all signals, regardless of whether there was a difference between the GM and FM.

```
create_campaign -args "--full64 -daidir simv.daidir -sff standard.sff
-dut_path <dut hierarchy path> -campaign fc1"
create_testcases -name test1 -exec "./simv" -args "+test1" -fsim_args
"--fsim= fault+dictionary"
```

```
create_testcases -name test2 -exec "./simv" -args "+test2" -fsim_args
"--fsim=fault+dictionary"
fsim
report -report vczoix.sff -overwrite -showfaultid
```

3. Review the Dictionary Report

The initial statuses in the Standard fault definition file are updated with the statuses after simulation. Report can be found inside fcm.dir.

Example:

```
StrobeData {
  StrobeList {"strobe4"
    Location {"$fs_drop_status, src/decoder.v : line 60"}
    Pins {
      0 " test.risc1.instdec.opcode";
    }
  }
  StrobeList {"strobe7"
    Location {"$fs_set_status, ./src/strobe.sv : line 42"}
    Pins {
      3 " test.risc1.alu_out";
    }
  }
  StrobeList {"strobe2"
    Location {"$fs_drop_status, src/decoder.v : line 46"}
    Pins {
      0 " test.risc1.instdec.opcode";
      1 " test.risc1.instdec.not_reset";
    }
  }
  StrobeList {"strobe6"
    Location {"$fs_set_status, ./src/strobe.sv : line 36"}
    Pins {
      3 " test.risc1.alu_out";
    }
  }
  FaultList {
    ["strobe2", 4000ps,
     1 GM: 0
     FM: 1]
    ON 1 {PORT "test.risc1.reseter.reset"}
  }
}
```

Also see:

- Example Coverage Report (No Dictionary)
- Transient Zero and Transient One Fault List Format
- Transient Toggle Fault List Format

MFO Faults

Multi fault origin (MFO) faults provide a mechanism for combining multiple fault locations under a single fault definition. All locations defined within an MFO are simulated in the same faulty machine (FM).

Basic MFO Example:

```
NA 1 {PORT "i1.F1" } + 1 { PORT "i2.F3"}
```

This fault is simulated with an SA1 on i1.F1 and an SA1 on i2.F3 in the same faulty machine. An MFO fault has a single fault status (there is no per-location status information).

An equivalent fault could also be defined using this alternate syntax:

```
NA 1 {PORT "i1.F1" + PORT "i2.F3"}
```

Both syntaxes can be combined to make more complex MFO definitions:

```
NA 1 {PORT "i1.F1" } + 1 { PORT "i2.F3" + PORT "i2.F4" }
```

MFO faults containing wildcarded fault locations are supported within the *FaultGenerate* block.

```
NA 1 {PORT "i1.*" } + 1 { PORT "i2.*"}
```

This fault will expand to all possible combinations of the wildcarded locations.

```
NA 1 {PORT "i1.F1" } + 1 { PORT "i2.F3" }
NA 1 {PORT "i1.F1" } + 1 { PORT "i2.F4" }
NA 1 {PORT "i1.F2" } + 1 { PORT "i2.F3" }
NA 1 {PORT "i1.F2" } + 1 { PORT "i2.F4" }
```

Reporting

The `vc_fdb_report` always displays MFO faults in expanded form. MFO faults do not collapse in

```
vc_fcc.
```

Static/ Dynamic Testability

MFO faults are not currently marked UB/NO.

For other testability statuses such as UT/NC, an MFO can be considered untestable if all individual model/location pairs can be marked untestable (not counting UB/NO).

If individual markings are a mix of N* or U*, the fault is marked according to these rules:

```
N* - MFO marked NT
NT 1 {PORT "i1.F1" + PORT "i2.F3"}
1 PORT "i1.F1" - NS
1 PORT "i2.F3" - NC
U* - MFO marked with untestable status of first model/location pair
UU 1 {PORT "i1.F1" + PORT "i2.F3"}
1 PORT "i1.F1" - UU
1 PORT "i2.F3" - UT
```

SFF MFO Input/ Output Format

The following is the SFF MFO input/ output format:

```
<mfo> ::= <status> <fault_location_hier> [ '+' <fault_location_hier> ]*
<fault_location_hier> ::= <fault_type> '{' <fault_class> <fault_location>
[ '+' <fault_class> <fault_location> ]* '}'
```

where,

<status> - Two-character fault status

<fault_type>- 0/1

<fault_class> - WIRE, PRIM, PORT, ARRY, VARI, FLOP

<fault_location> - Quoted string (wildcarding is supported in *FaultGenerate*)

Example:

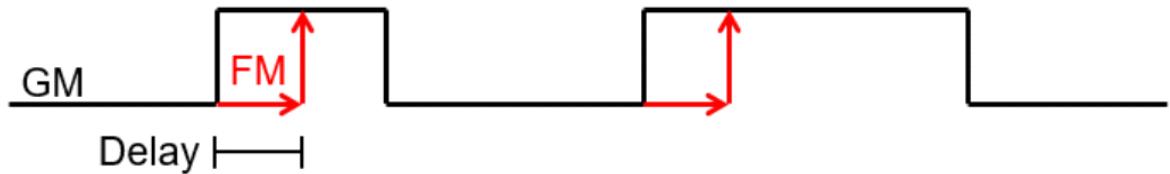
```
NA 1 {PORT "i1.F1" } + 1 { PORT "i2.F3" + WIRE "i2.w" } + 0 { PRIM
"i1.p.0" }
```

Transition Faults

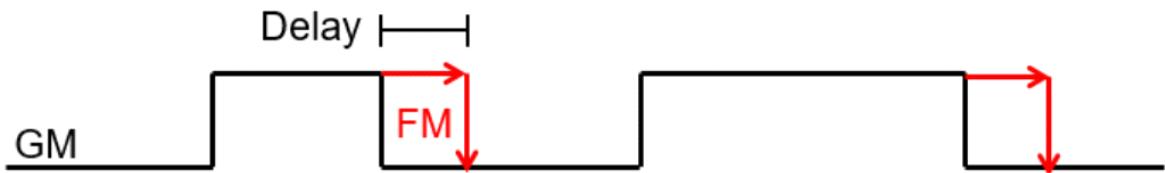
A transition fault is a fault model placed on a wire, primitive, or port that causes a delay on rising or falling edge transitions. There are two types of transition fault, Slow-to-Rise (R), and Slow-to-Fall (F). A Slow-to-Rise fault is a fault that has a delay rising from 0 to 1, X or Z, or from Z to 1 or X, and a Slow-to-Fall fault has a delay falling from 1 to 0, X or Z, or from Z to 0 or X.

Transitions caused by force/release behavior are also delayed. The behavior for these transitions matches with that of non-force/release transitions.

Slow-To-Rise



Slow-To-Fall



Specifying Transition Faults in SFF

Transition faults can be identified in an SFF file by using the R or F fault markers along with the PRIM, WIRE, or PORT fault types.

Example:

```
NA R {PORT "top.A"}  
NA F {PRIM "top.A"}
```

Note:

The R and F indicate whether it is Slow-to-Rise or Slow-to-Fall.

Specifying the Delay

When simulating transition faults, a delay duration must also be specified. This can be done on the simulator command line using `-fsim=trans+delay+<n>`.

Example:

```
simv -fsim=trans+delay+5  
simv -fsim=trans+delay+5ns
```

This sets a global delay for all transition faults in the simulation.

The delay can also be specified on a per-fault basis in the SFF file by including the desired delay before the fault descriptor. A per-fault delay overrides the global delay for that particular fault.

Example:

```
NA R (2ps) {PRIM "test.dut.**"}
```

Transition Fault Collapsing

Transition faults in SFF can be collapsed when they share the same delay and have a single drive and a single fanout.

Transition Fault Merging

When merging SFF files, transition faults would only be merged if they are on the same location and have the same delay.

Fault Descriptors

A fault descriptor identifies the type, status, and location of a fault. The Standard Fault File (SFF) format allows description of complex fault models such as transient and multi-point models. It is a flexible format, capable of describing virtually any kind of fault. Included in the format are many features that will allow companies to customize their output.

There are three fields in the fault descriptor that use abbreviations to describe aspect of the fault type and behavior. These abbreviations are explained in the tables below, and include Class, Status, and Type.

Class is one of the following:

ARRY	Array fault ^{2,3}
PORT	Port fault ³
PRIM	Primitive input or output fault
VARI	Variable fault ^{1,2,3}
WIRE	Net fault
FLOP	Sequential UDP fault ³

¹ Use of the VARI class combined with an array <location> results in an expansion based on array words. A VARI fault is generated on each individual word of the array.

² Use of the VARI or ARRY class combined with a vector <location> results in an expansion based on the individual bits. VARI or ARRY fault is generated on each individual bit of the vector.

³ ARRY, PORT, VARI, and FLOP classes are supported when using the transient toggle model (~). No other fault classes are supported in combination with this model.

Transient Fault Class Specifiers

It is recommended to specify one or more fault classes when describing transient faults. This directs VC Z01X to generate faults on locations appropriate to the class specified. This allows for customization of fault placement for complex transient faults. The following rules are followed when determining fault location placement:

- [PRIM] class specified: The fault is placed on the corresponding sequential/combinational primitive pin.
- [PORT] class specified: The fault is described on the port. The fault is placed on the corresponding pin of the sequential/combinational primitive.
- [FLOP] class specified: The fault is back traced to the output of the driving memory unit or will iterate the specified location and generate faults on applicable flop/latch locations as reported by VCS. (See “Flagging Flip-Flop and Latch Locations”). This specifier must be used from a port or instance location.
- *No class specified*: When no fault class is specified, VC Z01X defaults to FLOP fault class specifier behavior noted above.

For more details on the fault status, see Fault Status Definition.

Type is one of the following:

0	Stuck-at 0
1	Stuck-at 1
~	Transient SEU or SET fault
sub status 0-9	Excluded fault. Excluded statuses are named EN (0-9). See Verdi Fault Analysis documentation.

For stuck-at and transition faults, the fault origin is the instance name of a primitive or module including the terminal or port.

The VC Z01X fault descriptor identifies the output terminal of a primitive as 0 and the input terminals as the positional number of the terminal starting with 1.

FLOP Fault Class

In addition to the fault classes described in [Fault Descriptors](#), Standard Fault Format (SFF) supports the FLOP fault class. FLOP is a special class used only by SFF to describe elements of a design that hold a signal.

For VC Z01X, FLOP maps to the output pins of sequential user-defined primitives (UDPs) or sequential elements identified during compilation in RTL.

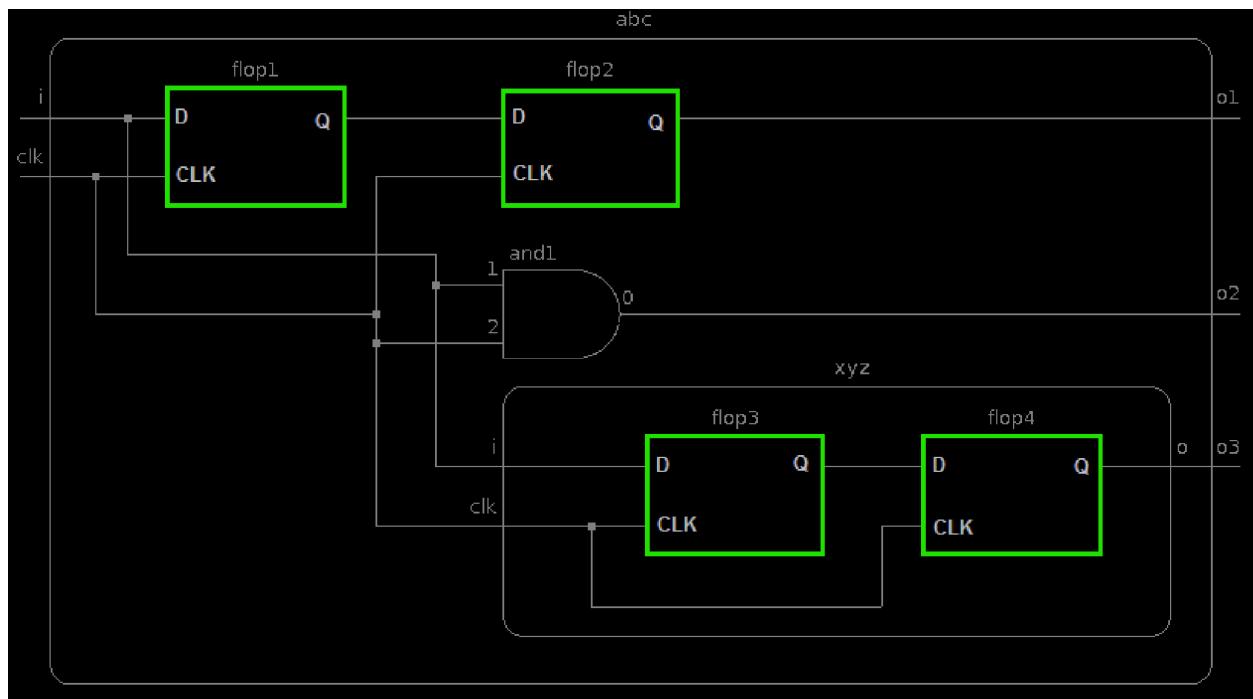
For more information, see “Flagging Flip-Flop and Latch Locations”.

Using the FLOP Fault Class

The FLOP fault class is used to filter fault definitions that map to multiple locations. For example, a wildcarded fault location, which uses FLOP as the fault class, places a fault on all sequential primitives that match the location and does not place any faults on the combinational primitives.

[Figure 10](#) shows a module instance in a larger test design. The instance abc contains several flip-flops and gates, some of which are in the xyz module instance. The flip-flops are highlighted in green.

Figure 10 Schematic for the abc Module Instance



To place a transient toggle fault only on the flip-flops in this design, use the following FaultGenerate block:

```
FaultGenerate
{
Timing("clock1", CycleTime 1ns)
UseTiming("clock1")
NA ~ (100:200) {FLOP "test.abc.**"}
}
```

The coverage report for this example contains:

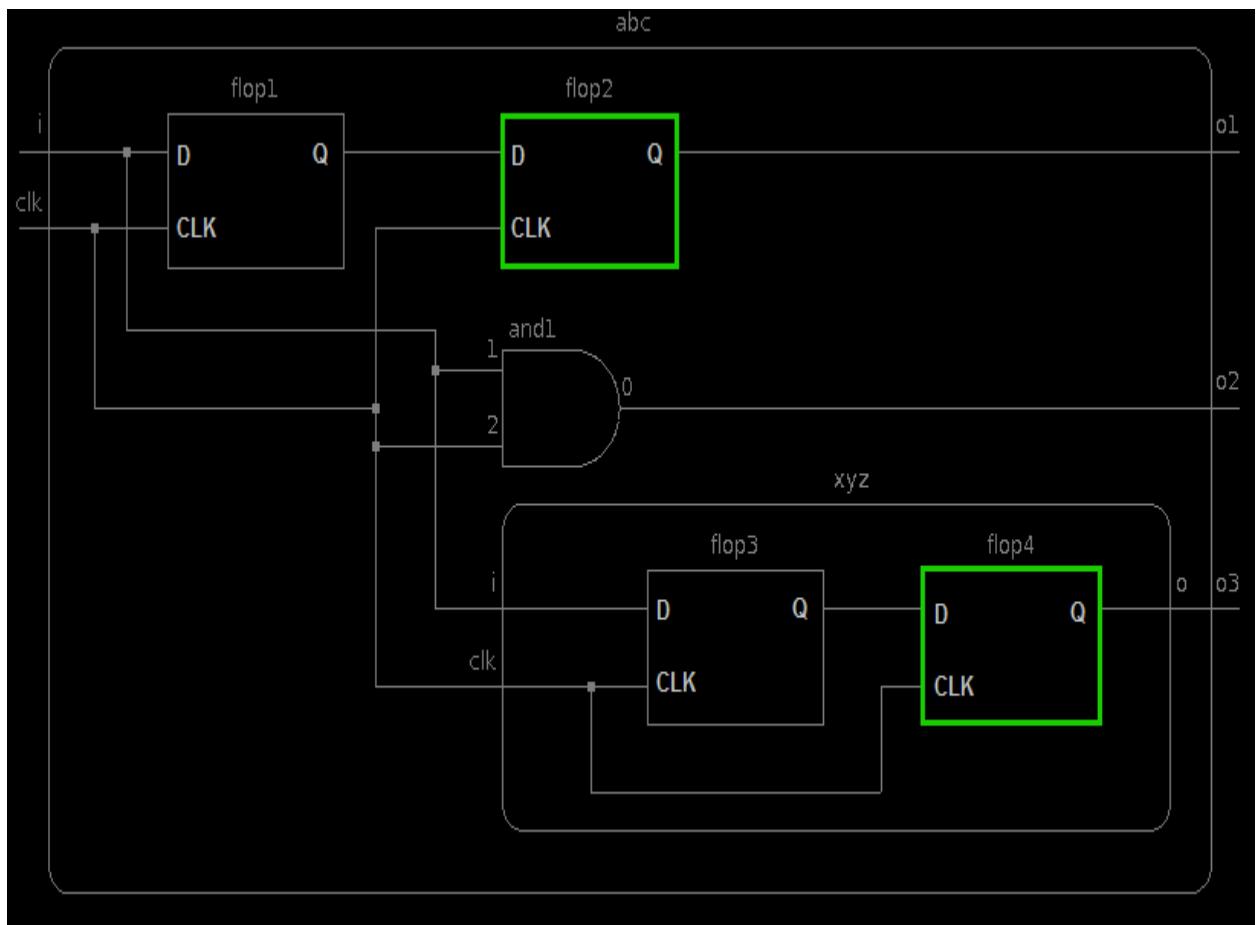
```
FaultList{
Timing("clock1", CycleTime 1ns)
UseTiming("clock1")
NA ~ (100:200) {"test.abc.flop1.Q"}
NA ~ (100:200) {"test.abc.flop2.Q"}
NA ~ (100:200) {"test.abc.xyz.flop3.Q"}
NA ~ (100:200) {"test.abc.xyz.flop4.Q"}
}
```

Behavior With `enable_portfaults and `suppress_faults

The FLOP fault class has a special behavior when used on module instances with port faults enabled. In this situation, only flip-flops that connect to the output and inout ports of the module instance are faulted. See [Fault Generation Compiler Directives](#) for more information on enabling port faults.

[Figure 11](#) shows the flip-flops (flop2 and flop4) that are faulted if module instances abc and xyz are defined in an `enable_portfaults block . Also the same fault descriptor from section [Using the FLOP Fault Class](#) is used.

Figure 11 Schematic for abc and xyz Module Instances (Both Instances Have `enable_portfaults Enabled)



As shown by the figure, flop2 and flop4 have faults placed on them. This is because flop2 is connected to the output port o1 of module instance *abc* and flop4 is connected to the output port o of module instance *xyz*.

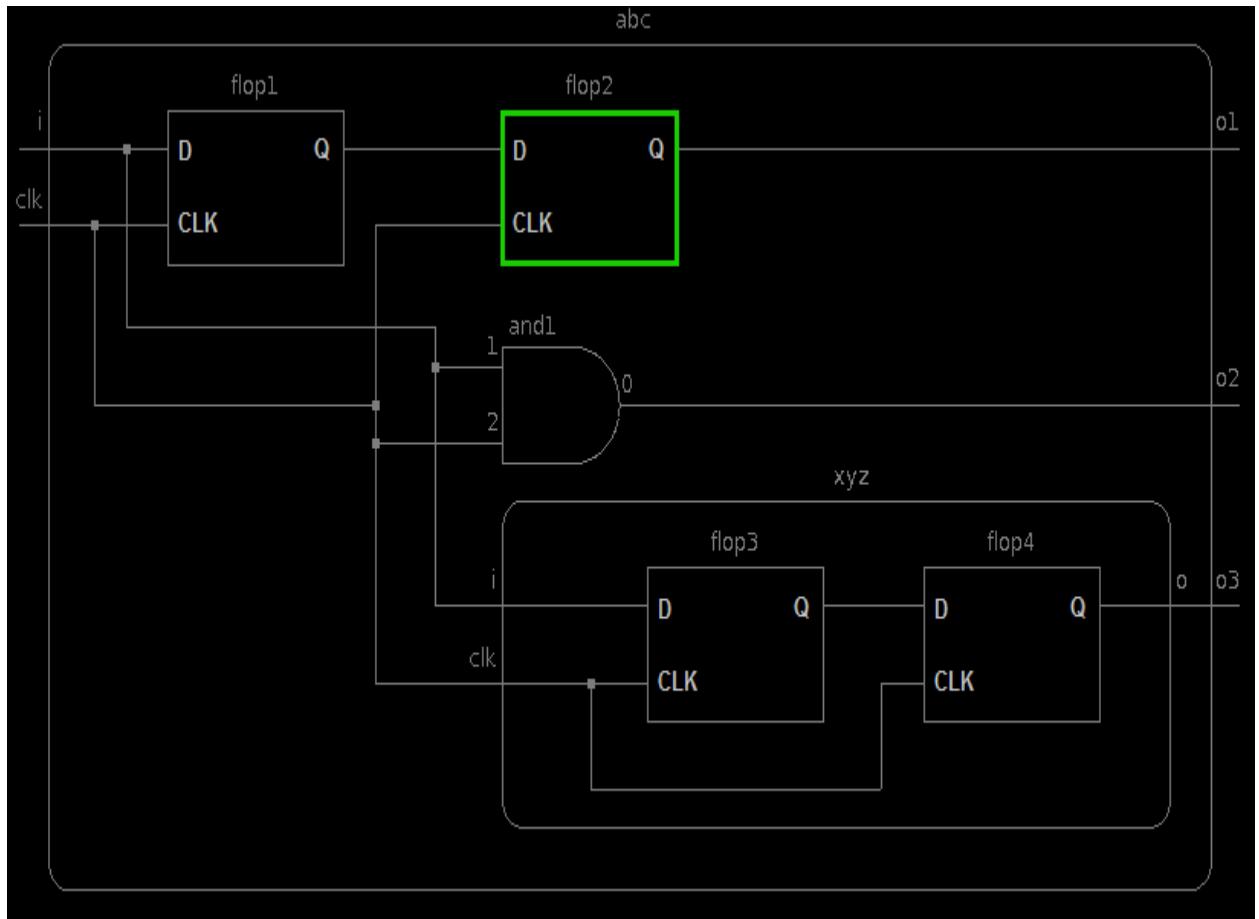
The resulting report output for this example is as follows:

```
FaultList {
  Timing("clock1", CycleTime 1ns) UseTiming("clock1")
  NA ~ (100:200) {"test.abc.flop2.0"}
  NA ~ (100:200) {"test.abc.xyz.flop4.0"}
```

If a wildcarded FLOP fault location encounters a module with both `enable_portfaults and `suppress_faults, only flip-flops at the top level, which connect to output pins, are faulted. If only `suppress_faults is enabled, no faults are placed on flip-flops.

Figure 12 shows the flip-flop (flop2) that is faulted when abc and xyz have `enable_portfaults and `suppress_faults enabled. Also the same fault descriptor from section 10.11.4.1 is used.

Figure 12 Schematic for abc and xyz Module Instances (Both Instances Have `enable_portfaults and `suppress_faults Enabled)



As shown in the figure, no faults are generated for the xyz module instance. A FLOP fault is placed on flop2 because the `enable_portfaults directive is enabled and flop2 is a top-level flip-flop that is connected to the output port.

The resulting report for this example is as follows:

```
FaultList{
  Timing("clock1", CycleTime 1ns)
  UseTiming("clock1")
  NA ~ (100:200) {"test.abc.flop2.0"}
}
```

Also see:

- [Standard Fault Format](#) to learn about the parts of a Standard Coverage Report.
- [Fault Status Definition](#) to learn about creating custom fault statuses and defining their behavior.
- [FaultList Section](#) to learn about viewing fault descriptors in Standard format.
- [Example Coverage Reports Using Standard Fault Format](#) for sample Coverage reports.

Fault Collapsing

To reduce simulation time VC Z01X equates faults which produce the same observable behavior. This process is called fault collapsing, and sometimes referred to as equivalence marking.

Faults are classified as either prime or collapsed. A prime fault is a fault which represents one or more faults. A collapsed fault is a fault which produces the same observable behavior as its equivalent prime fault. Only prime faults are simulated. In coverage reports, collapsed faults are listed directly below the prime fault.

Stuck-at faults are collapsed on an individual primitive according to the following table. Note that equivalent Boolean operators are given for continuous assigns..

Gate	Boolean	Input Fault Collapsed	Prime Output Fault
AND	&	S@0	S@0
NAND	combination of &, ~ assign a = ~(b & c)	S@0	S@1
OR		S@1	S@1
NOR	combination of , ~ assign a = ~(b c)	S@1	S@0
BUF	simple assign a = b	S@0, S@1	S@0, S@1
NOT	~	S@0, S@1	S@1, S@0
XOR	^	Based on untestable faults	Based on untestable faults

Stuck-at faults on the inputs of xor, xnor, bufif, notif, mos, tran and user-defined primitives cannot be collapsed to an output fault.

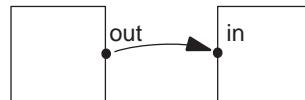
Stuck-at faults on the output of a primitive or cell which has a single fanout collapses to the input of the driven primitive or cell. Stuck-at faults on the output of a primitive or cell with multiple fanouts cannot be collapsed. These faults are not equivalent since the fault at the primitive or cell output affects all fanouts but the faults at the primitive or cell input affect only that primitive or cell.

Variable faults are collapsed to port faults if the faults are on the port fault locations. Only simple variable faults are collapsed (that is, those with only one read in the module).

Port faults will collapse through the cell depending on the content of the cell. For example, input port faults on a cell containing a buf primitive will collapse to the output port of that cell. The following figure shows three examples of port fault collapsing for cell faults (port faults).

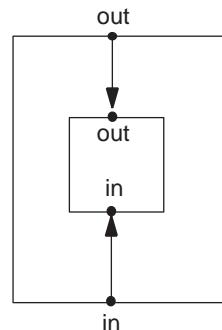
Figure 13 Examples of Cell Fault Collapsing

1) Across Modules



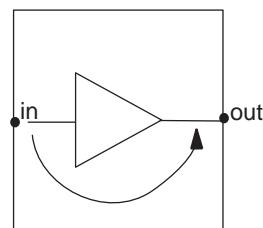
Ports collapse if they are directly connected across modules.

2) Through the hierarchy



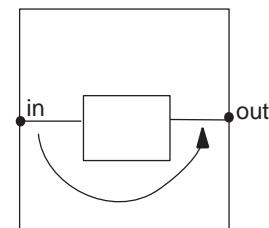
Port faults are directly connected through the hierarchy.

3) Through gates



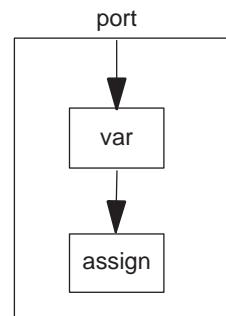
Input and output ports of the same module collapse through a gate.

4) Through continuous assigns



Input and output ports of the same module collapse through an assign.

5) var/port

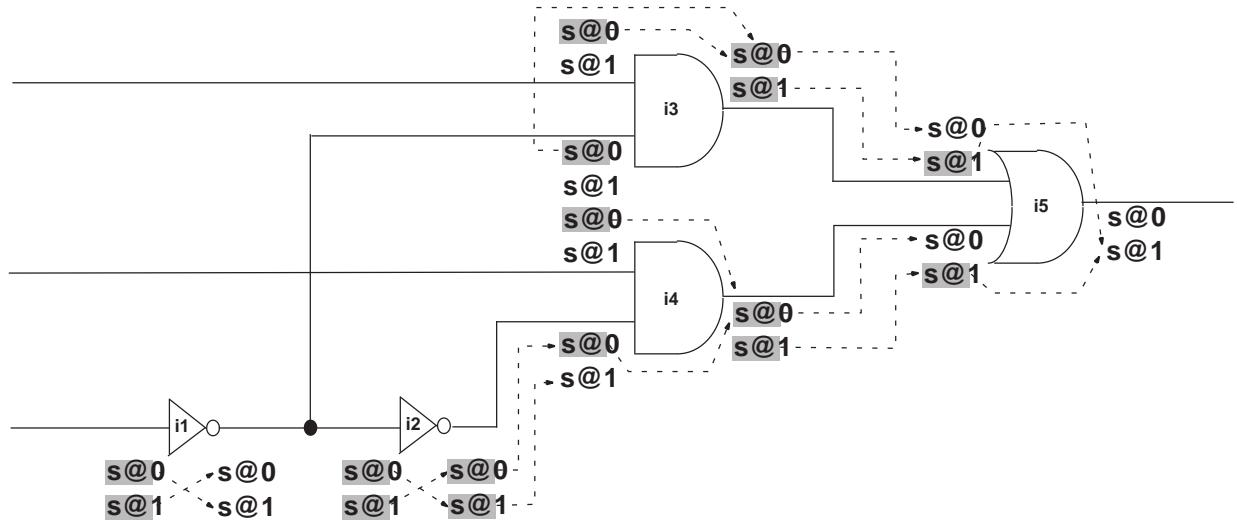


Variable collapses to the simple assign output. The simple assign output assigns to the port.

Figure 2 shows an example of stuck-at fault collapsing. In this example there are 26 total faults, 16 collapsed faults (shaded) and 10 prime faults. Fault collapsing is done as follows:

1. s@0 and s@1 on the input of i1 collapse to the output s@1 and s@0 respectively.
2. s@0 and s@1 on the output of i1 cannot be collapsed because i1 has multiple fanouts.
3. s@0 and s@1 on the input of i2 collapse to the output s@1 and s@0 respectively.
4. s@0 on both inputs of i3 collapse to the output s@0.
5. s@0 on both inputs of i4 collapse to the output s@0.
6. Output faults on i3 collapse to the input of i5 because i3 is single fanout.
7. Output faults on i4 collapse to the input of i5 because i4 is single fanout.
8. s@1 on both inputs of i5 collapse to the output s@1.

Figure 14 Stuck-At Fault Collapsing



Fault Placement

When a fault is injected in simulation it may be placed in a location that is not accessible for any external access at this location. The internal representation of the fault does not

always correspond to a specific Verilog construct. This can affect expected results as viewed from a waveform or when using PLI or CLI forces.

An input port fault is a good example. The port is referenced in the fault description by its port name. The placement in simulation is usually the internal load of that port. The internal loads do not show up in waveform tools.

Fault placement will move the fault location through zero delay buf or not gate. Language and forcelist forces are accounted for limited moving of the faults. PLI or CLI do not prevent moving the fault location.

Using `-fsim=portfaults+diag+hi` (runtime switch) will identify the specific fault location placement in the log file.

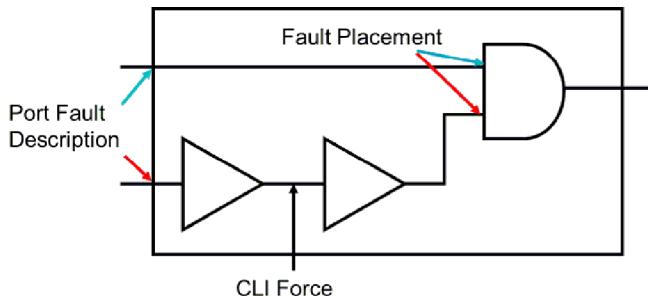
Example:

```
Fault(1) : NN 0 {PORT "test.risc1.alu1.accum[0]"}
```

Place Fault(1) on Behavioral Block in Instance <test.risc1.alu1.,10>

Fault Placement(SUCCESS) for fault(1)

Figure 15 Fault Placement



In the diagram the red and blue arrows represent a single fault showing the description is different than the placement. In the case of the blue line there is a net at the fault description that cannot be monitored for the faulty machine value because the placement is after the net, on the internal gate load terminal. Likewise, the red arrow represents a fault that was moved from the description to the internal gate load through two buf gates.

Moving the location influences the behavior of the CLI force placed on the net between the fault description and the fault placement. If the fault were placed on the first buffer, then the force would override the fault value. The force does not override the value based on the shown placement. This may lead to inconsistent results when using CLI or PLI forces. The Verilog language force and forces found in force list does not have this problem because the simulator can account for these force locations before the fault is injected.

9

Standard Fault Format

The Standard Fault Format (SFF) allows description of complex fault models such as transient and multipoint faults. It is a flexible format, capable of describing virtually any kind of fault. Included in the format are many features that will allow you to customize your fault injection environment.

This section describes the parts of the Standard fault format and provides sample Coverage reports using Standard fault format. It includes:

- [General SFF Considerations](#)
- [Test Information Section](#)
- [Fault Status Definition](#)
- [Coverage Section](#)
- [Strobe Section](#)
- [SafetyMechanism Section](#)
- [FailureMode Section](#)
- [Constraint Section](#)
- [FaultGenerate Section](#)
- [FaultInfo Section](#)
- [FaultList Section](#)
- [Common Keywords/Blocks/Constructs for FaultList and FaultGenerate](#)
- [Example Coverage Reports Using Standard Fault Format](#)

General SFF Considerations

This section provides information on the following:

- [Creating a Standard Fault Format File](#)
- [Header](#)

Creating a Standard Fault Format File

You may create an initial fault definition file in standard format and pass the resulting file into the fault campaign compiler. At a minimum, a file must include either a FaultGenerate or a FaultList section. Transient fault creation requires you to provide the necessary timing data along with the fault locations.

Also see:

- [FaultList Section](#) and the [FaultGenerate Section](#) for the format required when describing a fault.
- [Example Coverage Reports Using Standard Fault Format](#) for sample reports, both with and without integrated dictionary data.

The following table lists the sections of a standard fault file and summarizes the level of customization allowed:

Table 6 Standard Fault File Sections

Section	Purpose	Customization allowed
Header section	Contains information that applies to the file as a whole.	Output only. These features are seen only in the generated output file.
Block and test information	Lists the blocks and the tests used within the fault file.	Output only. Editing is not recommended. Editing this section may invalidate tests.
FaultInfo	Describes the per-fault information displayed before a fault line.	Output only.
Status definition	Used to create user-defined statuses and define their behavior.	Input is only required if using a user defined status. Output includes all statuses if not user-defined statuses are given on the input side.
Coverage	Provides a way for the user to override the default equations or provide user defined equations that will appear in the summary section of a standard fault report.	Input is only required if modifying the fault coverage equations.
Failure Mode	Creates a definition to link FMEDA failure modes with observation signals and safety mechanisms.	Input. Use in conjunction with \$fs_observe and \$fs_detect system tasks.

Table 6 Standard Fault File Sections (Continued)

Section	Purpose	Customization allowed
Safety Mechanism	Defines list of detection signals for simulating failure modes for quantifying FMEDA results.	Input Use with \$fs_detect for fault detection.
Constraint	Constraints are used to define signals that remain constant in the simulation. These signals are treated as if tied to power or ground for fault generation and the fault connected to them are marked safe.	Input if input signals such as test logic are known to remain constant throughout the simulation.
StrobeData	Defines which pins are used in the reporting sections to show the state of pins when a fault is found.	Output only. Enabled using dictionary commands.
Strobe	Define list of signals for automatically strobing during simulation.	Input. Strobing behaves as if signal list passed to \$fs_strobe_onevent.
FaultGenerate	Encloses all fault definitions. This section is used to specify to fault campaign compiler how and where to generate faults.	Used for input only. You can write the <i>FaultGenerate</i> section and use it as the initial fault definitions file.
FaultList	Lists faults to be included in the fault universe. This section is used to read a list of fault locations and create faults from those descriptions. The <i>FaultList</i> section of a Coverage report can also be used as the input to fault generation.	Used for both input and output. Users can write the <i>FaultList</i> section and use it as the initial fault definitions file.

Header

The header section contains information that applies to the file as a whole:

- [Version](#)
- [Date](#)
- [User](#)
- [Tool Information](#)

Version

The Version statement identifies the version of the fault file standard to which the file conforms (Optional).

Syntax:

```
Version ("<version string>")
```

Where, <version string> is of the form 1234.5.

Example:

```
Version("1.0")
```

Date

The Date statement indicates the date that the file was created (Optional).

Syntax:

```
Date ("<date>")
```

Example:

```
Date("03/05/2105 12:45")
```

User

The User statement indicates the user name of the user who created the file (Optional).

Syntax:

```
User ("<user name>")
```

Example:

```
User("kjohnson")
```

Tool Information

The Tool statement indicates the tool used to create the fault file (Optional).

Syntax:

```
Tool ("<tool>")
```

Example:

```
Tool("Hand Generated File")
'test.b1.expr$f8$0 (i) "test.v", 58'
```

Test Information Section

This section lists the blocks and tests used within the fault file. This section is for output only. *Do not edit this section*. Edited fault list counts in this section are invalid, and test information may be lost. Tests are assigned IDs in the order they are run in Fault Campaign Manager.

```
TestList
{
<testID> <testName> {Fault Counts}
}
```

The *Fault Counts* section shows the counts of the various fault status values encountered for each test represented in the file. The first field, *Results* shows the total number of faults simulated by the test. The remaining fields display fault statuses from the results of the simulated test. These statuses can include either built-in VC Z01X statuses or User Defined Fault Statuses as described later in this chapter.

```
{Results:1323 OD:1208 NC:15 HA:5}
Example: TestList Section
```

```
TestList
{
  2 TEST_ABC 10ns {Results:100 OD: 98}
  1 TEST_DEF 22ns {Results:2 OD:2}
}
```

Fault Status Definition

The `StatusDefinitions` block allows manipulation of fault status definitions, providing the user with the flexibility to create new fault statuses, redefine existing fault statuses, and define new status groups. All VC Z01X tools that accept fault status values as input can accept valid user-defined statuses, and VC Z01X tools that output fault status values include user-defined statuses.

See the following subsections:

- [User-Defined Fault Status \(UDFS\)](#)
- [Redefining Statuses](#)
- [Defining the Default Fault Status for Simulation](#)
- [Selecting Faults Statuses for Simulation](#)
- [Defining Status Interactions](#)
- [Creating User-Defined Fault Status Groups](#)

User-Defined Fault Status (UDFS)

The `StatusDefinitions` block allows you to create user-defined fault statuses. UDFS are created to integrate the fault injection flow into any environment and match the terminology being used. UDFS definitions consist of a status code and a status description.

Example:

```
>StatusDefinitions
{
OD "Observed Diagnosed";
}
```

The status code is a two-character string composed of alphanumeric characters. If a user-defined status code contains illegal characters, an error message will be displayed and the definition will be ignored. Fault status codes are not case-sensitive; two valid fault status codes that differ only by case are considered equivalent. Fault status codes default to upper-case when output by the tools.

Fault status codes are accompanied by quoted description strings. Fault status descriptions can be up to 2501 characters in length. All user-defined fault status description strings must be distinct (case-insensitive). If two user-defined fault statuses are encountered with identical description string values, the definition of the first status encountered in the file will be accepted, and the latter will be ignored with a corresponding warning message.

When attempting to define a new fault status whose two-character fault status matches that of an existing status, the existing fault status must be redefined in order to use its two-character status value to define the new fault status. An error occurs if the existing fault status is not first redefined. See “Redefining Statuses”.

You must also define other items in the `StatusDefinitions` block as detailed in the following sections of this chapter, such as fault status interactions and coverage formulas.

Redefining Statuses

Any built-in VC Z01X status or status group (except for the reserved *Excluded statuses E0-E9*) may be redefined using the `Redefine` keyword within the `StatusDefinitions` block.

A `Redefine` operation is required when creating a new user-defined fault status whose two-character status code matches that of an existing fault status. A `Redefine` operation will be required on the existing status to free its two-character status code for use by the new user-defined fault status. An error will be printed if the existing fault status is not first redefined. A `Redefine` fault status is not considered to be a *user-defined* status.

Example:

```
StatusDefinitions
{
Redefine DD DX "Dangerous";
DD "Dangerous Diagnosed";
}
```

The *Redefine* operation takes as input a two-character status value corresponding to an existing fault status or status group, followed by an unused two-character status and an unused description string. The status codes and description strings follow the same input value requirements as described in [Creating User-Defined Fault Status Groups](#).

The existing fault status or status group corresponding to the first status value is modified to use the new status value and description string. If the existing status is not redefined, VC Z01X prints an error message and the fault creation step will fail. The underlying behavior of the original fault status remains the same, but the new status code and description is used in place of the original status. This new status value is not required to be included in `PromotionTable` because previous interactions involving the status are retained and mapped to the new status value.

A *Redefine* operation can be used to update the status definition string without redefining the status. To do this, use the same two-character status code as both the old and new status code in the *Redefine* operation. For example, to update the built-in ND status definition string from *Not Detected* to *New Status Definition*, specify the following:

```
Redefine ND ND "New Status Definition";
```

For more examples on specifying the *Redefine* keyword, see the examples in the upcoming sections in this chapter.

When `vc_fcc` reads the *Redefine* operation inputs, the output is also preserved in the Standard Fault Format report.

Input.sff	Output.sff
StatusDefinitions{ Redefine OA RR "Danger";}	StatusDefinitions{Redefine OA RR "Danger";}

Defining the Default Fault Status for Simulation

The `DefaultStatus` operation is used to define the status of a fault when it has been simulated, but the final fault status is not set through a strobe system task. The `DefaultStatus` operation is required when using User-Defined Fault Status (UDFS).

Example:

```
DefaultStatus (NN)
```

`DefaultStatus` requires a valid fault status code. The status code may be either a VC Z01X built-in status or a UDFS. If an input status code is invalid, an error will be generated, and importing will be unable to proceed. If the number of inputs is not equal to one, an error is generated, and importing will be unable to proceed. A `DefaultStatus` directive will be ignored if there are no user-defined statuses present.

Synopsys recommends setting `DefaultStatus` to either the built-in ND (Not Detected) status, or to the status that has replaced ND through the Redefine operation. Using other statuses in this field may lead to unpredictable coverage results.

Selecting Faults Statuses for Simulation

The `Selected` operation is used to define which fault statuses will be considered for fault simulation. Certain statuses, such as the built-in Dropped Detected (DD) status are typically not simulated when fault simulation starts. Synopsys recommends that once a fault is detected, or diagnosed, by the safety mechanism, it does not need to be simulated in subsequent workloads.

Statuses that are listed as `Selected` are submitted for testability and fault simulation. At the end of the simulation, the resulting fault status will be merged with the original fault status according to the `PromotionTable`. The default statuses for `Selected` are NA, ND, PT, and DT.

Example:

```
Selected (NA, NN, NP, PN, OP, ON)
```

The `Selected` operation accepts as input any valid fault status codes or any defined status groups in a comma separated list. If a status group is input, all the two-character fault statuses that belong to the status group will be selected. If a `Selected` operation is not specified within the `StatusDefinitions` block, the default status selections of NA, ND, PT, and DT will be used. An individual tool may be set to override the selections made within the `Selected` directive. If an input status value is invalid, a warning is generated and the `Selected` directive is not processed.

Defining Status Interactions

User-defined fault status promotion is defined in the `PromotionTable{}` sub-block of the fault status definition or using the alternative `Promote/PromoteOrder` methodology. UDFS interactions are used when merging faults lists before or after a simulation. They do not define interactions during a fault simulation.

Any interactions that are not included in the `PromotionTable` but are required when merging faults will cause fault status importing or merging to give an error and exit the program.

Unless defined in the `PromotionTable` explicitly, the following built-in VC Z01X fault statuses have precedence over UDFS when merging. In all other cases, UDFS has precedence over built-in VC Z01X fault statuses when merging.

- Oscillating Group - OZ
- Illegal Group - IX, IA, IF
- Hyper Group - HA, HT
- Untestable Group - UU, UT, UB, UR, UI, UO
- Detected Group - DF
- Exclusion Status EN (E0-E9)

When defining a user-defined status, it is recommended to define the interaction of the status with the built-in VC Z01X statuses listed above to ensure expected user-defined status promotion behavior.

Example:

```
StatusDefinitions
{
S1 "Status Definition 1";
S2 "Status Definition 2";
S3 "Status Definition 3";
S4 "Status Definition 4";
PromotionTable
{
StatusLabels (S1,S2,S3,S4)
#-----OLD-----
# S1 S2 S3 S4 |
[ S1 S2 S3 S4 ; # S1 |
S2 S2 S3 S4 ; # S2 | NEW
S3 S3 S3 S4 ; # S3 |
S4 S4 S4 S4 ; # S4 |
]
}
```

`PromotionTable` consists of these required fields to fully define fault interactions.

- `StatusLabels (<S1>, <S2>, <S3>, <S4>, ...)` - Includes all statuses used in the promotion table. If a fault status code is used more than once within a `StatusLabels` directive, an error is produced, and `vc_fcc` exits. The `StatusLabels` define the number of columns and rows that will be used in the following `[Table]`. If there are 4 `StatusLabels`, the `[Table]` must contain 4 columns and 4 rows.
- `[Table]` - For each given New and Old status combination, enter the value to which the status should be promoted. For legibility, these are generally formatted to appear

as a table with commented labels. Unrecognized characters in a `PromotionTable` row trigger a syntax error.

The `[Table]` definition provides shorthand code that simplify definition and readability of the `PromotionTable`.

- |: Indicates that the promoted status takes the value of the OLD column label.
- - : Indicates that the promoted status takes the value of the New row label.

The input `PromotionTable` may be created with either the full status codes listed, the shorthand code, or a mix of both. When the coverage file is created the `PromotionTable` will be expanded to list the fault status codes as it appears in the first example below.

- `PromoteOrder` is an alternative to defining status promotions through the `PromotionTable`. `PromoteOrder` uses the '`<`' character to define a hierarchy of statuses within a single statement. The resulting statuses will be expanded by the fault campaign compiler to create a symmetrical `PromotionTable` that will be displayed in the output fault coverage file. `PromoteOrder` may also be used during definition of `StatusGroups` as shown below.

```
PromoteOrder(S1 < S2 < S3 ...)
```

- `Promote` is used to override or define individual entries in the `PromotionTable` generated with `PromoteOrder`. `PromoteOrder` and `Promote` may not be mixed with `StatusLabels` and `[Table]` in an SFF file.

```
Promote(S1 + S2 => S3)
```

Example:

A basic example of `PromotionTable` lists all UDFS and the merge results for all.

```
StatusDefinitions
{
D1 "Detect 1";
D2 "Detect 2";
D3 "Detect 3";
D4 "Detect 4";
PromotionTable
{
StatusLabels (D1,D2,D3,D4)
#-----Old-----
# D1 D2 D3 D4 |
[ D1 D2 D3 D4 ; # D1 |
D2 D2 D3 D4 ; # D2 | NEW
D3 D3 D3 D4 ; # D3 |
D4 D4 D4 D4 ; # D4 |
}]
```

```
}
```

```
}
```

Example:

The following example shows use of the promotion shorthand characters. This PromotionTable produces the same result as the example above.

```
StatusDefinitions
{
D1 "Detect 1";
D2 "Detect 2";
D3 "Detect 3";
D4 "Detect 4";
PromotionTable
{
StatusLabels ( D1,D2,D3,D4 )
#-----Old-----
# D1 D2 D3 D4 |
[ - | | | ; # D1 |
- - | | ; # D2 | NEW
- - - | ; # D3 |
- - - - ; # D4 |
]
}
}
```

Example:

A basic example of implementation a PromoteOrder lists all UDFS and the merge results for all. This example produces identical output to the PromotionTable example above.

```
StatusDefinitions
{
D1 "Detect 1";
D2 "Detect 2";
D3 "Detect 3";
D4 "Detect 4";
PromotionTable
{
PromoteOrder(D1<D2<D3<D4)
}
}

Resulting Promotion Table:
PromotionTable
{
StatusLabels (D1,D2,D3,D4)
#-----Old-----
# D1 D2 D3 D4 |
[ D1 D2 D3 D4 ; # D1 |
D2 D2 D3 D4 ; # D2 | NEW
D3 D3 D3 D4 ; # D3 |
D4 D4 D4 D4 ; # D4 |
```

```
]
}
```

Example:

The same `PromotionTable` can be created with an entry to override the symmetrical nature of the table.

```
StatusDefinitions
{
D1 "Detect 1";
D2 "Detect 2";
D3 "Detect 3";
D4 "Detect 4";
PromotionTable
{
PromoteOrder(D1<D2<D3<D4)
Promote(D1 + D3 => D4)
}
}
Resulting Promotion Table:
PromotionTable
{
StatusLabels (D1,D2,D3,D4)
#-----Old-----
# D1 D2 D3 D4 |
[ D1 D2 D4 D4 ; # D1 |
D2 D2 D3 D4 ; # D2 | NEW
D4 D3 D3 D4 ; # D3 |
D4 D4 D4 D4 ; # D4 |
]
```

Creating User-Defined Fault Status Groups

`StatusGroups` allow you to place multiple fault statuses into a container. Grouping fault statuses can make it easier to select these groups during simulation and reporting. All VC Z01X tools that accept a status as input recognize defined status groups. `StatusGroups` can be used with the default `PromotionTable` entries or in conjunction with `PromoteOrder` and `Promote`.

`StatusGroups` have several characteristics to be considered when being defined.

- `StatusGroups` must use unique two-character identifiers. The identifier must be unique among all defined fault statuses and other status groups.
- `StatusGroups` may contain any built-in or UDFS existing fault statuses.
- `StatusGroups` cannot be contained by other `StatusGroups`.

- A fault status must appear in one and only one *StatusGroups*. This includes both UDFS and built-in statuses. If a status is added to a multiple *StatusGroups*, the status is removed from the first group and added to the new group. The group defined last will be the only group that contains the status which has been re-used.
- When using *StatusGroups* with *PromoteOrder* and *Promote*, you must use the second syntax shown below that defines both the statuses included in the group and the promotion order of those statuses.

To define a new status group, the following syntax is used within the *StatusDefinitions* block of a standard fault format file:

```
	StatusGroups {
	<group_id_1> "<group_name_1>" (<status_1>, <status_2>, ...);
	<group_id_2> "<group_name_2>" (<status_3>, <status_4>, ...);
}
```

To define a new status group using *PromoteOrder* and *Promote*, the following syntax is used within the *StatusDefinitions* block of a standard fault format file:

```
	StatusGroups
{
<group_id_1> "<group_name_1>" (<status_1>, '<' <status_2>, ...);
<group_id_2> "<group_name_2>" (<status_3>, '<' <status_4>, ...);
}
```

Where:

- *group_id*: The unique two-character abbreviation used to identify the status group.
- *group_name*: The full name of the group
- *status*: A list of the two-character identifier of fault statuses contained by the group.

Example

This example shows how to create a user-specified group named “Dangerous” with a group identifier "DN" that contains the following built-in statuses: HA, OZ, and IF. In addition, this group has a user-defined status U1. The *StatusDefinitions* block is as follows:

```
	StatusDefinitions
{
    U1 "User defined fault 1";
    DefaultStatus(ND)
    StatusGroups
    {
        DN "Dangerous" (HA, OZ, IF, U1);
    }
}
```

This second example shows how to create the same user-specified group named “Dangerous” as in the first example, but using the alternative `PromoteOrder/Promote` syntax:

```
StatusDefinitions
{
U1 "User defined fault 1";
DefaultStatus(DD)
StatusGroups
{
DN "Dangerous" (HA < OZ < IF < U1);
}
}
```

Using Fault Status Groups

The following Fault Campaign Manager tool commands support predefined and user-defined fault status groups:

- `exclude_faults`
- `fsim`
- `remove_fault_results`
- `report`
- `show_fault_results`
- `show_status_groups`
- `show_statuses`
- `write_fault_results`

You can use `coverage` to generate coverage reports, which includes fault status group information. For more information, see [Appendix A: Fault Campaign Manager Commands](#).

Coverage Section

The *Coverage* section provides a way for the user to override the default equations or create user-defined equations. The coverage calculations will appear in the Summary section of a Standard Fault report. The default Fault and Test coverage equations can be found in *Test Coverage and Fault Coverage Calculations* section .

User-Defined Equation Syntax

The coverage block enables you to create coverage equations that you can use to calculate results that will be displayed in the Summary section of the coverage report.

When UDFS is used, coverage equations must be defined. If the coverage equations are not defined, no coverage result will appear in the final Summary section of the coverage report.

If the equation name is *Test Coverage* or *Fault Coverage*, then the new equation will override the default VC Z01X equations for these values.

A user-defined equation is of the form:

```
<Equation Name> = "<format>(<equation>)" ;
```

Where:

- **Equation Name:** Is the text string you want to show up in the summary report. You cannot have more than one equation of a given name. The name is case insensitive. If a space, semi-colon, or equal sign appears in the equation, enclose the name in quotes.
- **Format:** Is an optional value specifying the output format. Supported options are PCT, INT, or FLT. If no format is given, PCT is the default format. If one of these keywords is added, the rest of the equation should be in enclosed in parenthesis.
 - PCT - Formats the output with a trailing percent sign
 - INT - Formats the output as an integer
 - FLT - Formats the output as a floating point number
- **Equation:** Is the equation you want calculated for the summary. Valid values are:
 - Numbers: any non-exponential number. (that is 2, 5.0, 3.5, -3)
 - Operators: +, -, *, /, ^
 - Parenthesis: Standard precedence is supported

Example

```
Coverage
{
    MyCoverage = "INT(DD / (DD + PD) / 2)";
    "Fault Coverage" = "DD / (DD + PD + ND) / 2 ";
}
```

Strobe Section

Strobe points can be defined in the SFF file using a Strobe block. Signals in a Strobe block are considered to behave as `$fs_strobe_onevent()` (system task) signals with respect to both testability and fault simulation, but do not require you to insert the system task into your design.

Syntax:

```
Strobe
{
<List of Locations>
}
Example:
Strobe
{
"test.x"
"test.v"
}
```

Strobe signals are treated as though provided as arguments to `$fs_strobe_onevent()`. Like `$fs_strobe_onevent()`, the SFF Strobe blocks compare the values of the good machine (GM) to the faulty machine (FM) for the listed strobe points whenever the signal changes in either of the machines.

`<List of Locations>` is a whitespace separate list of signals. Each signal in the list must be a quoted string of a hierarchical path to a signal in the design.

The SFF Strobe block is available in all SFF flows. It can also be combined with other strobe-related `$fs_*tasks/functions` (including `$fs_set_status_onevent` and `$fs_drop_status_onevent`).

If there are no strobes defined in the design and in the SFF file, the Fault Campaign Compiler (FCC) attempts to automatically includes strobes on the inout/output ports of all top level instances that are not cells unless the design is compiled with `-debug_region +cell`.

If the design is compiled with the `-debug_region=cell+lib` switch, top level cells are considered for automatic strobe placement.

If automatic strobes are created, the FCC will display the following note with information on which instances the strobes were placed and continues.

```
Note-[FCC-AUTO-STROBES-ON-TOPLEVEL-OUTPUT-PORTS] Strobes on top level
output ports
There are no strobes defined in the design and in the SFF file.
Strobes are automatically added on the output ports of all the top level
instances.
The outputs of the following top level instance(s) are added as strobes:
top,
test2
```

In the case even the automatic placement fails to produce any strobes, FCC will display the warning below, disable pruning and continue.

```
Warning-[FCC-NO-STROBES] No strobes available
There are no strobes defined in the design and in the SFF file.
```

No strobes are automatically added as there are no output ports on the top level instances.

Please review the intent. Continuing without strobes, static testability and dynamic observability will be disabled.

Also during dynamic testability analysis, in case no strobing points are defined in the fault campaign, the NO *Not Observed Analysis* will be disabled.

This behavior can potentially produce different results from Coats compared to previous versions especially resulting in no faults with NO status and more faults with NC status.

SafetyMechanism Section

Safety mechanisms are used to create a list of detection signals for simulating failure modes to quantify FMEDA results. The `FailureMode` definition below allows the user to specify the detection points of the safety mechanism individually or the user can create individual `SafetyMechanism` blocks that can be re-used in the `FailureMode` section as an alternative to listing the detection points. The `SafetyMechanism` construct works in conjunction with `$fs_detect(system function)` and with providing a failure mode name to the [FaultGenerate Section](#) or [FaultList Section](#) sections.

```
SafetyMechanism <Name>
{
  Detect
  {
    <List of locations>
  }
}
```

The `<Name>` parameter is a user defined name to be referenced in the `FailureMode` Section defined later in this document.

The `Detect` subsection of the `SafetyMechanism` block must contain at least one signal.

The `List of Locations` in the `Detect` subsection is a white-space separated list with each item as one of the following:

- `<path>` – path to data object (wire, port, variable etc.), wildcard is allowed.
- `<instance_path>` - path to an instance, no wildcard is allowed. In this case, all output and inout ports of the instance are used.
- `<file>`- read list of data objects from the specified file. The `<` and `>` symbols are required when reading the data object from a file.

Example:

```
// Software test library safety mechanism
SafetyMechanism sm_stl
{
```

```

Detect
{
"top.dut.cpu.alarm"
}
}
// CPU lock step safety mechanism
SafetyMechanism sm_lockstep
{
Detect
{
"top.dut.lockstep.mismatch"
}
}

```

FailureMode Section

FailureMode is used in SFF to associate fault observation and detection information with an individual failure mode for purposes of FMEDA calculations. To enable this functionality the SFF file supports the following syntax to enable you to quantify failure mode detection results. These constructs work in conjunction with \$fs_observe (system function) and \$fs_detect (system function) and with providing a failure mode name to the [FaultGenerate Section](#) or [FaultList Section](#) sections.

```

FailureMode <Name>
{
Observe
{
<List of Locations>
}
Detect
{
<List of Locations>
}
}

```

Alternate Example:

```

FailureMode <Name>
{
Observe
{
<List of Locations>
}
SafetyMechanism(<Identifier1>, <Identifier2>, ..., <IdentifierN>)
}

```

Some requirements and consideration when using `FailureMode` are:

- The `<Name>` is an optional parameter. If `<Name>` is not specified, an unnamed failure mode will be created.
- `Observe` is required as part of the `FailureMode` definition to define the observation points for the failure mode.
- `Detect` or `SafetyMechanism` are required as part of the `FailureMode` definition to list the diagnostic points. You may specify one of either `Detect` and `SafetyMechanism` sub-sections, but not both. `SafetyMechanism` is useful because it allows you to reuse lists of diagnostic points with multiple `FailureMode` without listing the signals each usage.
- The signal list in `Observe` will be automatically used as the signal list for `$fs.observe` during fault simulation.
- The signal list in `Detect` or `SafetyMechanism` will be used as the signal list for `$fs_detect` during fault simulation.
- If the `SafetyMechanism` entry is present in the `FailureMode` section, it must first be specified in a separate `SafetyMechanism` top-level section.

The `<List of Locations>` in `Observe` and `Detect` is a white-space separated list with each item one of the following:

- `<path>` - path to data object (wire, port, variable etc.), wildcard is allowed.
- `<instance_path>` - path to an instance, no wildcard is allowed. In this case, all output and inout ports of the instance are used.
- `<file>`- read list of data objects from the specified file. The `<and>` symbols are required when reading the data object from a file.

Example:

```
Observe
{
  "top.dut.ram.data" // Full path to the signal
  "top.foo.bus" // All output & inout ports for instance 'bus'.
  "top.cpu_inst*.*" // All data objects directly under instances with
  //name starting with "cpu_inst".
  "cpu.**" // All data objects under the given
  // module (including all sub-instances)
  <signals.txt> // All data objects listed in the file 'signals.txt'.
}
```

Both `Observe` and `Detect` can optionally include `Exclude` blocks, which allows you to filter out previously specified signals, example:

```
Observe
{
  "top.dut.cpu" // All output and inout ports for instance "cpu"
  Exclude
  {
    "top.dut.cpu.debug*" // Except all debug ports.
  }
}
```

Similar to `Exclude` block behavior in the [FaultGenerate Section](#), `Exclude` will apply to all entries above it until the start of the section or another `Exclude`. `Exclude` can contain only data object paths (wildcard is allowed). Instances and included files are not allowed in `Exclude` blocks.

FailureMode Example:

```
# Software test library safety mechanism.
SafetyMechanism sm_stl
{
  Detect
  {
    "top.dut.cpu.alarm"
  }
}
# CPU lock step safety mechanism
SafetyMechanism sm_lockstep
{
  Detect
  {
    "top.dut.lockstep.mismatch"
  }
}
# Failure mode: some of CPU registers has a wrong value.
FailureMode fm_wrong_register_value
{
  Observe
  {
    "top.dut.cpu.registers.reg*"
    Exclude
    {
      "top.dut.cpu.registers.reg*_shadow" }
  }
  SafetyMechanisms(sm_stl, sm_lockstep)
}
FaultGenerate fm_wrong_register_value
{
  NA [0,1] { PORT "top.dut.cpu.**" }
```

Constraint Section

Constraints in the SFF file are used for the purpose of marking faults as UT, UB, UR, or a user-defined status. Constraints do not affect the behavior of the simulation and the validity of the constraints is dependent on the expert analysis of the user. Constraint marking is considered during fault generation and the signals that are constrained are generated as if they were tied to power or ground.

You may set only a single status for the constraints in the SFF file. The two constraint statuses are:

- **U*-** Fault may be marked UT, UB or UR after considering constraints.
- **User Defined-** The fault will be marked with the user defined status.

VC Z01X allows constraints to be defined on scalar or single-bit object in the design. To constrain a vector signal, you must constrain each bit individually by listing it in the constraint block.

Example:

The first example show how to constrain a signal and to mark all faults associated with that signal as the UDFS ED. This is useful for separating the fault results for constrained faults from the normal structural analysis for untestable faults.

```
>StatusDefinitions
{
    ED "Excluded Definition";
}
Constraint one
{
    ED "system.processor.mod1.mod2.i==0";
}
```

Example:

This example used the built-in U* fault statuses for the constrained signals. This will result in additional faults being marked UT, UB, or UR as the constrain is applied during fault generation.

```
Constraint safe
{
    U* "system.dut.te==0";
    U* "system.dut.mode1==1";
}
```

Limitation

VC Z01X does not support multi-bit constraints in an SFF file as the constraint value can only be 0 or 1.

FaultGenerate Section

The `FaultGenerate` section allows you to generate faults according to several input parameters. This section is present at the top-level of a Standard Fault Format file. When VC Z01X creates faults through the `FaultGenerate` section, it will use the algorithms for fault collapsing and untestable marking as described in [Fault Collapsing](#) and [Untestable Group](#). You can specify multiple `FaultGenerate` sections in the Standard Fault Format file. Statements are executed in the same order as encountered within a `FaultGenerate` section.

For more information on the `FaultGenerate` section, see the following subsections:

- [Generating Faults Using FaultGenerate](#)
- [Generating Multiple Fault List](#)
- [Wildcard Support](#)
- [Exclude Block](#)
- [Sampling](#)

Generating Faults Using FaultGenerate

The `FaultGenerate` block is created according to the syntax listed below. You can specify multiple `FaultGenerate` sections in the Standard Fault Format file. See [FaultGenerate Example](#).

Syntax:

```
FaultGenerate [<Name>]
{ (<Fault definition statement>)}
```

Where:

- `<Name>`: Optional parameter used to associate the `FaultGenerate` block with a previously defined `FailureMode`. For more information on creating named fault lists, see [Generating Multiple Fault List](#) section.
- `<Fault definition statement>`: For a complete list of supported fault descriptors, see [Fault Descriptors](#). In addition to the standard fault descriptors, there are some extensions supported in the `FaultGenerate` block as detailed below.

Statements are executed in the same order as encountered within a `FaultGenerate` section. Faults can be generated through fault descriptor statements within a `FaultGenerate` block as follows:

Syntax:

```
<status> <fault-type-list> '{' <fault-classlist> [<loc-filter-list> ]<location> '}'
```

Where:

- `<status>`: Specifies the status assigned to the fault when it is created. The `<status>` must be an existing VC Z01X status, or one created in the `FaultStatus` block or a warning message is issued and the `<Fault definition statement>` is ignored.
- `<fault-type-list>`: Specifies a list of VC Z01X supported fault types. See [Fault Descriptors](#) for a complete list of supported fault types. You can generate multiple fault types at once using this option as follows:

```
<fault-type-list> := '[' <fault-type> ( ',' <fault-type> )']'
```

- `<fault-class-list>`: Specifies a list of VC Z01X supported fault classes. See [Fault Descriptors](#) for a complete list of supported fault classes. You can generate multiple fault classes at once using this option as follows:

```
<fault-class-list> := '[' <fault-class> ( ',' <fault-class> )']'
```

Note:

It is recommended to specify one or more fault classes for transient faults to direct VC Z01X to generate faults on appropriate locations. For more information, see the special notes on transient fault class specifiers within [Fault Descriptors](#).

- `<loc-filter-list>`: Specifies properties which must be satisfied in order to place a fault on a location. Location filters are only relevant to PORT and PRIM fault classes. For all other fault classes, `<loc-filter-list>` option is ignored and an Error message is issued. If `<loc-filter-list>` is not specified, all the properties are considered.

Syntax:

```
<location-filter-list> := '[' <location-filter> ( ',',<location-filter> )* ']'
```

- `<loc-filter>`: Specifies one of the following keywords:
 - INPUT - Location is an input
 - OUTPUT - Location is an output

- INOUT - Location is an inout
- REG - Location is a reg
- LOGIC - Location is a logic
- BIT - Location is a bit
- BYTE - Location is a byte
- INTEGER - Location is an integer
- TIME - Location is a time
- <location>: Specifies the design locations where fault generation must occur.

Syntax:

<location> : = <instance-path> | <design-entity>

Where:

- <instance-path>: Defines a path to an instance in which faults are to be generated.
- <design-entity>: Defines the name of an entity within the design (that is, a module name and so on) in which faults are to be generated.

Both <instance-path> and <design-entity> arguments may include a number of wildcard variants.

Faults should be specified within a `FaultGenerate` block with a fault status of NA. `vc_fcc` reports the following warning message when a fault is specified in this block which is of non-NA status and resets the fault to NA status. This warning message is displayed only once.

Warning! One or more faults in the provided `FaultGenerate` blocks were specified with a starting status other than 'NA'. These faults will be reset to 'NA' status. Faults which you wish to exclude from simulation should be specified in an `Exclude` block. To import faults with a status other than 'NA', they may be specified in a `FaultList` block.

Faults may still be specified within an `Exclude` block with any status to be marked for exclusion.

For more information on using `Exclude` blocks, see the [Exclude Block](#) section. For more information on using the `FaultList` block, see the [FaultList Section](#).

Note:

Since the `\suppress_faults` and `\enable_portfaults` compiler directives are honored when generating faults through the `vc_fcc` tool from a Standard

Fault Format (SFF) file for wildcarded locations, faults are not generated within suppressed areas of the design as denoted by the `suppress_faults directive or through the `-fsim=portfaults` compile option. To generate faults in suppressed areas of the design, use the `-ignore_suppress vc_fcc` option. Similarly, faults are not generated on ports not enclosed by ``enable_portfaults`. To generate faults on ports not enclosed within ``enable_portfaults/`disable_portfaults`, use the `-fsim=portfaults` compile option.

FaultGenerate Example

The status of created faults is assigned based on the status specified in the `FaultGenerate` block. For example, if the `FaultGenerate` section is defined as follows, all faults are created with status NA.

```
FaultGenerate
{
    # Create faults on all reg types in hierarchy
    NA [0,1] { VARI "test.risc1.**" }
    # Create faults on all ports in hierarchy
    NA [0,1] { PORT "test.risc1.**" }
}
```

Generating Multiple Fault List

VC Z01X provides the ability to generate and report on named blocks of faults. During fault simulation all faults generated in the SFF file are simulated independent of the named fault list where they are defined.

This section targets the ability to specify one or more named fault lists within a Standard Fault Format (SFF) file to group one or more sets of faults. You can then generate fault reports on each named list of faults and provide coverage details as per the list. Reporting allows specific lists to be chosen for reporting through FCM report option `-failure_mode`.

For generating and reporting multiple fault lists, the key steps are as follows:

1. Specify the fault list names to the `FaultGenerate` and `FaultList` blocks in the SFF file. For more information, see [SFF Syntax and Fault Generation Using vc_fcc](#).
2. Run the `report FCM` command.
3. View reports. For more information on reports, see [Working with Results: Multiple Fault List](#).

To see the limitations associated with this feature, see the [Limitations: Multiple Fault Lists](#) section.

SFF Syntax and Fault Generation Using vc_fcc

To name FaultGenerate and FaultList blocks through SFF, add a string following the FaultGenerate and FaultList keywords in the SFF file. To name a fault list (<fault_list_name>), you can include any alphanumeric characters and underscores (a-z, A-Z, 0-9_). The names must not include whitespaces. VC Z01X reports an error message if the naming convention is not followed.

Example Message

```
Error! Line 1: syntax error at text: FL_name_1
```

In the following SFF, the named identifier is <fault_list_name>, which is an unquoted fault list name used to reference the block of faults. The fault list names are FL_name_1 and FL_name_2. Faults described within these named blocks are associated with the name specified. For faults described within multiple named blocks, the fault is associated with every name for the block in which it is included.

Syntax	Example
FaultGenerate <fault_list_name> {...} orFaultList <fault_list_name>{...}	FaultGenerate FL_name_1 { NA 0 {PRIM "tb.t1.**"} } orFaultList FL_name_2{ NA 1 {PRIM "tb.t2.s1.a1.0"} NA 0 {PRIM "tb.t2.s1.a1.0"} }

Example: SFF Specification With Multiple Named Blocks and Associations

This example shows an SFF specification.

```
FaultGenerate FL-name-1
{
NA 0 {PRIM "tb.dut.s1.**"}
NA 0 {PRIM "tb.dut.s2.**"}
}
FaultGenerate FL-name-2
{
NA 0 {PRIM "tb.dut.s2.**"}
NA 0 {PRIM "tb.dut.s3.**"}
}
```

The following associations are created:

- Faults within the tb.dut.s1 hierarchy are associated with FL-name-1 only.
- Faults within the tb.dut.s2 hierarchy are associated with both FL-name-1 and FL-name-2 lists.
- Faults within the tb.dut.s3 hierarchy are associated with FL-name-2 only.

For all named blocks, reference them in reporting by the names specified within the SFF file.

Wildcard Support

The following wildcard characters are supported for regular expressions:

- ? - Indicates a single character
- * - Indicates zero or more characters. This substitution does not proceed beyond the name segment containing the wildcard character (for example, 'test.*' is equivalent to 'test.wire_a', but not 'test.inst1.wire_b').

Note:

When utilized on its own as the trailing segment of a fault path, the fault generator will generate faults on all applicable elements within instance(s) resolved to via the preceding instance path prior to the trailing ".**". In other words, a standalone "*" character will not attempt to match instances unless the wildcard occurs in the middle of a wildcard path (that is, "test.dut.*.prim1.0").

- ** - Indicates that all locations within the instance(s) and all locations within all child instances must be considered for fault generation. This method must follow a '.' character and must be present as the last two characters in the path specified.

When using wildcard characters with escaped identifiers they must be preceded with a '\' character.

- \? - Indicates a single character (exactly 1 character)
- * - Indicates zero or more characters
- \\ - Indicates a search for the actual '\' character

Wildcarding is not allowed within square bracket characters used to specify indices for arrayed instances, bits, or part-selects.

When generating faults for a location with wildcards, VC Z01X attempts to find all design elements with hierarchical names that match the location. If the location matches a module instance and is not trailing single-star scenario (that is, not in the form of <instance_path>.*), a fault is generated on every valid element in that instance. In this case, child module instances of the instance are ignored. When a trailing single-star wildcard exists (that is, not in the form of <instance_path>.*), then a fault is generated on every valid element in the instance(s) resolved to via the preceding instance path specified prior to the trailing ".**". When the top level of a wildcarded location matches a Module definition, VC Z01X substitutes any instance of that module.

A warning message is issued if no matches are found for the specified wildcard pattern.

Note:

Since the substitution of complex wildcard patterns can be performance intensive, use wildcard characters sparingly.

All Standard Fault Format descriptor forms support expansion through the `FaultGenerate` section.

For example, sequential transient toggle faults can be generated as follows:

```
# Generate all transient toggle FLOP/PORT multi-point faults for all
# permutations of
cycles 3-4 as well as instances test1 and test2.
FaultGenerate
{
NA ~ (3:4) { FLOP "test1.*" } + 1 { PORT "test2.*" }
```

Non-Escaped Name Examples

The examples in this section are based on the following design:

```
test.v
-----
module test(a, e);
  output a, e;
  reg b, c, d;
  wire e;
  not (e, a);
  dut dut (a, b, c, d);
endmodule
`enable_portfaults
module dut(a, b, c, d);
  output a;
  input b, c, d;
  AND one (x, b, c);
  AND two (a, x, d);
endmodule
module AND(a, b, c);
  output a;
  input b, c;
  and a1 (a, b, c);
endmodule
```

Example 1

This example shows the `FaultGenerate` block specification to generate all stuck-at 1, PORT faults on instances matching the `test.dut?.*` pattern.

```
input.sff
-----
FaultGenerate
{
```

```

        NA 1 { PORT "test.du?.*" }
    }
output.sff
-----
FaultList
{
    NA 1 {PORT "test.dut.a"}
    NA 1 {PORT "test.dut.b"}
    NA 1 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.d"}
}
    
```

Example 2

This example shows the `FaultGenerate` block specification to generate all stuck-at 0 and stuck-at 1, PORT and PRIM faults within the hierarchy tree of the instance test. This includes the instance test and all lower levels of the hierarchy tree.

```

input.sff
-----
FaultGenerate
{
    NA [0, 1] { PORT "test.**" }
}
output.sff
-----
FaultList
{
    NA 0 {PORT "test.dut.a"}
    -- 0 {PORT "test.dut.two.a"}
    -- 0 {PORT "test.dut.two.b"}
    -- 0 {PORT "test.dut.two.c"}
    -- 0 {PORT "test.dut.d"}
    -- 0 {PORT "test.dut.one.a"}
    -- 0 {PORT "test.dut.one.b"}
    -- 0 {PORT "test.dut.one.c"}
    -- 0 {PORT "test.dut.b"}
    -- 0 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.a"}
    -- 1 {PORT "test.dut.two.a"}
    NA 1 {PORT "test.dut.two.b"}
    -- 1 {PORT "test.dut.one.a"}
    NA 1 {PORT "test.dut.two.c"}
    -- 1 {PORT "test.dut.d"}
    NA 1 {PORT "test.dut.one.b"}
    -- 1 {PORT "test.dut.b"}
    NA 1 {PORT "test.dut.one.c"}
    -- 1 {PORT "test.dut.c"}
}
    
```

Example 3

This example shows the `FaultGenerate` block specification to generate all stuck-at 0 and stuck-at 1, PORT faults within the instance `test.dut`. This includes the instance `test.dut`, but does not include lower levels of the hierarchy tree.

```
input.sff
-----
FaultGenerate
{
    NA [0, 1] { PORT "test.dut.*" }
}
output.sff
-----
FaultList
{
    NA 0 {PORT "test.dut.a"}
    -- 0 {PORT "test.dut.d"}
    -- 0 {PORT "test.dut.b"}
    -- 0 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.a"}
    NA 1 {PORT "test.dut.b"}
    NA 1 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.d"}
}
```

Example 4

This example shows the `FaultGenerate` block specification to generate all stuck-at 0 and stuck-at 1, PORT faults on inputs in the hierarchy of the instance `test.dut`.

```
input.sff
-----
FaultGenerate
{
    NA [0, 1] { PORT [ INPUT ] "test.dut1.*" }
}
output.sff
-----
FaultList
{
    NA 0 {PORT "test.dut.d"}
    -- 0 {PORT "test.dut.b"}
    -- 0 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.b"}
    NA 1 {PORT "test.dut.c"}
    NA 1 {PORT "test.dut.d"}
}
```

Example 5

This example shows the `FaultGenerate` block specification to generate a stuck-at 1 port fault on all ports named `a`.

```
input.sff
-----
FaultGenerate
{
NA 1 { PORT "*.a"}
}
output.sff
-----
FaultList
{
    NA 1 {PORT "test.dut.a"}
    -- 1 {PORT "test.dut.two.a"}
    NA 1 {PORT "test.dut.one.a"}
}
```

Example 6

This example shows a the `FaultGenerate` block specification for generating stuck-at 1 faults on primitives

in the generate block `test.dut.gen[1]`. This example uses a different design than previous examples.

```
test.v
-----
module test(a);
output [7:0] a;
reg [7:0] b;
DUT dut(a, b);
endmodule
module DUT(out, in);
output [7:0] out;
input [7:0] in;
genvar i;
generate
    for(i = 0; i < 4; i = i + 1)
        begin: gen
            and a1(out[i], in[i], in[i+4]);
            or o1(out[i+4], in[i], in[i+4]);
        end: gen
    endgenerate
endmodule

input.sff
-----
FaultGenerate
{
NA 1 {PRIM "test.dut.gen[1].**"}
}
output.sff
-----
FaultList
{
```

```

NA 1 {PRIM "test.dut.gen[1].o1.0"}
-- 1 {PRIM "test.dut.gen[1].o1.1"}
-- 1 {PRIM "test.dut.gen[1].o1.2"}
NA 1 {PRIM "test.dut.gen[1].a1.1"}
NA 1 {PRIM "test.dut.gen[1].a1.2"}
NA 1 {PRIM "test.dut.gen[1].a1.0"}
}

```

Escaped Name Examples

The examples in this section are based on the following design:

```

test.v:
-----
`enable_portfaults
module \dut_esc/esc1/esc2 (output Z);
reg Z;
reg A[0:3];
reg B[0:3];
endmodule
module \dut_esc/esc1* (output Z);
reg Z;
reg A[0:3];
reg B[0:3];
endmodule
module \dut_esc/esc1\* (output Z);
reg Z;
reg A[0:3];
reg B[0:3];
endmodule

```

Example 1

If faults are desired on the "\dut_esc/esc1<wildcard *>"location, use the following specification to make sure that the wildcard character is considered: "\dut_esc/esc1* "

```

input.sff:
-----
FaultGenerate
{
NA [0,1] { PORT "\dut_esc/esc1\* " }
}
output.sff
-----
FaultList
{
NA 0 {PORT "\dut_esc/esc1/esc2 .Z"}
NA 0 {PORT "\dut_esc/esc1* .Z"}
NA 0 {PORT "\dut_esc/esc1\* .Z"}
NA 1 {PORT "\dut_esc/esc1/esc2 .Z"}
NA 1 {PORT "\dut_esc/esc1* .Z"}
NA 1 {PORT "\dut_esc/esc1\* .Z"}
}

```

Example 2

If a fault is desired on the "\dut_esc/esc1*" location (matching the exact location string), use the following to indicate that the individual characters are substituted:

```
"\dut_esc/esc1\*"

input.sff:
-----
FaultGenerate
{
    NA [0,1] { PORT "\dut_esc/esc1\\* " }
}
output.sff
-----
FaultList
{
    NA 0 {PORT "\dut_esc/esc1\* .Z"}
    NA 1 {PORT "\dut_esc/esc1\* .Z"}
}
```

Example 3

A fault location exactly matching the "\dut_esc/esc1*" location requires no modifications: "\dut_esc/esc1*"

```
input.sff:
-----
FaultGenerate
{
    NA [0,1] { PORT "\dut_esc/esc1\* " }
}
output.sff
-----
FaultList
{
    NA 0 {PORT "\dut_esc/esc1\* .Z"}
    NA 1 {PORT "\dut_esc/esc1\* .Z"}
}
```

Exclude Block

The `Exclude` block can be used to exclude specified faults from the fault generation. Faults that are identified with the `Exclude` block are removed from the design and are not retained in the fault list. Fault generation follows the order of the commands inside the SFF file. To exclude faults, you must use it at the end of your fault generate block. If not at the end of the `FaultGenerate` block, other commands may overwrite the `Exclude` command.

Prime faults that are excluded will also excludes all faults that collapse to the prime fault. VC Z01X provides an option to turn this behavior off. For more information, see `excludecolloff`description in [Fault Generation](#).

`Exclude` blocks to not consider sampling directives. Faults are removed whether they are sampled or not. This may cause unexpected fault numbers to be reported when sampling because a sampled fault may later be removed through the `Exclude` block.

Syntax:

```
Exclude
{
( <Fault definition statement> )
}
```

Where,

- `<Fault definition statement>`: See [Fault Descriptors](#) for a complete list of supported fault descriptors. In addition to the standard fault descriptors, there are some extensions supported in the `Exclude` block as detailed below:

Table 7 Example:

Fault Descriptor	Fault(s) Excluded
Single fault exclusion	
<code>FaultGenerate{ NA 0 { WIRE "top.**" } Exclude { NA 0 { WIRE "top.a" } }}</code>	Exclude stuck-at 0 wire fault top.a. Note: If initial faults are:NA 0 { WIRE "top.a" }-- 0 { WIRE "top.b" }-- 0 { WIRE "top.c" }. All three will be excluded.
Non-Recursive Exclude	
<code>FaultGenerate { NA [0,1] { PORT "test1.**" } Exclude { NA 0 { PORT "test1.A.*" } }}</code>	Exclude all stuck-at 0 PORT faults in test1.A
When a module instance is specified, all faults within that module instance will be excluded.	
<code>FaultGenerate{ NA [0,1] { PORT "test1.**" } Exclude { * [*] {[*] "test1.blkA"} }}</code>	All faults in the module instance test1.blkA will be excluded.
Remove all faults under a single instance	
<code>FaultGenerate{ NA [0,1] { PORT "test1.**" } Exclude { * [*] {[*] "test1.blkA.*" } }}</code>	All faults matched in that module instance test1.blkA will be excluded. This is identical to specifying without a trailing wildcard (that is, same as example shown above).
Recursively remove faults under an instance	
<code>FaultGenerate{ NA [0,1] { PORT "test1.**" } Exclude { * [*] {[*] "test1.blkA.**" } }}</code>	All faults contained in the module instance and the entire hierarchy below module instance test1.blkA will be excluded.

Exclude only ARRY faults and VARI faults from a hierarchy	
FaultGenerate {NA [0,1] { [VARI,PORT,ARRY] "test1.**" } Exclude { * [*] {[ARRY,VARI] "test1.blkA.**"} } }	All ARRYand VARI faults contained in the module instance and the entire hierarchy below the module instance test1.blkA will be excluded. Faults with fault classes other than ARRY and VARI in the module instance and the entire hierarchy below will still be generated.
Exclude all transient faults below a hierarchy	
FaultGenerate { Timing("cycle1", CycleTime 2000ps) NA ~ (20:380) { FLOP "test1.**" } Exclude { * [*] (*) {[*] "test1.blkA.**"} } }	

Note:

Adding User-defined Exclude Statuses

You can define user-defined exclusion fault statuses in SFF as follows:

```
StatusDefinitions
{
X0 "My Status 0";
X1 "My Status 1";
ExcludeStatuses
{
J0 "My Excl 0";
J1 "My Excl 1";
}
...
}
```

X0, X1, J0, J1 are all user defined status. But J0, J1 are marked as excluded statuses. They are automatically added to the built-in exclude status group 'EG'.

Hence, J0, J1 cannot be added into any another status group, as they are automatically in EG. EG contains E0-E9, plus the marked excluded user defined statuses (J0, J1 here).

X0, X1 can be added into another status group.

Limitation: If the same fault is present in multiple FMs, and excluded in one of them, it is excluded everywhere (even if the other FM does not have the exclude statement).

Instance Based Exclusion

Exclude blocks offer a way to exclude all faults on a particular location without the need to specify status, fault type, timing (if excluding transient faults), or fault class. An asterisk (*) should be placed in place of each of these fields. For a fault to be excluded by instance-

based exclusion, it must have a single fault origin. Multiple origin faults (MFO faults) are not supported for instance-based exclusion. See [General Fault Descriptions](#) for more information on fields of a descriptor of Standard Fault Format. The fault class can be specified either explicitly, or as a wildcard. Specifying the fault class explicitly is useful when only faults of a certain class are desired to be excluded from an area of the design.

Instance based exclusion requires all fields except location information and fault class to be an asterisk. This means that if status, fault type, and/or timing are explicitly stated but a wildcard is present in another field (other than location) then the descriptor is ignored, and a warning is issued. Wildcarding status, fault type, fault class, and timing are reserved syntax for location exclusion and cannot be used individually, or outside of an exclude block definition. Fault class may be specified explicitly, or it can be an asterisk to specify all fault classes.

Sampling

The `Sampling` directive allows for definition of one or more configurations of fault generation sampling methodologies. A `Sampling` directive is contained within the `FaultGenerate` block.

For more information on using the `Sampling` directive, see the following subsections:

- [Specifying Sampling Methodologies](#)
- [Sampling for Permanent and Transient Faults, and Random Sampling](#)
- [Activating or Deactivating Active Sampling Configurations](#)
- [Viewing the Sampling Results](#)
- [Example: Sampling Directive](#)
- [Scale Factor Aware Sampling](#)
- **Note:**
 - 1) Faults that are reported when sampling is used are limited to only the faults that are included in the generated sample.
 - 2) Sampling directives do not apply to `Exclude` blocks. Faults defined for exclusion within an `Exclude` block will always be excluded, regardless of the active sampling definition.

Specifying Sampling Methodologies

The following sampling methodologies are supported:

- **Confidence model sampling:** For details about this methodology, see [Using Confidence Interval Sampling](#). The syntax is as follows:

```
# Confidence model sampling
Sampling ( <name>, ConfidenceInterval <confidence-interval>,
    ConfidenceLevel <confidence-level> [, Seed <seed>, CoverageEstimate
    <coverage-estimate> ] )
```

Coverage Estimate : Coverage Estimate is used in the statistical calculations for determining the sample size.

where,

sample size = $Z^2 * p*(1-p)/C^2$

C = confidence interval expressed as a decimal

Z = standard Z score based on confidence level

p = coverage estimate expressed as a decimal. By default, the value is 0.5, if Coverage Estimate is not provided.

Coverage Estimate is an estimate of the percentage coverage that you expect and is used in the statistical calculations for determining the sample size. If there is a value that you expect to get for the percentage coverage, you can use that value to reduce the sample size by setting Coverage Estimate to the decimal representation of the expected coverage percentage. A higher Coverage Estimate value will result in fewer faults being selected.

Example:

```
Sampling ("samp2", Seed 123456789, ConfidenceInterval 5.0,
    ConfidenceLevel 95,
    CoverageEstimate 0.25)
```

where $0 < \text{Coverage Estimate} < 1$. If Coverage Estimate is not specified, the default value of 0.5 is used.

- **Percentage based sampling:** In this methodology, the fault campaign compiler selects a percentage of faults from the potential fault set. The syntax is as follows:

```
# Percentage-based sampling
Sampling ( <name>, Percentage <percentage> [ , Seed <seed> ] )
```

- **Fixed number sampling:** This method uses the same approach as percentage-based sampling, except that a given fixed number of faults are selected rather than a

percentage. SFF Fixed fault sampling samples over the full sampling “section” rather than on a per-statement basis. The syntax is as follows:

```
# Fixed number sampling
Sampling ( <name>, Number <number> [ , Seed <seed> ] )
```

Example

For the following snippet, a total of 10 faults are generated.

```
Sampling("10 faults", Number 10, Seed 4824)

NA [0] { [PORT] "test.risc1.**" }
NA [1] { [PORT] "test.risc1.**" }
```

Here,

- <name>: Quoted string used to uniquely identify a sampling configuration.
- <confidence-interval>: Integer or floating point value greater than 0.0 and less than or equal to 100.0.
- <confidence-level>: Value from set of 70, 75, 80, 85, 90, 92, 95, 96, 98, 99.
- <coverage-estimate> : floating point number between 0 and 1, exclusive.
- <percentage>: Value between 0.0 and 100.0. For example, to specify 10 percent, you can either set <percentage> to 10 or 10.0.
- <seed>: Unsigned 32-bit integer value.
- <number>: Unsigned 32-bit integer greater than 0.

Note:

Sampling does not consider untestable faults by default.

Sampling for Permanent and Transient Faults, and Random Sampling

Random sampling occurs on a pseudo-random basis using the given seed value or the default if no seed was provided. The default seed value is 0x98724385. The seed value is used to initialize the pseudo-random number generator when the fault generator enters a FaultGenerate section.

Faults left out of the sample are not retained within the resulting fault list.

This section describes how sampling occurs for permanent and transient faults.

- Permanent fault sampling: Sampling occurs after generation of the complete fault list. Permanent fault sampling considers all generated prime faults. Any faults collapsing to a sampled prime fault are included in the sample and retain their status as a collapsed fault.
- Transient fault sampling: Sampling occurs as faults are generated. Transient fault sampling considers all faults (including collapsed). The fault collapsing is determined after sampling.
- Transient fault sampling corner cases: By default, corner case sampling for transient faults is not enabled. Set the `CornerCases` Sampling directive argument to 1 to enable it.

```
Sampling ( <name>, CornerCases < 0 | 1 > )
```

The `CornerCases` setting controls whether transient fault sampling includes cycle range corner cases in the sample. Default value is 0 (off).

After `CornerCases` is set to 1, when a fault definition statement containing a cycle range is detected, the injection times of all faults are evaluated so that two corner cases are guaranteed to be included in the sample. They are:

- Fault with the first injection time
- Fault with the last injection time

After isolation of the corner cases, the number of faults in a 100% sampling is calculated. Fault sampling is then performed according to the active sampling parameters.

Activating or Deactivating Active Sampling Configurations

Specify the `UseSampling` directive to activate an active sampling configuration. You can define multiple `UseSampling` directives anywhere within a `FaultGenerate` section to change the active sampling configuration as required. This enables you to dynamically change the sampling behavior applied to the hierarchies of a design. The `UseSampling` directive is useful when multiple sampling definitions have been defined and you must alternate between active sampling definitions throughout a `FaultGenerate` block since identical sampling definitions are not allowed to be defined.

The last defined `Sampling` or `UseSampling` directive is the active sampling rate. An empty `UseSampling` directive deactivates any active sampling configuration. If a `UseSampling` directive is not specified, the last `Sampling` directive defined is used as the default sampling configuration.

Note:

Sampling directives do not apply to Exclude blocks. Faults defined for exclusion within an Exclude block will always be excluded, regardless of the active sampling definition.

The syntax is as follows:

```
UseSampling ( [ <name> ] )
```

Where, <name> is a quoted string used to identify the sampling configuration that needs to be activated.

Example 1

The following FaultGenerate section shows the Sampling directive for confidence level sampling. The name that uniquely identifies the sampling configuration is "config1". The "config1" sampling configuration is activated by the UseSampling directive.

```
FaultGenerate {
    Sampling ("config1", ConfidenceInterval 2.0, ConfidenceLevel 99)
    Timing("clock1", CycleTime 10ns, Offset 0ns)

    UseTiming("clock1")
    UseSampling("config1")

    NA ~ (0:1000000001) {"test.u3.0"}
}
```

Example 2

The following FaultGenerate section allows a 20% sample to be applied to faults generated within the hierarchy "poly.multbx.***" and a 1% sample to be applied to faults generated within the hierarchy "poly.addax_2andbx.***":

```
FaultGenerate
{
    Sampling("20 percent", Percentage 20.0, Seed 9873)
    NA [0,1] { "poly.multbx.***" }

    Sampling("1 percent", Percentage 1.0, Seed 4824)
    NA [0,1] { "poly.addax_2andbx.***" }
}
```

Example 3

The following FaultGenerate section defines two different sampling definitions: "1 percent" and "20 percent". Since "20 percent" was defined after "1 percent", "20 percent" is the active sampling rate and is applied to the fault generation of hierarchy "poly.multbx.***".

The UseSampling("1 percent") directive then signifies that a 1% sampling rate should be applied to the fault generation of hierarchy "poly.addax_2andbx.***". While, the

`UseSampling("20 percent")` directive then signifies that the 20% sampling rate should be applied to the fault generation of hierarchy "poly.multx_2.**".

```
FaultGenerate
{
    Sampling("1 percent", Percentage 1.0, Seed 4824)
    Sampling("20 percent", Percentage 20.0, Seed 9873)

    NA [0,1] { "poly.multbx.**" }

    UseSampling("1 percent")
    NA [0,1] { "poly.addax_2andbx.**" }

    UseSampling("20 percent")
    NA [0,1] { "poly.multx_2.**" }
}
```

Viewing the Sampling Results

The number of faults sampled out of the possible fault set is reported in the `vc_fcc log` and the fault report output, as shown below:

Total faults generated: X (sampled from Y possible faults).

Example: Sampling Directive

This example shows a Standard Fault Format file (Input.sff), which contains the `FaultGenerate` section. The corresponding `vc_fcc log` (`vc_fcc.log`) and the fault report output (Output.rpt) are also shown.

```
Input.sffFaultGenerate{Sampling ("config1", ConfidenceInterval 50.0,
ConfidenceLevel 99)Timing("cycle1", CycleTime 2000ps, Offset 1800ps)NA
~ (20:30) {"test.risc1.alureg.d6.q"} }vc_fcc.logLoading design
information...

Done (CPU time: 0s, elapsed time: 0s, memory increased by 44Mb,
peak: 65Mb). Parsing fault list... Done (9 faults generated, CPU time:
0s, elapsed time: 0s, memory increased by 1Mb, peak: 67Mb). Sampling
(SMPA1)... Done (3 faults sampled, CPU time: 0s, elapsed time: 0s,
memory increased by 0Mb, peak: 67Mb). Collecting design strobes... Done
(CPU time: 0s, elapsed time: 0s, memory increased by 0Mb, peak:
67Mb). Loading design details... Done (CPU time: 0s, elapsed time: 0s,
memory increased by 1Mb, peak: 67Mb). Constant propagation... Done (CPU
time: 0s, elapsed time: 0s, memory increased by 0Mb, peak: 68Mb). Static
testability analysis... Done (CPU time: 0s, elapsed time: 0s, memory
increased by 0Mb, peak: 68Mb). Reconvergence check... Done (CPU time: 0s,
elapsed time: 0s, memory increased by 0Mb, peak: 68Mb). Pruning... Done
(CPU time: 0s, elapsed time: 0s, memory increased by 0Mb, peak:
68Mb). Collapsing... Done (CPU time: 0s, elapsed time: 0s, memory
increased by 0Mb, peak: 69Mb). Preparing to write to FDB... Done (CPU
time: 0s, elapsed time: 0s, memory increased by 0Mb, peak: 69Mb). Writing
fault campaign to FDB... Done (CPU time: 0s, elapsed time: 0s, memory
```

```
increased by 1Mb, peak: 70Mb).Total faults generated: 7 (sampled from
11 possible faults).Collapsed faults: 0.Untestable faults: 0.Fault
reduction: 0%.Output.sffFaultList(Timing("cycle1", CycleTime 2000ps,
Offset 1800ps)UseTiming("cycle1")NA ~ (20:21, 23, 25:26, 28:29)
{"test.risc1.alureg.d6.q"})}
```

Scale Factor Aware Sampling

Functional safety FMEDA calculations commonly include a failure mode distribution (FMD), or scale factor (SF). The FMD enables tools to specify which failure modes are more likely to be encountered by the device during operation on a global level. At a local or block level, tools like VC Functional Safety Manager use SF to assign the amount of a block's design data to a failure mode. To accurately sample from the blocks in a design VC Z01X has the ability to consider scale factors for the purposes of weighting fault generation when using statistical sampling.

`ScaleFactor` blocks are groups of fault descriptors being generated under the same active sampling directive. The potential fault population is created for the active sampling block and `ScaleFactor` is used to weight the sample for each scale factor. The `ScaleFactor` directive applies to all fault statements until a new `FaultGenerate`, `UseSampling`, `Sampling`, or `ScaleFactor` directive is executed. `ScaleFactor` values within the same sample need-not add up to 100%. Multiple `ScaleFactor` applied within the same sampling definition interact with one another for the purposes of determining the sample size needed for each `ScaleFactor` in the `ScaleFactor` block.

`ScaleFactor` must be specified within a `FaultGenerate` block after a sampling definition is activated. It supports both fixed-number and confidence interval sampling methods. `ScaleFactor` requires that systematic sampling is enabled or else an error will be issued when faults are generated. The number of calculated faults for an individual `ScaleFactor` will be rounded to the nearest integer. This might lead to the sample size being slightly less or slightly more than expected, especially if large numbers of `ScaleFactor` directives are used in the same sample.

Synopsys does not recommend that fault definitions overlap when using `ScaleFactor`. The faults generated do not consider duplicate faults and may be counted multiple times within the same block when overlapping fault definitions exist. The sample sizes might be lower than expected due these fault duplicates. If there are duplicates, `ScaleFactor` will only generate a single fault which may cause overall fault numbers to appear incorrect.

When applying a large `ScaleFactor` to a small fault population or a small `ScaleFactor` to a large fault population, there may not be enough faults to fulfill the faults required by the sample. This discrepancy is noted as a warning and leads to the overall sampled fault size being less than requested.

Syntax:

```
ScaleFactor( <value> )
```

Where:

- <value>: is either a floating point or integer value specifying the effective sample weight to apply to all the fault definitions which follow it within the range 0.0 to 100.0 inclusive. If no <value> is specified, the default of 100% applies. Floating point values are accurate to two decimal places. The <value> specified to `ScaleFactor` directive corresponds to a percentage. Specifying 5.25 means a scaling of 5.25% is targeted. A value of 0.525 specified to `ScaleFactor` means a scaling of 0.525%.

Example:

The following is an example of utilizing `ScaleFactor` within SFF, along with the internal calculations to determine the proper sample size from each fault block. The example uses fixed-number sampling. Confidence level/interval sampling would be identical, but requires an internal tool calculation of the Targeted Sample Size prior to calculating each `ScaleFactor` value.

```
SFF file:  
FaultGenerate  
{  
Sampling("S1", Number 1000)  
ScaleFactor(20.0)  
NA [0,1] { PORT "top.dut.BlkA.**" }  
ScaleFactor(80.0)  
NA [0,1] { PORT "top.dut.BlkB.**" }  
ScaleFactor(10.0)  
NA [0,1] { PORT "top.dut.BlkC.**" }
```

ScaleFactor Sample Calculations:

<i>Design Block</i>	BlkA	BlkB	BlkC	Total
<i>Targeted Sample_Size</i>				1,000
<i>Fault_Population</i>	10,000	4,000	20,000	34,000
<i>Scale Factor</i>	20%	80%	10%	
<i>SF_Faults (internal)</i>	2,000	3,200	2,000	7,200
<i>% SF_Faults (internal)</i>	27.78%	44.44%	27.78%	
<i>Final ScaleFactor Sample_Size</i>	278	444	278	1000

Internal Calculations:

BlkA:

$SF_Faults = Fault_Population \times SF$	$2,000 = 10,000 \times 20\%$
$SF_Faults "Total" = \sum SF_Faults$	$7,200 = \sum 2,000 + 3,200 + 2,000$
$\%_SF_Faults = \frac{SF_Faults}{SF_Total} \times 100$	$27.78\% = \frac{2,000}{7,200} \times 100$
$Final ScaleFactor Sample Size = Sample_Size \times \%_SF_Faults$	$278 = 1,000 \times 27.78\%$

BlkB:

$SF_Faults = Fault_Population \times SF$	$3,200 = 4,000 \times 80\%$
$SF_Faults "Total" = \sum SF_Faults$	$7,200 = \sum 2,000 + 3,200 + 2,000$
$\%_SF_Faults = \frac{SF_Faults}{SF_Total} \times 100$	$44.44\% = \frac{3,200}{7,200} \times 100$
$Final ScaleFactor Sample Size = Sample_Size \times \%_SF_Faults$	$444 = 1,000 \times 44.44\%$

BlkC:

$SF_Faults = Fault_Population \times SF$	$2,000 = 20,000 \times 10\%$
$SF_Faults "Total" = \sum SF_Faults$	$7,200 = \sum 2,000 + 3,200 + 2,000$
$\%_SF_Faults = \frac{SF_Faults}{SF_Total} \times 100$	$27.78\% = \frac{2,000}{7,200} \times 100$
$Final ScaleFactor Sample Size = Sample_Size \times \%_SF_Faults$	$278 = 1,000 \times 27.78\%$

FaultInfo Section

The `FaultInfo` section describes the per fault information displayed before a fault line. This section is for Output only.

Fault descriptors are detailed later in this chapter, including how the `FaultInfo` data is displayed. Each entry within the `FaultInfo` section is a keyword for a piece of fault information. The keywords is:

- `TestNum`: The test number associated with this fault.

If other keywords appear that are not supported by VC Z01X, a warning is printed, and that position within the fault information section is ignored. Keywords are case insensitive.

Example: The FaultInfo Section

```
FaultInfo
{
  TestNum;
}
```

Each item has a predefined type for the data that is displayed. `TestNum` is a 32-bit unsigned integer value.

FaultList Section

The `FaultList` statement encloses all fault definitions.

```
FaultList [<Name>]
{<FaultList information>}
```

The `FaultList` section is used to provide a list of faults as input to fault generation. This list may be the output of a previous coverage report. It is also possible to use the fault list section from reporting as the input to fault generation. It is recommended to use the `FaultGenerate` section if using wildcard characters to search the hierarchy or hierarchies of the design (that is, `testbench.dut.**`).

The optional parameter `<Name>` is used to associate the `FaultList` block with a previously defined `FailureMode`.

See the following subsections:

- [General Fault Descriptions](#)
- [Fault Timing and Cycle Information](#)
- [Setting Active Cycle Timing](#)
- [Injection Faults Based on Verilog Condition](#)

Also see [Common Keywords/Blocks/Constructs for FaultList and FaultGenerate](#).

General Fault Descriptions

Fault Descriptions

The fault descriptor is the syntax used to define any fault that will be used during fault simulation. The descriptor communicates all the information needed to provide data about the type, location, status, and basic behavior of a fault.

A fault definition terminates at the end of a line; use the \ (back-slash) character to continue on a new line.

The general format is as follows.

Syntax:

Stuck-at Fault:

```
<fault info> <status> <fault type> {<location info>} [results]
```

Transient Fault:

```
<fault info> <status> <fault type> (<timing info>) {<location info>} [results]
```

Where:

- <fault info>: Optional, but if used then the `FaultInfo` section needs to be defined earlier in the SFF. The fault-info section is enclosed in <>. If there are too few or too many items in the list compared to the `FaultInfo` definition section, then warnings will be issued, and the missing/extra items will be ignored. This section is generated by the FCM `coverage` command and is not required when creating the initial SFF.
- <status>: Is one of the status values either from the `StatusDefinition` section or one of the VC Z01X built-in fault statuses.
- <fault type>: Either a 0/1 for a value-based fault, or a ~ (tilde) character to represent a bit flip of the current simulation value.
- <timing info>: Is present only for transient faults and is described in detail in the transient fault section below.
- <location info>: Consists of two fields:
 - <Location Type>: Is the type of construct where the fault will be placed. The location type will vary for different faults and will come from the list of support types, PORT, FLOP, ARRY, WIRE, PRIM, or VARI.
 - <path>: Is a quoted path to the location hierarchical location where the fault is injected.

Example

```
# Not attempted stuck-at 0 fault on the port of a cell
NA 0 { PORT "testbench.dut.blkA.Z" }
# Dropped detected transient fault at cycle 50 on a flip-flop
DD ~ (50) { FLOP "testbench.dut.blkA.flopB" }
```

See the following for individual fault type timing and location information:

- [Stuck-At Fault List Format](#)
- [Transient Toggle Fault List Format](#)

- Transient Hold Fault List Format
- Multiple Fault Origin Format

Stuck-At Fault List Format

Stuck-at fault are the simplest type of fault to describe in SFF. The descriptor can contain the following fields:

Syntax:

```
<fault info> <status> <fault type> {<location info>} [results]
In the <location info> field, the location can be expanded to:
{ <location type> "<path>" + <location type> "<path>" + ...}
```

The fault descriptor fields for stuck-at faults can contain:

- <fault info>: Optional, but if used then the `FaultInfo` section needs to be defined earlier in the SFF. The fault-info section is enclosed in <>. If there are too few or too many items in the list compared to the `FaultInfo` definition section, then warnings will be issued, and the missing/extra items will be ignored. This section is generated by `fault_report` and is not required when creating the initial SFF.
- <status>: Is one of the status values either from the `StatusDefinition` section or one of the VC Z01X built-in fault statuses.
- <fault type>: Either a 0/1 for a value-based fault.
- <location info>: The location info can contain a single location in the design, or a plus separated list of locations. If a list is provided a fault will be injected at each location but simulated in a single FM as one fault.
 - <location type>: Type of location for faults. Can be one of the following: WIRE, PRIM, PORT, ARRY, FLOP or VARI.
 - <path>: Quoted path to the instance that should receive the fault. VARI, ARRY, and PORT faults can take vector part selects with the syntax `[msb:lsb]`.
 - + (plus) :- Create a separate fault at each injection point at each location.

Table 8 Examples:

Fault Descriptor
Not attempted stuck-at 0 fault on the port of a cell
NA 0 { PORT "testbench.dut.blkA.Z"}
Dropped detected stuck-at 1 fault on a cell port with included fault info
<1 1 1 0> DD 1 { PORT "testbench.dut.blkA.B"}

Single fault with multiple fault injection points
NA 0 { PORT "system.processor.alu.in1"} + { PORT "system.processor.alu.in5"}NA 0 { PORT "system.processor.alu.in1"} + 1 { PORT "system.processor.alu.in5"}

Transient Toggle Fault List Format

Transient toggle faults, also known as Single Event Upset (SEU) or Single Event Transient (SET) are the most common transient faults used for functional safety. The SEU fault model determines the value of the flip-flop or latch where it is injected at the time of injection and sets the fault value to the opposite state. The fault location will remain in that state until the next update of the sequential element causes it to be overwritten.

Transient faults may be created in SFF as cycle-based faults. The Timing section is required for transient toggle fault list when using cycle-based timing to inject faults, otherwise `vc_fcc` reports a warning message and does not generate the fault. Absolute timing is not supported yet.

Syntax:

```
<fault info> <status> <fault type> (<timing info>) {<location info>}
  [results]
  ("timingID" <timing_options> [cycle1,cycle2,...])
```

The fault descriptor fields for transient toggle faults can contain:

- <fault info>: Optional, but if used then the `FaultInfo` section needs to be defined earlier in the SFF. The fault-info section is enclosed in <>. If there are too few or too many items in the list compared to the `FaultInfo` definition section, then warnings will be issued, and the missing/extraneous items will be ignored. This section is generated by the FCM `coverage` command and is not required when creating the initial SFF.
- <status>: Is one of the status values either from the `StatusDefinition` section or one of the VC Z01X built-in fault statuses.
- <fault type>: ~ for the transient toggle fault model.
- <timing info>: Timing info is contained within ()'s and can contain:
 - timingId: An optional quoted string used to override the current `timingId`. If used, it must be the first item and must be a unique name. Duplicate names will cause an error.
 - ,: A comma ends the current expansion of faults and starts a new fault definition.

- <timing options>: Can be any combination of the following:
- [start:end]: An inclusive range of cycles to generate faults. One fault is generated for each unique cycle in the listed range. The start and end cycles need to be integer values to use cycle times from a Timing directive. Absolute timing with time value (ns,ps, and so on) is not allowed.
- <location info>: The location info can contain a single location in the design or a plus separated list of locations. If a list is provided a fault will be injected at each location but simulated in a single FM as one fault.
 - <location type>: Type of location for faults. Can be one of the following: PRIM, PORT, ARRY, FLOP or VARI. If the location type is not specified, the fault generator searches for a location matching any of the supported location types.
 - <path>: Quoted path to the instance that should receive the fault. VARI, ARRY, and PORT faults can take vector part selects with the syntax [msb:lsb].
 - ,: A comma used to separate path names indicates that a new fault should be defined for each path in the list.

Table 9 Examples

Fault Descriptor	Fault Created
Individually Specified Cycle Based	
NA ~ (50,87,352) { FLOP "instanceA.flopB"}	Cycle 50 flopB Cycle 87 flopB Cycle 352 flopB
Comma separated list, combine timing and location	
NA ~ (8,9) {FLOP "inst1.flopA", FLOP "inst2.flopC"}	Cycle 8 inst1.flopA Cycle 8 inst1.flopC Cycle 9 inst1.flopA Cycle 9 inst1.flopC
Cycle based range (times from UseTiming)	
NA ~ ([8:10]) { FLOP "top.blkA.flop7"}	Cycle 8 top.blkA.flop7 Cycle 9 top.blkA.flop7 Cycle 10 top.blkA.flop7
Absolute range with frequency defined	
ND ~ (4ps:44ps 10ps) { VARI "system.alu.reg1"}	4ps system.alu.reg1 14ps system.alu.reg1 24ps system.alu.reg1 34ps system.alu.reg1 44ps system.alu.reg1
Absolute range with default frequency	
ND ~ (6ps:10ps) { VARI "system.alu.reg1"}	6ps system.alu.reg1 7s system.alu.reg1 8ps system.alu.reg1 9ps system.alu.reg1 10ps system.alu.reg1

Specify Timing block with fault definition – Cycle timing from UseTiming “clock2”	
NA ~ (“clock2” [8:10]) { FLOP “top.blkA.flop7”}	Cycle 8 top.blkA.flop7 Cycle 9 top.blkA.flop7 Cycle 10 top.blkA.flop7

Transient Hold Fault List Format

Transient hold faults are an extension of transient toggle and transient 0/1 fault models. The behavior of the fault is similar to those models and extends the capability to provide an amount of time where the fault is held to the new value before being released in simulation. After release, the fault location will remain in that state until the next update of the sequential element causes it to be overwritten. For transient hold faults placed on the output of sequential logic, the effect of the fault release is not realized until the next input change on the element causes the output value to be updated. For transient hold faults placed on combinational logic, the effect of the fault will be realized immediately upon release time.

The `Timing` section is necessary for transient toggle fault list, otherwise `vc_fcc` reports a warning message and does not generate the fault.

Syntax:

```
<fault info> <status> <fault type> (<timing info>) {<location info>}  
[results]
```

The fault descriptor fields for transient hold faults can contain:

- `<fault info>`: Optional, but if used then the `FaultInfo` section needs to be defined earlier in the SFF. The fault-info section is enclosed in `<>`. If there are too few or too many items in the list compared to the `FaultInfo` definition section, then warnings will be issued, and the missing/extraneous items will be ignored. This section is generated by the `FCM coverage` command and is not required when creating the initial SFF.
- `<status>`: Is one of the status values either from the `StatusDefinition` section or one of the VC Z01X built-in fault statuses.
- `<fault type>`: Either a 0/1 for a transient 0/1 fault or a `~` for a bit flip.
- `<timing info>`: Timing info is contained within `()`'s and can contain:
 - `timingId`: An optional quoted string used to override the current `timingId`. If used, it must be the first item and must be a unique name. Duplicate names will cause an error.
 - `,`: A comma ends the current expansion of faults and starts a new fault definition.

- <timing options>: Can be any combination of the following.
- [start^end]: Define the hold period. The fault is either toggled or set to 0/1 at the start of the time and the new value is held until the end of the time.
- <location info>: The location info can contain a single location in the design or a plus separated list of locations. If a list is provided a fault will be injected at each location but simulated in a single FM as one fault.
 - <location type>: Type of location for faults. Can be one of the following: PRIM, PORT, ARRY, FLOP or VARI. If the location type is not specified, the fault generator searches for a location matching any of the supported location types.
 - <path>: Quoted path to the instance that should receive the fault. VARI, ARRY, and PORT faults can take vector part selects with the syntax [msb:lsb].
 - ,: A comma used to separate path names indicates that a new fault should be defined for each path in the list.
 - + (plus): Create a separate fault at each location.

Table 10 Examples

Fault Descriptor	Fault Created
Individually Specified Cycle Based	
NA ~ (50^51,87^89) { FLOP "instanceA.flopB"}	Cycle 50-51 flopB Cycle 87-89 flopB
Comma separated list, combine timing and location	
NA 1 (8^9,9^11) {FLOP "inst1.flopA", FLOP "inst2.flopC"}	Cycle 8^9 inst1.flopACycle 8^9 inst1.flopCCycle 9^11 inst1.flopA Cycle 9^11 inst1.flopC
Specify Timing block with fault definition – Cycle timing from UseTiming “clock2”	
NA 1 ("clock2" [8^10]) { FLOP "top.blkA.flop7"}	Cycle 8^10 top.blkA.flop7

Multiple Fault Origin Format

In addition to standard faults types described earlier in this chapter, VC Z01X supports multiple fault origin faults. These faults extend the capabilities of the tool to place a single fault in multiple locations in the simulation. The fault types, values, and times of injection may be of mixed types from the fault descriptors above, but combine two or more locations

where a fault occurs. These fault descriptors allow description and simulation of multi-point fault for functional safety.

Syntax:

<fault descriptor> + <fault descriptor>

The fault descriptor fields for transient hold faults can contain:

- <fault descriptor>: Each fault descriptor in a multi-fault origin is a fault descriptor from one of the previous sections of this chapter.
- + (plus): Separate fault descriptors.

Table 11 Examples

Fault Descriptor	Fault Created
Mixed transient toggle and stuck-at fault definitions	
ND ~ (10) { FLOP "test.u1" } + 1 { PORT "test.u2.O" }	One stuck-at fault on PORT O and one transient toggle fault on flop u1 at cycle 10
Transient hold faults at two different locations at two different times	
ND 0 (10^15) {PORT "test.a.b.I1"} + 1 (13^20) {PORT "test.a.b.c.I2"}	One fault is created with a transient zero hold time from cycle 10-15 at test.a.b.i1 and a transient one hold time from cycle 13-20 at test.a.b.c.i2.
Toggle/Hold with additional cycle information	
NA ~ ([8^14]) { FLOP "i1.F1" } + ([5:7]) { FLOP "i1.F2" }	#1 Cycle 5 "i1.F2", Cycle 8 toggle "i1.F1" hold until cycle 14 #2 Cycle 6 "i1.F2", Cycle 8 toggle "i1.F1" hold until cycle 14 #3 Cycle 7 "i1.F2", Cycle 8 toggle "i1.F1" hold until cycle 14
# Multiple transient zero/ one definitions	
NA 0 ([8^15]) {PORT "i1.F1"} + 1 ([8^20]) {PORT "i1.F2"}	#1 Cycle 8-15 i1.F1 stuck-at 0 Cycle 8-20 i1.F2 stuck-at 1

Fault Timing and Cycle Information

The *Timing* section specifies the *CycleTime* and *Offset* values with an associated ID. The *Timing* directive is used to determine the injection times for cycle based transient faults.

It is recommended to specify *CycleTime* and *Offset* parameters as integer values. The use of floating-point values might require scaling to meet the design timescale precision

for the simulator and can cause loss in precision. In the case that floating point values must be used, `vc_fcc` reports a warning message if precision is lost due to scaling. If a value cannot be scaled appropriately to meet the design precision, an error message is reported.

Cycle based faults are delayed when using `$fs_inject`, `create_testcases -fault_injection_time <string>`, or `-fsim=fault+inject+<string>`. See [Defining Fault Timing for Transient Faults](#) for more information.

Syntax:

```
Timing("<ID>", CycleTime <time>, Offset <time> [, Condition
"<Expression>" | "<Condition_Id>"])
```

Where:

- **ID:** A quoted string of characters to uniquely identify a particular timing group.
- **CycleTime:** Is required and indicates the duration of the clock cycle. The cycle time may optionally include a timescale.
- **Offset:** An optional value indicating the time within the cycle to inject the fault. The `Offset` value must be less than the `Cycle` value. Alternatively, you can specify the `Offset` value as a percentage of the `CycleTime`. If no `Offset` is specified, the fault is injected at the first time of the cycle.
- **Expression:** Verilog expression. See “[Supported Condition Expression Operators and Operands](#)” for supported constructs. The fault will only be injected in cycles where the expression evaluates to True.
- **Condition_ID:** ID of an existing `Condition` section to associate with this `Timing` section.

Examples:

```
#50ns cycle with 40ns offset
Timing("clock1", CycleTime 50ns, Offset 40ns)
#25ns cycle with 5ns offset
Timing("clock2", CycleTime 25ns, Offset 5ns)
# 50ns cycle with 10% offset
Timing("T1", CycleTime 50ns, Offset 10%)
```

Setting Active Cycle Timing

The `UseTiming` section defines which timing definition to use. This can be changed at any time within the fault descriptor section. The timing section can also be overridden when declaring the cycle information and in individual fault descriptors.

Syntax:

```
UseTiming("<ID>")
```

Where:

- ID: A quoted string of characters to uniquely identify a particular timing group.

Example:

```
UseTiming("cycle1")
```

Injection Faults Based on Verilog Condition

The Condition section specifies a condition for injection of transient faults based on a Verilog expression. A fault associated with a Condition is only injected during simulation if the expression evaluates to true at the time of the potential injection.

Syntax:

```
Condition("<ID>", "<Expression>")
```

Where:

- ID: A quoted string of characters to uniquely identify a particular timing group.
- Expression: Verilog expression. See “*Supported Condition Expression Operators and Operands*” for supported constructs.

Example:

```
Condition("condition1", "testbench.dut.rst == 1")
Timing("clock1", CycleTime 10ns, Condition "condition1")
```

Common Keywords/Blocks/Constructs for FaultList and FaultGenerate

This section describes the following:

- [Condition Information](#)
- [Example of Using FaultList and FaultGenerate Together](#)

Condition Information

The Condition section specifies a condition for injection of an SEU or SET fault based on a Verilog expression. A fault associated with a Condition is only injected during simulation if the expression evaluates to true at the time of the potential injection.

Syntax:

```
Condition("<Id>", "<Expression>")
```

Where:

- Id: A quoted string of characters to uniquely identify a particular timing group.
- Expression: Verilog expression. See the following table or supported constructs.

Table 12 Supported Condition Expression Operators and Operands

Supported Condition Expression Operators		Supported Condition Expression Operands
Relational	>, >=, <, <=	Constant number
Logical equality	==	Net
Logical inequality	!=	Variable
Logical negation	!	
Logical AND	&&	
Logical OR		

Example

```
Condition("condition1", "testbench.dut.rst == 1")
Timing("clock1", CycleTime 10ns, Condition "condition1")
```

Example of Using FaultList and FaultGenerate Together

This example shows usage of a FaultList and FaultGenerate block together in a single SFF file. Individual fault statements are processed in the same order as the input file is read. If a fault to be imported/generated already exists, a merge is initiated. Merged fault statuses are resolved according to the PromotionTable.

Example:

```
input.sff:
FaultGenerate
{
NA 1 { PORT "test1.*" }
}
FaultList
```

```
{
NA 1 { PORT "test1.A" }
DD 1 { PORT "test1.B" }
NA 1 { PORT "test1.C" }
}
Output.sff:
FaultList
{
NA 1 { PORT "test1.A" }
DD 1 { PORT "test1.B" }
NA 1 { PORT "test1.C" }

}
```

Example Coverage Reports Using Standard Fault Format

See the following subsections:

- [Example Coverage Report](#)

Example Coverage Report

```
report -report coverage.sff
```

Sample Coverage

```
Tool("VC Z01X")
Info(" Type: Fault Coverage Report")
TestList {
1 design {"No tests found"}
}
FaultList{
Timing("clock1", CycleTime 10ns)
UseTiming("clock1")
ND ~ (12:51, 62:101) {"test.u3.0"}
DD ~ (52:61, 102:111) {"test.u3.0"}
IX ~ (0:11) {"test.u3.0"}
}
#-----
# Fault Coverage Summary for Default List
#
# Total
#
# Number of Faults: 112 100.00%
#
# Untestable Faults: 0 0.00% 0.00%
# Untestable Unused UU 0 0.00% 0.00%
#
# Testable Faults: 112 100.00% 100.00%
# Dropped Detected DD 20 17.86% 11.96%
```

Chapter 9: Standard Fault Format

Example Coverage Reports Using Standard Fault Format

```
# Dropped Potential PD 0 0.00% 0.00%
# Impossible x-state IX 12 10.71% 10.71%
# Not Detected ND 80 71.43% 71.43%
# Coverage -----
# Test Coverage 17.86%
# Fault Coverage 17.86%
#-----
```

10

Fault Generation

the following:

- [Fault Generation Compiler Directives](#)
 - [Design Compilation Options](#)
 - [Fault Selection Using System Tasks](#)
 - [Understanding Fault Placement](#)
-

Fault Generation Compiler Directives

By default, VC Z01X faults are injected to all devices in the circuit. However, you can enable VC Z01X to inject faults, or not to inject faults, in specific parts of the design. You may wish to do this in designs that contain devices that are not actual primitives. Such devices can find their way into a circuit, for example, during model library development, when simulation models are included.

The following compiler directives can be used to control fault creation.

- `enable_portfaults / `disable_portfaults compiler directives can be used to enable faults on the ports of a module.

These directives are often used in conjunction with `suppress_faults to fault only the ports of a cell while omitting faults on the internal description of the cell.

Example:

```
`suppress_faults
`enable_portfaults
module MUX2(out, sel, a, b);
    output out;
    input sel, a, b;
    and sel_ag(sel_a, a, sel);
    and sel_bg(sel_b, b, sel_bar);
    inv sel_barg(sel_bar, sel);
    or out_g(out, sel_b, sel_a);
endmodule
`disable_portfaults
`nosuppress_faults
```

This example produces faults on the ports (out, sel, a, b) but no faults internal to the module. In modules that contain multiple primitives, port faults may not provide an accurate answer. For example, a cell that cascades two nands has five prime port faults but six prime primitive faults. It is possible to detect the extra primitive fault without detecting any other faults. Using port faults gives an optimistic result. Port faults should only be used for complex cells such as sequential cells.

Note:

Since the `suppress_faults and `enable_portfaults compiler directives are honored when generating faults through the vc_fcc tool from a Standard Fault Format (SFF) file for wildcarded locations, faults are not generated within suppressed areas of the design as denoted by the `suppress_faults directive or through the -fsim=portfaults compile option. To generate faults in suppressed areas of the design, use the -ignore_suppress vc_fcc option. Similarly, faults are not generated on ports not enclosed by `enable_portfaults. To generate faults on ports not enclosed within `enable_portfaults/`disable_portfaults, use the -fsim=portfaults compile option.

- `begin_faultfree / `end_faultfree can be used to block all fault effects from propagating outside the enclosed instantiated primitives and modules. Blocking of fault effect propagation is done both on the ports and hierarchical references.

This compiler directive is useful when using a testbench for fault simulation. In the testbench, instantiate the DUT between the `begin_faultfree and `end_faultfree compiler directives. This will prevent the testbench from diverging due to fault effects from the DUT propagating into the testbench. It is important to not allow the testbench to diverge because this may cause incorrect fault coverage.

In addition to inserting these compiler directives, faults should be limited to the DUT with instance-specific fault generation/exclusion. Sensing must be done inside the DUT or on the DUT ports, since the fault effects will be blocked and will not go outside to the testbench.

Example:

In this example, fault effects will not be propagated into or out of b2 and b3.

```

module test (o, i1, i2, i3);
    output o;
    input i1, i2, i3;
    wire l1, l2, l3;
    buf b1 (l1, i1);
    `begin_faultfree
    buf b2 (l2, i2);
    BUF b3 (l3, i3);
    `end_faultfree
    and a1 (o, l1, l2, l3);
endmodule

module BUF (o, i);
    output o;
    input i;
    wire z;
    buf B1 (z, i);
    buf B2 (o, z);
endmodule

```

- **`begin_faultprevent / `end_faultprevent: Compile directives`**
``begin_faultprevent` and ``end_faultprevent` can be placed around module instances to allow the instance to behave only as a good machine (GM). This behavior is similar but opposite to ``begin_faultfree` and ``end_faultfree`. Instead of preventing fault effects from propagating outside of the specified hierarchy, ``begin_faultprevent/`end_faultprevent` prevents fault effects from propagating inside the specified hierarchy. The fault origins cannot be placed within the specified prevent hierarchy and any faults specified in the hierarchy for generation will not be created.

Example:

```
module test (o, i1, i3);
    output o;
    input i1, i3;
    wire i1, i3
    buf b1 (i1, i1);
    `begin_faultprevents
    buf b2 (i2, i1);
    BUF b3 (i3, i3);
    `end_faultprevents
    and a1 (o, i1, i2, i3);
endmodule
```

Fault on b1.i1 won't affect b2.i1. b2.i1 will use Good machine value.

vczoix cannot generate fault on b2 since it is inside fault prevents area.

Note:

`faultprevent` only works in concurrent mode. Serial mode ignores the `faultprevent` compiler directives.

Design Compilation Options

VC Z01X provides an easy way to generate port faults only for all cell modules. A cell module is one which is compiled from a library directory (-y) or a library file (-v) or subsequent to the ``celldefine` compiler directive.

The `-fsim=portfaults` command-line option can be used when compiling the design to produce the same effect as using the compiler directives ``suppress_faults` and ``enable_portfaults` for all cell modules. Use `-suppress_cell` during `vc_fcc` can also generate faults in cells, except PORT faults.

Fault Selection Using System Tasks

By default, VC Z01X creates faults for the entire design. However, there are several system tasks that allow you to control the creation of faults.

See the following subsections:

- [Suggested Methodology](#)
- [\\$fs_inject\(\)](#)

Suggested Methodology

When using a testbench for stimulus, fault simulation tasks should be placed in the testbench module.

\$fs_inject()

The default VC Z01X fault injection method is at the end of the initial start of simulation. This is often referred to as time 0 in the simulation. `$fs_inject()` is an optional system task that injects faults into a simulation at a user specified time step.

When using `$fs_inject()`, faults are applied at the end the time step it was invoked.

`$fs_inject()` should only be called once. A warning will be issued on subsequent calls to `$fs_inject()`, and the call will be ignored. Note that transient faults using absolute timing will not be affected by specifying a later inject time.

Example Message:

```
Warning-[FSIM-FSII] Ignoring $fs_inject call
Call to system task $fs_inject ignored following initial $fs_inject
```

Different fault simulation results can occur when injecting at the beginning of a time step versus the end of that time step. For example, assume a signal is currently at a logic value 0 and changes to a logic value 1 during a time step. If a stuck-at 0 fault is injected on that signal at the beginning of the time step, the transition from 0 to 1 does not occur for the fault. If the fault is injected at the end of the time step, then the transition 1 to 0 will occur for the fault. The 1 to 0 transition can cause more events for the fault during that time step. Sequential UDPs and event control statements are often sensitive to these transition patterns.

You can override the default VC Z01X fault injection method or the `$fs_inject()` injection time with the following Fault Campaign Manager variables:

Delaying Injection Time

The following FCM option delays the time in simulation when faults are introduced into the circuit. Faults are normally injected at time 0. Faults will be injected at the end of the specified time step.

```
create_testcases -fault_injection_time <time>
```

Causing Fault Injection to Occur at the Beginning of the Time Step

```
-fsim=fault+async
```

VC Z01X always delay fault injection to the end of the time step. User can disable this behavior and do fault injection immediately by using this switch during runtime. It may cause Faults injected prior to initialization of always @(*) blocks. This can result in faults unexpectedly failing to propagate when a value change at a fault location occurs during the same time step as the fault injection. In this scenario, the presence of the stuck-at fault does not allow the change to take place within the faulty machine.

Note that the time of fault injection also affects testability. Toggle counting does not begin until the fault injection time, which can affect testability and fault selection.

Understanding Fault Placement

VC Z01X places faults in the design according to the type of fault given by the fault descriptor and according to the structure of the design. The placement of the fault is the "fault location" and may be different from the hierarchical location given in the fault descriptor. The following rules and examples describe the fault placement used for PORT, VARI, and WIRE stuck-at faults created by VC Z01X.

Generating PORT Faults

PORT faults are created on the boundaries of cells and are the most commonly used type of fault location. VC Z01X uses the following rules when creating PORT faults and setting the fault location where the fault will be injected into the design:

- PORT faults described on an output which have a single driver will have the fault placed on the single driver, regardless of the number of levels of hierarchy between the port and the driver.
- PORT faults described on an output which have multiple drivers will have the fault placed on the multiply-driven net.
- PORT faults described on an input will have the fault placed on the local and internal fanouts of the connected local wire.
- PORT faults described on an output which have multiple drivers but a single fanout will have the fault placed on the highest hierarchical output connection.
- PORT faults described will not have the fault placed on external fanouts.
- Faulting a port which is directly driven by a variable or register will block that variable or register to be updated in simulation.

These examples of PORT fault placement demonstrate how faults are created in VC Z01X. All examples assume the proper compiler directives and/or command line switches have been used to create PORT faults.

Example 1:

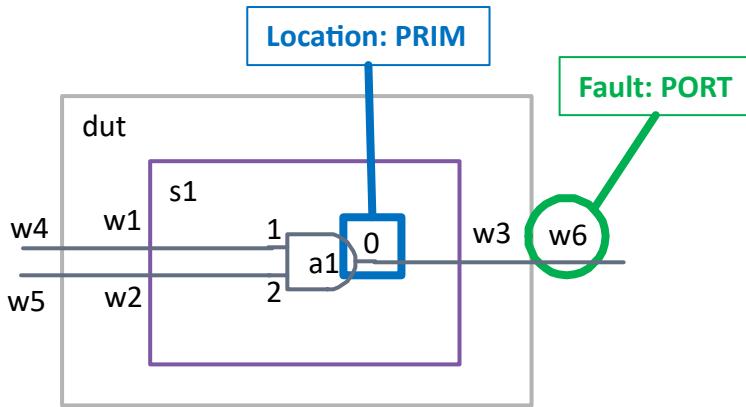
The PORT fault is placed on a port with a single driver (primitive driver) placed on output with no load internal to the cell.

- Fault Descriptor:

```
NA 0 {PORT "dut.w6"}
```

- Fault Location:

The fault is placed on the single primitive output pin driver `dut.s1.a1.0`.



Example 2:

In this example, the fault is defined on the output PORT driven by a single register driver. The result is that the fault is located on the register.

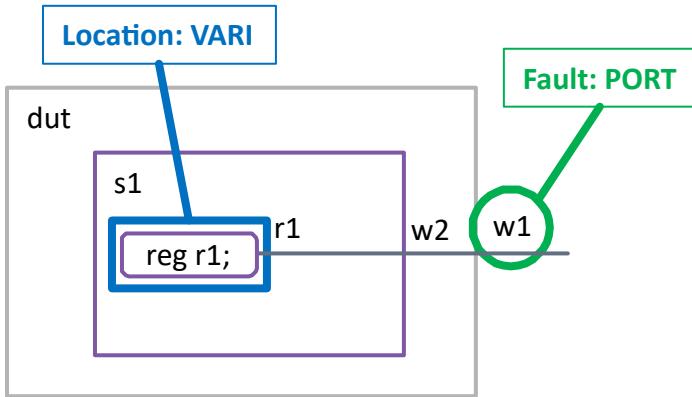
- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the register "`dut.s1.r1`".

- Register "`dut.s1.r1`" will not be updated during simulation.



Example 3:

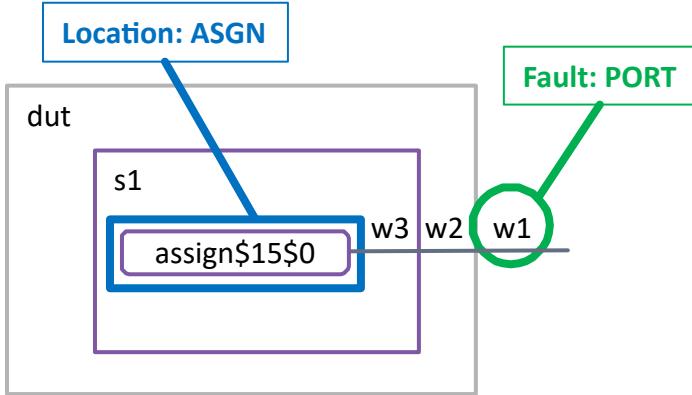
This example shows fault placement for an output fault with a single continuous assignment driving the output port. The output PORT has no load.

- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the continuous assignment "dut.s1.w3".



Example 4:

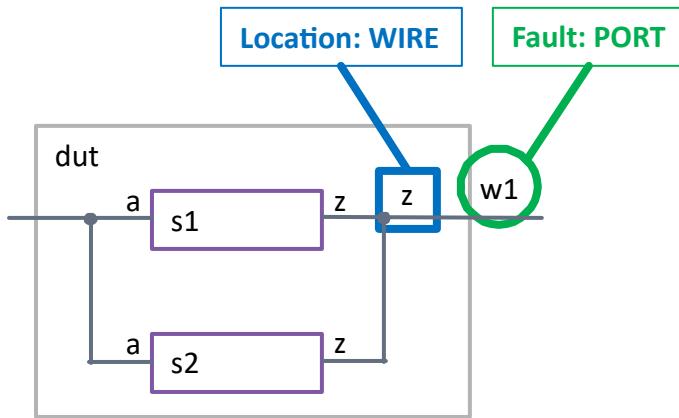
This example shows a fault on an output port with no external fanout that is internally driven by a multi-drive wire.

- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the multi-drive wire dut.z.



Example 5:

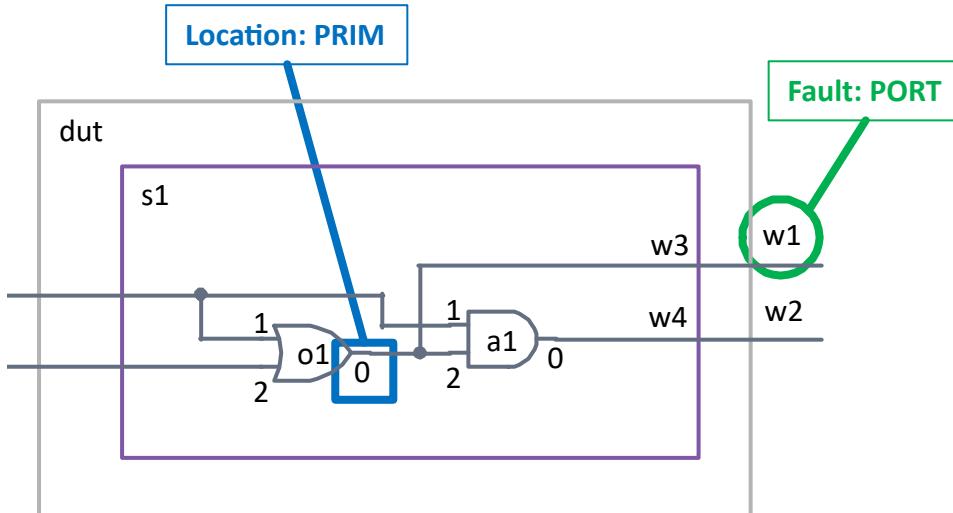
When a fault is placed on an output port with a single driver the fault will be located on the output of the driver (a primitive in this case). This holds true even when the driver is connected to a multi-fanout wire.

- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the multi-drive wire dut.s1.o1.0.



Example 6:

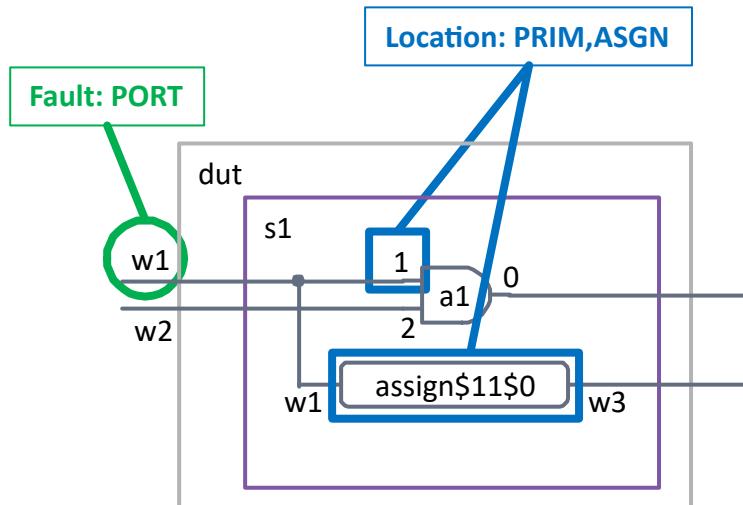
A fault on the input PORT in the example below show that the multi-fanout (internal primitive and continuous assign loads) will cause the fault to be place on the inputs of the internal primitive and continuous assignment statement.

- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the internal continuous assign dut.s1.assign\$11\$0 and primitive input pin “dut.s1.a1.1” fanouts.



Example 7:

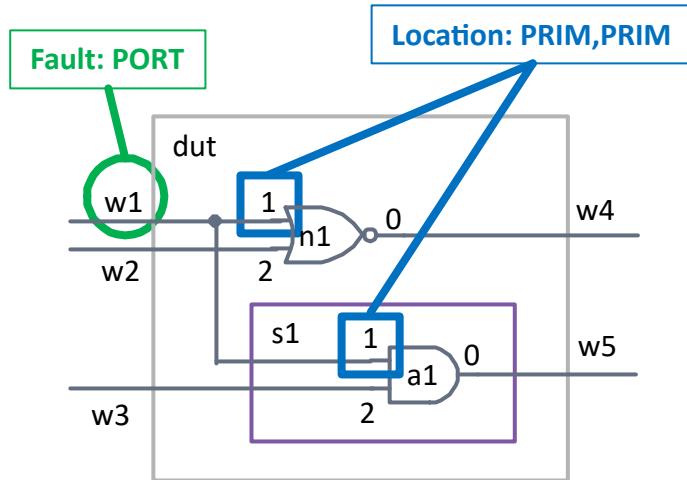
This example shows a fault placed on an input port of the cell when it is connected to a multi-fanout wire. In this case, because the wire connects internally to two primitives, the fault is placed on the input of each of these primitives. This is true even though the two primitives are represented at different levels of hierarchy within the design.

- Fault Descriptor:

```
NA 0 {PORT "dut.w1"}
```

- Fault Location:

The fault is placed on the internal primitive input pin dut.s1.a1.1 and local primitive input pin dut.n1.1 fanouts.



Example 8:

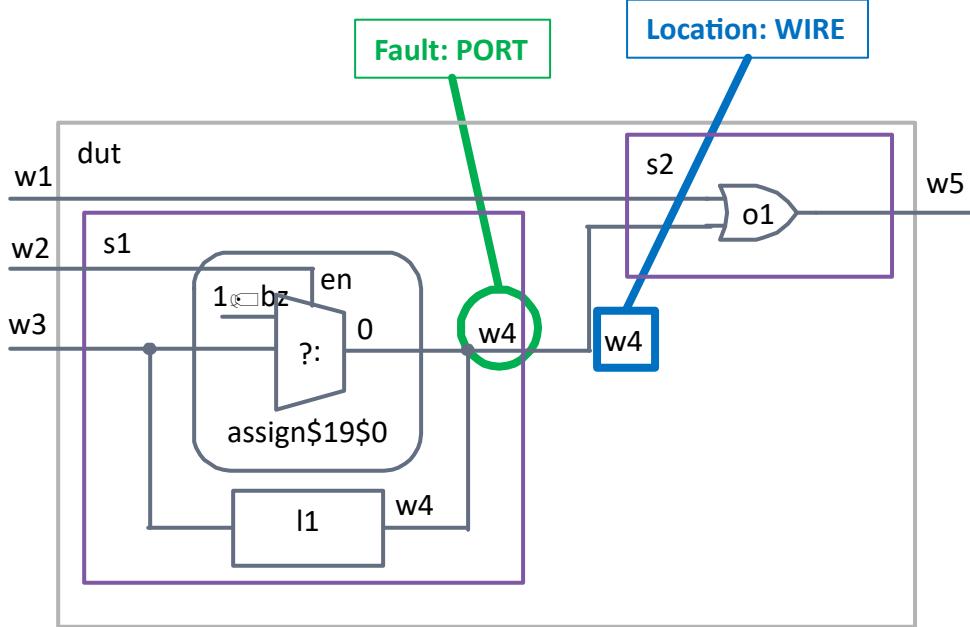
Multi drive, single load placed on output.

- Fault Descriptor:

```
NA 0 {PORT "dut.s1.w4"}
```

- Fault Location:

The fault is placed on the single fanout net dut.w4.



Example 9:

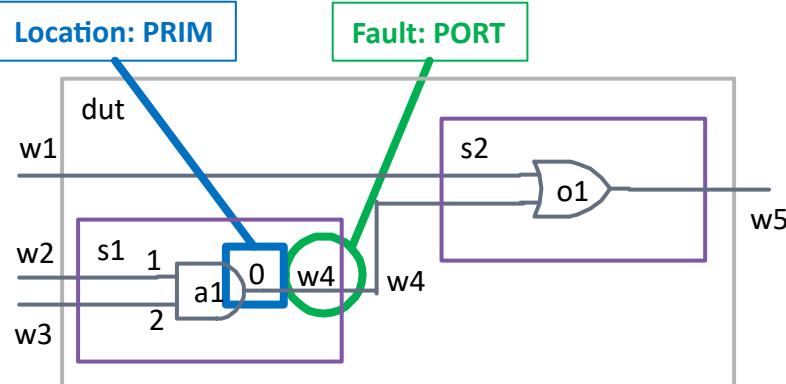
This example shows a fault placed on an output port with a single driver and single load. The resulting fault placement is on the output of the driving primitive.

- Fault Descriptor:

```
NA 0 {PORT "dut.s1.w4"}
```

- Fault Location:

The fault is placed on the output of the driving primitive dut.s1.a1.0.



Example 10:

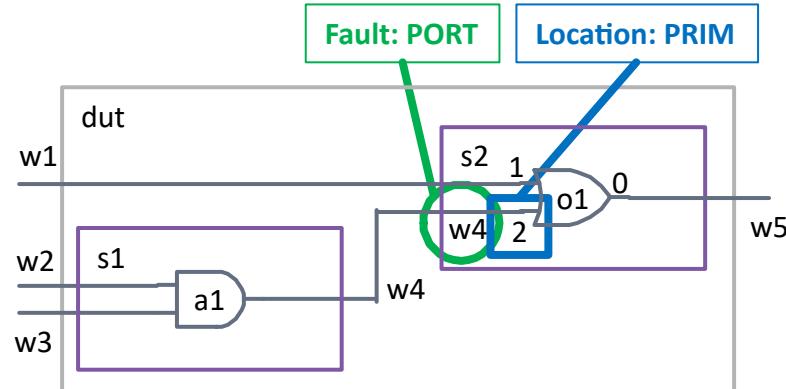
The final PORT fault example shows a fault placed on an input port with a single driver and single load. The resulting fault placement is on the input of the driven primitive.

- Fault Descriptor:

```
NA 0 {PORT "dut.s2.w4"}
```

- Fault Location:

The fault is placed on the input of the driven primitive dut.s2.o1.2.



Generating VARI Faults

VARI faults are created on RTL registers. VC Z01X uses the following rules when creating VARI faults and setting the “fault location” where the fault will be injected into the design:

- VARI faults cannot be placed on any object which isn’t a variable.
- VARI faults placed on a variable will not allow that variable to change value.

These examples of VARI fault placement demonstrate how faults are created in VC Z01X. All examples assume the proper compiler directives and/or command line switches have been used to create VARI faults.

Example 1:

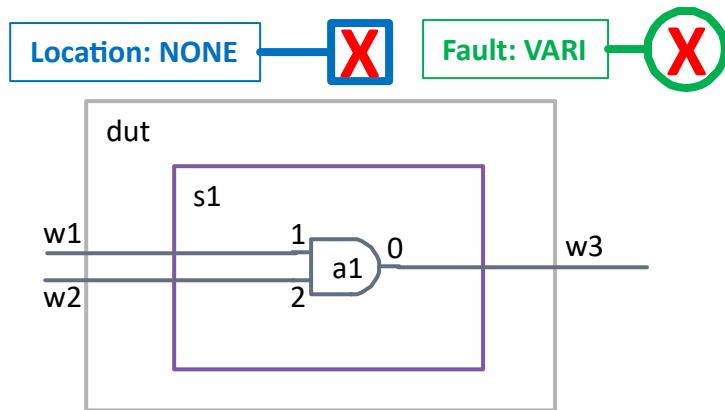
In this example, no VARI fault can be created because no register exists on which to create a VARI fault.

- Fault Descriptor:

```
NA 0 {VARI "dut.w3"}
```

- Fault Location:

No fault is created; no registers exist.



Example 2:

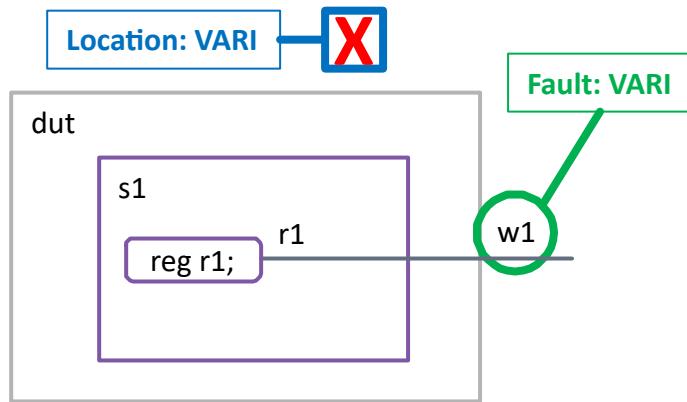
An attempt to create a VARI fault on a design object which is not a register, will not create the fault even though a register is present in the design description and is connected to the PORT where the fault is described.

- Fault Descriptor:

```
NA 0 {VARI "dut.w1"}
```

- Fault Location:

No VARI fault is generated because dut.w1 is not a register.



Example 3:

The final example for VARI faults shows that a variable is present in the design and the fault is correctly described for that register.

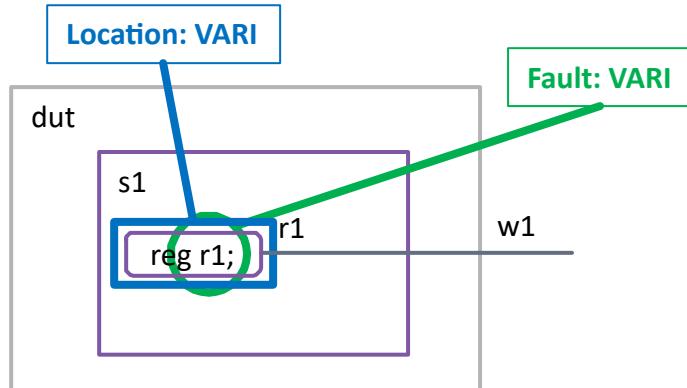
- Fault Descriptor:

```
NA 0 {VARI "dut.s1.r1"}
```

- Fault Location:

The VARI fault is placed on register dut.s1.r1.

- Register dut.s1.r1 will not be updated during simulation because the fault is placed on the register.



Generating WIRE Faults

WIRE faults are created on wires (nets) throughout the design. VC Z01X uses the following rules when creating WIRE faults and setting the “fault location” where the fault will be injected into the design:

- WIRE fault placement depends on the object (net or variable) and its drivers. Nets which are single drive originating from a primitive will have the WIRE fault placed on the driving primitive output. If a net is driven by multiple primitives or any non-primitive driver, then the WIRE fault is placed on the wire itself.
- A WIRE fault may be placed on an identical net in a different level of hierarchy in respect to where the fault was described. The fault affects the value of all connected nets through ports at every level of hierarchy but does not affect the drivers.
- Faulting the net connected to a register leaves the register free to be updated during simulation but will not affect the net value.
- Faulting a net which is named identical to a variable will leave the variable free to update during simulation. The faulted net, however, will not be free to update due to the presence of the fault even though the variable can update.

These examples of WIRE fault placement demonstrate how faults are created in VC Z01X.

Example 1:

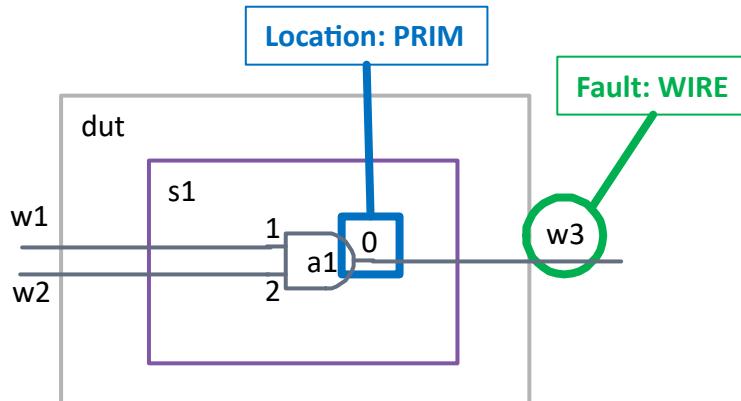
The first example shows a net or WIRE with a single driver that is a primitive.

- Fault Descriptor:

```
NA 0 {WIRE "dut.w3"}
```

- Fault Location:

The WIRE fault is placed on the output of the primitive dut.s1.a1.0.



Example 2:

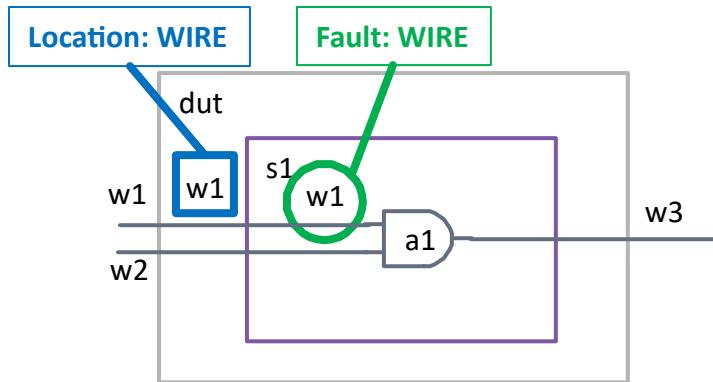
In the case of a WIRE fault with no net driver, but a single load or fanout, the fault is placed on the net at the highest hierarchical level.

- Fault Descriptor:

```
NA 0 {WIRE "dut.s1.w1"}
```

- Fault Location:

The WIRE fault is placed on the output of the primitive dut.w1.



Example 3:

The following example shows fault placement when the faults is described on a wire with a single driver that is a register.

- Fault Descriptor:

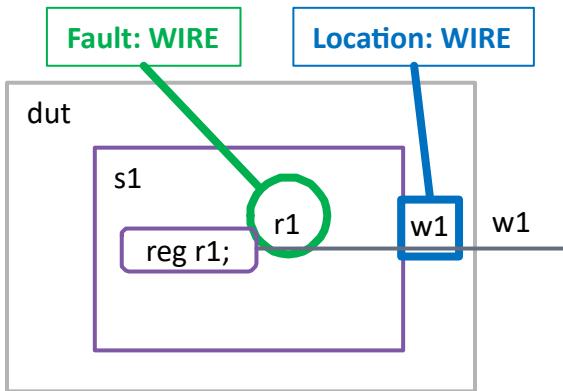
```
NA 0 {WIRE "dut.s1.r1"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w1.

Note:

The `dut.s1.r1` can change value during simulation, but the affect will not be seen for stuck-at fault location `dut.w1`.



Example 4:

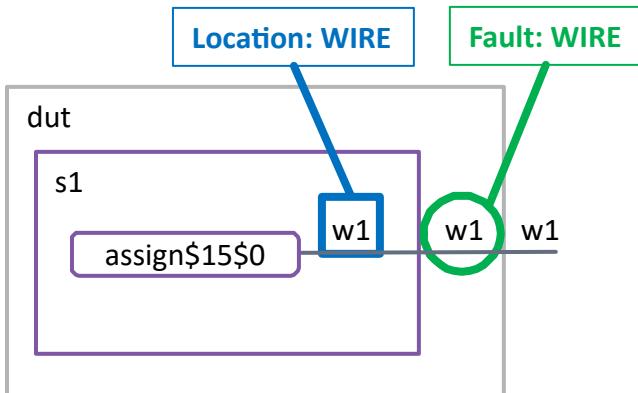
A WIRE fault on a single drive net with continuous assignment as the driver will behave the same as the earlier cases of a fault with a single driver.

- Fault Descriptor:

```
NA 0 {WIRE "dut.s1.w1"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w1.



Example 5:

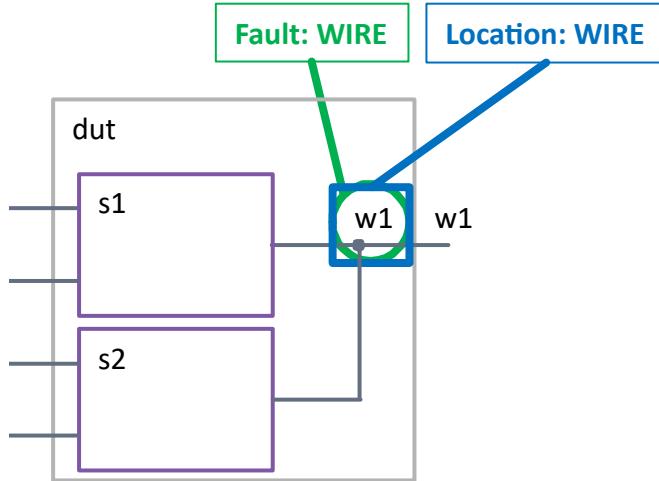
When a WIRE fault is described on a multi-drive net, the fault will be placed directly on that multi-drive net.

- Fault Descriptor:

```
NA 0 {WIRE "dut.w1"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w1.



Example 6:

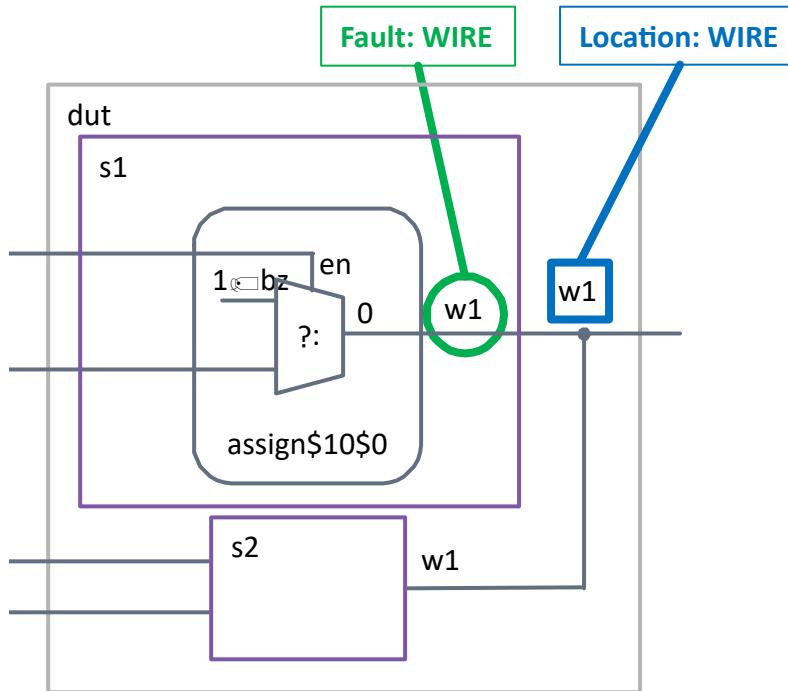
This example shows a fault described on a net with a single driver that has a load on a net which is multiply driven. The fault is placed on the highest hierarchical level.

- Fault Descriptor:

```
NA 0 {WIRE "dut.s1.w1"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w1.



Example 7:

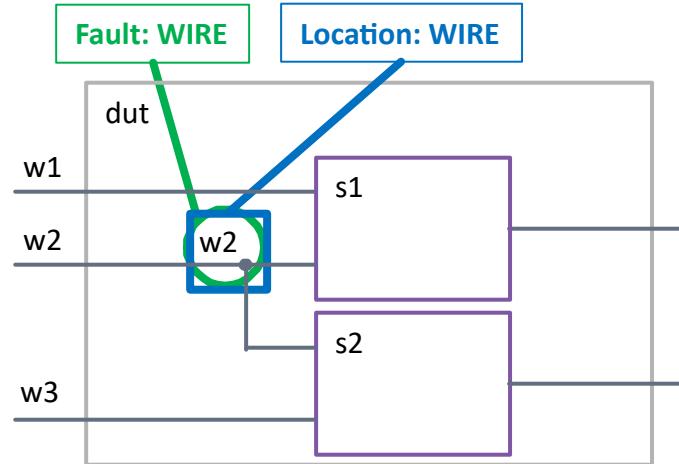
When a WIRE fault is created on a net with no net driver and multi-fanout, the fault is located directly on the WIRE.

- Fault Descriptor:

```
NA 0 {WIRE "dut.w2"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w2.



Example 8:

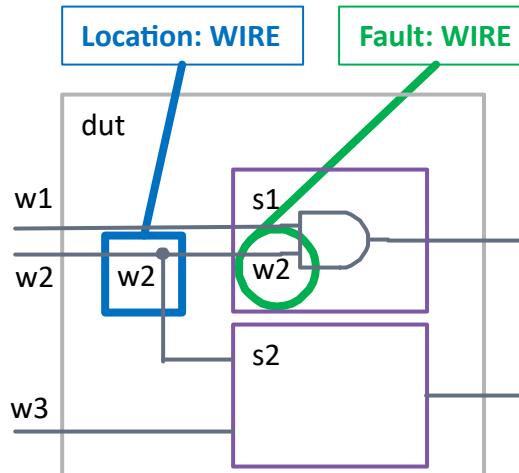
The final example shows fault placement for a WIRE fault described on a single drive net where the driving net is multi-fanout at a higher level in the design.

- Fault Descriptor:

```
NA 0 {WIRE "dut.s1.w2"}
```

- Fault Location:

The WIRE fault is placed on the net at the highest hierarchical connection dut.w2.



11

Fault Simulation

This chapter describes how to use Fault Campaign Manager to run fault simulation. Fault Campaign Manager controls and synchronizes the entire fault simulation process, from creating the fault list through obtaining fault coverage reports. Fault Campaign Manager runs testability analysis on the tests and orders them in a near optimal sequence. The steps in this chapter assume that you have set up your simulation network according to the guidelines in the [Setup](#) and [Compilation](#) chapters.

This chapter has the following sections:

- [Preparing for Fault Simulation](#)
- [Procedural Flow](#)
- [Invoking and Controlling Fault Campaign Manager](#)
- [Generating or Importing a Fault List](#)
- [Adding Tests](#)
- [Configuring and Running Testability](#)
- [Configuring Fault Simulation](#)
- [Starting Fault Simulation](#)
- [Reporting](#)
- [Behavior Analysis and Debug](#)
- [Quitting Fault Campaign Manager](#)
- [Example VC_FCM Scripts](#)
- [Sharing Results Between Failure Modes](#)

Preparing for Fault Simulation

To run fault simulation, you will provide:

- The `simv` and `simv.daidir` generated by the VC Z01X compiler. These can be the same files used for logic simulation. `simv` and `simv.daidir` are default names. Specify the `-o <filename>` compile switch to rename.
 - Stimulus file(s) containing sensing data. See the [Stimulus](#) chapter.
 - Any peripheral files, such as memory initialization files (.dat) needed by the simulation as specified in a `$readmemb` or `$readmemh` system task.
-

Procedural Flow

The following describes the steps you and Fault Campaign Manager take to optimize your test set.

1. **Compile and logic simulate the design with VC Z01X before launching Fault Campaign Manager.** This verifies that the design was correctly compiled, and also generates the database (`simv.daidir`) file and (`simv`) executable that Fault Campaign Manager requires.
2. **Create the project. The first time a FCM is started, Fault Campaign Manager initializes the fault database.** For subsequent Fault Campaign Manager sessions, the design is loaded from the `fdb`.
3. **Generate a fault list.** You perform a one-time operation to set fault creation or import options. See the [Fault Generation](#) chapter for more information on generating faults.
4. **Define the test patterns.** You will define the list of tests that will be run for a fault campaign. See [Adding Tests](#).
5. **Configure and initiate fault simulation.** You setup configuration for fault simulation. Toggle simulation and testability analysis are automatically invoked when you launch fault simulation; or you may invoke them separately for purposes of reporting and/or test selection. Fault Campaign Manager will simulate the tests you specify that have not been previously simulated. Toggle and testability analysis results are written to the Fault Database and are available as reports. You can use these results to decide the test sequence for fault simulation.
6. **Initial status and status promotion.** At the start of simulation, all faults are considered NA (Not Attempted). As simulation progresses, statuses are updated according to the status detection table. See “Detect Status Promotion”. At the end of simulation, imported faults are compared to their imported status and given their highest status in the final Coverage Report.

7. **Fault Campaign Manager runs toggle simulation.** This simulation is only performed once per test. It is not run on a test unless you select that test for fault simulation. During the toggle simulation, VC Z01X dumps an FSDB file to record the data used during testability analysis.
8. **Fault Campaign Manager performs testability analysis.** Relying on data gathered during toggle simulation, Fault Campaign Manager determines which test will most effectively detect the greatest number of faults.
9. **Fault Campaign Manager runs fault simulation on the top- ranked test.** Test analysis, test selection, fault simulation and fault database updates are automatically performed as necessary. Fault simulation results are written to the fdb. Storing results in the fdb allows simulation results for a test pair to be removed without compromising other results.
10. **Fault Campaign Manager continues the simulation/analysis cycle until selected tests are exhausted, or according to parameters you set.**
11. **View simulation results in report form.** Fault Campaign Manager offers several useful reporting formats, which are customizable to show the information that interests you most. Data integrity is critical to Fault Campaign Manager's decisions, and consistency checks ensure that data presented is up-to-date.

Invoking and Controlling Fault Campaign Manager

There are a number of basic commands used to interact with `vc_fcm`. These commands perform some of the key functions of the tool and allow you to get information about the state of the current session. They are available both from an interactive command line and from a run script. Some of the resource components that `vc_fcm` uses are configurable by the user. The following topics are discussed in this section:

- [Invoking Fault Campaign Manager](#)
- [Help](#)
- [Displaying Variable Settings](#)
- [Displaying Variable Settings](#)
- [Renaming or Disabling the Fault Campaign Manager Log](#)

Invoking Fault Campaign Manager

Fault Campaign Manager is run through the executable `vc_fcm`. The `vc_fcm` executable provides both a batch and command line interface through which you can interact with Fault Campaign Manager.

Syntax:

```
$VCS_HOME/bin/vc_fcm \
[-tcl_script <filename>]
[-connect]
```

See “[vc_fcm](#)” for information on available options.

When FCM is run interactively on the GRID, you must request a pseudo-terminal as follows:

- For SGE grid: `qsub -Ip...`
- For LSF grid: `bsub -Ip...`

Help

To obtain a list of possible `vc_fcm` commands, the tcl command line interface provides the ability to use the <tab> key to show a list of possible commands. To display the FCM specific tcl commands, use the following syntax:

Syntax:

```
vc_fcm> <tab>
```

You may also get a list of detailed command information for any command in `vc_fcm`.

Syntax:

```
< command> <tab>
```

Displaying Variable Settings

Fault Campaign Manager will display the current state of all variables in effect in a `vc_fcm` session.

Syntax:

```
get_config
```

Renaming or Disabling the Fault Campaign Manager Log

Logging is automatically enabled in `vc_fcm` when starting Fault Campaign Manager. By default, the log file is named `vc_fcm.log`. The log file is located in the `fcm.dir/latest` directory.

Syntax:

```
vc_fcm -log <name>.log
```

Generating or Importing a Fault List

Fault Campaign Manager controls fault generation by reading input files describing the fault universe to be created. The tool can read multiple formats including SFF, TestMax™ ATPG, FastScan™, and Verifault.

Fault Campaign Manager allows you to create and manipulate more than one fault list in `vc_fcm`. Using multiple fault lists provides more flexibility in your fault simulation operations but does require you to specify the fault list being used for specific Fault Campaign Manager commands.

- [Generating the Fault List](#)
 - [Creating the Project](#)
-

Generating the Fault List

As discussed in the [Fault Generation](#) chapter, the method for creating the fault list is to use the Fault Campaign Compiler (`vc_fcc`). This is the default method within `vc_fcc`. Fault Campaign Compiler writes results to the fault database, which contains all of the faults that have been defined for a given circuit, as well as their locations, collapsing data, and pre-simulation detectability status if assigned by `vc_fcc`. Generating a fault list includes:

- [Invoking the Fault Generator](#)
- [Using Percentage Sampling](#)
- [Using Confidence Interval Sampling](#)

Invoking the Fault Generator

You can change the default behavior with the following settings that control the fault generator. The `create_campaign` command is used to trigger FCC to create the fault list.

Syntax:

```
create_campaign -args "<vc_fcc_arguments>"
```

Using Percentage Sampling

FCC can limit the size of the fault universe by sampling a percentage of faults from the total possible faults. Sampling may also be specified directly in the SFF file.

Syntax:

```
create_campaign -args "-sample percent:<percent>"
```

Using Confidence Interval Sampling

Confidence Interval Sampling uses standard statistical sampling to randomly select faults, which reduces simulation time substantially while providing a statistically valid representation of true fault coverage.

Syntax:

```
create_campaign -args "-sample ci:<float>,cl:<integer>"
```

Where:

- : cl:<float>: The interval + or - the sample fault coverage (mean), also called margin of error. For example, if a sample fault coverage was 50% and the confidence interval was 10%, the interval would be [50-10, 50+10] = [40, 60]%. Possible values: An integer or floating-point value greater than 0.0 and less than or equal to 100.0.
- ci:<integer>: A percentage that shows how likely a sample fault coverage will fall within the confidence interval. For example, if a confidence level is 99% and the confidence interval is [45, 55], then the sample fault coverage will fall within the interval [45,55] (confidence interval) 99% (confidence level) of the time. Possible values: 70, 75, 80, 85, 90, 92, 95, 96, 98, and 99, usually 95 or 99.

Fault Campaign Manager does not need to generate many faults to represent the true fault coverage within a specified confidence level and interval. When you specify a confidence interval and a confidence level, Fault Campaign Manager uses these two parameters to randomly generate faults. Given a normal distribution (which can be assumed with sufficient sample sizes), general practice says $n > 30$ is good enough to make this method valid.

The distribution of the sample size must be random for this to work. In other words, there is no guarantee every array or every memory address is sampled, but if you repeat the sample size with the same confidence interval and level. The resulting faults will correctly represent the fault list.

About Confidence Interval and Confidence Level

In statistics, a confidence interval is a type of interval estimate of a population parameter and is used to indicate the reliability of an estimate. Confidence level is a measure of the reliability of a result. A confidence level of 95 per cent, or 0.95, means that there is a probability of at least 95 percent that the result is reliable.

Confidence Interval Calculation

VC Z01X uses standard statistical formulas to determine the required sample size.

$$\text{new sample size} = \frac{\text{sample size}}{1 + \frac{(\text{sample size} - 1)}{\text{population}}}$$

where:

$$\text{sample size} = \frac{Z^2 * p*(1-p)}{C^2}$$

population= total number of faults

C = confidence interval expressed as a decimal

p = coverage estimate expressed as a decimal. By default, the value is 0.5, if Coverage Estimate is not provided

Z = standard Z score based on confidence level

Note:

For more information on sampling, see “Specifying Sampling Methodologies”.

About Z Score

Z score is a statistical measurement of a score's relationship to the mean in a group of scores. A Z-score of 0 means the score is the same as the mean. A Z-score can also be positive or negative, indicating whether it is above or below the mean and by how many standard deviations.

Table 13 Confidence Level and Z Score

Confidence Level	Z Score
70%	1.04
75%	1.15
80%	1.28
85%	1.4
90%	1.645
92%	1.75
95%	1.96
96%	2.05
98%	2.33
99%	2.58

Example 1

Assume a design can generate a maximum of 10,000 array faults from many different arrays. If you specify you would like to be 99% confident that the fault coverage falls into an interval of 1%, how many faults would Fault Campaign Manager need to generate to ensure this confidence level?

Using a standard Z-score in the calculation, the answer is 6,247 faults. This means 6,247 faults will represent the 10,000 faults. If the detect rate is 88%, then you may say with 99% confidence the true fault coverage is between 87% and 89%. Thus, Fault Campaign Manager only must simulate approximately 62% of the faults to achieve the specified confidence level.

$$\text{new sample size} = \frac{16,641}{1 + \left(\frac{(16,641 - 1)}{10,000} \right)} = 6247$$

$$\text{sample size} = \frac{2.58^2 \times 0.25}{0.01^2} = 16,641$$

$$\text{population} = 10,000$$

$$C = 0.01$$

$$Z = 2.58$$

Example 2

Given a design of 100,000 faults, a confidence interval of 5%, and a confidence level of 99%, one would only need 661 faults to represent the population. This means we only need 0.661% of all the faults to be 99% confident the answer will be between within the specified interval. If the fault coverage percentage is 90%, we can say with 99% confidence the true fault coverage is between 85% and 95%.

$$\text{new sample size} = \frac{665}{1 + \left(\frac{(665 - 1)}{100,000} \right)} = 661$$

$$\text{sample size} = \frac{2.58^2 \times 0.25}{0.05^2} = 665$$

$$\text{population} = 100,000$$

$$C = 0.05$$

$$Z = 2.58$$

Example 3

Given the same design of 100,000 faults, a confidence interval of 0.5% and a confidence level of 99% would need only 39,963 faults to represent the design. In other words, approximately 40% of the faults selected at random can tell with 99% confidence that the measured fault detect percentage is between the interval of the true value + or - 0.5%. This means that if the true fault coverage percentage is 98%, the fault coverage will fall into the interval (97.5% - 98.5%) 99% of the time.

$$\text{new sample size} = \frac{66,564}{1 + \left(\frac{(66,564 - 1)}{100,000} \right)} = 39,963$$

$$\text{sample size} = \frac{2.58^2 \times 0.25}{0.005^2} = 66,564$$

$$\text{population} = 100,000$$

$$C = 0.005$$

$$Z = 2.58$$

When the design has more faults, this method becomes more powerful because you can be very confident and keep the interval low, and the calculated sample size is still very low. Example 3 only selects 40% of the faults but is extremely confident that the true value lies in a very small interval.

Using Systematic Sampling Selection Method

Systematic sampling is a sampling technique that populates a sample via 'n' selections at a fixed interval. This interval is usually calculated to be population_size / n. (This differs from simple random sampling, which creates a sample by selecting 'n' elements at random from the population.)

Systematic sampling is the default sampling method. You may disable systematic sampling with the following command:

Syntax:

```
create_campaign -args "-no_systematic_sampling"
```

Note:

Systematic sampling is made default in FCC. It could be turned off by providing the no_systematic_sampling switch.

Example:

Population -

2, 5, 1, 3, 7, 8, 4, 6, 9

- **Simple Random Sampling-** Assume a sample of size n=3 is desired. An iteration of simple random sampling may produce the following sample:

Random selection #1 - 4

Random selection #2 - 9

Random selection #3 - 1

2, 5, 1, 3, 7, 8, 4, 6, 9

Final Sample: {1, 4, 9}

- **Systematic Sampling -** The systematic sampling process proceeds in a more ordered fashion. The sampling interval is calculated to be $9 / 3 = 3$. The following assumes a starting position at the first element in the population:

Systematic selection #1 - 1

Systematic selection #2 - 8

Systematic selection #3 - 9

2, 5, 1, 3, 7, 8, 4, 6, 9

Final Sample: {1, 8, 9}

VC Z01X systematic sampling is initialized using a common statistical approach to systematic sampling. A randomized position in the fault list is selected as the starting point for sampling. The sampling interval is calculated according to the total number of faults to include in the sample and the set of all possible faults.

`sampling_interval = possible_fault_set_size / fault_sample_size`

VC Z01X adds a touch of randomness to the interval based systematic selection process. Instead of selecting a fault for every nth element (where n is the calculated interval), VC Z01X randomly selects a single fault within that interval.

Creating the Project

FCM uses a project to run a fault campaign on a design. You may create the project with the following command. If `vc_fcm` is started with the `-connect` option a project named `default` is automatically created.

Syntax:

```
create_project -fdb_project <string>
```

Adding Tests

You must provide information about each test to be simulated. A test consists of a stimulus file, which may be a testbench, eVCD, or FSDB, as well as additional support files and runtime options.

This section describes the following tasks:

- [Defining a Test](#)
- [General Stimulus Options](#)
- [Using a Testbench Stimulus](#)
- [Using Scripts](#)
- [Listing a Test](#)
- [Deleting a Test](#)

Defining a Test

To include a test in simulation, you must place it under Fault Campaign Manager's control.

When you add a test to a design, you are creating a test specification. The test specification may contain a name and an input stimulus file. You can associate additional files with a test; for example, files used by behavioral elements of the design. These files are copied into the test directory and are made available to the toggle and fault simulations. You can also add test specific runtime command line arguments to be used with toggle simulation, fault simulation, or all processes related to the test.

Syntax:

```
create_testcases
-args "<string>"
[-campaign | -fc] <string>
-daidir <directory>
-desc "<string>"
-exec <filename>
[-fault_injection_time | -fit] <string>
-fit "<string>"
-fsdb <filename>
-fsim_args "<string>"
-max_faults <integer>
-name <string>
-path <directory>
-post_script <filename>
-pre_script <filename>
-stim=type:<string>
-timeout <integer>
```

```
-tsim_args "<string>"  
-user_attr <list>
```

Note:

For more details on `create_testcases`, see [create_testcases](#).

General Stimulus Options

Enable Verification

Enable automatic verification of correct logic or good machine simulation. Automatic verification can be performed on eVCD and FSDB stimulus types.

Syntax:

```
create_testcases -tsim_args "-stim=verify"
```

If the stimulus data contains no strobes during verification:

- When the stimulus file or the testbench contains a strobe call or strobe system task, Fault Campaign Manager prints a warning and continues with the simulation.

Example:

```
Warning! Test t1 contains no stimulus data to compare with strobes
```

- When there are no strobes in the stimulus file and no strobe calls, Fault Campaign Manager prints an error and removes that test from fault simulation.

Example:

```
Error: Test t1 contains no strobes
```

Define Maximum Number of Mismatches

Define the maximum number of mismatches between the expected and simulated values before the simulation is terminated.

Syntax:

```
create_testcases -tsim_args "-stim=verify_mismatch_limit:<integer>"
```

Using a Testbench Stimulus

Using \$readmemb/h and \$test\$plusargs to Provide Stimulus

1. Design your testbench using `$readmem` and `$test$plusargs` to provide stimulus. Stimulus must be applied within the design using one of the following:

- `$value$plusargs`
- `$test$plusargs`

Example:

```
reg [15:0] stimulus_memory [15:0];
initial
begin
    if ($test$plusargs("test1")) //Start test1
        $readmemb("test1.dat", stimulus_memory);
    if ($test$plusargs("test2")) //Start test2
        $readmemb("test2.dat", stimulus_memory);
end // initial block
```

2. Invoke Fault Campaign Manager with a script designed for using a test bench for stimulus:

```
vc_fcm -tcl_script fcm.tcl
```

Script for testbench stimulus:

```
create_testcases -name test1 -args "+test1"
create_testcases -name test2 -args "+test2"
fsim
report -report=coverage.txt
```

Using \$readmemb/h to Provide Stimulus

1. Design your testbench using `$readmemb/h` to provide stimulus.

Example:

```
reg [15:0] stimulus_memory [15:0]; initial begin $readmemb/
h("stimulus.dat", stimulus_memory); end // initial block
```

2. Invoke Fault Campaign Manager with a script designed for using a test bench for stimulus:

```
vc_fcm -tcl_script fcm.tcl
```

Script for testbench stimulus:

```
create_testcases -name test1 -args "+test1" -pre_script "cp
test1/test.dat ./stimulus.dat" create_testcases -name test2 -args
```

```
+test2" -pre_script "cptest2/test.dat ./stimulus.dat"fsimreport
-report=coverage.txt
```

Using Scripts

You can include scripts for execution both before and after fault and toggle simulation. Arguments may be placed after the script name as if running the script directly from the command line. This allows you to manually manage files if needed. Note that you may also place shell commands in the `-pre_script` argument.

Be sure your script returns a non-zero exit status in case it encounters an error. Fault Campaign Manager interprets a non-zero exit status as a failure and terminates.

Syntax:

Before a simulation:

```
create_testcases -pre_script <filename>
```

After a simulation:

```
create_testcases -post_script <filename>
```

Listing a Test

The `show_testcases` command can be used to display a list of tests.

Syntax:

```
show_testcases
[-campaign | -fc] <string>
-coats_results
[-testcases | -tcs] <list of string>
```

Deleting a Test

Fault Campaign Manager allows you to delete tests and fault results from tests. When you remove the fault results from a test, those faults are downgraded in the master fault definitions file as well.

Syntax:

To remove results from the entire design:

```
remove_testcases
-all
[-campaign | -fc] <string>
-force
[-testcases | -tcs] <list of string>
```

Configuring and Running Testability

Fault Campaign Manager runs toggle simulation and testability automatically and it also allows you to run these steps manually. To complete the testability portion of the run, `vc_fcm` initiates a toggle simulation. Toggle simulation is only run a single time on a test unless some of the test options have been changed. During toggle simulation, VC Z01X records the faults of each signal in the `fsdb` from the start till the end of simulation. This happens at time 0 in the simulation by default, but might be delayed with `$fs_inject` (system task) or `create_testcases -fault_injection_time`.

Configuring and running testability and toggle simulation may involve these items:

- [Invoking Testability](#)
- [Importing Toggle Results](#)
- [Reporting Unselected Faults](#)

Invoking Testability

You can start the toggle simulations and testability analysis independently from running the fault simulations.

Syntax:

```
coats
-localhost
-rerun
[-testcases | -tcs] <list of string>
-verbose
-wait_for_tsim
-wait_for_tsim_timeout <integer>
```

Importing Toggle Results

Instead of running toggle simulation for testability, it is possible to import toggle simulation results from a previous `vc_fcm` run. When adding a new test, the `-fsdb` argument can be used to specify a FSDB file for import:

```
create_testcases -fsdb <filename>
```

FSDB files can be found in the test directory:

```
fcm_tsim_fsdb
```

When `vc_fcm` needs to update testability, it attempts to import the results from the specified FSDB file. If any tests fail to import FSDB results, they are reported. Then `vc_fcm` runs toggle simulation for those test as if no results were ever imported.

Note:

Please be careful when importing FSDB results. The design used to generate the results and the one that's importing them must match. Any changes to compile can invalidate an FSDB file. In addition, make sure the test setup matches between the generation and import. Failure to do so can lead to severely incorrect testability results.

Example Message:

```
Error-[FCM-FSDB-OUTDATED] Fsdb file(s) outdated
The fsdb files for testcases 'test1, test2' are older than testcase executable.
Using such fsdb files for dynamic testability analysis might lead to incorrect results.
Please remove all files starting with mentioned testcase names from fsdb directory */fcm_tsim_fsdb/<Campaign_name>
```

If required, this error can be downgraded to a warning by setting `set_config -fsdb_outdated_behavior 0` while executing `fsim`.

Reporting Unselected Faults

Fault Campaign Manager can produce a report that shows why untestable faults were not chosen for fault simulation. This report can be useful in determining where a portion of a design is blocking faults from propagating to observable outputs. It can be sorted either by fault, location, observability, or controllability. For more details, see [Unselected Fault Report](#).

Syntax:

```
report -unselected <string>
```

Configuring Fault Simulation

There are a number of variable settings that can be used to control the behavior of fault simulation. Each of these variables and their purpose are detailed in the following section. The variables available to configure the fault simulation run include:

- [Controlling Basic Fault Simulation Options](#)
- [Configuring Distributed Fault Simulation](#)
- [Setting Fault Detection Options](#)
- [Controlling Oscillation Detection](#)

- [Controlling Hyperfault Detection](#)
 - [Controlling Assertion Checking and Illegal Verilog Faults'](#)
-

Controlling Basic Fault Simulation Options

You can explicitly set fault simulation command line arguments and how `vc_fcm` will deal with a simulation that fails for various reasons. Basic fault simulation options are:

- [Setting Fault Simulation Command-Line Options](#)
- [Enabling Simulation Messages](#)

Setting Fault Simulation Command-Line Options

Fault simulation may require additional command line arguments when it is run. This is the same as the command line arguments required for toggle simulation. You can share the command line arguments between the two processes.

Syntax:

```
create_testcases -args "<string>"
```

Alternatively, you can set command line arguments that will apply only to the fault simulation.

Syntax:

```
set_config -fsim_std_args "<string>"
```

or

```
create_testcases -fsim_args "<string>"
```

Enabling Simulation Messages

Fault simulation does not display output from the PLI or Verilog system tasks, such as `$display`, `$monitor`, and so on. These messages are suppressed to improve performance of fault simulation. They can be enabled and will be displayed in the fault simulation logfiles.

Syntax:

```
create_testcases -fsim_args "-fsim=fault+messages"
```

By default, `-fsim=fault+messages` only displays good machine output. To display messages of a particular faulty machine, use the optional value `+<fault id>`. Where `<fault id>` is an integer value starting at 1.

Syntax:

```
create_testcases -fsim_args -fsim=fault+messages+<fault id>
```

It is possible to display all outputs of all machines at the same time using the optional value `+all`. In this mode, fault simulation will add a `<FID>` tag to the beginning of any faulty machine output that is divergent from the good machine.

Syntax:

```
create_testcases -fsim_args -fsim=fault+messages+all
```

Example Output:

```
$display 0
<FID2> $display 1
```

Controlling Simulation Retry

Fault Campaign Manager may be configured to attempt to restart failed jobs in toggle and fault simulation. Often a job that was run on a machine that became unavailable can be restarted on a new machine and will then run successfully. By default, `vc_fcm` will not attempt to restart a job before considering that job to have failed. In distributed fault simulation, if one partition failed, the remaining partitions will continue and complete and the partial results will be recorded to the `fdb`. You can enable or modify the number of retry attempts.

Syntax:

```
set_config -max_task_reruns <integer>
```

Configuring Distributed Fault Simulation

Fault Campaign Manager can cut processing time by distributing processing among multiple hosts in parallel. Fault Campaign Manager automatically divides the fault list into several jobs according to the number of parts you specify.

When you request distributed fault list simulation in Fault Campaign Manager, the required toggle simulations are distributed over the host list as well. Toggle simulations are triggered by your request for testability analysis of a test for the first time. If you select a group of tests for testability analysis or fault simulation, each test is farmed out to an available host. Separate fault simulations are run in parallel on the hosts specified, with a portion of the fault list simulated on each host. The results are combined as the simulation results for that test.

Fault Campaign Manager allows precise control over queues and simulation tool tasks. This can be useful for large designs, where you may want to carefully control distribution of tasks over available resources.

There are nine pre-defined queue types: **rsh, ssh, sh, sge, lsf, pbs, rtqa, nb, and custom**. For more flexibility in the fault simulation process, you can customize these queues. See [Viewing All Defined Host Information](#). You can use the custom queue type to specify your own batch scheduling system.

Note:

Due to the design of the SGE queue system it is possible to have inconsistent queue, cpu, and wall time statistics. The reason for this is when SGE sets the submit time, the time from the submitting host is used. When the start and end times are written to the accounting files, the time from the execution host is used. If there is a time discrepancy between the submitting and execution hosts, it is possible to show the job being started before it was submitted, or it could potentially lead to longer times being displayed.

This section includes:

- [Setting Distributed Simulation Queues](#)
- [Setting a Queue's Tool Groups](#)
- [Testing Grid Configuration](#)
- [Viewing All Defined Host Information](#)

Setting Distributed Simulation Queues

The following command is used to enable distributed simulation. The `set_submit_cmd` command is required.

Syntax:

```
set_submit_cmd
-cmd {<string>}
-disable_submit_cmd_check
-grid_type <string>
-hosts_file <string>
-max_jobs <integer>
-remote_host <string>
-submit_timeout <integer>
-task_type <string>
-worker_submit_delay <integer>
-worker_submit_size <integer>
-worker_timeout <integer>
```

Setting a Queue's Tool Groups

To set a queue's setting to run tools for a specific tool groups:

Syntax:

```
set_submit_cmd -grid_type <string> -cmd {<cmd>} -task_type <string>
```

Available `task_type` options are:

Table 14 Task Options

Tool group	Function
default	All steps
fcc	Fault Campaign Compiler (fault generation)
coats	Testability analysis
tsim	Toggle simulation
fsim	Fault simulation
report	Coverage reporting

Examples:

The following specifies an `sge` queue should run the `fsim` tool group. All other tools are run using the `lsf` queue.

```
set_submit_cmd -grid_type LSF -cmd {bsub ...}
set_submit_cmd -grid_type SGE -cmd {qsub ...} -task_type fsim
```

Testing Grid Configuration

The following command can be used to test distributed simulation.

Syntax:

```
test_grid_config
-timeout <integer>
-retries <integer>
```

Viewing All Defined Host Information

Use Fault Campaign Manager's `show` command to view all currently defined queues.

Syntax:

```
show_host_infos
```

Setting Fault Detection Options

VC Z01X allows user control of several options that control how faults are detected. These include thresholds for when to drop faults, how to handle force and release constructs, and when to inject faults.

- [Delaying Fault Injection](#)

Delaying Fault Injection

Fault simulation starts simulating fault at time 0. This time can be delayed until later in the simulation with the `create_testcases -fsim_injection_time` option.

Syntax:

```
create_testcases -fsim_injection_time <string>
```

Controlling Oscillation Detection

VC Z01X performs oscillation fault checking for zero-delay oscillations. A fault is oscillating if the injection of the fault causes the simulation to enter a zero delay loop. You can control the action VC Z01X takes when it finds an oscillation during fault simulation as described below.

- [Zero-Delay Event Threshold](#)
- [Zero-Delay Loop Iteration Threshold](#)

Zero-Delay Event Threshold

You can disable VC Z01X checking to categorize faults as causing zero-delay oscillation. The default is to consider a fault zero delay oscillating if the fault has 10 times the total number of nets in the design.

Syntax:

```
set_config -fsim_std_args "-fsim=noosc"
or
create_testcases -fsim_args "-fsim=noosc"
```

Zero-Delay Loop Iteration Threshold

By default, VC Z01X drops faults as zero delay oscillating because of a loop construct (for, while, repeat, and so on) at 1,000,000 iterations. You can adjust this limit or turn off the feature by setting the threshold to 0.

Syntax:

```
set_config -fsim_std_args "-fsim=limit+loop+<integer>"
or
create_testcases -fsim_args "-fsim=limit+loop+<integer>"
```

Controlling Hyperfault Detection

Fault Campaign Manager provides several options for controlling how the fault simulator handles hyperfaults.

Hyperactive Faults

Hyperactive faults are faults that generate many simulation events when compared with the good machine. They may be disabled, or the number of events required to classify a fault as hyperactive may be configured. The default value is 20 percent.

Syntax:

```
set_config -fsim_std_args "-fsim=limit+hyperactive+<integer>"
```

or

```
create_testcases -fsim_args -fsim=limit+hyperactive+<integer>
```

Where,

<integer> - is a percent number. That is, if the number of any FM events is greater than <integer>% of GM events, it is considered as hyperactive.

A value of 0 indicates that dropping of hyper active faults is disabled. Then no faults are dropped as hyper active.

Hyperactive dropping can also be disabled by `-fsim=fault+disable+HA+...`, which is general disable method.

Controlling Assertion Checking and Illegal Verilog Faults'

Fault Campaign Manager provides several options for controlling how the fault simulator handles assertion checking and illegal Verilog faults.

Disabling Assertion Checking and Illegal Verilog Faults

Assertion checking faults and illegal Verilog faults can be disabled. Fault simulation behavior has been defined for cases where a fault meets the criteria for fault classification but the fault status has been marked as disabled:

- **DE:** Fault simulation will continue as if the assert had not been executed.
- **DF:** Fault simulation will continue as if the system task had not been executed.
- **IA:** The resulting access in the faulty machine will return an 'x' value.
- **IF:** A call to a system function from the faulty machine will return a zero value. A call to a system task from the faulty machine will be ignored.

Syntax:

```
set_config -fsim_std_args "-fsim=fault+disable+<string>"
```

or

```
create_testcases -fsim_args "-fsim=fault+disable+<string>"
```

Managing Memory in Fault Simulation

Fault simulation is a memory-access-intensive activity, and accurately predicting the memory requirements of a fault simulation is not possible, so VC Z01X includes several features designed to manage memory use to maximize simulation performance. VC Z01X simulates the fastest when it can do as much work as possible (the maximum number of faults per pass) without causing the host system to waste time paging. The memory management features in VC Z01X work automatically, with optional guidance from the user, to maximize the number of faults per pass while avoiding paging, and to handle out-of-memory conditions in the worst case.

- [Setting the Maximum Faults Per Pass](#)
- [Setting the Minimum Faults Per Task](#)

Setting the Maximum Faults Per Pass

The `-max_fault_per_fsim_task` config setting can be used to set a maximum value for the faults simulated in a single simulation process. The default value is 2046. Ideally, you should run as many faults as possible per pass in the available amount of computer memory. This number will vary depending on the number of faults and the amount of faulty machine activity. If the value is “too high”, the simulation process will begin paging because of the amount of memory consumed by fault effects.

Syntax:

```
set_config -max_faults_per_fsim_task <integer>
```

or

```
fsim -max_faults_per_task <integer>
```

Setting the Minimum Faults Per Task

The `-min_faults_per_fsim_task` config setting can be used to set a minimum value. The default value is 2046. Ideally, you should run as many faults as possible per pass in the available amount of computer memory. This number will vary depending on the number of faults and the amount of faulty machine activity. If the value is “too high”, the simulation process will begin paging because of the amount of memory consumed by fault effects.

Syntax:

```
set_config -min_faults_per_fsim_task <integer>
```

Starting Fault Simulation

When you start fault simulation of a design for the first time Fault Campaign Manager first performs a toggle simulation (if necessary) and uses the results to analyze the testability of each test. It then performs fault simulation of the tests, records the results. Fault Campaign Manager updates the Coverage Statistics, the Simulation History, and the Test Simulation area, and usually evaluates all selected tests for ordering. By default, fault simulation and results recording proceeds for each test until all the tests are exhausted.

Syntax:

```
fsim [-failure_modes | -fms] <list of string> -fids <list of IDs>
-fsim_args <string> -localhost -max_faults_per_task <integer> -no_coats
-post_task_proc <string> -selected_status <string> -task <string>
[-testcases | -tcs] <list of string> -verbose
```

FCM gets all faults where the `Promoted status` is in the `-selected_status` enumeration.

Example:

`-selected_status {IA HA}`: will select all faults where the promoted status is IA or HA.

The FCM then executes `fault_sim` (either serial or concurrent, based on the mode set) with the tests where the status is in the `-selected_status` enumeration or the selected status from the SFF (or default).

Example:

```
3 tests, after concurrent t1=NN, t2=IA, t3 has no result stored for the
fault.
Promoted status is IA. Selected in SFF is {NA NN}.
set_config -fsim_mode serial
fsim -selected_status IA
```

This would execute a serial fault sim on all 3 tests, until a test result is not found in {IA NN NA}.

Reporting

You can generate a number of different reports and data that reflect the status of the testability analysis and fault simulations. This section describes the commands used to generate reports. See [Working with Results](#) for more information about fault reports.

The following topics are discussed in this section:

- [Fault Coverage](#)
- [Coverage Summary](#)

Fault Coverage

Fault reports provide the status of each defined fault in a fault simulation. You can report on fault coverage at any time during or after fault simulation to get the status of the faults for fault simulations that have been completed. The `report` command will allow you to generate a report based on a number of different parameters. These include faults attempted a certain number of times, faults potentially detected a certain number of times, limiting faults to particular fault status, showing certain fault models, specific reporting, and test specific reporting.

Syntax:

For test specific reporting:

```
report
-aggregatedsummary
-args <string>
[-campaign | -fc] <string>
-csv
[-failure_modes | -fms] <list of string>
-faultlimit <integer>
-faultstatus <string>
-filterbyattributes <string>
-format <string>
-gz
-hierarchical [<integer>]
-log <filename>
-overwrite
-promotedsummary
-report <filename>
-showallattributes
-showexcludedfaults
-showfaultid
-showsimdetails
-showspecifiedattributes <string>
-showtestresults <string>
-showtimingid
-sort <string>
-summaryonly
-summarystyle
-targeted <integer>
-test <list of string>
[-testcases | -tcs] <list of string>
-tool <string>
```

Coverage Summary

As an alternative to creating a full fault report, you may wish to view an abbreviated summary of the current coverage state of the design and tests in Fault Campaign Manager. You may also display this report to the screen or direct its output to a file.

```
report -summaryonly
```

Behavior Analysis and Debug

See the following subsections:

- [Dumping GM/ FM Waveforms](#)
-

Dumping GM/ FM Waveforms

VC Z01X can be used to analyze behavior or debug a single fault within a design. Use the dump command to insert faults and output the faulty machine behavior in the same way the logic simulation (good machine) is output. Results can be viewed in Synopsys' Verdi Fault Analysis tool. See[Appendix D: Using VC Z01X with Verdi](#).

Example:

```
dump-args <string>
[-failure_modes | -fms] <list of string>
-fids <list of IDs>
-fsdb <filename>
-mode <string>
[-testcases | -tcs] <list of string>
```

Quitting Fault Campaign Manager

Any of the following commands will be interpreted as a reason to end a Fault Campaign Manager session.

Syntax:

quit

or

exit

Example VC_FCM Scripts

Example 1:

This example shows how to create a test with testbench stimulus to run a fault simulation.

```
create_campaign -args "-full164 -daidir simv.daidir -campaign fc1 -sff
    user.sff -dut
    test.dut"
create_testcases -name test1 -exec "$::env(PWD)/simv" -args
    "+UVM_TESTNAME=test1"
catch {fsim}
report -campaign fc1 -report user_coverage.sff -overwrite
```

Example 2:

This example shows how to create the set of tests through eVCD stimulus to run a fault simulation.

```
create_campaign -args "-full164 -daidir simv.daidir -campaign fc1 -sff
    user.sff -duttest.dut0"
set_config -global_max_jobs 4 -tsim_std_args "-stim=verify"
set_submit_cmd -grid_type SGE -cmd {qsub -b y -cwd -V -P bnormal
    -los_bit=64,os_version='CS7.0' -o /dev/null -e /dev/null}
create_testcases -path ../evcd -exec "$::env(PWD)/simv" -args
    "-stim=file:input.evcd -stim=type:evcd -stim=inst:test.dut0
    -stim=clk:clk1+clk2" -fsim_args ""
report -campaign fc1 -report before_fsim.sff -overwrite
catch {fsim}
report -campaign fc1 -report after_fsim.sff -overwrite
```

Example 3:

FCM can also be setup to run multiple Verilog testbench patterns when the testbench stimulus can be controlled at runtime. The individual pattern stimulus is controlled through the `-args` switch when each test is added.

```
create_campaign -args "-full164 -daidir simv.daidir -campaign fc1 -sff
    user.sff -duttest.dut"
set_config -global_max_jobs 4
set_submit_cmd -grid_type SGE -cmd {qsub -b y -cwd -V -P bnormal
    -los_bit=64,os_version='CS7.0' -o /dev/null -e /dev/null}
set_config -max_faults_per_fsim_task 4000
create_testcases -name test1 -exec "$::env(PWD)/simv" -args "+test1"
create_testcases -name test2 -exec "$::env(PWD)/simv" -args "+test2"
report -campaign fc1 -report before_fsim.sff -overwrite
catch {fsim}
report -campaign fc1 -report after_fsim.sff -overwrite
```

Sharing Results Between Failure Modes

This section explains about sharing results between failure modes.

If the same fault (model, type, location) is present in multiple failure modes, the results may or may not be shared.

For example, consider fault ID 100 is present in two failure modes (and the campaign has two failure modes):

If you use `-fm_result_sharing on` at FCC, then not specifying a failure mode in the `write_fault_results` results in:

```
<fcml> write_fault_results -fids 100 -status OD -tc test1
```

Wrote 1 results to FDB.

If you did not use `-fm_result_sharing on` at FCC, then not specifying a failure mode in the `write_fault_results` results in:

```
<fcml> write_fault_results -fids 100 -status OD -tc test1
```

Wrote 2 results to FDB.

If you do not specify the failure mode, `write_fault_results` writes the results to all failure modes (or the shared one).

The sharing is more important if you run simulations. Then it only qualifies the fault once, hence saves simulations.

12

Working with Results

This chapter explains how to control and interpret the outputs of VC Z01X simulations. Procedures for simulating are described in the Logic Simulation and Fault Simulation chapters.

This chapter covers the following information:

- [Obtaining Results of Logic Simulation](#)
- [Working With Results of a Fault Simulation](#)
- [Verdi Fault Analysis](#)

Obtaining Results of Logic Simulation

See the following subsections:

- [Generating Logic Simulation Results](#)

Generating Logic Simulation Results

There are many output forms available to confirm that a logic simulation is running correctly. Some of the more common methods include:

- Use the automatic verification feature if your stimulus is eVCD or FSDB. The logic simulation will display a message indicating pass or failure of the simulation.
- Direct VC Z01X to write logic simulation results to an eVCD or FSDB file. Accomplish this using `$dumpvars`, `$dumports`, or `$fsdbDumpvars`, `$fsdbDumpfile` during compile and generate eVCD or FSDB file during logic simulation. The eVC D file that VC Z01X writes conforms to IEEE formatting standards and can be used by post-process tools.
- You can direct VC Z01X to write value changes dynamically to standard out or to a file, for example, or simply print a message indicating a particular signal's value. Usage of these system tasks conforms to Verilog standard IEEE 1364-1995. Standard output includes the time of the value change; the device's new value; and the device's hierarchical name.

```
$display()  
$monitor()  
$strobe()  
$fwrite()
```

- Use the PLI to produce C output as a result of I/O interaction between C code and VC Z01X.
- Direct the simulator to create a log file of the simulation output where results can be verified.

Working With Results of a Fault Simulation

Fault simulation reports provide all manner of data generated during the fault simulation run. In addition to the fault coverage report, you can generate reports on the summaries and history of all actions undertaken by Fault Campaign Manager, and complete fault dictionaries from the simulation.

See the following subsections:

- [Simulation Logs](#)
- [Fault Campaign Manager Statistics](#)
- [Fault Simulation Log Statistics](#)
- [Fault Coverage Report](#)
- [Hierarchical Fault Coverage Report](#)
- [Fault Dictionary](#)
- [Fault Campaign Manager Usage Statistics](#)
- [Unselected Fault Report](#)
- [Comma Separated Value \(CSV\) Fault Coverage Summary](#)
- [Working with Results: Multiple Fault List](#)
- [Limitations: Multiple Fault Lists](#)

Simulation Logs

Each time you perform fault simulation, VC Z01X creates logfiles of the simulation. Both Fault Campaign Manager and the fault simulator create log files with information regarding the tool status and detect information.

By default, Fault Campaign Manager generates `fcm.dir`. Each Fault Campaign Manager will create a unique session inside `fcm.dir`. Log information is directed to the display and saved inside each session. Toggle, coats and fsim generate individual tasks inside the FCM session and each process generates `out.log` inside their task.

For example,

Fault Campaign Manager log

```
fcm.dir/session0001_2023-02-01-12:40:01_vgtest-cos74_PID32231/vc_fcm.log
```

coats simulation log

```
fcm.dir/session0001_2023-02-01-12:40:01_vgtest-cos74_PID32231/
tasks/0xxxx/0xx/coats:test2:task1/out.log
```

Fault Campaign Manager Statistics

When you complete a task in Fault Campaign Manager, FCM will print out the progress message in the terminal. With long testcases, FCM will print out status updates for every 900s. User can change the update time interval by using the `-update_interval <integer>` switch in FCM script.

Time	Workers	Tasks		Time [s]	Mem [MB]	results from completed task(s)	from completed task(s)
		active	done				
05:07:40	-	2	-	(0%)	-	-	<START>
05:22:41	1	2	0	(0%)	0	-	<STATUS> workers: 1
		busy, 0 idle					
05:37:41	1	2	0	(0%)	0	-	<STATUS> workers: 1
		busy, 0 idle					

Fault Simulation Log Statistics

The fault simulation log files are similar to logic simulation log files but contain fault statistics at the end of each pass. You can find fault simulation log for each fault simulation tasks inside `fcm.dir/<session name>/tasks/<task name>/out.log`. An example of the fault statistics is as follows:

Fault Simulation Summary		
Faults Simulated	349	
Not Observed	Not Diagnosed	59 (16.91%)
Not Observed	Diagnosed	18 (5.16%)
Observed	Not Diagnosed	24 (6.88%)

Observed Potentially Diagnosed	3 (0.86%)
Observed Diagnosed	245 (70.20%)

Fault Coverage Report

The fault coverage report is generated through the `report` command in Fault Campaign Manager. The report command will create a report providing fault coverage statistics on the entire design. Optionally, it will also provide individual fault detail for all faults in the desired area of reporting.

You can report on fault coverage at any time during or after fault simulation to get the status of the faults for all simulations that have been completed. Results of simulations still in process will not be available until completion of the test.

For faults specified in the `report` command, there is a corresponding section in the fault report, followed by a fault model summary table. Faults can be sorted by status, hierarchy, status then hierarchy, or hierarchy then status using the `-sort <argument>` option. At the end of the report is a combined summary table of all specified fault models. Individual fault details are described in [Fault Descriptors](#).

Example 1: Examples for using the sort argument

Example of an unsorted fault list:

```
ND 0 {VARI "test.n3.i"}
PT 1{VARI "test.n2.i"}
PD 0 {VARI "test.n1.i"}
DD 0 {VARI "test.n4.i"}
DD 1{VARI "test.n1.i"}
```

This list can be sorted using `-sort status` as follows:

```
ND 0 {VARI "test.n3.i"}
DD 1{VARI "test.n1.i"}
DD 0 {VARI "test.n4.i"}
PD 0 {VARI "test.n1.i"}
PT 1{VARI "test.n2.i"}
```

This list can be sorted using `-sort hier` as follows:

```
DD 1{VARI "test.n1.i"}
PD 0 {VARI "test.n1.i"}
PT 1{VARI "test.n2.i"}
ND 0 {VARI "test.n3.i"}
DD 0 {VARI "test.n4.i"}
```

This list can be sorted using `-sort desc` as follows:

```
PD 0 {VARI "test.n1.i"}
ND 0 {VARI "test.n3.i"}
DD 0 {VARI "test.n4.i"}
```

```
DD 1 {VARI "test.n1.i"}
PT 1 {VARI "test.n2.i"}
```

Mixed fault locations are only sorted by the first location. Any additional locations are ignored in the search. The following list remains the same regardless of using the `sort` option:

```
DD ~ (0) {"test.b2.i" + "test.n3.i"}
DD ~ (0) {"test.b2.i" + "test.n4.i"}
DD ~ (0) {"test.b2.i" + "test.n1.i"}
DD ~ (0) {"test.b2.i" + "test.n2.i"}
```

In the Coverage report, you can request information about all valid faults, including prime, untestable, and collapsed faults. Requesting this information does not affect the data in the statistics section.

Sample Fault Coverage Report

```
Input fault list with collapsing:
FaultList
{
NA 0 { WIRE "test.dut1.b" }
-- 0 { WIRE "test.dut1.one.b" }
-- 0 { WIRE "test.dut2.b" }
-- 0 { WIRE "test.dut1.one.b" }
}
```

Output faultlist with -collapseoff:

```
FaultList
{
NA 0 { WIRE "test.dut1.b" }
NA 0 { WIRE "test.dut1.one.b" }
NA 0 { WIRE "test.dut2.b" }
NA 0 { WIRE "test.dut1.one.b" }
}
#-----
# Fault Coverage Summary for Default List
#
# Total
#-----
# Number of Faults: 4 100.00%
#
# Testable Faults: 4 100.00% 100.00%
# Not Attempted NC 4 100.00% 100.00%
#-----
```

The SFF coverage report contains definitions of user-defined status groups and redefined status groups. The definitions appear within the StatusDefinitions block with the following format:

```
StatusGroups{
  <two_char_name> "<group_description>" (<status1>,<status2>,...);
  ...
}
```

For example, the following report snippet shows user-defined status groups:

```
DefaultStatus (ND)
PromotionTable{
  StatusLabels (U1,U2,U3)
  [
    U1 U2 U3 ;
    U1 U2 U3 ;
    U1 U2 U3 ;
  ]
}
StatusGroups{
  ER "Error" (HM,U2,U3);
  WN "Warning" (HA,U1);
}
```

The `vc_fdb_report` reports IDDQ toggle faults as follows:

```
NA D {PORT "test.d1.i"}
NA D {WIRE "test.d1.tmp"}
NA U {PORT "test.d1.i"}
NA U {WIRE "test.d1.tmp"}
```

Where:

D => IDDQ toggle 0

U => IDDQ toggle 1

L => IDDQ pseudo stuck-at 0

H => IDDQ pseudo stuck-at 1

Test Coverage and Fault Coverage Calculations

Test Coverage refers to the fault detect percentage across the entire design, excluding faults with Untestable statuses. Fault Coverage refers to the detect percentage when Untestable faults are included. They are calculated as follows:

Test Coverage = Weighted_Total / Total

Fault Coverage = Weighted_Total / (Total + UB + UT + UU + UR + UO + UI)

Where, Weighted_Total is the total of each fault status multiplied by its weight value:

$$\text{Weighted_Total} = (\text{DD} * \text{DD_weight}) + (\text{DT} * \text{DT_weight}) + (\text{PD} * \text{PD_weight}) + (\text{PT} * \text{PT_weight}) + (\text{NA} * \text{NA_weight}) + \dots$$

Default weights:

1.0 - DD, DT, DE, DF

0.5 - PD, PT

0.0 - NA, ND, OZ, OA, HA, HM, HT, IA, IX, NC, NO, NS, NI, NT, UU, UT, UB, UR, UI, UO, EN

Collapsed faults inherit weight from their equivalent/prime faults.

Coverage Weight section in Fault Report

Coverage Calculation can be adjusted inside SFF file with user define coverage equation.

Example:

```
Coverage {
  "Diagnostic Coverage" = "DD/ (NA + DA + DN + DD) ";
}
```

Interpreting Autogenerated Gate Names

Unless auto naming is disabled, VC Z01X reports a name for each unnamed instance of a signal specified for strobing. Instance names are of the form:

```
<primitive_type>$<unnamed instance sequence number>$<primitive
sequence_number>
```

Where

primitive_type	Is a built-in primitive name (for example, and, or, buf) or user-defined primitive (listed as "UDP").
unnamed instance sequence number	The sequence number for unnamed instance. Start from integer 1. Recalculate for each Verilog file.
sequence_number	An integer 0 or greater indicating how many unnamed instances of the primitive have been encountered within the current module. In the case where the generated identifier conflicts with a user-defined identifier within the module, the sequence_number is incremented until no conflict exists.

Example

This example shows a UDP definition, which has been instantiated without an instance name. The Verilog design file is called `test.v`.

```
test.v
primitive dffudp (q,clk,d); //UDP definition
  input clk,d;
  output q;
  reg q;
  initial
    q=1'b1;
  table
    // clkd q q+
    p 0:? :0 ;
    p 1:? :1 ;
    n ?:? :- ;
    ? *:? :- ;
  endtable
endprimitive
15
moduledff (q,qb,clk,d);
  input clk,d;
  output q,qb;
  dffudp (qi,clk,d); //UDP instantiation without instance name
  buf (q,qi); // buffer without instance name
  not (qb,qi); // not gate without instance name
endmodule
```

The resulting coverage report shows the auto-generated names for each instance that is not named. The report provides a unique name for each instance. This example only generates s@0 fault on Primitives.

demo.rpt

```
FaultList {
  NA 0 {PRIM "dff.buf$2.0"} //fault 1
  -- 0 {PRIM "dff.buf$2.1"}
  NA 0 {PRIM "dff.not$3.0"}
  NA 0 {PRIM "dff.not$3.1"}
  NA 0 {PRIM "dff.udp$1.0"}
  NA 0 {PRIM "dff.udp$1.1"}
  NA 0 {PRIM "dff.udp$1.2"}
}
```

For the first fault inside `FaultList`, \$2 means the second unnamed instance inside `test.v`. .0 represents the output port of that UDP. Similarly, you can interpret the other autogenerated instance names.

Comparing auto-generated names between fault reports

Names in fault reports generated by the same design source should be constant from one simulation run to the next.

However, note that modifications to the design source (for example, change or add instance order) will change order of unnamed instance and will require you to recompile the simulator executable. The fault report resulting from the modified design source will contain autogenerated signal names that are different from those in a prior fault report.

Using vc_fcc to Merge Separate Fault Report Files

Using `vc_fcc` it is possible to merge several fault report files as if all tests were run in a single session. The following command creates fault campaign called test and save it inside FDB:

```
vc_fcc -full164 -daidir simv.daidir -sff 1.sff 2.sff -campaign test
-dut_path tb.top
```

The following generates a report named `test_report.sff` for fault campaign test:

```
vc_fdb_report -campaign test
```

You can pass any command line arguments to the fault import program and report program. See the [vc_fcc](#) and [vc_fdb_report](#) utility for a description of available command-line options.

When merging multiple fault reports using `vc_fdb_report`, the results are merged using two hierarchies: test and individual fault.

- **Test merging**
 - Each test and its fault results are preserved in the resulting merged file.
 - When multiple `FaultList` are merged, identically named tests will saved into FDB with file name.
- **Fault merging**

The merged status for a fault that is in more than one test is done according to the *Promotion Table*. If the merged status is not NA or ND, the associated test is the test that resulted in the merged status. If more than one test results in the merged status, the test with the highest coverage is the associated test. This yields a ranking of best to worst test.

When viewing the results of multiple tests merged using `vc_fcc`, the top-level summary counts may not match the detail test specific fault status counts. When a fault is detected in multiple tests, the summary reflects the fault detected in only the test with

the highest coverage in which the fault was detected, while the individual test data shows it detected in multiple tests.

Note:

Merging fault list in `vc_fcc`only supports Standard fault format. For other formats, such as fastscan, tetramax, and verifault. Convert it to the standard fault format first, and then use `vc_fcc` to merge standard fault format.

Hierarchical Fault Coverage Report

The Hierarchical Fault Coverage report can be useful to quickly identify hierarchies in the design with low coverage. It can be configured to show N levels of hierarchy.

To create a hierarchical fault coverage report, use the `l` or `report` command in Fault Campaign Manager. Set the `-hierarchical <arg>` option to the number of levels of hierarchy. By default, the number of levels is 1. Setting the number of levels to 0 reports on the entire hierarchy, including cells.

Each fault status in the `report` command has a corresponding column in the report. Each level of hierarchy is sorted according to the order of the fault status specified. If the '--' (collapsed) fault status is specified, total faults are reported, otherwise prime faults are reported. Default is all fault statuses.

The following report shows sample contents of a coverage report file, created with the following command:

```
report -campaign riscdemo -report hier_out.rpt -overwrite -hierarchical 1
-faultstatus OD,ON,NN
```

Hierarchical Fault Coverage Report

```
# Unified Fault Platform Hierarchical Report
# Version: T-2022.06-SP2
# Date: "02/09/23 13:31:14"
# User: "user1"
# Command: "vc_fdb_report -campaign riscdemo -report hier_out.rpt
#           -overwrite -hierarchical -faultstatus OD,ON,NN"
# FailureMode: Default
# Statuses: OD, ON, NN
Total OD ON NN Scope
-----
412 296 (71.84%) 14 (3.40%) 40 (9.71%) test
412 296 (71.84%) 14 (3.40%) 40 (9.71%) -risc1
```

Fault Dictionary

The VC Z01X fault dictionary file is generated by simulation in Fault Campaign Manager. It provides fault detection details, including the observation and detection points on which

faults are detected, when the fault detects occurred, and the differences in fault signatures between good and faulty machines.

To generate the fault dictionary file:

- Enable fault dictionary by enabling the creation of the fault dictionary. This setting causes the fault simulator to save the detect time and status of all faults during fault simulation.

Example:

```
set_config -fsim_std_args "-fsim=fault+dictionary"
```

or

```
create_testcases ... -fsim_args "-fsim=fault+dictionary"
```

- Find dictionary report inside corresponding `fsim` tasks folder. If design doesn't have any detects then it will not generate the dictionary file.

Sample Fault Dictionary File (default format)

```
StrobeData{
  StrobeList{"strobe1"
    Location{"$fs_drop_status, test.v : line 22"}
    Pins{
      1 "test.q3";
      2 "test.q2";
    }
  }
  StrobeList{"strobe2"
    Location{"$fs_drop_status, test.v: line 27"}
    Pins{
      1 "test.q3";
      2 "test.q2";
      3 "test.q1";
    }
  }
  FaultList{
    Timing("clock1", CycleTime 10ns)
    Timing("clock2", CycleTime 10ns,Offset 1ns)
    UseTiming("clock1")
    ND ~ (12:51, 62:101) {"test.u3.0"}
    IX ~ (0:11) {"test.u3.0"}
    ["strobe1", 62, 1 0/1]
    DD ~ (52:61) {"test.u3.0"}
    ["strobe2", 112, 1 0/1]
    DD ~ (102:111) {"test.u3.0"}
```

\$fs_set_status in Fault Dictionary

All calls to \$fs_set_status for each fault is recorded until it is dropped from simulation in the standard fault format dictionary report. They are listed in the strobes under the Fault List section. The intermediate strobes where \$fs_set_status is being called may list different statuses than what the final statuses are for each fault underneath it. This is because each strobe will record what the fault statuses were being set to when \$fs_set_status was called, not the final statuses for the faults. The same applies to \$fs_set_status_onevent.

Fault Campaign Manager Usage Statistics

Fault Campaign Manager will display usage statistics for any given fcm tasks. FCM prints out execution summary at the end of FCM log, include statistics about time spent in each task, time queued versus time running and time breakdown by partition time. Refer to the figure below for an example of the Fault Campaign Manager time usage breakdown.

Fault Campaign Manager Usage Statistics Example

Execution Summary

```
=====
=====
Task |Wall clock(hh:mm:ss)|Time in s |Peak mem in MB
|(start->end) |Wall |CPU |RSS |Virtual
|Host/Jobs | | | |
=====
===
Fault Gen ( 0%) | - | - | - | - | -
=====
===
Combined ( 78%) | 00:00:22 | 17 | 1 | 95 | 345
-----
---
Toggle Sim | - | 13 | 1 | 81 | 345
setup | vgtest-cos74 | 1 | - | - | -
Total Queued | - | 0 | - | - | -
test1 | vgtest-cos74 | 12 | 1 | 81 | 345
-----
---
Testability | - | 4 | 0 | 95 | -
test1 | vgtest-cos74 | 4 | 0 | 95 | -
=====
===
Fault Sim ( 22%) | 00:00:06 | 4 | 1 | 81 | 361
setup (FDB) | vgtest-cos74 | 1 | - | - | -
Total Queued | - | 0 | - | - | -
test1 | 1 job | 3 | 1 | 81 | 361
=====
===

```

```
Total (100%) | 00:00:28 | 21 | 2 | 95 | 361
=====
=====
```

Unselected Fault Report

Fault Campaign Manager can also produce a report that shows additional information about faults that were not chosen for fault simulation. This report can be useful in determining where a portion of a design is blocking faults from propagating to observable outputs or where faults are not controlled. See the figure below for an example of an unselected fault report.

The report contains the standard header information in SFF format. This includes the date and version of the software used to generate the report. It also contains the `TestList` section which provides an index of the tests that will be used in the report detail for individual faults. Each fault is reported with the relevant information about where it was not controlled or blocked immediately below the fault location. The blocking or controllable information is specific to the individual test listed and tests are combined when a common blocking or controllable location is identified.

```
set_config -vc_coats_std_args "-collect_reason_unselected"
report -campaign <campaign name> -unselected fault -report unselected.rpt
```

Example N C Fault:

```
NC 0 {PRIM "poly.multax_2.multbx.matrix.botrow7.U3.I0"}
# Test: 1
# Location: poly.multax_2.multbx.matrix.column7.multcell8.W3
# Test: 2
# Location: poly.multax_2.multbx.matrix.column7.multcell11.W3
# Test: 3, 6
# Location: poly.multax_2.multbx.matrix.column7.multcell14.W3
# Test: 4, 5
# Location: poly.multax_2.multbx.matrix.column7.multcell15.W3
```

This fault located at stuck-at 0 for the primitive in the first line above is not controllable. The location that causes the fault to be inactive is different for several of the tests. The location where the fault is not active (toggling) is indicated for each test or group of tests that have a common location. This is an indication that if these locations can be activated with either a new test or modification to a current test, this fault may become selected and detectable through fault simulation.

Example NO Fault:

```
NO 1 {PRIM "poly.multax_2.multbx.matrix.botrow7.U4.I0"}
# Test: 1, 2, 3, 4, 5, 6
# Location: poly.multax_2.multbx.matrix.botrow7.W2
```

The fault indicated in the first line is blocked from reaching a strobe point in all six tests. The first location in the circuit where activity stops is at the wire poly.multax_2.multbx.matrix.botrow7.W2. Further examination of the simulation waveforms is required to determine what is necessary to enable this location to propagate the activity (toggling of values) further through the circuit and enable this fault to be selected and possibly detected.

Unselected Fault Report Example

```
Date("02/15/23 08:26:08")
User("user")
Tool("REPORT")
Info(" Type: Unified Fault Platform Unselected Report")
Info("Command: fcm::report -campaign riscdemo -unselected fault -report
      unselected.rpt")
TestList {
  1 t1 0ps {F:18620 NA:12517 NC:3893 NO:2210}
  2 t2 0ps {F:18620 NA:12179 NC:4109 NO:2332}
  3 t3 0ps {F:18620 NA:11334 NC:4663 NO:2623}
  4 t4 0ps {F:18620 NA:9252 NC:5941 NO:3427}
  5 t5 0ps {F:18620 NA:9101 NC:6036 NO:3483}
  6 t6 0ps {F:18620 NA:11810 NC:4347 NO:2463}
}
FaultList{
  NC 0 {WIRE "poly.Overflow"}
  # Test: 1, 2, 3, 4, 5, 6
  # Location: poly.multax_2.multbx.matrix.botrow15.W3
  NC 1 {PRIM "poly.multax_2.multbx.matrix.botrow7.U6.I0"}
  -- 1 {PRIM "poly.multax_2.multbx.matrix.botrow7.U4.O"}
  -- 0 {PRIM "poly.multax_2.multbx.matrix.botrow7.U4.I0"}
  -- 0 {PRIM "poly.multax_2.multbx.matrix.botrow7.U4.I1"}
  # Test: 1, 2, 3, 4, 5, 6
  # Location: poly.multax_2.multbx.matrix.botrow7.W3
  NO 1 {PRIM "poly.multax_2.multbx.matrix.botrow7.U4.I0"}
  # Test: 1, 2, 3, 4, 5, 6
  # Location: poly.multax_2.multbx.matrix.botrow7.W2
  NC 0 {PRIM "poly.multax_2.multbx.matrix.botrow7.U3.I0"}
  # Test: 1
  # Location: poly.multax_2.multbx.matrix.column7.multcell18.W3
  # Test: 2
  # Location: poly.multax_2.multbx.matrix.column7.multcell11.W3
  # Test: 3, 6
  # Location: poly.multax_2.multbx.matrix.column7.multcell14.W3
  # Test: 4, 5
  # Location: poly.multax_2.multbx.matrix.column7.multcell15.W3
  NO 1 {PRIM "poly.multax_2.multbx.matrix.column15.multcell1.U5.I1"}
  # Test: 1, 2, 3, 4, 5, 6
  # Location: poly.multax_2.multbx.matrix.gnd
  NO 1 {PRIM "poly.multax_2.multbx.matrix.column15.multcell1.U4.I1"}
  # Test: 1, 2, 3, 4, 5, 6
  # Location: poly.multax_2.multbx.matrix.column15.ccarry[0]
  NC 0 {PRIM "poly.multax_2.multbx.matrix.column15.multcell1.U3.O"}
```

```
-- 0 {PRIM "poly.multax_2.multbx.matrix.column15.multcell1.U3.I1"}
# Test: 1
# Location: poly.multax_2.multbx.matrix.column15.gnd
# Test: 2, 3, 4, 5, 6
```

Unselected Report

Unselected reports can be sorted on different criteria. There are four different sorting options for unselected reports generated through `vc_fcm`: fault, location, observability, and controllability.

Sorting by Fault:

```
report -campaign <campaign name> -unselected fault -report unselected.rpt
```

Sorting by fault organizes the report by listing each unselected fault and printing the blocked location beneath them. Multiple faults can be grouped together if they all have the same blocked location. Sorting by fault is the default organization method of the unselected report if no sorting option is specified.

Example: Unselected Fault Report Sorting Example Verilog

```
module test;
reg a, b, c, f, g, j;
dut dd1 (b, c, f, g, e, i, j);
initial begin
c = 0; b = 0; j = 0; f = 0; g = 0;
#1 c = 1; b = 0; f = 1; g = 0;
#1 c = 0; b = 1; f = 0; g = 1;
#1 c = 1; b = 1; f = 1; g = 1;
#1 $fs_strobe(test.dd1.e, test.dd1.i);
end
endmodule
module dut(b, c, f, g, e, i, j);
output e, i;
input b, c, f, g, j;
buf t (a, j);
and q (d, c, b);
and p (e, d, a);
and r (h, f, g);
and s (i, a, h);
endmodule
```

Sorting by fault:

```
report -campaign <campaign name> -unselected fault -report unselected.rpt
Info(" Type: Unified Fault Platform Unselected Report")
Info(" Command: fcm::report -unselected fault -report 'unselected.rpt'
      -campaign test ")
TestList {
1 test1 {Results:28 DD:6 NC:14 NO:8}
}
```

```

FaultList {
  NC 0 {PRIM "test.dd1.p.0"}
  -- 0 {PRIM "test.dd1.p.1"}
  -- 0 {PRIM "test.dd1.p.2"}
  -- 0 {PRIM "test.dd1.q.0"}
  -- 0 {PRIM "test.dd1.q.1"}
  -- 0 {PRIM "test.dd1.q.2"}
  # Test: 1 (test1)
  # Location: test.j
  NO 1 {PRIM "test.dd1.p.1"}
  -- 1 {PRIM "test.dd1.q.0"}
  # Test: 1 (test1)
  # Location: test.dd1.d is blocked by test.dd1.a at test.v, line 19, 1
    driver to test.dd1.e
  NO 1 {PRIM "test.dd1.q.1"}
  # Test: 1 (test1)
  # Location: test.dd1.d is blocked by test.dd1.a at test.v, line 19, 1
    driver to test.dd1.e
  ...
}

```

Sorting by location:

```
report -campaign <campaign name> -unselected location -report
unselected.rpt
```

Sorting by location organizes the report by listing each blocked location followed by the fault(s) which are blocked by that location.

Unselected Fault Report Sorted by Location Example:

```

Info(" Type: Unified Fault Platform Unselected Report")
Info(" Command: fcm::report -unselected location -report 'unselected.rpt'
      -campaign test")
TestList {
  1 test1 {Results:28 DD:6 NC:14 NO:8}
}
FaultList {
  ### Location: test.j
  ## Number of Faults: 14
  # Test: 1 (test1)
  NC 0 {PRIM "test.dd1.p.0"}
  -- 0 {PRIM "test.dd1.p.1"}
  -- 0 {PRIM "test.dd1.p.2"}
  -- 0 {PRIM "test.dd1.q.0"}
  -- 0 {PRIM "test.dd1.q.1"}
  -- 0 {PRIM "test.dd1.q.2"}
  NC 0 {PRIM "test.dd1.s.0"}
  -- 0 {PRIM "test.dd1.r.0"}
  -- 0 {PRIM "test.dd1.r.1"}
  -- 0 {PRIM "test.dd1.r.2"}
  -- 0 {PRIM "test.dd1.s.1"}
  -- 0 {PRIM "test.dd1.s.2"}
  NC 0 {PRIM "test.dd1.t.0"}
}

```

```
-- 0 {PRIM "test.dd1.t.1"}
### Location: test.dd1.d is blocked by test.dd1.a at test.v, line 19, 1
  driver to test.dd1.e
## Number of Faults: 4
# Test: 1 (test1)
NO 1 {PRIM "test.dd1.p.1"}
-- 1 {PRIM "test.dd1.q.0"}
NO 1 {PRIM "test.dd1.q.1"}
NO 1 {PRIM "test.dd1.q.2"}
```

Sorting by controllability:

```
report -campaign <campaign name> -unselected controllability -report
unselected.rpt
```

Sorting by controllability will sort the faults in the report by the location that is controlling them. This means that the locations controlling the most faults will be at the top of the report. Toggling these locations will be the most likely to make the most faults selectable. However, sometimes faults are controlled by more than one location. Unselected reports only map faults to a single control location, so in that situation toggling all the control locations is necessary to make the fault selected. This requires rerunning the report after toggling a control location to see if there are other control locations for the same fault. Faults without control locations will be excluded from the report.

Example: Unselected Fault Report Sorted by Controllability

```
Info(" Type: Unified Fault Platform Controllability Sorted Unselected
Report")
Info(" Command: fcm::report -unselected controllability -report
'unselected.rpt'")
TestList {
1 test1 {Results:28 DD:6 NC:14 NO:8}
}
FaultList {
### Location: test.j
## Number of Faults: 22
# Test: 1 (test1)
NC 0 {PRIM "test.dd1.p.0"}
-- 0 {PRIM "test.dd1.p.1"}
-- 0 {PRIM "test.dd1.p.2"}
-- 0 {PRIM "test.dd1.q.0"}
-- 0 {PRIM "test.dd1.q.1"}
-- 0 {PRIM "test.dd1.q.2"}
NO 1 {PRIM "test.dd1.p.1"}
-- 1 {PRIM "test.dd1.q.0"}
NO 1 {PRIM "test.dd1.q.1"}
NO 1 {PRIM "test.dd1.q.2"}
NO 1 {PRIM "test.dd1.r.1"}
NO 1 {PRIM "test.dd1.r.2"}
NC 0 {PRIM "test.dd1.s.0"}
-- 0 {PRIM "test.dd1.r.0"}
-- 0 {PRIM "test.dd1.r.1"}
```

Sorting by observability:

```
report -campaign <campaign name> -unselected observability -report
unselected.rpt
```

The targeted report creates an SFF file based off the unselected observability sorted report with a strobe block containing the blocking locations. All the NO/NT faults from the unselected observability report are marked NA and included in the fault list. The goal is that if the targeted report is simulated it will cause more of the NO faults to become selected due to the strobe block.

Example: Unselected Fault Report Sorted by Observability

```
Info(" Type: Unified Fault Platform Observability Sorted Unselected
      Report")
Info(" Command: fcm::report -unselected observability -report
      'unselected.rpt' ")
TestList {
  1 test1 {Results:28 DD:6 NC:14 NO:8}
}
FaultList {
  ### Location: test.dd1.d is blocked by test.dd1.a at test.v, line 19, 1
  driver to test.dd1.e
  ## Number of Faults: 4
  # Test: 1 (test1)
  NO 1 {PRIM "test.dd1.p.1"}
  -- 1 {PRIM "test.dd1.q.0"}
  NO 1 {PRIM "test.dd1.q.1"}
  NO 1 {PRIM "test.dd1.q.2"}
  ### Location: test.dd1.h is blocked by test.dd1.a at test.v, line 21, 1
  driver to test.dd1.i
  ## Number of Faults: 4
  # Test: 1 (test1)
  NO 1 {PRIM "test.dd1.r.1"}
  NO 1 {PRIM "test.dd1.r.2"}
  NO 1 {PRIM "test.dd1.s.2"}
  -- 1 {PRIM "test.dd1.r.0"}
```

Comma Separated Value (CSV) Fault Coverage Summary

You can generate a simplified fault coverage report summary in Comma Separated Value (CSV) format. The CSV file can be directly imported into any tool that reads CSV.

To generate the fault coverage report summary in CSV format, use the FCM commands.

- The `report -csv` options. For example, to generate only a CSV format summary report, specify the following:

```
report -csv
report tool creates CSV directory save two CSV files into that
directory:
```

```
Directory:  
<report_name>_csv_files  
DEFAULT_faultlist.csv  
DEFAULT_summary.csv
```

The fault coverage report summary in CSV format has the following columns:

- **Category:** A high level description of the type of data contained in each line.
- **Name:** Denotes the name of the data field.
- **Label:** Contains the fault status as a two-character representation.
- **Prime Cnt:** Denotes the prime fault count for the current status or group.
- **Prime Pct:** Denotes the prime fault percentage associated with the current status, group, or coverage equation.
- **Prime Sub Pct:** Shows prime fault percentage of the fault status within the fault status group.
- **Total Cnt:** Denotes the total fault count for the current status or group.
- **Total Pct:** Denotes the total fault percentage associated with the current status, group, or coverage equation.
- **Total Sub Pct:** Shows fault percentage of the faults status within the fault status group.

The fault coverage report detail in CSV format has the following columns:

- **FID:** Lists the fault ID number.
- **Test Name:** Denotes the test name of the test data.
- **Prime:** Contains the word "yes" if the fault is prime, or the number of the FID for the corresponding prime fault.
- **Status:** Denotes the fault status as a two-character string.
- **Model:** Lists the fault model.
- **Timing:** Name of the timing cycle used as created in the SFF input.
- **Cycle Injection:** Specifies the cycle where the fault is injected.
- **Cycle End:** Shows the end cycle time for a transient hold fault.
- **Class:** Lists the fault class used.
- **Location:** Shows the hierarchical location of the fault.

Chapter 12: Working with Results
 Working With Results of a Fault Simulation

Each line in the CSV file represents a record. Each value of each record represents a field. Fields are separated by a comma (,). A pair of empty quotes ("") are written out to the CSV file if no data is required for a particular field of a particular record.

Only fault statuses/groups that have counts greater than zero are written to the CSV file, except for the default status which is always present in the CSV coverage summary file regardless of count.

The CSV Fault Report Summary is written with the merged total coverage appearing first. This coincides with the coverage summary printed at the bottom of a normal fault coverage summary report.

The CSV format and the Original Format are shown below.

Example DEFAULT_faultlist.csv:

A	B	C	D	E	F	G	H	I	J	K	L
1 FID	Test Name	Prime	Status	Model	Timing	Cycle Inje	Cycle End	Class	Location		
2 1	test1	yes	OD	0					PORT	test.risc1.alu1.accum[0]	
3 2	test1		1 OD	0					PORT	test.risc1.alureg.d1.q	
4 3	test1		1 OD	0					PORT	test.risc1.alureg.r[1]	
5 4	test1	yes	OD	1					PORT	test.risc1.alu1.accum[0]	
6 5	test1		4 OD	1					PORT	test.risc1.alureg.d1.q	
7 6	test1		4 OD	1					PORT	test.risc1.alureg.r[1]	
8 7	test1	yes	OD	0					PORT	test.risc1.alu1.accum[1]	
9 8	test1		7 OD	0					PORT	test.risc1.alureg.d2.q	
10 9	test1		7 OD	0					PORT	test.risc1.alureg.r[2]	
11 10	test1	yes	OD	1					PORT	test.risc1.alu1.accum[1]	

Example DEFAULT_summary.csv:

A	B	C	D	E	F	G	H	I
1 Category	Name	Label	Prime Cnt	Prime Pct	Prime Sub Pct	Total Cnt	Total Pct	Total Sub Pct
2 General	Number of Faults		412	100		550	100	
3 General	Untestable Faults:		40	9.71	100	40	7.27	100
4 Fault	Untestable	UU	40	9.71	100	40	9.71	100
5 General	Testable Faults:		372	90.29	100	510	92.73	100
6 Fault	Not Contr	NC	12	2.91	3.23	20	2.91	3.92
7 Fault	Not Obser	NN	44	10.68	11.83	52	10.68	10.2
8 Fault	Not Obser	ND	5	1.21	1.34	5	1.21	0.98
9 Fault	Potentiall	PN	6	1.46	1.61	8	1.46	1.57
10 Fault	Observed	ON	44	10.68	11.83	60	10.68	11.76
11 Fault	Observed	OP	3	0.73	0.81	3	0.73	0.59
12 Fault	Observed	OD	258	62.62	69.35	362	62.62	70.98
13 Group	Dangerou	DD	263	70.7		367	71.96	
14 Group	Dangerou	DN	53	14.25		71	13.92	
15 Group	Safe	SA	40	10.75		40	7.84	
16 Group	Dangerou	SU	56	15.05		72	14.12	
17 Coverage	Diagnostic Coverage			83.23%			83.79%	

Working with Results: Multiple Fault List

Named fault list grouping are included in the following types of reports:

- [Coverage Report](#)
- [Dictionary Report](#)
- [CSV Report](#)

Coverage Report

In the Coverage report, named fault lists are reported in the same order as they were defined in the input SFF file. For each named `FaultGenerate/FaultList` block defined in the input SFF file, an equivalent named `FaultList` block is generated in an output coverage report.

Each named `FaultList` block is followed by a coverage summary that includes only the faults that are part of that `FaultList` block.

The following example shows a Coverage Report that is generated by using the input SFF shown in the example. In this example, `FL_name_1` contains two prime Dropped Detected (DD) faults. In the total column, there are six DD faults reported. `FL_name_2` contains a total of three DD faults and three PD faults. The Total Fault Coverage Summary section shows the total number of faults (nine) across the two named fault lists, `FL_name_1` and `FL_name_2`. This section also shows the total faults categorized by fault type; six DD faults and three PD faults.

Example of Coverage Report

```

FaultList FL_name_1 {
  DD 0 {PRIM "tb.dut.s1.a1.0"}
  -- 0 {PRIM "tb.dut.s1.a1.2"}
  -- 0 {PRIM "tb.dut.s1.a1.1"}
  DD 0 {PRIM "tb.dut.s2.a1.0"}
  -- 0 {PRIM "tb.dut.s2.a1.2"}
  -- 0 {PRIM "tb.dut.s2.a1.1"}
}
#-----
#----- Fault Coverage Summary for "FL_name_1"
#
#                               Total
#-----#
# Number of Faults:           6 100.00%
#
# Testable Faults:            6 100.00% 100.00%
#   Dropped Detected          DD 6 100.00% 100.00%
#   Dropped Potential          PD 0 0.00% 0.00%
#   Not Detected               ND 0 0.00% 0.00%
#
# Status Groups -----

```

Chapter 12: Working with Results
 Working With Results of a Fault Simulation

```

#     Detected          DG      6 100.00%
#     Potential         PG      0  0.00%
#-----
#
FaultList FL_name_2 {
DD    0 {PRIM "tb.dut.s2.a1.0"}
--    0 {PRIM "tb.dut.s2.a1.2"}
--    0 {PRIM "tb.dut.s2.a1.1"}
PD    0 {PRIM "tb.dut.s3.a1.0"}
--    0 {PRIM "tb.dut.s3.a1.2"}
--    0 {PRIM "tb.dut.s3.a1.1"}
}
#-----
# Fault Coverage Summary for "FL_name_2"
#
#                               Total
#-----
# Number of Faults:           6 100.00%
#
# Testable Faults:           6 100.00% 100.00%
#   Dropped Detected          DD      3 50.00% 50.00%
#   Dropped Potential         PD      3 50.00% 50.00%
#   Not Detected              ND      0  0.00% 0.00%
#
# Status Groups -----
#   Detected                  DG      3 50.00%
#   Potential                 PG      3 50.00%
#
#-----
```

Dictionary Report

Dictionary reports are created in the following format:

```

StrobeData{
StrobeList{"strobe4"
Location{"$fs_strobe, ./src/strobe.sv : line 47"}
Pins{
2 " test.riscl.mem1.memory";
}
}
StrobeList{"strobe3"
Location{"$fs_strobe, ./src/strobe.sv : line 32"}
Pins{
3 " test.riscl.alu_out";
}
}
StrobeList{"strobe1"
Location{"$fs_strobe, src/decoder.v : line 45"}
Pins{
0 " test.riscl.instdec.opcode";
1 " test.riscl.instdec.not_reset";
}
```

```

}
FaultList{
["strobe3", 208500ps,
3 GM: 11000000
FM: 0....]
DD ~ (100) {FLOP "test.risc1.alureg.d1.d1.0"}
3 GM: 11000000
FM: 0....]
DD ~ (101) {FLOP "test.risc1.alureg.d1.d1.0"}
3 GM: 11000000
FM: 0....]
DD ~ (102) {FLOP "test.risc1.alureg.d1.d1.0"}
["strobe3", 216500ps,
3 GM: 01101000
FM: 1....]
DD ~ (104) {FLOP "test.risc1.alureg.d1.d1.0"}
3 GM: 01101000
FM: 1....]
DD ~ (105) {FLOP "test.risc1.alureg.d1.d1.0"}
3 GM: 01101000
FM: 1....]
DD ~ (106) {FLOP "test.risc1.alureg.d1.d1.0"}
}
}

```

CSV Report

The CSV files will include one pair of summary and faultlist files for each named fault list. The name of the fault list is pre-pended to the CSV file name.

Limitations: Multiple Fault Lists

The following limitations exist with this feature:

- **Fault simulation:** During fault simulation, all faults generated in the SFF file are simulated independent of the named fault list where they are defined.

Verdi Fault Analysis

Verdi Fault Analysis is Synopsys' graphical user interface for exploring the quality of a design's fault coverage. Verdi Fault Analysis is part of the Verdi Ultra package or may be purchased as an add-on to existing Verdi licenses. Contact Synopsys Sales for information on evaluating or purchasing Verdi Fault Analysis.

From Verdi Fault Analysis, you can:

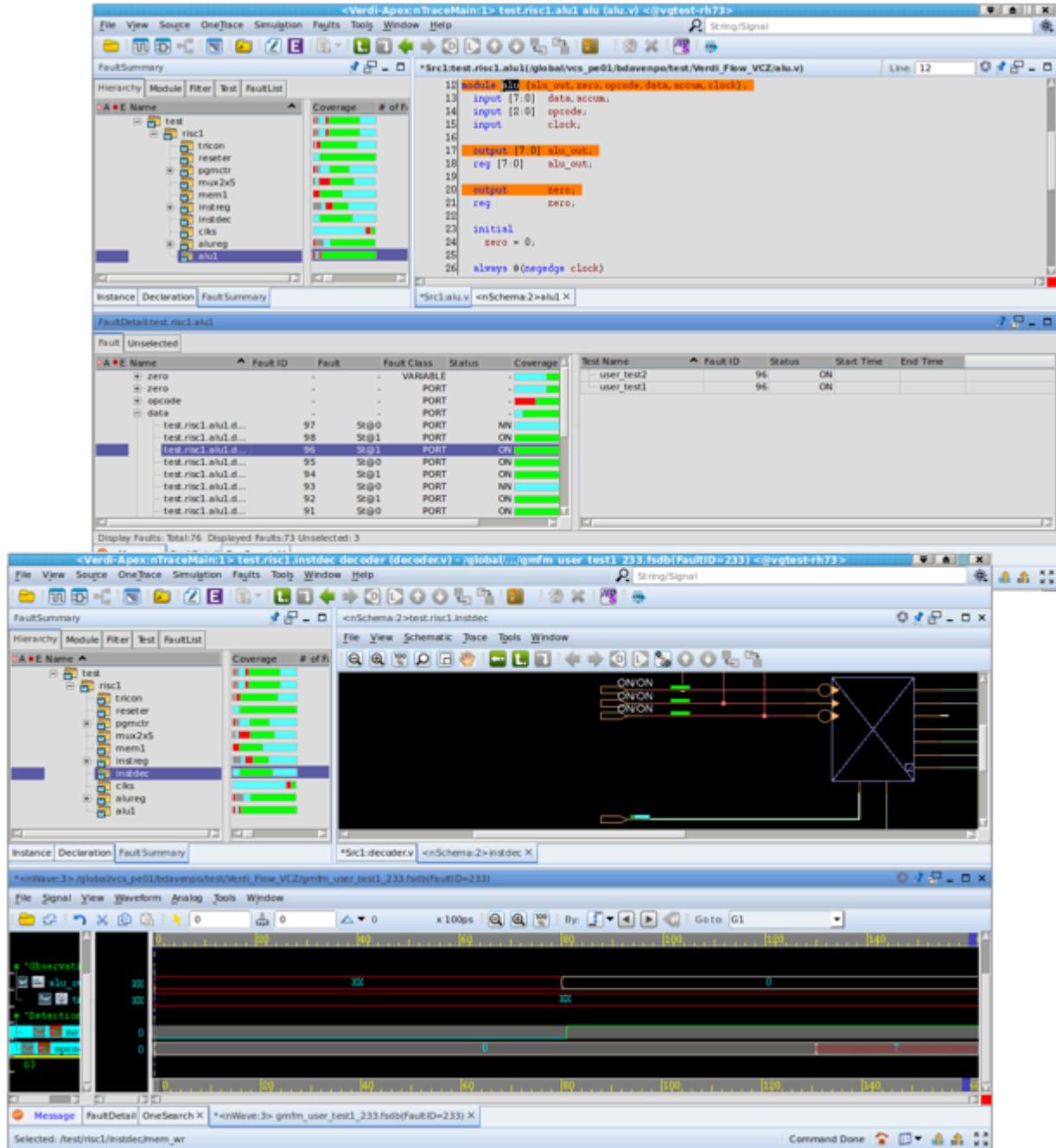
- Navigate a design in Definition or Instance view.
- Use coverage bars to quickly scan a design for areas of low coverage.

- View and edit Verilog code from the Source Pane.
- Visualize the context of undetected faults using the Schematic Pane.
- Survey signals behavior over time in a Waveform Pane.
- Compare signals for good and bad fault machines in the Waveform Pane.
- Annotate transient faults against selected signals in the Waveform Pane.

Chapter 12: Working with Results Verdi Fault Analysis

- View coverage details in the Fault Pane and Blocked Pane.

Figure 16 Verdi Fault Analysis



For more information, see *Verdi User Guide and Tutorial*.

13

Advanced Features

This chapter describes the following:

- [Defining and Reporting Custom Fault Attributes](#)
-

Defining and Reporting Custom Fault Attributes

VC Z01X provides the ability to define custom fault attributes using key, value pairs. The fault coverage is reported in the coverage report for each key, value pair. For more information, see the following sections:

- [Introduction](#)
 - [Specifying Custom Fault Attributes Using Key, Value Pairs](#)
 - [Error Messages](#)
 - [Examples](#)
-

Introduction

The following system tasks are used to define and report custom fault attributes using key, value pairs:

- `$fs_set_attribute`
- `$fs_add_attribute`

If `$fs_set_attribute` or `$fs_add_attribute` is called from simulation, the defined fault attribute is set as an active fault attribute. A new active fault attribute replaces any existing active fault attribute with a matching key. Each active fault attribute key is active for only one value at a time. By using these system tasks, you can perform the following:

- Replace or append active fault attributes.
- Delete active fault attributes based on attribute keys.
- Delete all existing active attributes.

For information on options, usages, and impact on active attributes, see the [Specifying Custom Fault Attributes Using Key, Value Pairs](#) section.

During fault detection, all available active attributes are attached to the detected fault. A fault detection is considered when any of the following occurs:

- An action causes a fault to be dropped from simulation (`$fs_drop_status`, auto-detection of PD, DD, DE, DF, HA, HT, OZ, IA, IF, IX)
- A fault status is set through a call to `$fs_set_status`

A call from the GM affects all non-diverged FMs in existence. A call from the FM only affects the current FM. The following figures illustrate that calls to `$fs_set_attribute` or `$fs_add_attribute` from both the GM and the FM are effective.

Figure 17

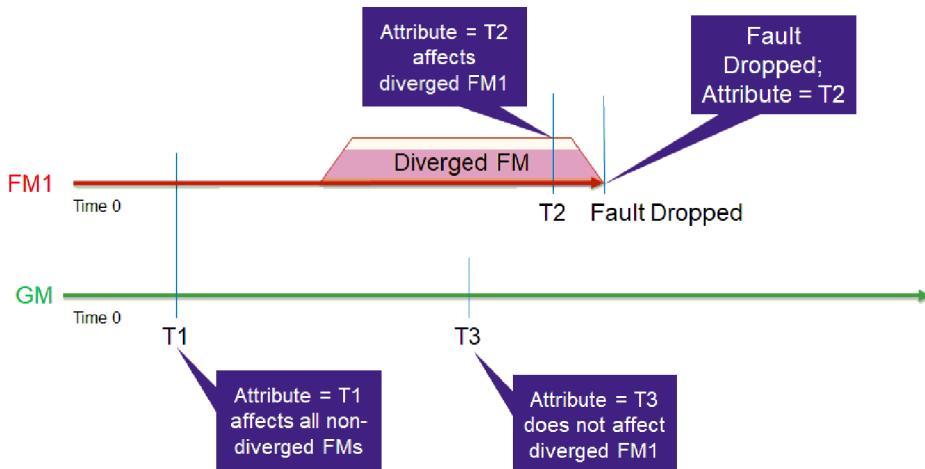
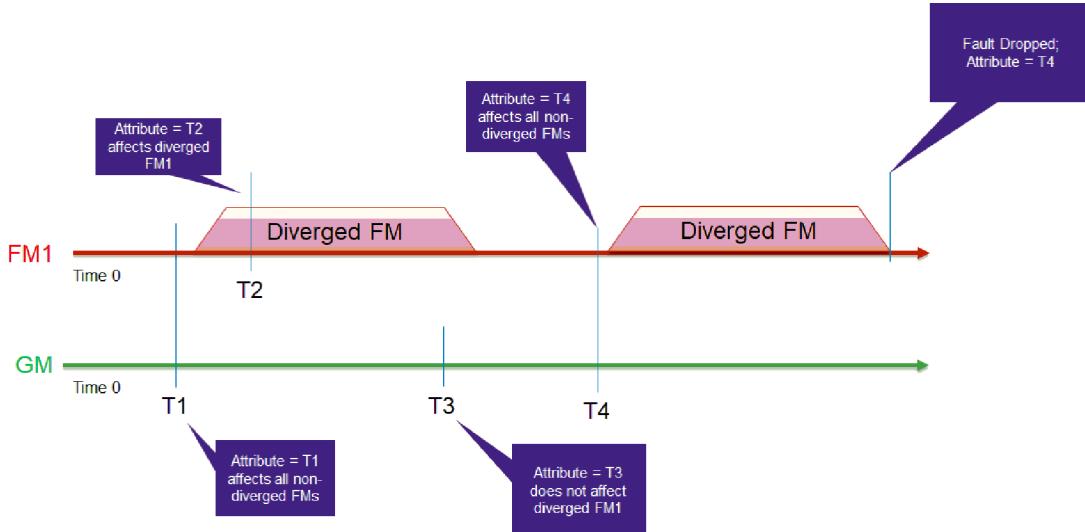


Figure 18



Specifying Custom Fault Attributes Using Key, Value Pairs

To define fault attributes, use either of the following system tasks:

- `$fs_set_attribute`
- `$fs_add_attribute`

Though these system tasks are similar in function, the key difference between the `$fs_set_attribute` and `$fs_add_attribute` system tasks is that the `$fs_set_attribute` system task replaces an existing key, value pair, while the `$fs_add_attribute` system task appends an existing key, value pair.

The following table shows the usages and the impact on active attribute.

Table 15 Usages and Impact on Active Attribute

System task	Usages	Impact on Active Attribute
<code>\$fs_set_attribute</code>	<code><verbose>, <key>, [<format>, <value>]</code>	<code>\$fs_set_attribute</code> - An active key, value attribute pair is enabled. The given key, value pair is stored to any fault detected while the key, value pair is active. This replaces any key, value already associated with the fault (not the "active" key, value).

Table 15 Usages and Impact on Active Attribute (Continued)

System task	Usages	Impact on Active Attribute
	<verbose>, <key>	The active key, value pair is deleted.
	<verbose>	All active key, value pairs is deleted.
\$fs_add_attribute	<verbose>, <key> [<format>, <value>]	An active key, value attribute pair is enabled. The given key, value pair is stored to any fault detected while the key, value pair is active. This appends the current <value> of <key>, if one already exists for that fault. System tasks are used in both the GM and FM. Note: An existing key, value is not appended a second time.
	<verbose>, <key>	The active key, value pair is deleted.
	<verbose>	All active key, value pairs is deleted.

The \$fs_set_attribute and \$fs_add_attribute system tasks use the same options, which are described as follows:

- verbose: Specifies whether verbose mode should be enabled. This is a boolean option. Specify 1 to enable verbose mode.

Note:

Verbose mode is functional as the good machine (GM) in logic simulation for \$fs_add_attribute and \$fs_set_attribute system tasks.

Verbose mode produces a message that describes the <key>, <value>, file location, and current simulation time. The message is output when called from the GM. The task calls are ignored when called during logic simulation. FM messages appear regardless of -fsim=fault+messages.

- **key:** Specifies a constant string that represents a key
- **format:** Specifies the format of the attribute value. You can specify the following format specifiers:
 - %t: Time
 - %d: Decimal (separated every 3 digits with "_")
 - %h: Unsigned hexadecimal (separated every 4 digits with "_")
 - %b: Binary (separated every 8 digits with "_")
 - %s: String

Note:

Quotes are mandatory to specify the format, without which, VC Z01X generates a compile error.

- **value:** Specifies the value for a key. The value can be either a constant string or a numeric literal. For a numeric literal, the size of the value is 64-bit. The attribute value conforms to the format specified. The tool attempts to convert to the desired format when the format is specified, but the value is in a different format. If conversion fails, the following error is reported:

Error! Invalid <format> format.

Error Messages

This section shows some of the error messages reported when the \$fs_set_attribute or \$fs_add_attribute specification is incorrect.

The following error message is reported when verbose mode is not enabled in the \$fs_set_attribute or \$fs_add_attribute specification:

Error! <verbose> argument is required.

The following error message is reported when more than four options are passed to the \$fs_set_attribute or \$fs_add_attribute system tasks:

Error! System task takes at most four arguments.

The following error message is reported when the <format> argument does not match a supported value:

Error! \$fs_set_attribute: Invalid <format> format,
Simulation time: time

The following message is reported when only 3 arguments are detected.

Error! Only 3 arguments detected. <format> and <value> arguments must be specified together.

Examples

FCM Coverage Usage

The following FCM command enables reporting of attribute key, value pairs. Key, value pair information is included in the coverage report if `-showallattributes` or `-showsspecifiedattributes` options are used.

```
report -showallattributes
```

The following command specifies the key, value pairs that must be reported:

```
report -showsspecifiedattributes <string>
```

where, `<string>` is a comma-separated list of key or key, value pairs.

Example:

```
report -showsspecifiedattributes key1+key2=value3
```

The `<string>` argument list is used to list the key, value pairs that will be reported. The `<string>` has the following characteristics:

- Use the plus sign (+) to separate the keys and key, value pairs, as shown in the following example:

```
report -showsspecifiedattributes KEY1+KEY2=VALUE1+KEY3
```

- To report an empty value key, specify the key name followed by the equal sign (=). In the following example specification, "KEY1=" denotes an empty value key:

```
report -showsspecifiedattributes KEY1=+KEY2=VALUE1+KEY3
```

- To include the subset of faults with no attribute associations, use the equal sign (=), as shown in the following specification:

```
report -showsspecifiedattributes KEY1==+KEY2=VALUE1+KEY3
```

The coverage report includes a summary of the number of faults that contain each attribute key, value pair.

Example:

```
-----
# Attribute Summary for Default List
#
# * Displaying all attributes in FaultList *
#
```

```
# Key Value Count
#-----
# key1 value1 82
# key2 value2 2
#-----
```

Coverage Report With Fault Attributes

The following example shows how to specify the `fcm` command and the shows the generated coverage report.

Example:

FCM Command:

```
report -showsspecifiedattributes key1=value1+key2=value3
```

Generated Coverage Report:

As the key, value pairs are given in the argument list, the summary section displays information for each key, value entry. In this example, the information pertains to the following key, value pairs:

- key1, value1
- key2, value3

The coverage report is as follows:

```
FaultList{
DD 0 {PRIM "test.dut.xor$30$0.1"} (* key1=value1; key2=value3; *)
ND 1 {PRIM "test.dut.xor$30$0.1"}
ND 0 {PRIM "test.dut.xor$30$0.2"}
DD 1 {PRIM "test.dut.xor$30$0.2"} (* key1=value1; key2=value3; *)
DD 0 {PRIM "test.dut.xor$30$0.0"} (* key1=value1; key2=value3; *)
ND 1 {PRIM "test.dut.xor$30$0.0"}
}
#-----
# Fault Coverage Summary for Default List
#
# Total
#
# Number of Faults: 3 100.00%
#
# Testable Faults: 3 100.00% 100.00%
# Dropped Detected DD 3 100.00% 100.00%
# Dropped Potential PD 0 0.00% 0.00%
# Not Detected ND 0 0.00% 0.00%
#
# Status Groups -----
# Detected DG 3 100.00%
# Unselected NG 0 0.00%
# Potential PG 0 0.00%
```

Chapter 13: Advanced Features
Defining and Reporting Custom Fault Attributes

```
#-----
#-----
# Fault Coverage Summary for -showattributes key1=0
#
# KEY=key1
# VALUE=0
# Total
#-----
# Number of Faults: 3 100.00%
#
# Testable Faults: 3 100.00% 100.00%
# Dropped Detected DD 3 0.00% 0.00%
# Dropped Potential PD 0 0.00% 0.00%
#-----
#-----
# Fault Coverage Summary for -showattributes key2=1
#
# KEY=key2
# VALUE=1
# Total
#-----
# Number of Faults: 3 100.00%
#
# Testable Faults: 3 100.00% 100.00%
# Dropped Detected DD 3 100.00% 100.00%
#-----
#-----
# Attribute Summary for Default List
#
# * Displaying all attributes in FaultList *
#
# Key Value Count
#-----
# key1 0 3
# key2 1 3
#-----
```

14

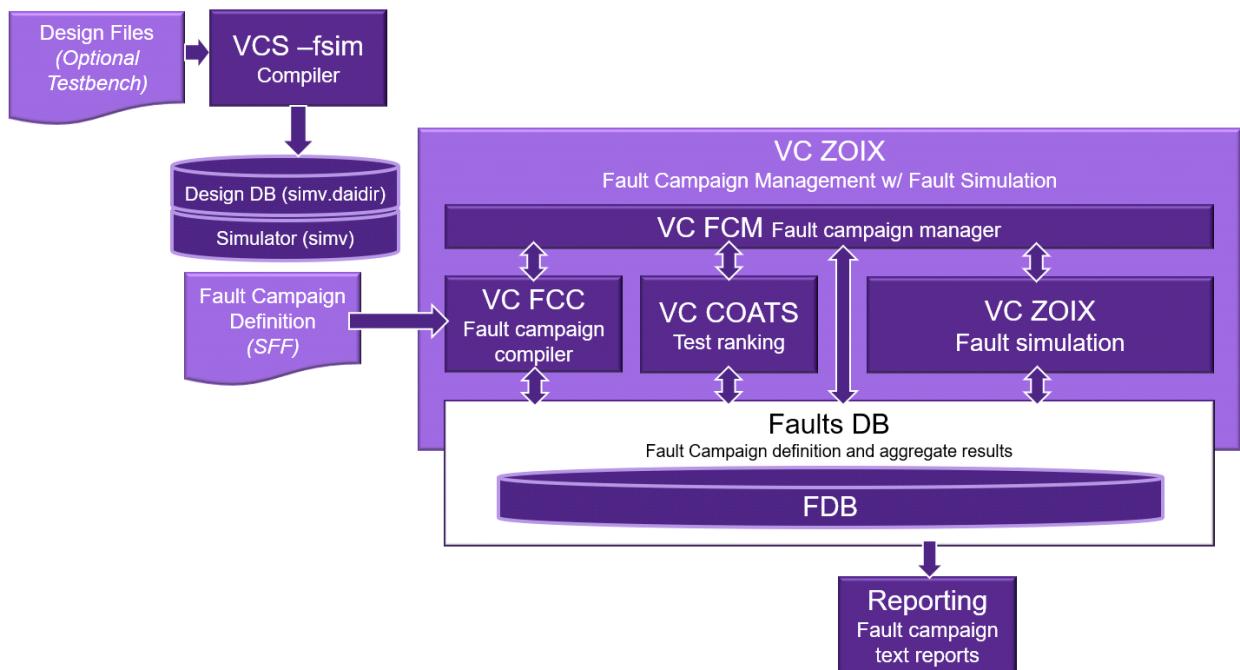
VC Z01X Methodology

VC Z01X is frequently used in functional safety applications to quantify diagnostic coverage by injecting faults into a simulated hardware design and measuring the results. This chapter outlines the safety-qualified guidelines to safely use VC Z01X for measuring your diagnostic coverage. Detailed information on command options and usage can be found in the individual chapters of this *VC Z01X User Guide*.

In all use cases, you are requested to review and abide by the Conditions of Use (CoU) and Assumptions of Use (AoU) in the *VC Z01X Functional Safety Manual*. These CoU and AoU include some basic assumptions such as reviewing all messages (Info, Warning, Error, Fatal) in the log files as well as some VC Z01X specific requirements for adhering to functional safety guidelines.

[Figure 19](#) shows an overview of the flow using the VC Z01X fault injection tool set.

Figure 19 VC Z01X Flow



Fault Campaign Manager

Fault Campaign Manager is the primary interface to the VC Z01X fault injection process. The Fault Campaign Manager (`vc_fcm`) is the executable that you will use to deliver commands to run VC Z01X either through an input script or on the interactive command line. Fault Campaign Manager follows a recommended procedural flow that is discussed in the following sections. The details of the FCM syntax are available in [Appendix A: Fault Campaign Manager Commands](#) and [Appendix B: Command Syntax](#) sections.

VC Z01X Example Flows

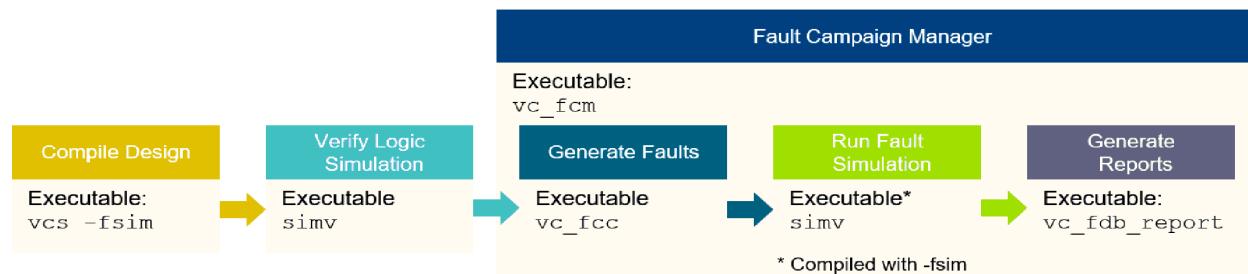
At the end of this chapter, several common flows are described. The described flows are verified reliable for use in functional safety. Use of VC Z01X in a safety environment is not limited to these flows.

- Concurrent Simulation Flow
- Using Software Test Libraries
- Resolving Hyperfaults
- Serial Fault Simulation Flow

VC Z01X Fault Injection Procedural Guidelines

The typical VC Z01X fault injection flow can be described in the following subsections:

- [Compile](#)
- [Logic Simulation](#)
- [Fault Generation](#)
- [Testability](#)
- [Fault Simulation](#)
- [Coverage](#)



Compile

The first step in using VC Z01X is to compile your design, including your netlist, libraries, optional testbench, and strobe system tasks. During the compile step you should take some steps to correctly setup your design for fault simulation. Two of the most important considerations are inserting system tasks for the purpose of categorizing fault results and defining where faults are to be placed in the design.

The following subsections discuss the considerations you must make during the compile phase of VC Z01X:

- [Strobe System Tasks](#)
- [Fault Locations](#)
- [Testbench Setup](#)
- [Timing Checks](#)

Strobe System Tasks

VC Z01X provides system tasks designed for use in functional safety fault injection simulations. You must insert the strobe system tasks into the testbench or safety mechanisms to enable observation and detection of faults during the fault simulation. These system tasks do not change the behavior of the reference simulation and report a warning message if executed in the good machine.

These User Controlled Fault Detection (UCFD) tasks include:

- `$fs_compare`: Compare the GM/FM values of the listed signals.
- `$fs_observe`: Compare the GM/FM values of the signals from the SFF *Observe* block.
- `$fs_detect`: Compare the GM/FM values of the signals from the SFF *Detect* block.
- `$fs_drop_status`: Set the status of the fault and remove it from simulation.
- `$fs_set_status`: Set the status of the fault but continue simulating it.
- `$fs_default_status`: Set the default status of faults in the simulation.
- `$fs_get_status`: Return the current status of a fault.
- `$fs_drop_status_onevent`: Set the status of a fault and remove it from simulation on any GM/FM differences.
- `$fs_set_status_onevent`: Set the status of a fault, but continue to simulate it, on any GM/FM differences.
- `$fs_disable_onevent`: Cancel monitoring of signals for `$fs_drop_status_onevent` and `$fs_set_status`.

Strobe system tasks are effective only in the FMs during fault simulation. See “User Controlled Fault Detection” for complete details of strobe system task interaction with the GM and FM in a fault simulation. Placement of these tasks must be implemented correctly to ensure fault coverage is valid. The following paragraphs will discuss considerations for inserting strobe system tasks into your design environment.

Insert the system tasks in the testbench or safety mechanisms where:

- A device failure can be observed; usually on the safety related output pins of the device.
- In a safety mechanism where the fault is detected as a failure.
- In a safety mechanism where the fault is corrected, and the simulation can continue with correct behavior.

Blocks that are activated through signal changes, such as the Verilog always procedure, are preferred for use with UCFD tasks. Placing UCFD in an initial procedure with a time delay will not have the desired effect because not all FMs can be diverged for that block.

Examples of good constructs for inserting these tasks include:

```
always @(<signal list>) blocks
always @(*) blocks
forever @(<signal list>) blocks
```

The sensitivity list or signal activation of statements such as those in the examples above cause a divergence of the block for a FM. The divergence will enable the strobe system tasks to be executed in the FM and complete the desired action, such as setting the status of the fault.

The execution of procedural blocks in simulation to control strobing system tasks (\$fs_compare, \$fs_set_status, \$fs_drop_status, and so on.) are subject to the same restrictions and considerations for any other Verilog simulation behavior, particularly race conditions. Creating two separate blocks of procedural code for observation and detections points must consider that the blocks might execute in the same time step and create a possible race condition between the two.

Example:

```
always (negedge clk)
  cmp = $fs_compare(observe_signal);
  if (cmp==1)
    $fs_set_status("ON", observe_signal);
  always (negedge clk)
    status = $fs_get_status();
    cmp = $fs_compare(detect_signal);
    if (cmp==1)
      if (status == "ON")
        $fs_drop_status("OD", detect_signal);
```

```
else
$fs_drop_status("ND", detect_signal);
```

These two always blocks might execute in either order within the simulation. The fault result is not deterministic because second always block depends on the result of the first always block. A better implementation of these strobe system tasks is to implement them in a single procedural block:

Example:

```
always (negedge clk)
cmp = $fs_compare(observe_signal);
if (cmp==1)
$fs_set_status("ON", observe_signal);
status = $fs_get_status();
cmp = $fs_compare(detect_signal);
if (cmp==1)
if (status == "ON")
$fs_drop_status("OD", detect_signal);
else
$fs_drop_status("ND", detect_signal);
```

Fault Locations

Fault locations are discussed here because there are considerations you must make prior to the compilation of your circuit that affect how faults are created for fault injection. You define the locations and types of faults to inject in the design. VC Z01X injects faults on different constructs throughout the design based on this user input. These locations provide accurate modeling of the physical defect that may occur. Fault locations are specific to different types of faults.

The following locations are recommended as possible fault injection locations:

Table 16 Possible Fault Injection Locations

Design Stage	Fault Type	Fault Model	Fault Injection Location
Register Level Transfer (RTL)	Permanent	Stuck-at	- Ports of module cells- Variables of type reg, logic, bit, and byte- Arrays (and memories) of reg, logic, bit, and byte
	Transient	Single Event Upset (SEU)	- Variables of type reg, logic, bit, and byte- Arrays (and memories) of reg, logic, bit, and byte
Gate Level	Permanent	Stuck-at	- Ports of module cells- Multiple driven nets

Table 16 Possible Fault Injection Locations (Continued)

Design Stage	Fault Type	Fault Model	Fault Injection Location
	Transient	Single Event Upset (SEU)	- Output pins of Verilog sequential UDPs - Arrays (and memories) of reg, logic, bit, and byte

Port faults and flop fault locations should be facilitated through one of the following recommended methods:

Port Fault Compiler Directives

Many standard library cells contain compiler directives to suppress internal faults and create only port faults. This is one method for defining which library cells should have faults created only on the instance ports.

```
`suppress_faults
`enable_portfaults
module lib1(...);
...
endmodule
`disable_portfaults
`nosuppress_faults
```

Command Line Switches

Some library cells do not contain the compiler directives necessary to define the locations for port faults. VC Z01X can automatically identify library cells for creating port faults through compile time command line switches.

For an RTL design, the compiler option `-fsim=portfaults` will enable faults on cell ports throughout the entire design. Synopsys recommends using this option only for RTL designs. Use the compile command line options `-fsim=suppress+cell` and `+nolibcell` to define cells in a gate level netlist as locations to create port faults.

Flop Identification

Transient faults (SEU) commonly used in functional safety applications are placed on sequential elements in the circuit (flip-flops and latches). In the SFF fault descriptor, these are both denoted with the keyword FLOP. VC Z01X can automatically locate and identify sequential elements in your design both at RTL and gate level.

No additional command line options are required to create FLOP faults for designs. These faults are created by using the FLOP keyword in your SFF input file. .

Exceptions:

- Faults inside suppressed cells Many standard library cells contain compiler directives to suppress internal faults and create only port faults.

```
`suppress_faults
`enable_portfaults
module lib1(...);
...
endmodule
`disable_portfaults
`nosuppress_faults
```

Two common areas that are affected by these compiler directives are memory and flip-flop or latch models:

- Creation of SEU or stuck-at faults on memory arrays
- Creation of FLOP faults for sequential UDPs

In both cases the compiler directives must be removed or relocated to enable the fault locations to be used. For a memory, this means the declaration of the memory array must be outside the compiler directives. When creating FLOP faults on sequential elements for SEU faults, the sequential UDP must be moved outside any compiler directives.

Testbench Setup

Verilog testbenches are commonly used with VC Z01X during fault injection. There are multiple methods for design testbench stimulus and certain types can be incorporated into the VC Z01X flow more easily than others. Testbenches that dynamically load stimulus at runtime are preferred because they do not require you to re-compile your design for each new testbench to run.

One of the most common methods for implementing this type of testbench is to provide the stimulus by reading the stimulus data into a memory which then executes the test instructions from that memory. This is a frequently used technique when simulating software test libraries and other object code to execute instructions during simulation.

Example Testbench:

```
reg [15:0] image_memory [15:0]; string filename;
initial begin
  filename = $value$plusargs(TESTNAME);
  $readmemb(filename, image_memory);
end
```

FCM script example:

```
create_campaign -args "-full164 -daidir simv.daidir -campaign fc1 -sff
  input.sff -dut test.dut"
```

```
create_testcases -name test1 -exec "$::env(PWD)/simv" -args
  "+TESTNAME=stl1.dat"
create_testcases -name test2 -exec "$::env(PWD)/simv" -args
  "+TESTNAME=stl2.dat"
fsim
report -report mycoverage.sff
```

In this example, you need to compile the design only once, but can execute any number of tests by adding `+TESTNAME=<file>.dat` on the command line. The `$value$plusargs` will read the file and then the testbench can apply the stimulus that will be loaded into `image_memory`.

There are other techniques that can be used to create this style of testbench. Most leverage

`$value$plusargs` or `$test$plusargs` to dynamically create data through the simulation command line arguments.

Timing Checks

Timing checks are used in simulation to verify correct timing of signals within module cells. VC Z01X supports timing checks in logic simulation. During fault simulation, timing checks are not relevant, because faults often cause timing violations in the faulty machine. VC Z01X requires timing checks to be disabled during fault simulation by adding `+notimingcheck` to the compile time command line.

Logic Simulation

Logic simulation is a pre-requisite to running fault simulation. Fault simulation results are based on comparing results of faults injected into the design against the golden results of the good machine which is equivalent to logic simulation. Prior to running any of the remaining steps in the VC Z01X procedural steps, you should run logic simulation to ensure your environment is complete and the simulation is producing the expected outputs.

There are numerous methods for validating logic simulation which are outlined in For further information on the syntax and options, see [Logic Simulation](#). These methods include auto-verification features of VC Z01X when using external stimulus like FSDB, and eVCD. For other types of stimulus, you must visually confirm the correct messages appear in a self checking testbench or interactive Tcl script, or create and review waveforms of the simulation with a waveform viewer.

Fault Generation

Creation and definition of the fault universe is enabled through the Standard Fault Format. Users control fault generation, detection, and coverage calculations through the SFF file.

SFF also provides the mechanisms through which you can customize some of the fault injection to your environment and terminology. The following subsections describe some of the most important aspects to consider when creating an SFF file:

- [User Defined Fault Status \(UDFS\)](#)
- [Defining UDFS Interactions and Merging Fault Lists](#)
- [Defining and Generating Faults](#)
- [Excluding Faults From a Fault List](#)
- [Defining Fault Timing for Transient Faults](#)
- [Extracting Flops and Latches for Transient faults](#)
- [Sampling](#)
- [Coverage Calculations](#)
- [Reading the SFF file in Fault Campaign Manager](#)
- [Testability](#)

Note:

For further information on the syntax and options, see [Standard Fault Format](#).

User Defined Fault Status (UDFS)

VC Z01X provides a set of standard fault classifications, but SFF allows you to redefine and add new definitions to create classifications that fit customized functional safety flows (UDFS). The UDFS consists of defining new statuses, redefining existing statuses, creating status groups, and defining status interactions.

The status is created from two alphanumeric characters. There is no requirement for how UDFS are defined. A common approach is to assign significance to each character with the first character representing the observation status of the fault and the second character the detection status. Any two-character status can be used and the definitions created that are relevant for your functional safety environment.

Any re-use of a VC Z01X standard fault classification must first re-define the built-in status before the two-character code is available for use as a UDFS. When using UDFS, you must also indicate how the new statuses will interact with the fault simulation environment. This includes defining the default status when a fault status is not explicitly set by the simulation, deciding which statuses will be selected for fault simulation between multiple workloads, creating the promotion table for fault status interaction, and creating coverage equations to calculate your final coverage results.

All fault status definitions and interaction are contained in the `StatusDefinitions` block in SFF. Status interactions are created in the `PromotionTable` section of the `StatusDefinitions` block. See “Defining Status Interactions” for more information on the promotion table.

For example,

```

StatusDefinitions
{
  # Redefinition of some built-in status definitions
  Redefine ND NX "Not Detected";
  Redefine DD DX "Redefine DD";
  # Creation of new fault status definitions
  NN "Not Observed Not Diagnosed";
  NP "Not Observed Potential Diagnosed";
  ND "Not Observed Diagnosed";
  PN "Potential Observed Not Diagnosed";
  OP "Observed Potentially Diagnosed";
  ON "Observed Not Diagnosed";
  OD "Observed Diagnosed";
  # Any fault not set by a strobe system task will be set to this status
  DefaultStatus (NN)
  # Any fault of this status will be chosen by the simulation for injection
  Selected (NA, NN, NP, PN, OP, ON)
  #Define the merging of faults
  PromotionTable
  {
    StatusLabels (NN,NP,ND,PN,OP,ON,OD)
    # NN NP ND PN OP ON OD
    [ - | | | ON | | ; # NN
    - - | | | | ; # NP
    - - - | OD | | ; # ND
    - - - - ON | | ; # PN
    ON - OD ON OP | | ; # OP
    - - - - - | ; # ON
    - - - - - - ; # OD
  }
}
}

```

Defining UDFS Interactions and Merging Fault Lists

User Defined Fault Status (UDFS) provided a mechanism for defining custom fault status definitions during fault simulation. Most fault injection campaigns consist of multiple simulation runs. To resolve status differences between runs and when merging fault lists, the interaction between faults of different status must be defined for UDFS. An example is provided in the previous section of a basic `PromotionTable` that can be used for defining interactions with UDFS.

In addition to the `PromotionTable` syntax shown in the above example, a more compact format is available that may be easier to use when you are creating the fault status

interaction details. `PromoteOrder` and `Promote` SFF keywords can be used to define the promotion table. `PromoteOrder` is a shorthand method for quickly creating a symmetrical promotion table and `Promote` can be used to override some of the entries in that table with specific promotions. The same table above can be created with:

```
StatusDefinitions
{
Redefine ND NX "Not Detected"
Redefine DD DX "Redefine DD";
NN "Not Observed Not Diagnosed";
NP "Not Observed Potential Diagnosed";
ND "Not Observed Diagnosed";
PN "Potential Observed Not Diagnosed";
OP "Observed Potentially Diagnosed";
ON "Observed Not Diagnosed";
OD "Observed Diagnosed";
DefaultStatus (NN)
Selected (NA, NN, NP, PN, OP, ON)
PromotionTable
{
PromoteOrder( NG < HG < NN < NP < ND < PN < OP < ON < OD < UG )
Promote( OP + NN => ON )
Promote( OP + ND => OD )
Promote( PN + OP => ON )
}
}
```

Defining and Generating Faults

Fault Generation

Fault Generation in SFF is defined in the `FaultGenerate` block. Synopsys recommends creating faults on all locations within the device that will be simulated for fault injection. The `FaultGenerate` block may contain multiple statements that affect which faults are created for fault injection. Statements are executed in the order they appear in the `FaultGenerate` block. SFF files can use multiple `FaultGenerate` blocks that will be processed in order the appear in the input file. Faults are generated with the following considerations:

Faults in a `FaultGenerate` block can be defined with either absolute paths that create individual faults or using wildcarded paths. Wildcarding of locations can be used to include entire blocks or hierarchies of the design. The following example generates permanent port faults (stuck-at) on the entire DUT under the top module and transient (SEU) on all flip-flops from cycles 1-1000 in the same DUT:

```
FaultGenerate
{
NA [0, 1] { PORT "top.**" }
NA ~ (1:1000) { FLOP "top.**" }
}
```

Port fault compiler directives in Verilog cells are honored in all fault generate statements that contain paths with wildcards. You may override this behavior with the `-fsim=suppress+cell` option for the fault generation tools. Port faults will not be generated on cells that are not contained within the port fault compiler directives or defined as cells using compiler command line options `-fsim=suppress+cell` at the gate level or with `-fsim=portfaults` for RTL designs.

The following statement will honor port fault compiler directives when creating the faults for simulation.

```
FaultGenerate
{
NA [0, 1] { PORT "top.**" }
NA ~ (1:1000) { FLOP "top.**" }
}
```

Faults specified with absolute paths, will be created regardless of the port fault compiler directives or command line options used at compile time. The following example will create the faults given in the `FaultGenerate` block even if they are contained within a suppressed block.

```
FaultGenerate
{
NA [0, 1] { PORT "top.dut.blka.cell1.z" }
NA ~ (1:1000) { FLOP "top.dut.blkb.flop4.Q" }
}
```

Faults should be created with a fault status of Not Attempted (NA) or to the status that NA has been redefined to in the `StatusDefinitions` block. The `vc_fcc` reports the following warning message when a fault is specified in this block which is of non-NA status and resets the fault to NA status. This warning message is displayed only once.

Warning! One or more faults in the provided `FaultGenerate` blocks were specified with a starting status other than 'NA'.
 These faults will be reset to 'NA' status. Faults which you wish to exclude from simulation should be specified in an `Exclude` block.
 To import faults with a status other than 'NA', they may be specified in a `FaultList` block.

Defining the Fault List

In addition to creating faults by traversing the design with `FaultGenerate`, you may create a fault list from a text list using the `FaultList` block. The fault list in the `FaultList` block can be specified directly or imported from a previous coverage file with VC Z01X. Faults that are created with a status other than NA, retain that status when they are imported into the fault list used by VC Z01X. The general format of a fault origin within the `FaultList` section is:

```
<status> <fault-type> (<timing-info>) {<Location-Info>}
Example:
FaultList
```

```
{
Timing("cycle1", CycleTime 50ns, Offset 40ns)
UseTiming("cycle1")
# Stuck-at fault
NA 0 { "top.dut.cell1.A" }
# Transient fault
NA ~ (5) { "top.dut.ff1.Q" }
}
```

The first fault in the list creates a stuck-at 0 fault on port A of the specified cell. The second fault creates a transient (SEU) fault on the flip-flop at cycle 5.

Excluding Faults From a Fault List

There are various reasons for excluding faults from a fault list as follows:

- a fault that is safe by design.
- a fault that does not violate any safety goal.
- a fault on pre-configured registers.
- a hierarchy that is not exercised with the use case of the design.

Coverage results can be measured more accurately by excluding these faults. You can exclude specific blocks, instances, gates, or pins from the fault list. Faults that are excluded through this mechanism will not be shown in the final coverage report because they are not created or imported as part of the fault universe. Statements in the `FaultGenerate` block are executed in order of appearance so the order in the SFF file is important.

Excluding Faults from a Fault List with Exclude

You may specify objects to be excluded using the `Exclude` block in SFF, as shown in the following example. You can use wildcarding to remove entire hierarchies or target individual faults.

```
FaultGenerate
{
NA [0, 1] { PORT "top.**" }
Exclude
{
NA [0, 1] {PORT "top.sub_block_A.adder.**" }
NA [0, 1] {PORT "top.io.demux.alu.**" }
NA [0, 1] {PORT "top.reg_bank.bank2.reg5.Q" }
}
```

For more details of options and usage on fault sites and fault format, see [Fault Models](#) and [Standard Fault Format](#).

Excluding Faults from a Fault List with Constraints

Some faults may be considered safe when the device is operating within a constrained set of inputs. This is common for inputs connected to test logic that will not be active when the device is operating in mission mode and for other scenarios. These faults can be removed with the `Constraint` block. Faults removed by constraints are set to a status provided by the user and are not removed completely from the fault list. The status may be used from the Exclude Group of fault statuses or be a user defined status. Only one status for constraints can be used in a single SFF file. When a fault location is constrained, it is treated by the fault generation tool as if it were tied to power or ground based on the constrained given in the `Constraint` block.

Example:

```
StatusDefinitions
{
  ED "Excluded Definition";
}
Constraint one
{
  ED "system.processor.mod1.mod2.i==0";
}
```

Defining Fault Timing for Transient Faults

In addition to a fault origin, all transient faults (SEU or SET) must contain timing information. There are multiple methods you can leverage for creating timing information for transient faults. You can define timing in the SFF file through the `Timing` keyword. It contains the cycle time (`CycleTime`) and optionally an offset into that time (`Offset`).

In the SFF file, you can define multiple timing blocks and activate each block for different sections of the fault list by using the `UseTiming` directive. The timing cycles and injection times are calculated based on the time in the simulation faults are injected. Faults are injected at time 0 in the simulation unless delayed with `$fs_inject` or the equivalent command line option. Absolute timing faults do not follow this behavior and are injected at the time given in the fault descriptor.

The following examples demonstrate three methods for creating the same fault list for transient fault simulation.

Example:

```
FaultGenerate
{
  Timing("clock1", CycleTime 5ps, Offset 1ps)
  UseTiming("clock1")
  NA ~ (10:100) {FLOP "testbench.dut.blkA.**"}
}
```

Alternatively, you can specify the `Offset` value as a percentage of the `CycleTime`.

Example:

```
FaultGenerate
{
  Timing("clock1", CycleTime 5ps, Offset 20%)
  UseTiming("clock1")
  NA ~ (10:100) {FLOP "testbench.dut.blkA.**"}
}
```

Finally, you can create absolute timing faults. These faults are not delayed by \$fs_inject or create_testases -fault_injection_time <string> command option.

Example:

```
FaultGenerate
{
  Timing("clock1", CycleTime 5ps, Offset 20%)
  UseTiming("clock1")
  NA ~ (12ps:102ps|5ps) {FLOP "testbench.dut.blkA.**"}
}
```

Extracting Flops and Latches for Transient faults

VC Z01X can generate faults on flip-flop and latch locations throughout the design. Flop and latch locations are identified during compilation when flip-flops and latches are modeled at the RTL level and are identified automatically in the fault generation stage when flip-flops and latches are modeled with sequential UDPs.

The second step is to create FLOP type faults in the input SFF file FaultGenerate section.

Example:

```
FaultGenerate
{
  Timing("clock1", CycleTime 10ns, Offset 9ns)
  UseTiming("clock1")
  NA ~ (1:100) {FLOP "testbench.dut.**" }
}
```

Sampling

Using a random sample of faults rather than all possible faults can reduce the total runtime for large designs and provide a reasonable estimate of the coverage that can be achieved for the full fault universe. You can create a random sample of faults using the Sampling block in the SFF file. VC Z01X provides three separate fault sampling options: confidence model, percentage based, and fixed number.

Sampling From SFF

The confidence model is a method that uses statistical sampling to determine the level of confidence that a fault coverage result will lie within a specified range of the measured

result. For example, if a confidence level is 99 and the confidence interval is 10, then there is a 99% confidence that a final result of 50% coverage would lie between 45-55% if all faults had been injected.

Example:

```
FaultGenerate
{
  Sampling("c1", ConfidenceInterval 1.0, ConfidenceLevel 99)
  UseSampling("c1")
  NA [0,1] { PORT "test.dut.blkA.***" }
}
```

Percentage-based sampling selects a pseudo-random percentage of the faults to be injected. The pseudo-random seed is set, but you can override it.

Example:

```
FaultGenerate
{
  Sampling("p1", Percentage 10.0)
  UseSampling("p1")
  NA [0,1] { PORT "test.dut.blkA.***" }
}
```

Finally, the fixed number sampling method creates the specified fixed number of faults from random locations or times in the simulation. SFF Fixed fault sampling samples over the full sampling section rather than on a per-statement basis.

Example:

```
FaultGenerate
{
  Sampling("f1", Number 10000)
  UseSampling("f1")
  NA [0,1] { PORT "test.dut.blkA.***" }
}
```

VC Z01X generates all faults when using permanent fault sampling. Faults are selected by one of the random selection methods after the generation of the fault list is completed. When sampling transient faults, VC Z01X calculates the total number of faults according to fault locations and injection times. From that calculation a random number of faults are created to meet the selection criteria.

For both permanent and transient fault sampling, the faults not selected are not retained as part of the fault list. A message appears in both the import tool log and the coverage report that indicates the number of faults generated from the full fault universe as part of sampling:

X faults sampled from Y possible faults

You can combine the Sampling option with the capabilities of defining faults and removing faults from a fault list file through the Exclude block for flexibility in selecting fault placement.

In the SFF file, samples are calculated for each statement in the `FaultGenerate` block. A statement is processed, and sampling completed before moving on to the next statement. This allows you to control sampling on a statement by statement basis.

Example:

```
FaultGenerate
{
  Sampling ( sample1, Percentage 10.0)
  NA [0,1] { PORT "testbench.dut.blk1.**" }
  NA [0,1] { PORT "testbench.dut.blk2.**" }
}
```

In this example, VC Z01X will create a sample for 10% of the faults that exist in blk1 and 10% of the fault in blk2 separately. If blk1 has 100 faults and blk2 500 faults, the sampled fault list will contain a sample of 10 faults from blk1 and 50 faults from blk2.

Example:

```
FaultGenerate
{
  Sampling ( sample1, Percentage 10.0)
  NA [0,1] { PORT "testbench.dut.blk1.**" }
  Sampling ( sample2, Number 5)
  NA [0,1] { PORT "testbench.dut.blk2.**" }
}
```

This example shows a more complex case where the sampling is changed between fault generation statements. VC Z01X will create a sample for 10% of the faults that exist in blk1 and will sample 5 faults from blk2 separately.

VC Z01X Sampling Selection

Systematic sampling is a technique that populates a sample by selecting a random starting point and then sampling from a fixed, periodic interval. Systematic sampling has the benefit of eliminating the possibility of clustered results because it guarantees the sample is spread uniformly across the entire set of faults to be sampled.

A setting is provided to enable systematic sampling as a default sampling method.

Syntax:

```
vc_fcc -no_systematic_sampling
```

VC Z01X implements systematic sampling through the following process:

1. Calculate the sampling interval.
2. Select a random starting point in the fault list.
3. Select a random fault in the sampling interval group of faults.

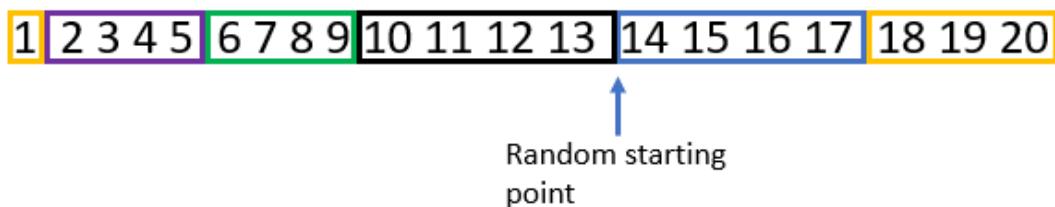
The sampling interval is calculated according to the total number of faults to include in the sample and the set of all possible faults.

`sampling_interval = full_fault_set_size / fault_sample_size`

For example, if you have 20 faults and want to sample 25%, your full fault set size is 20 and your fault sample size is 5. The sampling interval can be calculated as 4.

After VC Z01X selects the random starting point in the fault list, it will randomly select a fault from each group of 4 faults in the fault list. In the image below, VC Z01X selects a starting point in the fault list and then will randomly select a fault from each of the different boxes, which show the sampling intervals.

Figure 20 Sampling and Fault Collapsing



When choosing faults for sampling, VC Z01X considers prime and collapsed faults equally. If a collapsed fault is selected for simulation through sampling, it will be promoted to a prime fault in the sampled fault list and simulated. This technique avoids bias in the generated sample by considering all faults in the fault list instead of just prime faults.

Coverage Calculations

Defining the Coverage Formula

The formula used to create the final coverage number from VC Z01X is defined in the SFF file. The coverage equation is required when UDFS and status groups are used. The statuses included in the coverage calculations can be include both UDFS and any of the VC Z01X built-in statuses or status groups.

Formula:

Test Coverage = Weighted_Total / (Total)

Fault Coverage = Weighted_Total / (Total+UB+UI+UR+UT+UU+UO)

Where,

Total = the total number of testable faults.

Weighted_Total = the total of each fault status multiplied by its weight value:

Weighted_Total = (DD * DD_weight) + (DT * DT_weight) + (PD * PD_weight) + (PT * PT_weight) + (NA * NA_weight) + ..

Usage:

Coverage

```
{Test Coverage = (DD+DT+DE+DF+0.5*PD+0.5*PT)/(Total); Fault Coverage = (DD+DT+DE+DF+0.5*PD+0.5*PT)/(Total+UB+UI+UR+UT+UU+UO); }
```

Example SFF:

```
Coverage
{
"Diagnostic Coverage" = "DD/ (NA + DU + DA + DN + DD) ";
}
```

Reading the SFF file in Fault Campaign Manager

One of the first items that needs to be set up in `vc_fcm` is where and how to create the fault universe that will be used throughout the simulation. The `create_campaign` command is used inside `vc_fcm` to define and create the fault universe as in the example below. You may optionally run `vc_fcc` before starting `vc_fcm` to create the fault universe in the fault database.

Example:

```
create_campaign -args "-full164 -daidir simv.daidir -sff test.sff
-campaign fcl -dut test.dut"
```

Testability

Testability is an internal set of processes run by Fault Campaign Manager to maximize throughput of fault simulation. Testability is run automatically by `vc_fcm` and requires no intervention by you to complete successfully. Synopsys recommends that you use the default settings to allow testability to optimally order multiple workloads and to select faults that are relevant for each test in your fault simulation campaign.

Testability for stuck-at

Testability is the combination of processes executed by `vc_fcm` to measure the ability of each test pattern to detect individual faults. Testability provides early feedback of potential simulation coverage and helps identify areas of the design likely to have low coverage. Fault Campaign Manager runs toggle simulation once for each test pattern and updates

the necessary testability information automatically after each test pattern is simulated to update the status of fault coverage. Testability is calculated starting when the faults will be injected into the simulation. This happens at time 0 in the simulation by default, but might be delayed with [\\$fs_inject \(system task\)](#).

The automatic testability calculations are effective when analyzing permanent (stuck-at) faults.

Fault Simulation

Fault simulation is a compute intensive process. VC Z01X fault simulation is optimized for injecting faults into RTL and gate-level netlists using functional vectors (workloads) as stimulus. Fault coverage results are dependent on the quality of input workloads to detect faults. VC Z01X implements the concurrent simulation algorithm, which is the fastest way to inject and simulate faults into a design.

Defining the Workloads for Simulation

Prior to running testability and fault simulation, you must define the workload or list of workloads that will be used to provide stimulus for measuring coverage. Functional vectors are defined as tests and are imported into `vc_fcm` with the `create_testcases` command. This command creates tests with virtual test names that you define. The stimulus type and location are determined when executing the `create_testcases` command as in the following examples. Using a Verilog Testbench does not require you to list a stimulus file; it is included as part of the compilation step. An example is as follows:

```
create_testcases -name test_result -exec "$::env(PWD)/simv"
```

Using an eVCD file requires you to specify the location of the stimulus file:

```
create_testcases -name user_test1 -exec simv -args "-stim=file:./user.evcd"
```

Using an FSDB File requires you to specify the stim type and location of the stimulus file:

```
create_testcases -name user_test1 -exec simv -args "-stim=file:./user.fsdb"
```

Simulation

Fault simulation is executed through the `fsim` command in `vc_fcm`. The `fsim` command is dependent on compilation of the design, initial creation of the fault list, and definition of workloads that will be used for simulation. Logic simulation is a highly recommended prerequisite to running Fault Campaign Manager.

Coverage

The final step in `vc_fcm` is to create your coverage reports. There are several considerations when deciding which faults to include in the report as discussed in the following subsections:

- [Interpreting Fault Status](#)
- [Generating a Fault Coverage Report](#)
- [Collapsed and Uncollapsed Fault Lists](#)
- [Unselected vs. Untestable](#)
- [Example Coverage Reports](#)

Interpreting Fault Status

Faults are assigned to classes corresponding to their current fault detection or detectability status. A two-character code is used to specify a fault class. Fault classes can be grouped together to form fault status groups. Faults are only assigned to fault classes, but the fault status groups may be used for reporting. Fault classes and status groups may include both UDFS and built-in statuses. The fault class hierarchy for all fault classes is described in “Interpreting Fault Status”.

Generating a Fault Coverage Report

You can use the `report` command inside Fault Campaign Manager to write a fault list to a file for analysis or to read back in for future sessions. There are numerous formats and information that can be conveyed through VC Z01X coverage reports. These examples demonstrate some common combinations of basic coverage reports. For a full list of reports that can be generated, see [Working with Results](#).

Examples:

1. Write a fault list for all faults:

```
report -report coverage.sff -overwrite
```

2. Write a fault list containing only NN class faults (UDFS fault class):

```
report -report faults_NN.sff -faultstatus NN -overwrite
```

By default, the list of faults is uncollapsed unless the `-faultstatus` switch is used. When the `-faultstatus` switch is included in the `report` command the coverage report contains only the prime faults unless the `--` option is included to display uncollapsed faults. The following command includes the uncollapsed faults by using the `--` option:

```
report -campaign fc1 -report coverage.sff -faultstatus DD,PD,--  

-overwrite
```

Collapsed and Uncollapsed Fault Lists

To improve performance, VC Z01X collapses all equivalent faults and injects only the collapsed set. For example, the stuck-at faults on the input pin of a BUF device are considered equivalent to the stuck-at faults on the output pin of the same device. The collapsed fault list contains only the faults at one of these pins, called the primary fault site. The other pin is then considered the equivalent fault site.

Coverage Summary

You can generate a fault summary report using the `-summaryonly` option of the `report` command. All faults are listed in the summary.

Example: Collapsed and Uncollapsed Fault Summary Reports

```
report -report summary.sff -summaryonly
-----
# Fault Coverage Summary for Default List
#
# Total
-----
# Number of Faults: 550 100.00%
#
# Untestable Faults: 40 7.27% 100.00%
# Untestable Unused UU 40 7.27% 100.00%
#
# Testable Faults: 510 92.73% 100.00%
# Not Controlled NC 13 2.36% 2.55%
# Not Observed Not Diagnosed NN 48 8.73% 9.41%
# Not Observed Diagnosed ND 5 0.91% 0.98%
# Potentially Observed Not Diagnosed PN 8 1.45% 1.57%
# Observed Not Diagnosed ON 18 3.27% 3.53%
# Observed Potentially Diagnosed OP 3 0.55% 0.59%
# Observed Diagnosed OD 415 75.45% 81.37%
#
# Status Groups -----
# Dangerous Diagnosed DD 420 76.36%
# Dangerous Not Diagnosed DN 29 5.27%
# Safe SA 40 7.27%
# Dangerous Unobserved SU 61 11.09%
#
# Coverage -----
# Diagnostic Coverage 93.54%
-----
```

The following example is a fault list that contains one fault entry per line. Each entry consists of a single fault separated by one or more spaces. The first item indicates the two-character fault class code, the second item is the stuck-at value (sa0 or sa1), and the third item is the pin path name to the fault site.

If the fault list contains equivalent faults, the equivalent faults must immediately follow the primary fault on subsequent lines. Instead of a class code, an equivalent fault is indicated by a fault class code of --.

Example: Typical Fault List Showing Equivalent Faults

```
FaultList
{
  PD 0 { PORT "top.CLK" }
  PD 1 { PORT "top.CLK" }
  DD 1 { PORT "top.RSTB" }
  DD 0 { PORT "top.RSTB" }
  NA 1 { PORT "top.i22.mux2.A" }
  DD 0 { PORT "top.i22.reg1.MX1.D" }
  -- 0 { PORT "top.i22.mux1.X" }
  -- 0 { PORT "top.i22.mux1.MUX2_UDP_1.Q" }
  PD 1 { PORT "top.i22.reg2.r.CK" }
  PD 0 { PORT "top.i22.reg2.r.CK" }
  DD 1 { PORT "top.i22.reg2.r.RB" }
  NO 0 { PORT "top.i22.out0.EN" }
  NO 1 { PORT "top.i22.out0.EN" }
  UT 1 { PORT "top.i22.reg2.lat1.SB" }
  UR 0 { PORT "top.i22.mux0.MUX2_UDP_1.A" }
    -- 0 { PORT "top.i22.mux0.A" } # equivalent to UR fault above it
}
```

You can control whether the fault list contains equivalent faults or primary faults by using the `-faultstatus` option of the `report` command.

Unselected vs. Untestable

Two fault groups that should be understood and accounted for are the unselected and untestable fault groups.

Untestable Faults

An untestable fault is a fault that is structurally impossible to cause a device failure independent of the stimulus used to exercise the device. These faults are logically identifiable because they are connected to power or ground, tied off to a constant value in the Verilog netlist or library cells, or unconnected in the netlist. Untestable faults are safe faults in the functional safety space. They are logically undetectable and will appear in the coverage report as tied, redundant, blocked, or unused fault statuses. Definition of strobe locations can affect the structural analysis when classifying untestable faults, primarily when one or more outputs are not defined as observation or detection points, the faults leading up to them are categorized as untestable.

Unselected Faults

An unselected fault is a fault that is not activated with the current workload. Unselected faults must be considered as potentially dangerous because with different stimulus they may become observable and detectable. Unselected faults will be categorized as not

controllable, not observable, not strobed, or not injected (transient faults only). These faults can often be made detectable by adding additional workloads or modifying the current stimulus to exercise the fault location.

Example Coverage Reports

The following example shows a typical coverage report in SFF format. The equivalent faults always immediately follow the primary fault and are identified by two dashes (--) in the second column.

Example:

```

Tool("REPORT")
Info(" Type: Fault Coverage Report")
Info(" Version: T-2022.06-SP2")
Info(" FDB Storage Path: fdb")
Info(" FDB Server: server:11111")
Info(" FDB Project: default")
Info(" FDB Campaign: fc")
Info(" Command: <truncated>")

TestList {
    1 test1 {Results:510 NC:20 NN:52 ND:5 PN:8 ON:60 OP:3 OD:362}
    2 test2 {Results:156 NC:39 NN:48 ND:1 ON:16 OD:52}
}

FaultInfo {
    TestNum;
}

StatusDefinitions {
    Redefine DD DX "Redefine DD";
    Redefine PD PX "Redefine PD";
    Redefine ND NX "Redefine ND";
    NN "Not Observed Not Diagnosed";
    NP "Not Observed Potentially Diagnosed";
    PD "Potentially Observed Diagnosed";
    ND "Not Observed Diagnosed";
    PN "Potentially Observed Not Diagnosed";
    PP "Potentially Observed Potentially Diagnosed";
    ON "Observed Not Diagnosed";
    OP "Observed Potentially Diagnosed";
    OD "Observed Diagnosed";
    AN "Assumed Dangerous Not Diagnosed";
    AD "Assumed Dangerous Diagnosed";
    AP "Assumed Dangerous Potentially Diagnosed";
    IS "Invalid Status Promotion";

    DefaultStatus(NN)

    Selected(NA, NN, ON)
}

```

```

PromotionTable {
    StatusLabels (NN, NP, ND, PD, PN, PP, ON, OP, OD, AP, AN, AD, IS)
    [
        NN NN ND PD PN PP ON OP OD IS IS IS IS ;
        NN NP ND PD PN PP ON OP OD IS IS IS IS ;
        ND ND ND PD PD PD OD OD OD IS IS IS IS ;
        PD PD PD PD PD PD OD OD OD IS IS IS IS ;
        PN PN PD PD PN PP ON ON OD IS IS IS IS ;
        PP PP PD PD PN PP ON OP OD IS IS IS IS ;
        ON ON OD OD ON ON ON ON OD IS IS IS IS ;
        OP OP OD OD ON OP ON OP OD IS IS IS IS ;
        OD OD OD OD OD OD OD OD IS IS IS IS ;
        IS IS IS IS IS IS IS IS AP AN AD IS ;
        IS IS IS IS IS IS IS IS IS AN AN AD IS ;
        IS IS IS IS IS IS IS IS IS AD AD AD IS ;
        IS ;
    ]
}

StatusGroups {
    DA "Dangerous Assumed" (HA, HT, IA, IF, IX, OZ);
    DD "Dangerous Diagnosed" (ND, NP, OD);
    DN "Dangerous Not Diagnosed" (ON, OP, PN);
    SA "Safe" (UB, UR, UT, UU);
    SU "Dangerous Unobserved" (NC, NN, NO, NT);
}
}

Coverage {
    "Diagnostic Coverage" = "DD/(NA + DA + DN + DD)";
}

FaultList {
    < 1> OD 0 {PORT "test.risc1.alu1.accum[0]"}
        -- 0 {PORT "test.risc1.alureg.d1.q"}
        -- 0 {PORT "test.risc1.alureg.r[1]"}
    < 1> OD 1 {PORT "test.risc1.alu1.accum[0]"}
        -- 1 {PORT "test.risc1.alureg.d1.q"}
        -- 1 {PORT "test.risc1.alureg.r[1]"}
<results truncated>
    < 1> OD 0 {PORT "test.risc1.tricon.o[5]"}
    < 1> OD 1 {PORT "test.risc1.tricon.o[5]"}
    < 1> OD 0 {PORT "test.risc1.tricon.o[6]"}
    < 1> OD 1 {PORT "test.risc1.tricon.o[6]"}
    < 1> OD 0 {PORT "test.risc1.tricon.o[7]"}
    < 1> OD 1 {PORT "test.risc1.tricon.o[7]"}
}
#-----
# Fault Coverage Summary for Default List
#
#-----                                     Total
#-----                                     550 100.00%
# Number of Faults:
```

```

#
# Untestable Faults:                                40  7.27% 100.00%
#   Untestable Unused                            UU  40  7.27% 100.00%
#
# Testable Faults:                                 510 92.73% 100.00%
#   Not Controlled                               NC  13  2.36% 2.55%
#   Not Observed Not Diagnosed                  NN  48  8.73% 9.41%
#   Not Observed Diagnosed                      ND   5  0.91% 0.98%
#   Potentially Observed Not Diagnosed        PN   8  1.45% 1.57%
#   Observed Not Diagnosed                      ON  18  3.27% 3.53%
#   Observed Potentially Diagnosed              OP   3  0.55% 0.59%
#   Observed Diagnosed                          OD  415 75.45% 81.37%
#
# Status Groups -----
#   Dangerous Diagnosed                         DD  420 76.36%
#   Dangerous Not Diagnosed                     DN   29  5.27%
#   Safe                                         SA   40  7.27%
#   Dangerous Unobserved                        SU  61 11.09%
#
# Coverage -----
#   Diagnostic Coverage                         93.54%
#-----

```

Example Flows

Concurrent Simulation Flow

VC Z01X injects and simulates faults for functional safety and can create a coverage report that includes all information about faults that are dangerous and detected by the safety mechanisms. The features of the software provide the ability to obtain this result within a single pass fault injection run when the Fault Campaign Manager, Standard Fault File, and VC Z01X are used.

Define the observability and diagnostic strobe points in the testbench or safety mechanisms to mark faults as observed or diagnosed and to set the fault status in some cases based on the status of the fault in simulation. The following example shows how faults can be marked observed and then later be marked diagnosed in the same simulation.

Example:

```

always @ (negedge tb.dut.sysclk)
begin
#25
// Compare all DUT output signals
cmp = $fs_compare(tb.dut);
if (1 == cmp) // Compare has diffs in GM/FM
begin
$fs_set_status("ON", tb.dut);

```

```

break;
end
else
if (2 == cmp) // Compare with X values in FM
begin
$fs_set_status("PN", tb.dut);
break;
end
end
always @ (negedge tb.dut.sml.valid)
begin
// Get status of fault
status = $fs_get_status();
// Compare diag point in safety mechanism
cmp = $fs_compare(tb.dut.sml.diag);
// Compare has diffs in GM/FM
if (1 == cmp)
begin
if ((status == "ON") || (status == "PN"))
// if status is observed, drop observed diagnosed
$fs_drop_status("OD");
else
// if status was not observed, drop not observed, diagnosed
$fs_drop_status("ND");
break;
end
else if (2 == cmp)
// Compare with X values in FM
begin
if ((status == "ON") || (status == "PN"))
// if status is observed, drop observed pot. diagnosed
$fs_drop_status("OP");
else
// if status == not observed, drop not observed, pot. Diag.
$fs_drop_status("NP");
break;
end
end

```

Using Software Test Libraries

Software Test Libraries (STLs) are a common form of workload to use during the fault injection process. VC Z01X and Fault Campaign Manager are useful tools in determining which STLs provide the highest contribution to fault coverage.

STLs can be most effectively simulated in a Verilog environment as object code that is loaded into a memory and then run by the design. There are multiple methods to incorporate memory loading such that it operates effectively in Fault Campaign Manager. These methodologies are suggested because they do not require the design to be recompiled to run different STLs.

Example Testbench:

```
reg [15:0] image_memory [15:0]; string filename;
initial begin
filename = $value$plusargs(TESTNAME);
$readmemb(filename, image_memory); end
```

vc_fcm script example:

```
create_campaign -args "-full64 -daidir simv.daidir -sff input.sff
-campaign fc1 -dut test.dut"
create_testcases -name test1 -exec "$::env(PWD)/simv" -args
"+TESTNAME=stl1.dat"
create_testcases -name test2 -exec "$::env(PWD)/simv" -args
"+TESTNAME=stl2.dat"
fsim
report -campaign fc1 -report mycoverage.sff -overwrite
```

The combination of the `$value$plusargs` in the testbench with the `-args` setting in the Fault Campaign Manager script causes the simulation to be run with the command-line argument `+TESTNAME=stl*.dat` so the correct memory image is loaded into the simulation.

Resolving Hyperfaults

Hyperfaults are one of the performance related side effects of concurrent fault simulation. These faults are removed from the simulation by default, and Synopsys recommends that you retain this setting.

Hyperfaults are considered dangerous faults for coverage calculations. To determine if these faults can be detected, Synopsys recommends the following methodology for simulating the faults. Using Fault Campaign Manager, run fault injection with default hyperfault settings.

- Run fault injection with the default settings and allow hyperfaults to be dropped from the simulation.
- Disable hyperfault checking
- Re-run the hyperfaults in fault simulation
- Report the final coverage. Note that the faults in the second fault simulation that started as hyperfaults (HA) and were not detected will remain reported as hyperfaults (HA). These faults will only be promoted if they result in a higher promoted fault status in the fault simulation with hyperfault checking disabled.

The following example uses the fault statuses defined in [User Defined Fault Status \(UDFS\)](#) section. If the other fault statuses or status groups are created in your SFF, it is requested you review the following commands correctness:

Step 1 - Example vc_fcm script:

```
create_campaign -args "-full164 -daidir simv.daidir -sff input.sff
-campaign fc1 -dut test.dut"
create_testcases -path stim -exec "$::env(PWD)/simv"
fsim
set_config -fsim_std_args "-fsim=limit+hyperactive+0"
fsim -selected_status HA
report -report coverage.sff
```

15

Appendix A: Fault Campaign Manager Commands

This appendix lists the commands available for use in Fault Campaign Manager. The context for these commands is presented in the preceding chapters of this user guide.

Command Executed on the vc_fcm UNIX Command Line

The `vc_fcm` command is executed on the UNIX command line.

Synopsis:

```
vc_fcm  
-connect  
-disable_auto_setup  
-fc | -campaign] <string>  
-fcm_dir <directory>  
-fdb_connection_timeout <integer>  
-fdb_mgmt_only  
-fdb_path <directory>  
-fdb_project <string>  
-fdb_server <string>  
-fdb_storage_path <directory>  
-h | -help  
-log <filename>  
-no_echo_script  
-no_exit  
-no_log  
-output <filename>  
-tcl <filename>  
-tcl_script <filename>  
-temp_session_dir
```

Description:

Launches Fault Campaign Manager. Do not create session directory in the `fcm.dir` directory. This option can only be used if you want to use FCM to manage/query FDB, such as add/remove a user, check what campaign exists, query faults or results, or any other `show_*` command. If you want to execute campaign with commands such as `tsim`,

coats, fsim, dump, then the fcm.dir/session* directory is required. It will ignore the -no_log option.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-connect	Directly connect to server; if specified also set project and campaign.
-disable_auto_setup	Disables (a) automatic connection to FDB when -fdb_server/-fdb_project options or SNPS_FDB_SERVER/SNPS_FDB_PROJECT variables are defined. (b) automatic setting of campaign when -campaign option or SNPS_FDB_FAULT_CAMPAIGN variable is defined.
-fc <string>	Name of the fault campaign. When used, relevant configuration is also set.
-fcm_dir <directory>	Name of the directory to place all generated files created by simv invoked from FCM. The default name is fcm.dir.
-fdb_connection_timeout <integer>	Timeout value (in sec) when writing to fault database. When used, relevant configuration is also set. The default value is 10 seconds.
-fdb_mgmt_only	Runs FCM with no license check to allow FDB management commands. Commands to execute a fault campaign are not enabled in this mode.
-fdb_path <directory>	Path to the fault database directory.
-fdb_project <string>	Set the name of the FDB project or fault database.
-fdb_server <string>	Specifies the name of the FDB server (host:port).
-fdb_storage_path <directory>	Path for FDB storage location. The default is fdb.
-h [-help]	Produce help message.
-log <filename>	Log file name. The default file is fcm_debug.log and is created in the FCM directory.
-no_echo_script	Don't echo commands executed by the script. Only valid when -tcl_script option is used.
-no_exit	Do not exit at the end of tcl script and return to FCM prompt. Only when -tcl_script option is used.

Option	Description
-no_log	Don't create session directory at all in <i>fcm.dir</i> directory.
-output <filename>	File name or path where standard streams are copied. Default file <i>vc_fcm.log</i> is created in the FCM directory.
-tcl <string>	When the argument is empty, opens TCL interactive shell. When the argument is non-empty, executes TCL version of FCM command and exit. Must be the last FCM argument. Please do remember to quote/escape brackets.
-tcl_script <filename>	Name of the TCL script to execute.
-temp_session_dir	Use temporary session directory latest instead of time-stamped directory. Any existing latest will be overwritten.

:

Commands Executed on the Fault Campaign Manager Command Line

The following commands are executed from the Fault Campaign Manager shell (*vc_fcm*) command prompt. Obtain the prompt by using the *vc_fcm* command.

add_attribute

Synopsis:

```
add_attribute
  -key <string>
  [-testcases | -tcs] <list of string>
  -value <string>
```

Description:

Adds an attribute to the defined object in FDB. If the key for object already exists, it is overwritten. *add_attribute* requires the FDB to be connected (see [fdb_connect](#) command) and the campaign to be set (see [set_campaign](#) command). When an invalid object identifier (such as task, test case, and so on) is specified, an error is generated.

Parameters

Option	Description
-key <string>	When -key option is provided, only the defined keys (if it exists in the FDB object) are returned. If the key is not found, then it is not the case of an error (in this case a warning is generated). An empty key, "", returns all the keys.
-tcs <list of string>	Alias for -testcases.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.
-value <string>	Value to set for the specified attribute.

Example:

```
add_attribute -testcases TC001 -key ATTR001 -value VAL001
```

add_metadata

Synopsis:

```
add_metadata
-attributes
[-campaign | -fc] <string>
-name <string>
```

Description:

Allows users to add metadata to a fault campaign. Returns the index of the created metadata entry.

Parameter

Option	Description
-attributes	A list of key/value pairs, For example, { <key1> <value1> <key2> <value2> ... }
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-name <string>	The name of the metadata block.

Example:

```
add_metadata -name MD_001 -attributes {KEY_001 VAL_001}
```

add_user

Synopsis:

Alias for [user_mgmt](#)

coats

Synopsis:

```
coats
  -localhost
  -rerun
  [-testcases | -tcs] <list of string>
  -verbose
  -wait_for_tsim
  -wait_for_tsim_timeout <integer>
```

Description:

This command does the following:

- Runs controllability, observability, and testability analysis.
- Requires the FDB to be connected (see `fdb_connect` command) and campaign to be set (see `set_campaign` command).
- When `-testcases` is not provided, all the test cases are used. When the provided test case is not found in the FDB, the command returns with an error.

Error Message:

```
Error-[FCM-NO-TC] No testcase found in campaign
The current campaign contains no testcases. Please use
create_testcases command to create testcases.
```

- When the grid submit commands are not defined, the simulation is launched on the local machine. If the grid submit commands are defined and the simulation is to be executed on the local machine, `-localhost` option should be used.
- Returns false when an error is encountered, otherwise, returns true. When partial simulations result in failures, use `show_task_failures` command to view more information.

Parameters:

Option	Description
<code>-localhost</code>	Run the testability tasks on the local host instead of using the defined compute farms.

Option	Description
-rerun	Update the testability data for all testcases.
-tcs <list of string>	Alias for -testcases.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.
-verbose	Print messages for worker start/stop and task start.
-wait_for_tsim	Wait for other FCM instance to create FSDB files. Valid only for hybrid FDB.
-wait_for_tsim_timeout <integer>	Timeout for -wait_for_tsim option. The default timeout 48*60*60 second or 48 hours.

Example:

```
coats -testcases TC_001
```

cp_fault_results

Synopsis:

```
cp_fault_results
  [-campaign | -fc] <string>
  -dest_fm <list>
  -raw
  -src_fm <string>
```

Description:

Copy existing fault results between two failure modes. Both failure modes must exist in the fault campaign.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-dest_fm <list>	Destination failure mode. The destination FM may be a single failure mode or a list of failure modes.
-fc <string>	Alias for -campaign.
-raw	Copies the raw results to the destination fm. These results include per test results instead of only the final fault statuses.

Option	Description
-src_fm <string>	Source failure mode.

Example:

```
cp_fault_results -src_fm FM_001 -dest_fm FM_002
```

create_campaign

Synopsis

```
create_campaign
  -args <string>
  -localhost
  -only_compile
```

Description:

Creates a new campaign using `vc_fcc` in the connected FDB.

Parameters

Option	Description
-args <string>	Quoted string contains the arguments to be passed to <code>vc_fcc</code> . This argument is mandatory and must not be empty.
-localhost	Run <code>vc_fcc</code> on the local host instead of through grid submission. This option overrides any grid submission settings.
-only_compile	Causes the <code>create_campaign</code> command not to set the newly created campaign as the current campaign.

Example:

```
create_campaign -args "-full164 -daidir simv.daidir -campaign fc1 -sff
in.sff -dut mydut"
```

create_fc

Alias for [create_campaign](#).

create_project

Synopsis:

```
create_project
  -dump_file <filename>
  -fdb_project <string>
  -fdb_server <string>
```

Description:

Create a new project in the existing fault database.

Parameters:

Option	Description
-dump_file <filename>	Restore the project from the listed dump file.
-fdb_project <string>	Set the name of the FDB project or fault database. The default project name is <code>default</code> . Required argument.
-fdb_server <string>	Specifies the name of the FDB server (host:port).

Example:

```
create_project -fdb_project PRJ_001
```

create_tcs

Alias for [create_testcases](#).

create_testcases

Synopsis:

```
create_testcases
  -args <string>
  [-campaign | -fc] <string>
  -daiadir <directory>
  -desc <string>
  -exec <filename>
  [-fault_injection_time | -fit] <string>
  -fsdb <filename>
  -fsim_args <string>
  -max_faults <integer>
  -name <string>
  -path <directory>
```

```
-post_script <filename>
-pre_script <filename>
-stim=type:<string>
-timeout <integer>
-tsimm_args <string>
-user_attr <list>
```

Description:

Create test cases in the fault database inside the currently defined fault campaign.
 Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
-args <string>	Quoted string specifying the arguments to be passed to the simulation executable. These arguments will be used for both toggle and fault simulation.
-campaign <string>	Name of the fault campaign. If not provided, the current campaign will be used. If the campaign is not set and -campaign is not provided, the command will return an error.
-daidir <directory>	Specify the compiled daidir directory for a test. This is necessary when multiple tests are compiled with different testbenches. Each test will have a separate daidir used for simulation.
-desc <string>	Provide a user defined description for the testcase.
-exec <filename>	Required argument to provide the location and name of the simulator executable. It may be a user defined script, wrapper, or simulation executable name.
-fault_injection_time <string>	Define the fault injection time for a test. This overrides any \$fs_inject compiled in the design. The input may contain a time and timescale (example: 100ns). Toggle simulation will start filtering of value changes from the delayed fault injection time specified using this switch. Fault simulation will inject faults at this delayed fault injection time specified by the user.
-fc <string>	Alias for -campaign.
-fit <string>	Alias for -fault_injection_time.
-fsdb <filename>	Give the location of the FSDB file to be used for testability analysis. When given, the toggle simulation step in FCM will be skipped.

Option	Description
-fsim_args <string>	Quoted string used to pass arguments to the simulation executable for fault simulation only. The following are the runtime options: -fsim=fault+progress+<secs> <ul style="list-style-type: none"> - Report one line prefixed with “FSIM Progress” on the overall simulation progress (wall time, sim time, CPU+elapsed, peak mem, fault status). For example, -fsim=fault+progress+600-fsim=fault+monitor+drop- Reports one line prefixed with “FSIM Drop” whenever a fault is dropped (with details). -fsim=fault+monitor+strobe - Reports whenever FM strobing happens (with details). -fsim=fault+stats - Reports the events for all FMs at the end of fsim.
-max_faults <integer>	The -max_faults option sets the maximum number of faults which can be simulated in single FCM task. If the option is not defined, the configuration setting max_faults_per_fsim_task is used.
-name <string>	Provide the name of the testcase.
-path <directory>	Path to directory containing test stimulus (.evcd, .fsdb and .lst force files) files. This generates a testcase for each file based on their name to eliminate the need to explicitly add each testcase individually.
-post_script <filename>	Define a script which can be used for clean-up after the test operation is finished.
-pre_script <filename>	Used to define a script which is sourced before starting the actual test operation. This file can also be used to set environment variables for the test operation. The -pre_script must be used in sh format.
-stim=type:<string>	Type of stimulus input to be used for the test. Valid options include evcd, wgl, stil fsdb.
-timeout <integer>	The -timeout option defines the test-specific timeout for the FCM task in seconds. When this option is not provided, the default timeout of the grid submission command is used.
-tsim_args <string>	Quoted string used to pass arguments to the simulation executable for toggle simulation only.
-user_attr <list>	Allows you to assign user-defined attributes to the test cases. The following is a Tcl command example with user_attr option: create_testcases ... -user_attr {key1 value1 key2 value2 ...}

Example:

```
create_testcases -name test_result -exec "$::env(PWD)/simv"
```

dump

Synopsis:

```
dump
  -args <string>
  [-failure_modes | -fms] <list of string>
  -fids <list of int>
  -fsdb <filename>
  -mode <string>
  [-testcases | -tcs] <list of string>
```

Description:

Dumps an FSDB file with signal value changes of a single faulty machine.

Parameters

Option	Description
-args <string>	Specify additional runtime arguments to the simulation command for dumping.
-failure_modes <list of string>	Lists the name of failure modes.
-fids <list of int>	Dump the list of Fault IDs. See <code>report</code> for a description of how to obtain fault IDs.
-fms <list of string>	Alias for <code>-failure_modes</code> .
-fsdb <filename>	Path (relative to CWD or absolute) to fsdb file.
-mode <string>	Specify the dumping mode. Valid inputs are <code>fm</code> or <code>gmfm</code> . The default is <code>fm</code> .
-tcs <list of string>	Alias for <code>-testcases</code> .
-testcases <list of string>	Specify the testcase to run in order to create the dump of the faulty machine.

Example:

```
dump -fids 5 -fsdb fm5.fsdb -mode gmfm
```

exclude_faults

Synopsis:

```
exclude_faults
  [-campaign | -fc] <string>
  [ -failure_modes | -fms] <list of string>
  -fault_locations <filename | string>
  -fids <list of IDs>
  -sff_faultlist <filename>
  -status <string>
```

Description:

Writes fault results with exclude statuses (that is, E0-E9) to FDB as tool `USER_CREATED_RESULT` and testcase `ExcludedFaults`.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fault_locations <filename string></code>	Either (a) a path to a file containing fault locations (one per line) or (b) list of fault locations. Can contain wildcards.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fids <list of IDs></code>	VC Z01X can be used to analyze behavior or debug a single fault within a design. Use the FCM <code>dump</code> command to dump the faulty machine behavior of the specified fault into an FSDB file. Results can be viewed in Synopsys' Verdi Fault Analysis tool.
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-sff_faultlist <filename></code>	Path to SFF file with FaultList block. Other blocks are ignored.
<code>-status <string></code>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.

Example:

```
exclude_faults -sff_faultlist exclude.sff
```

fdb_connect

Synopsis:

```
fdb_connect
-fdb_path <directory>
-fdb_project <string>
```

Description:

Connect to an existing fault database. If a fault database does not exist, one is automatically created.

Parameters

Option	Description
-fdb_path <directory>	Set the location of the FDB. The default is <code>fdb</code> in the directory where <code>vc_fcm</code> was started.
-fdb_project <string>	Set the name of the FDB project.

Example:

```
fdb_connect -fdb_path myfdb
```

fdb_disconnect

Synopsis:

```
fdb_disconnect
```

Description:

Disconnects from the current FDB server. Disconnecting does not stop the running server (see `fdb_stop` command outside of `vc_fcm`).

Parameter

None.

Example:

```
fdb_disconnect
```

fsim

Synopsis:

```
fsim
[-failure_modes |-fms] <list of string>
-fids <list of int>
-fsim_args <string>
-localhost
-max_faults_per_task <integer>
-no_coats
-post_task_proc <string>
-post_fsim_task_proc <TCL proc>
-selected_status <list>
[-testcases |-tcs] <list of string>
-verbose
```

Description:

Simulate faults using the VC Z01X command defined by [create_testcases](#). This command manages the cluster/farm jobs automatically. Running `fsim` requires the FDB to be connected (see [fdb_connect](#) command) and the campaign option to be set (see [set_campaign](#)).

The command returns false when an error is encountered, otherwise it returns true. When partial simulations result in failures, use [show_task_failures](#) command to view additional information.

Note:

The failures container is cleared before every call to the [show_task_failures](#) command.

Parameters

Option	Description
-failure_modes <list of string>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file. When failure modes are not provided, all failure modes are simulated. When the provided failure mode is not found in the FDB, the command returns with an error.
-fids <list of int>	Simulate the list of Fault IDs. See report for a description of how to obtain fault IDs.
-fms <list of string>	Alias for -failure_modes.
-fsim_args <string>	Quoted string used to pass arguments to the simulation executable for fault simulation only.

Option	Description
-localhost	Run the fault simulation tasks on the local host instead of using the defined compute farms. If the grid submit commands are not defined, the simulation is launched on the local machine. If the grid submit commands are defined and the simulation should be executed on the local machine, then the <code>-localhost</code> option should be used.
<code>-max_faults_per_task <integer></code>	<code>-max_faults_per_task</code> option defines the number of simultaneous faults that each job needs to simulate. When <code>-max_faults_per_task</code> option is not provided, the value from the <code>-max_faults_per_fsim_task</code> configuration is used.
<code>-no_coats</code>	Run fault simulation without first running testability analysis. Testability analysis will be automatically performed before starting fault simulation unless this option is used. Fault simulation will start after testability has completed.
<code>-post_task_proc <string></code>	Function name to execute after each <code>fsim</code> task.
<code>-post_fsim_task_proc <TCL proc></code>	<p>Calls a TCL proc whenever <code>fsim</code> task is completed. The TCL proc can be defined in the FCM script or in a file and be sourced in the FCM script. The TCL proc can have a subset (or all, or even none) of the following arguments: <code>task_dir</code> <code>task_id</code> task status testcase failure mode <code>fids_status</code> The arguments are assigned with values according to their names (not their position within the argument list). The following arguments are supported: <code>task_dir</code>: The directory the task was executed in <code>task_id</code>: The task number. <code>task_status</code>: SUCCESS / FAILED / TIMEOUT / MEMOUT. <code>exit_code</code>: The exit code of the <code>fsim</code> process. <code>testcase</code>: The testcase used. <code>failure_mode</code>: The failure mode used. <code>fm</code>: An alias for <code>failure_mode</code>. <code>fids_status</code>: A list of pairs consisting of fids and their associated fault status.</p> <p>Usage Example: It outputs all arguments on the screen and then changes the first fault results to IX.</p> <pre> proc post_fsim_hook { {task_dir task_id task_status testcase failure_mode} {fids_status} { puts "task_dir: \$task_dir" puts "task_id: \$task_id" puts "task_status: \$task_status" puts "testcase: \$testcase" puts "failure_mode: \$failure_mode" puts "fids_status: \$fids_status" } } # Overwrite the first result with new status: # First extract the first fid/status pair: set first_result [lindex \$fids_status 0] # Extract the fid from the pair: set fid [lindex \$first_result 0] puts "fids: \$fid" # Now write a new fault status "IX" for this fm / tc / fid combination: write_fault_results -fm \$failure_mode -tc \$testcase -fids \$fid -status "IX" puts "TCL proc end" </pre> <p><code>set_config -post_fsim_task_proc post_fsim_hook</code></p>

Option	Description
-selected_status <list>	The selected status option allows you to pass a list of statuses to the fault simulation. When this option is used, only the faults that have the status specified will be considered for simulation. Example: <code>fsim -selected_status {HA HT}</code> . In this case, it selects all faults where the promoted status is HA or HT. These faults are scheduled for <code>fsim</code> . While simulating, these faults are simulated until a result status is no longer in <code>{NA ND HA HT}</code> . Note: Do not use the <code><list></code> argument if you are using <code>all</code> with this option.
-tcs <list of string>	Alias for <code>-testcases</code> .
-testcases <list of string>	List of testcase(s) defined through the create_testcases command. When test cases are not defined, then all testcases are simulated. When the provided test case is not found in the FDB, the command returns with an error.
-verbose	Print messages for worker start/stop and task start.

Example:

```
fsim
```

get_config

Synopsis:

```
get_config

[ -campaign | -fc]
-compress_task_logs
-confirmation
-delete_successful_task_logs
-fdb_connection_timeout
-fdb_host
-fdb_port
-fdb_project
-fdb_server
-fdb_storage_path
-fsim_layout
-fsim_mode
-fsim_std_args
-global_max_jobs
-max_faults_per_fsim_task
-max_task_reruns
-min_faults_per_fsim_task
-post_fsim_job_proc
```

```
-report_stats
-run_sim_in_sub_dir
-scheduling_params
-session_dir
-show_campaign_on_prompt
-show_cov_equations_layout
-show_det_pts_layout
-show_fault_results_layout
-show_faults_layout
-show_fcs_layout_detail
-show_fcs_layout_main
-show_fdb_project_on_prompt
-show_fdb_server_on_prompt
-show_fms_layout_detail
-show_fms_layout_main
-show_obs_pts_layout
-show_projects_layout
-show_promotion_tables_layout
-show_sms_layout
-show_status_groups_layout
-show_statuses_layout
-show_tcs_layout
-show_users_layout
-show_worker_task_start
-tsdb_dir
-tsdb_layout
-tsdb_std_args
-update_interval
-vc_coats_std_args
-verbose
```

Description:

Gets value of a configuration option.

Parameters

Option	Description
-campaign	Returns the name of the fault campaign.
-compress_task_logs	Returns the status of automatic compression of task logfiles.
-confirmation	Show the setting for confirmation when a data removal command is executed. If enabled, the user is asked to confirm the data removal after executing the command.
-delete_successful_task_logs	Display the setting to show if logs from tasks that completed successfully will be automatically deleted upon completion of the task.
-fc	Alias for -campaign.

Option	Description
-fdb_connection_timeout	Show the timeout for connecting to the FDB server.
-fdb_host	Print the name of the FDB server host machine.
-fdb_port	Get the system port number where the FDB server is running.
-fdb_project	Show the name of the FDB project or fault database.
-fdb_server	Display the server name and port in host:port format.
-fdb_storage_path	Path for FDB storage location. The default is <code>fdb</code> .
-fsim_layout	Layout configuration for fault simulation output (fsim command).
-fsim_mode	Fault simulation mode. Possible values are <code>concurrent</code> , <code>serial</code> , <code>serial_fsdb_ref</code> , <code>serial_no_ref</code> . Where, <code>concurrent - 2046</code> faults can be run in a single simulation. It is the fastest method and the default method. <code>serial_fsdb_ref/serial - serial</code> fault simulation that uses a reference FSDB for the GM values to compare GM and FM. <code>serial</code> is an alias to <code>serial_fsdb_ref</code> . <code>serial_no_ref</code> - serial fault simulation that runs without a GM as reference, so there is no check on strobe locations during fault simulation. The status of a fault must be set by user through <code>\$fs_set_status/\$fs_drop_status</code> functions or after simulation from outside of <code>simv</code> . Currently, <code>-fsim_mode serial</code> is aliased to <code>-fsim_mode serial_fsdb_ref</code> as this is the most often used serial case.
-fsim_std_args	Fault simulation standard arguments.
-global_max_jobs	Maximum number of jobs which can be created on all host groups.
-max_faults_per_fsim_task	Maximum number of faults which could be handled by single FCM task.
-max_task_reruns	Maximum number of rerun/retries for each failing FCM task.
-min_faults_per_fsim_task	Minimum number of faults which could be handled by a single FCM task.
-post_fsim_job_proc	Procedure which should be executed after each <code>fsim</code> job.
-report_stats	Show if <code>-reportstats</code> will be added to simulation runtime command.
-run_sim_in_sub_dir	Show the setting for running simulations in the local task sub-directory. By default, the simulations are run in the same directory where <code>vc_fcm</code> is started.

Option	Description
-scheduling_params	How to schedule faults/testcases/failure modes in fault sim phase.
-session_dir	FCM session directory path.
-show_campaign_on_prompt	Show/hide campaign configuration on/from prompt.
- show_cov_equations_layout	Layout configuration for show_cov_equations command.
-show_det_pts_layout	Layout configuration for show_detection_points command.
- show_fault_results_layout	Layout configuration for show_fault_results command.
-show_faults_layout	Layout configuration for show_faults command.
-show_fcs_layout_detail	Layout configuration for detail data in show_campaigns command.
-show_fcs_layout_main	Layout configuration for main data in show_campaigns command.
- show_fdb_project_on_prompt	Show/hide FDB project configuration on/from prompt.
- show_fdb_server_on_prompt	Show/hide FDB project configuration on/from prompt.
-show_fms_layout_detail	Layout configuration for detail data in show_campaigns command.
-show_fms_layout_main	Layout configuration for main data in show_failure_modes command
-show_obs_pts_layout	Layout configuration for show_observation_points command.
-show_projects_layout	Layout configuration for show_projects command.
- show_promotion_tables_layout	Layout configuration for show_promotion_tables command
-show_sms_layout	Layout configuration for show_safety_mechanisms command.
- show_status_groups_layout	Layout configuration for show_status_groups command
-show_statuses_layout	Layout configuration for show_statuses command.

Option	Description
-show_tcs_layout	Layout configuration for show_testcases command.
-show_users_layout	Layout configuration for show_users command.
-show_worker_task_start	Show/hide when a worker or task starts running on a machine
-tsim_fsdb_dir	Path for FSDB storage, default is <code>tsim_fsdb_dir</code> in the current working directory
-tsim_layout	Layout configuration for toggle simulation output (<code>coats</code> and <code>fsim</code> command)
-tsim_std_args	Toggle simulation standard arguments
-update_interval	Time interval in seconds of periodic status updates (0 = disabled).
-vc_coats_std_args	Coats standard arguments.
-verbose	Prints extra information related to the configurations.

Example:

```
get_config -fsim_std_args
```

getobj_attributes

Synopsis:

```
getobj_attributes
-key <list>
[-testcases | -tcs] <list of string>
```

Description:

Returns a list of `ObjAttributes` objects from the FDB object. Requires the FDB to be connected (see [fdb_connect](#)) and the campaign option to be set (see [set_campaign](#)).

Parameters

Option	Description
-key <list>	When <code>-key</code> option is provided, only the defined keys (if it exists in the FDB object) are returned. If the key is not found, a warning is generated. An empty key, "", returns all the keys.

Option	Description
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>-testcases <list of string></code>	List of testcase(s) defined through the create_testcases command.

Tcl Object:

```
::oo::class create ObjAttributes {
    variable obj_id, keys, values;
    method getObjId {} { return $obj_id } # task ID or testcase name
    method getKeys {} { return $keys }
    method getValues {} { return $values }
    method show {} { # prints object contents in readable format
}
```

Example:

```
getobj_attributes -testcase TC_001
```

getobj_campaigns

Synopsis:

```
getobj_campaigns
[-campaign | -fc] <string>
```

Description:

Returns a list of fault campaign objects. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign. When provided, only the defined campaign is returned in the list.
<code>-fc <string></code>	Alias for <code>-campaign</code> .

Tcl Object:

```
::oo::class create FaultCampaign {
    variable name dut creation_tool unsampled_faults creation_time;
    variable fault_count fm_names sm_names test_names;
    variable raw_result_count, cov_eq_count;
```

```

method getName {} { return $name };
method getDut {} { return $dut };
method getCreationTool {} { return $creation_tool};
method getUnsampledFaults {} { return $unsampled_faults};
method getFaultCount {} { return $fault_count };
method getFmNames {} { return $fm_names };
method getSmNames {} { return $sm_names };
method getTestNames {} { return $test_names };
method getRawResultCount {} { return $raw_result_count };
method getCoverageEquationCount {} { return $ce_count };
method show {} { # print object contents in readable format}
}
  
```

Example:

getobj_campaigns

getobj_ces

Alias for [getobj_cov_equations](#)

getobj_cov_equations

Synopsis:

```

getobj_cov_equations
[-campaign | -fc] <string>
-name <string>
  
```

Description:

Returns a list of coverage equations objects.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-name <string>	Name of the equation.

Tcl Object:

```

::oo::class create CoverageEquation {
  variable name equation fc_name;
  method getName {} { return $name }
  
```

```

method getEquation {} { return $equation }
method getFcName {} { return $fc_name }
method show {} { # print object contents in readable format }
}
  
```

Example:

```
getobj_cov_equations
```

getobj_detection_points

Synopsis:

```
getobj_detection_points
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
[-safety_mechanism | -sm] <string>
```

Description:

Returns a list of detection point objects.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-failure_modes <list of string>	Name of the safety mechanism.
-fc <string>	Alias for -campaign.
-fms <list of string>	Alias for -failure_modes.
-safety_mechanism <string>	Name of the safety mechanism.
-sm <string>	Alias for -safety_mechanism.

Tcl Object:

```
::oo::class create StrobePoint {
variable location fm_names sm_names type;
method getLocation {} { return $location }
method getFmNames {} { return $fm_names }
method getSmNames {} { return $sm_names }
method getType {} { return $type }
method show {} { # print object contents in readable format }
}
```

Example:

```
getobj_detection_points -sm SM_001
```

getobj_dps

Alias for [getobj_detection_points](#)

getobj_failure_modes

Synopsis:

```
getobj_failure_modes
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
```

Description:

This command returns a list of failure mode objects. It requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
-campaign <string>	Name of the fault campaign. When campaign argument is not provided, failure modes from all campaigns in the project are listed.
-failure_modes <list of string>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file. When <code>-failuremode</code> or <code>-fm</code> option is provided, only the defined FMs are returned in the list.
-fc <string>	Alias for <code>-campaign..</code>
-fms <list of string>	Alias for <code>-failure_modes</code> .

Tcl Object:

```
::oo::class create FailureMode {
variable name fault_count result_count sm_names obs_pts det_pts fc_names;
method getName {} {return $name};
method getFaultCount {} {return $fault_count};
method getSmNames {} {return $sm_names};
method getObsLocations {} {return $obs_pts}; # returns string of paths
method getDetLocations {} {return $det_pts}; # returns string of paths
method getFcNames {} {return $fc_names};
```

```
method show {}; { # print object contents in readable format }
```

Example:

```
getobj_failure_modes -failure_modes DEFAULT
```

getobj_fault_results

Synopsis:

```
getobj_fault_results

[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-fids <list of int>
-only_collapsed
-only_prime
-raw
-result <list>
-status <list>
[ -testcases | -tcs] <list of string>
-tool <list>
```

Description:

Returns a list of Fault Result objects. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-failure_modes <list of string>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
-fc <string>	Alias for -campaign.
-fids <list of int>	Get the list of Fault IDs. See report for a description of how to obtain fault IDs.
-fms <list of string>	Alias for -failure_modes.
-only_collapsed	Return only collapsed faults. The status returned is the status of the prime fault this fault collapses to.
-only_prime	Return only prime faults.

Option	Description
-raw	Outputs individual results per test and per tool (results without promotion).
-result <list>	Show results only for the results given. Valid results are NOT_EXECUTED, PASSED, TEST_FAILED, TEST_ERROR, PROMOTION_ERROR, INVALID_FAULT_ID and INCONSISTENT_RESULT.
-status <list>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.
-tcs <list of string>	Alias for -testcases.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.
-tool <list>	Get results for the tool provided. Valid tools are CERTITUDE, ZOIX, FCC, ZEBU, FTA, VCFORMAL, VERDI, VCS, REPORT, GLOBAL, VC_FSM, SER_CREATED_RESULT, DYNAMIC_TESTABILITY and ACSIM

Tcl Object:

```
::oo::class create FaultResult {
variable fid, fc_name, fm_name, tc_name, obs_time, obs_pt, det_time,
det_pt;
variable inject_start_time, inject_end_time, fault_status, result_status;
variable error_message, tool, is_prime;
method getFid {} { return $fid };
method getFcName {} { return [internal_get_fc_name $fc_ptr] };
method getFmName {} { return [internal_get_fm_names $fm_ptr] };
method getTcName {} { return [internal_get_tc_name $tc_ptr] };
method getObsLocation {} { return $obs_pt };
method getDetLocation {} { return $det_pt };
method getToolName {} { return $tool };
method getFaultStatus {} { return $fault_status };
method getResultStatus {} { return $result_status };
method getErrorMessage {} { return $error_message };
method getObsTime {} { return $obs_time };
method getDetTime {} { return $det_time };
method getInjectStartTime {} { return $inject_start_time };
method getInjectEndTime {} { return $inject_end_time };
method getIsPrime {} { return $is_prime };
method show {} { # prints object contents in readable format };
}
```

Example:

```
getobj_fault_results -fids 12345 -fc FC_001 -testcases TC_001 -result
PASSED
```

getobj_faults

Synopsis:

```
getobj_faults
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
```

Description:

Returns a list of fault objects. Requires the FDB to be connected (see [fdb_connect](#)).

When the `-campaign` option is not provided, the campaign configuration is used. When both `-campaign` and `-failure_modes` are not provided, all faults from all campaigns in the connected project are returned.

Parameter

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .

Tcl Object:

```
::oo::class create Fault {
variable fid, fc_name, fm_names, locations, bridge_locations;
variable init_status, type, class, mfo_parent_fid, collapsing_fid;
method getFid {} { return $fid }
method getFcName {} { return fc_name }
method getFmNames {} { return fm_names } # empty for sub faults
method getLocation {} { return locations }
method getBridgeLocation {} { return bridge_locations }
method getInitStatus {} { return init_status } # empty for master fault
method getFaultType {} { return $type } # empty for master fault
method getFaultClass {} { return $class } # empty for master fault
method getMfoParentFid {} { return $mfo_parent_fid }
method getCollapsingFid {} { return $collapsing_fid }
```

```
method show {} { # prints object contents in readable format }
```

Example:

getobj_faults

getobj_fcs

Alias for [getobj_campaigns](#).

getobj_fms

Alias for [getobj_failure_modes](#).

getobj_host_groups

Synopsis:

getobj_host_groups

[-host_group | -hg] <list>

Description:

Returns a list of GridHostGroup objects.

Parameters

Option	Description
-hg <list>	Alias for -host_group.
-host_group <list>	View only the specified host group. The default is to view all host groups.

Tcl Object:

```
::oo::class create GridHostGroup {
variable name, group_max_jobs, referenced_host_info_ids;
method getName {} { return $name }
method getGroupMaxJobs {} { return $group_max_jobs }
method getRefHostInfoIds {} { return $referenced_host_info_ids }
method show {} { # prints object contents in readable format }
}
```

Example:

```
getobj_host_groups
```

getobj_host_infos**Synopsis:**

```
getobj_host_infos
-id <list>
```

Description:

Returns a list of `GridHostInfo` objects.

Parameters

Option	Description
<code>-id <list></code>	Used to return only the specified host information.

Tcl Object:

```
::oo::class create GridHostInfo {
variable id, grid_type, command, group_name, work_dir, remote_host_name;
method getId {} { return $id }
method getGridType {} { return $grid_type }
method getCommand {} { return $command }
method getNamesOfGroups {} { return $list_of_names }
method getRemoteHostName {} { return $remote_host_name }
method show {} { # prints object contents in readable format }
}
```

Example:

```
getobj_host_infos
```

getobj_metadata**Synopsis:**

```
getobj_metadata
[-campaign | -fc] <string>
-index <list>
-name <string>
```

Description:

The VC_FCM getobj_metadata command allows users to access TCL class representations of metadata from a fault campaign.

Parameter

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-index <list>	List of indexes corresponding to metadata entries.
-name <string>	The name of the metadata block which is requested to be returned.

Tcl Object:

```
::oo::class create Metadata {
variable name obj_id keys values;
method getName {} { return $name }
method getObjId {} { return $obj_id }
method getKeys {} { return $keys }
method getValues {} { return $values }
method show {} { # prints object contents in human readable format }
}
```

Example:

```
getobj_metadata -name MD_001
```

getobj_observation_points

Synopsis:

```
getobj_observation_points
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
```

Description:

Returns a list of observation point objects.

Parameter

Option	Description
-campaign <string>	Name of the fault campaign.
-failure_modes <list of string>	Name of the failure mode.
-fc <string>	Alias for -campaign.
-fms <list of string>	Alias for -failure_modes.

Tcl Object:

```
::oo::class create StrobePoint {
variable location fm_names sm_names type;
method getLocation {} { return $location }
method getFmNames {} { return $fm_names }
method getSmNames {} { return $sm_names }
method getType {} { return $type }
method show {} { # print object contents in readable format }
}
```

Example:

getobj_observation_points

getobj_ops

Alias for [getobj_observation_points](#).

getobj_projects

Synopsis:

```
getobj_projects
-fdb_server <string>
```

Description:

Return a list of FdbProject objects.

Parameters

Option	Description
-fdb_server <string>	Specifies the name of the FDB server (host:port).

Tcl Object:

```
::oo::class create FdbProject {
variable name owner access_privileges, size, description, fc_count;
method getName {} { return $name };
method getOwner {} { return $owner };
method getSize {} { return $size };
method getDescription {} { return $description };
method show {} { # prints object contents in readable format }
}
```

Example(s):

```
getobj_projects -fdb_server myhost:123456
```

get_obj_promotion_tables

Synopsis:

```
getobj_promotion_tables
-all
[-campaign | -fc] <string>
-promoted_status <list>
-status <list>
```

Description:

Returns a list of promotion table objects.

Parameters

Option	Description
-all	List all built-in and user defined statuses.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-promoted_status <list>	List of promoted statuses to filter.
-status <list>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.

Tcl Object:

```
::oo::class create PromotionTable {
variable fc_name promotions;
```

```
method getFcName {} { return $fc_name }
method getPromotedResult {old new} { return [dict get [dict get
  $promotions $old] $new]}
method show {} { # prints object contents in readable format }
}
```

Example:

`getobj_promotion_tables`

getobj_pts

Alias for [get_obj_promotion_tables](#).

getobj_safety_mechanisms

Synopsis:

```
getobj_safety_mechanisms
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
[-safety_mechanism | -sm] <list>
```

Description:

Returns a list of `SafetyMechanism` objects. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign. When <code>-campaign/-fm</code> arguments are not provided, safety mechanisms from all <code>-campaigns/-fms</code> in the project are included.
<code>-failure_modes <list of string></code>	Name of the failure mode.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-safety_mechanism <list></code>	List of safety mechanism(s). The list is a <code>tcl</code> formatted list of safety mechanism(s) defined through the SFF file. When <code>-safety_mechanism</code> or <code>-sm</code> option is provided, only the defined SM is returned to the list.

Option	Description
-sm <list>	Alias for -safety_mechanism..

Tcl Object:

```
::oo::class create SafetyMechanism {
variable name det_pts fc_names fm_names;
method getName {} { return $name };
method getDetPts {} { return $det_pts } # returns string of path
method show {} { # prints object contents in readable format}
}
```

Example:

```
getobj_safety_mechanisms -campaign FC_001 -sm SM_001
```

getobj_sgs

Alias for [getobj_status_groups](#)

getobj_sms

Alias for [getobj_safety_mechanisms](#)

getobj_status_groups

Synopsis:

```
getobj_status_groups
-all
[-campaign | -fc] <string>
-name <string>
-redefined <list>
```

Description:

Returns a list of fault status group objects.

Parameters

Option	Description
-all	List all built-in and user defined statuses.
-campaign <string>	Name of the fault campaign.

Option	Description
-fc <string>	Alias for -campaign..
-name <string>	Name of the status group.
-redefined <list>	List of old names to filter by (only matches if status group has been redefined).

Tcl Object:

```
::oo::class create FaultStatusGroup {
variable name description oldName fc_name members;
method getName {} { return $name }
method getDescription {} { return $description }
method getOldName {} { return $oldName; }
method getFcName {} { return $fc_name; }
method getStatusNames {} { return $members; }
method show {} { # prints object contents in readable format }
}
```

Example(s):

getobj_status_groups

getobj_statuses

Synopsis:

```
getobj_statuses
-all
[-campaign | -fc] <string>
-status <list>
-redefined <list>
```

Description:

Returns a list of fault status objects.

Parameters

Option	Description
-all	List all built-in and user defined statuses.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign..

Option	Description
-status <list>	List of statuses to filter.
-redefined <list>	List of old statuses to filter by (only matches if status has been redefined).

Tcl Object

```
::oo::class create FaultStatus {
variable status oldStatus description fc_name;
method getStatus {} { return $status }
method getOldStatus {} { return $oldStatus }
method getDescription {} { return $description }
method getFcName {} { return $fc_name }
method show {} { # prints object contents in readable format }
```

Example:

getobj_statuses

getobj_tcs

Alias for [getobj_testcases](#)

getobj_testcases

Synopsis:

getobj_testcases

```
[-campaign | -fc] <string>
[-testcases | -tcs] <list of string>
```

Description:

Returns a list of Testcase objects.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-tcs <list of string>	Alias for -testcases.

Option	Description
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.

Tcl Object:

```
::oo::class create Testcase {
variable name description command args pre_script post_script;
variable name campaign_name max_faults timeout simulation_runtime;
variable name coats_total_analyzed coats_not_strobed;
variable name coats_not_observable coats_not_controllable;
variable name coats_fault_reduction fsdb_path daidir_path;
variable name fault_injection_time tsim_args fsim_args;
method getName {} { return $name }
method getDescription {} { return $description }
method getCommand {} { return $command }
method getArgs {} { return $args }
method getPreScript {} { return $pre_script }
method getPostScript {} { return $post_script }
method getCampaignName {} { return $campaign_name }
method getMaxFaults {} { return $max_faults }
method getTimeout {} { return $timeout }
method getSimulationRuntime {} { return $simulation_runtime }
method getCoatsTotalAnalyzed {} { return $coats_total_analyzed }
method getCoatsNotStrobed {} { return $coats_not_strobed }
method getCoatsNotObservable {} { return $coats_not_observable }
method getCoatsNotControllable {} { return $coats_not_controllable }
method getCoatsFaultReduction {} { return $coats_fault_reduction }
method getFsdbPath {} { return $fsdb_path }
method getDaidirPath {} { return $daidir_path }
method getFaultInjectionTime {} { return $fault_injection_time }
method getTsimArgs {} { return $tsim_args }
method getFsimArgs {} { return $fsim_args }
method show {} { # prints object contents in readable format }
}
```

Example:

getobj_testcases

getobj_users

Synopsis:

```
getobj_users

-fdb_project <string>
-fdb_server <string>
-user <string>
```

Description:

Returns a list of User objects.

Parameters

Option	Description
-fdb_project <string>	Set the name of the FDB project or fault database.
-fdb_server <string>	Specifies the name of the FDB server (host:port).
-user <string>	User name to search for.

Tcl Object:

```
::oo::class create User {
variable name roles projects;
method getName {} { return $name }
method getRoles {} { return $roles }
method getProjects {} { return $projects }
method show {} { # prints object contents in readable format}
}
```

Example:

```
getobj_users -user foo
```

obj_mgmt**Synopsis:**

```
obj_mgmt
-class <list>
-destroy
-get
-obj <list>
-show
```

Description:

Perform the described operations on Tcl object(s).

Parameters

Option	Description
-class <list>	The <code>-class</code> option is used to restrict the operation only on the defined Tcl class names (for example, <code>FdbProject</code> , <code>FailureMode</code> , and so on). An invalid class name is not an error.
-destroy	Destroy the specified object(s).
-get	Return the specified object(s) names.
-obj <list>	The <code>-obj</code> option is used to restrict the operation only on the defined Tcl object names (for example, <code>::oo::Obj32</code>). An invalid object name is an error. When <code>-obj</code> option is not given, all objects are considered for the operation.
-show	Prints object information in tabular readable format on <code>stdout</code> .

Example:

```
obj_mgmt -obj ::oo::Obj46 -show
```

remove_attribute

Synopsis:

```
remove_attribute
-key <list>
[-testcases | -tcs] <list of string>
```

Description:

Removes the defined key attributes from the defined object from the FDB.

Parameters

Option	Description
-key <list>	When <code>-key</code> option is provided, only the defined keys (if it exists in the FDB object) are returned. If the key is not found, a warning is generated. An empty key, "", returns all the keys.
-tcs <list of string>	Alias for <code>-testcases</code> .
-testcases <list of string>	List of <code> testcase(s)</code> defined through the create_testcases command.

Example:

```
remove_attribute -tc TC -key mykey1
```

remove_campaign

Synopsis:

```
remove_campaign
[-campaign | -fc] <string>
-force
```

Description:

Removes one fault campaign from the current project (including all fault campaign results). FCM will interactively ask for confirmation unless `-force` is used. Requires the FDB to be connected (see [fdb_connect](#)). A set campaign cannot be removed. Use [unset_campaign](#) command to unset it first.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign. The campaign name is mandatory for this command.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-force</code>	Bypass the interactive confirmation to remove the campaign.

Example:

```
remove_campaign -fc FC_001
```

remove_fault_results

Synopsis:

```
remove_fault_results
-all
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-fids <list of int>
-only_collapsed
-only_prime
-result <list>
-status <list>
```

```
[-testcases | -tcs] <list of string>
-tc <list>
```

Description:

This command removes results from the fault campaign. Select specific results based on a combination of criteria (an exhaustive selection), for example, tests, tool, failure mode, and so on. When multiple options are provided, they perform AND operation. When no selections (or `-all`) are provided, an error is generated. In interactive mode, a confirmation is required.

Parameters

Option	Description
<code>-all</code>	The option <code>-all</code> can be used to erase all results belonging to the campaign.
<code>-campaign <string></code>	Name of the fault campaign. When campaign option is not provided, the active campaign is used.
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fids <list of int></code>	Remove the list of Fault IDs. See report for a description of how to obtain fault IDs.
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-only_collapsed</code>	Remove only collapsed faults. The status returned is the status of the prime fault this fault collapses to.
<code>-only_prime</code>	Remove only prime faults.
<code>-result <list></code>	Show results only for the results given. Valid results are NOT_EXECUTED, PASSED, TEST_FAILED, TEST_ERROR, PROMOTION_ERROR, INVALID_FAULT_ID and INCONSISTENT_RESULT.
<code>-status <list></code>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>-testcases <list of string></code>	List of testcase(s) defined through the create_testcases command.

Option	Description
-tool <string>	Get results for the tool provided. Valid tools are CERTITUDE, ZOIX, FCC, ZEBU, FTA, VCFORMAL, VERDI, VCS, REPORT, GLOBAL, VC_FSM, SER_CREATED_RESULT, DYNAMIC_TESTABILITY and ACSIM

Example:

```
remove_fault_results -tool VCFORMAL -fm FM_003
```

remove_fc

Alias for [remove_campaign](#).

remove_metadata

Synopsis:

```
remove_metadata
-all
[-campaign | -fc] <string>
-index <string>
-name <string>
```

Description:

The vc_fcm remove_metadata command allows users to remove metadata from a fault campaign.

Parameters

Option	Description
-all	Remove all metadata from campaign.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign..
-index <string>	List of indexes corresponding to metadata entries.
-name	The name of the metadata block which is requested to be removed.

Example(s):

```
remove_metadata -index {1 2 3}
```

remove_project

Synopsis:

```
remove_project
-fdb_project <string>
-fdb_server <string>
```

Description:

Removes the project. If you are in interactive mode, the command asks for your confirmation. You can reuse a project if there are requests to remove the currently opened project. If required, you can disconnect from the project first using [fdb_disconnect](#) command.

Parameters

Option	Description
-fdb_project <string>	The name of the FDB project or fault database.
-fdb_server <string>	Specifies the name of the FDB server (host:port).

Example

```
remove_project
```

remove_tcs

Alias for [remove_testcases](#).

remove_testcases

Synopsis:

```
remove_testcases
-all
[-campaign | -fc] <string>
-force
[-testcases | -tcs] <list of string>
```

Description:

Removes testcase(s) in FDB.

Parameters

Option	Description
-all	Remove all testcases.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-force	Force removes testcase with results and tasks.
-tcs <list of string>	Alias for -testcases.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.

Example

```
remove_testcases -fc FC_001 -force -all
```

remove_user

Alias for [user_mgmt](#).

report**Synopsis:**

```
report

-aggregatedsummary
-args <string>
[-campaign | -fc] <string>
-collapseoff
-csv
[-failure_modes | -fms] <list of string>
-faultlimit <integer>
-faultstatus <list>
-filterbyattributes <string>
-flat
-format <string>
-gz
-hidenoattributes
-hierarchical [<integer>]
```

```
-localhost
-log <filename>
-metadataonly
-nostatusdefinitions
-overwrite
-primesonly
-print
-promotedsummary
-report <filename>
-showallattributes
-showexcludedfaults
-showfaultid
-showfub
-showmetadata
-showmetadataid
-showsimdetails
-showspecifiedattributes <string>
-showtestresults <list>
-showtimingid
-sort <string>
-summaryonly
-summarystyle
-targeted <integer>
-tests <list of string>
[-testcases | -tcs] <list of string>
-tool <string>
-unselected [<string>]
```

Description:

Create a coverage report. See the parameter below for a detailed list of options to include in the report.

Parameters

Option	Description
-aggregatedsummary	Create a combined summary across all fault lists. By default, duplicate faults across fault lists will have each status counted once for each fault list, if only one status is desired per unique fault see <code>-promotedsummary</code> .
-args <string>	Quoted string contains the arguments to be passed to <code>vc_fdb_report</code> . This argument cannot be used with other arguments of the <code>report</code> command.
-campaign <string>	Name of the fault campaign.
-collapseoff	Turn off collapse marking of "--" and show the corresponding prime fault's status.

Option	Description
-csv	Create CSV tables of fault lists and fault list summaries. Will create CSV files in directory <report_name>_csv_files.
-failure_modes <list of string>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
-faultlimit <integer>	Hide child instances that contain less than a specified fault count. Useful for readability when hierarchical reports are dominated large areas of the design with few faults.
-faultstatus <list>	Optional argument to display only faults with specified statuses. Use a tcl formatted list of individual statuses or status groups. When a status group is encountered all statuses in that status group will be marked for display.
-fc <string>	Alias for -campaign.
-filterbyattributes <string>	Display faults in the fault list that contain attributes that match the specified filter.
-flat	Print a single descriptor for each transient fault, rather than combine transients that only differ by cycle number
-fms <list of string>	Alias for -failure_modes.
-format <string>	Create the report in the provided format. Valid options are standard (default), tetramax, fastscan, and verifault.
-gz	Write report as a compressed gzip file. It is recommended to use -report and name the report with postfix ".gz".
-hidenoattributes	Display only faults that contain attributes in each fault list.
-hierarchical [<integer>]	Create a hierarchical report. Optional argument specifies the levels of hierarchy to report on. The default is to report on 1 level. Specify 0 to report on the entire hierarchy.
-localhost	Run the reporting tasks on the local host instead of using the defined compute farms.
-log <filename>	Specify a filename for the logfile. Default is vc_fdb_report.log.
-metadataonly	Print only the metadata of the SFF report. Useful for a quick look at the history of the campaign, often used with -print.
-nostatusdefinitions	Omit the StatusDefinitions block from the report.
-overwrite	Overwrite an existing file if the file already exists.

Option	Description
-primesonly	Limit report to prime faults.
-print	Print output of report to screen and log file.
-promotedsummary	Change methodology of total summaries. By default, when the same fault spans multiple fault lists <code>vc_fdb_report</code> will count a status for each fault per fault list. This option is for enabling the methodology of counting faults that span failure modes only once and to use the promoted status between the fault lists.
-report <filename>	Specify the filename for creation of the report. By default <code>vc_fdb_report</code> will use <code><campaign>.report.sff</code> , unless -test was specified, then it will be <code><campaign>_<test>.report.sff</code> .
-showallattributes	Display all the attributes attached to faults.
-showexcludedfaults	Display faults with excluded statuses in the fault list. By default, faults with excluded statuses are not displayed.
-showfaultid	Display a comment after each descriptor showing its associated fault ID.
-showfub	Display <code>FunctionalBlock</code> . <code>FunctionalBlock</code> was replaced with <code>TestList</code> in SFF reports. However, it may be useful to get the old style for importing to other SFF compilers that still require the fub information.
-showmetadata	Print fault campaign metadata in report. The metadata block contains a logged history of the campaign.
-showmetadataid	Print fault campaign metadata in report with commented indices. The metadata block contains a logged history of the campaign. The indices are useful for add/removal of entries.
-showsimdetails	Optional argument to display "Attempt" count, and "NumPot" count (if applicable) for each fault. <code>NumPot</code> is only displayed when no user defined fault statuses were used in the campaign. <code>Attempt</code> count is the number of times the fault was attempted for fault simulation. <code>NumPot</code> is the number of potential results (for example, PD,PT) that occurred in this campaign run for each fault.
-showspecifiedattributes <string>	(Takes a filter to show specified attributes. The switch takes any number of keys or key/value pairs, separated by '+'. To specify key/value pair use <code><key>=<value></code> . Use brackets when filter will contain spaces or comma's. Example: To show all attributes that match key2, key5/"value 9", and key10: <code>-showspecifiedattributes {key2+key5=value 9+key10}</code>)

Option	Description
-showtestresults <list>	Display the non-promoted test results in a comment following the fault description. Specify the status, or status group to display.
-showtimingid	Optional argument used to display every transient fault's timing ID in its descriptor. For example, ND ~ ("timingid", 25) {PRIM "..."}
-sort <string>	-sort arg Optional argument to specify sorting type. Valid arguments: hier (sort by hierarchy), (sort by status), and (sort by descriptor starting from fault type (~,0,1).
-summaryonly	Print only the summaries of the SFF report. Useful for quick counts, often used with -print.
-summarystyle	Optional argument to specify an alternative summary style. Different summaries are basic, prime, coverage, and zoix. Default summary is basic.
-targeted <integer>	Create the targeted report using the unselected report. Requires the -unselected option and an argument for the number of blocked or uncontrolled locations to include in the report. 0 means all the locations will be included.
-tcs <list of string>	Alias for -testcases.
-tests <list of string>	Optional argument to either get results from a test level if only a single test was specified. Example: -test <test_name>. Otherwise, multiple test names may be specified separated by a comma and the promoted results between the relevant tests will be calculated. Example: -test {test1,test2,test3}.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.
-tool <string>	Get results for the tool provided. Valid tools are CERTITUDE, ZOIX, FCC, ZEBU, FTA, VCFORMAL, VERDI, VCS, REPORT, GLOBAL, VC_FSM, SER_CREATED_RESULT, DYNAMIC_TESTABILITY and ACSIM
-unselected [<string>]	Create an unselected report. Optional argument specifies the mode of unselected. Different options are: fault, location, observability and controllability.

Example:

```
report -campaign fcl -report fsim_v.rpt -overwrite -sort desc
```

set_campaign

Synopsis:

```
set_campaign
[-campaign | -fc] <string>
```

Description

This command sets the active campaign for later commands. The campaign must exist in the FDB. Requires the FDB to be connected (see [fdb_connect](#)). Returns true when successful, false otherwise. To view the currently active campaign name, use [get_config](#) -campaign option.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.

Example(s):

```
set_campaign -campaign FC_001
```

set_config

Synopsis:

```
set_config
-compress_task_logs <string>
-confirmation <string>
-delete_successful_task_logs <string>
-fdb_connection_timeout <integer>
-fdb_storage_path <directory>
-fsim_layout <string>
-fsim_mode <string>
-fsim_std_args <string>
-global_max_jobs <integer>
-max_faults_per_fsim_task <integer>
-max_task_reruns <integer>
-min_faults_per_fsim_task <integer>
-post_fsim_job_proc <string>
-post_fsim_task_proc <TCL proc>
-report_stats <string>
-run_sim_in_sub_dir <true|false>
```

```

-scheduling_params <string>
-show_campaign_on_prompt <string>
-show_cov_equations_layout <string>
-show_det_pts_layout <string>
-show_fault_results_layout <string>
-show_faults_layout <string>
-show_fcs_layout_detail <string>
-show_fcs_layout_main <string>
-show_fdb_project_on_prompt <string>
-show_fdb_server_on_prompt <string>
-show_fms_layout_detail <string>
-show_fms_layout_main <string>
-show_obs_pts_layout <string>
-show_projects_layout <string>
-show_promotion_tables_layout <string>
-show_sms_layout <string>
-show_status_groups_layout <string>
-show_statuses_layout <string>
-show_tcs_layout <string>
-show_users_layout <string>
-show_worker_task_start <string>
-tsdb_dir <directory>
-tsdb_layout <string>
-tsdb_std_args <string>
-update_interval <integer>
-use_absolute_path_task_details <true|false>
-vc_coats_std_args <string>
-write_fault_results_from_fcm
-write_fault_results_bulk_size <positive integer>
-write_fault_results_interval <seconds>,
  
```

Description:

Sets value for a configuration option.

Parameters

Option	Description
-compress_task_logs <string>	Compress task logs in gzip format. Possible values are true or false.
-confirmation <string>	Ask for confirmation in interactive mode, when a potential data removal command is called. Possible values are true or false.
-delete_successful_task_logs <string>	Delete the successful task logs to save space. Possible values are true or false.
-fdb_connection_timeout <integer>	Time out in seconds for FDB connection. The default is 10 seconds.

Option	Description
<code>-fdb_storage_path <directory></code>	Path for FDB storage location. The default is <code>fdb</code> .
<code>-fsim_layout <string></code>	Layout configuration for fault simulation output (<code>fsim</code> command).
<code>-fsim_mode <string></code>	Specify the fault simulation mode. Default is <code>concurrent</code> . Possible values are <code>concurrent</code> , <code>serial</code> , <code>serial_fsdb_ref</code> , <code>serial_no_ref</code> . Where, <code>concurrent</code> - 1000s of faults can be run in a single simulation. It is the fastest method and default method). <code>serial</code> - one fault at a time. Recommended for running Illegal group fault statuses such as IA, IF, IP. Also, to run SDF fault simulation in serial mode. <code>serial_fsdb_ref</code> - serial fault simulation that uses a reference FSDB for the GM values to compare GM and FM. <code>serial_no_ref</code> - serial fault simulation that runs without a GM as reference, so no strobing happens. The status of a fault must be set by to <code>\$fs_set_status/\$fs_drop_status</code> functions or after simulation from outside of <code>simv</code> , similar to the Synopsys Certitude method. Currently, <code>-fsim_mode serial</code> is aliased to <code>-fsim_mode serial_fsdb_ref</code> as this is the most often used serial case.
<code>-fsim_std_args <string></code>	Standard arguments for all fault simulation tasks.
<code>-global_max_jobs <integer></code>	Global maximum number of grid jobs which can be run in parallel at a time for all host groups. Only enforced when it is set by the user, the default is 4.
<code>-max_faults_per_fsim_task <integer></code>	Maximum number of faults assigned per <code>fsim</code> task. The default is 2046.
<code>-max_task_reruns <integer></code>	Maximum number of retries for FCM tasks when they fail. The default is 2.
<code>-min_faults_per_fsim_task <integer></code>	Minimum number of faults which could be handled by single FCM task.
<code>-post_fsim_job_proc <string></code>	Procedure that defines user action at the end of every <code>fsim</code> task.

Option	Description
-post_fsim_task_proc <TCL proc>	<p>Calls a TCL proc whenever <code>fsim task</code> is completed. The TCL proc can be defined in the FCM script or in a file and be sourced in the FCM script. The TCL proc must exactly use these proc arguments in the given order: <code>task_dir task_id task_status testcase failure_mode fids_status</code> Usage</p> <p>Example: It outputs all arguments to the screen and then changes the first fault results to <code>IX.proc post_fsim_hook {task_dir task_id task_status testcase failure_mode fids_status} { puts "task_dir: \$task_dir" puts "task_id: \$task_id" puts "task_status: \$task_status" puts "testcase: \$testcase" puts "failure_mode: \$failure_mode" puts "fids_status: \$fids_status" # Overwrite the first result with new status: # First extract the first fid/status pair: set first_result [lindex \$fids_status 0] # Extract the fid from the pair: set fid [lindex \$first_result 0] puts "fids: \$fid" # Now write a new fault status "IX" for this fm / tc / fid combination: write_fault_results -fm \$failure_mode -tc \$testcase -fids \$fid -status "IX" puts "TCL proc end"}set_config -post_fsim_task_proc post_fsim_hook</code></p>
-report_stats <string>	Add <code>-reportstats</code> to <code>simv</code> , <code>default: true</code> .
-run_sim_in_sub_dir <true false>	Run all simulations in dedicated sub-directory. <code>false: simv</code> is executed at the current directory and FCM directs the output to the FCM task directory. <code>true: simv</code> is run in the FCM created task directory and the output also goes there. The default behavior is <code>false</code> . For testbenches that need to load memory files, use the default value <code>false</code> or absolute path in Verilog. To avoid overrides of external files (such as <code>.hex</code> , <code>.dat</code> files and so on), set this option to <code>true</code> .

Option	Description
-scheduling_params <string>	<p>How to schedule faults/testcases/failure modes during fault sim. This option considers the following values:</p> <ul style="list-style-type: none"> tc_selection:fmsh- Picks the best test and initially runs all possible faults on this test. This option can still run the next test in parallel, if there are no more fault sim jobs for the original test. disable_same_fault_in_concurrent_testcases - Does not run the same fault in multiple testcases in parallel. This option is applicable if there are no more unique faults. ignore_tsim_time- Considers only testable faults per test to select the best test and does not consider toggle sim time. dont_shuffle - Keeps faults with a similar fault origin (pathname) together. This is the default value for all the shuffle-based options. shuffle always_shuffle - Shuffles the faults picked for a fault sim job (scramble faults). Leads to a more uniform distributed runtime and peak runtime memory of fault sim jobs. However, tends to increase the overall CPU time during fault simulation for GL designs. auto_shuffle - Starts faultsim without shuffling and automatically turns-on shuffling towards the end. This only makes a difference versus dont_shuffle if each worker simulates multiple tasks. The default flip-point is 2.5 of the remaining tasks per worker. <p>Example1: 100k faults, 10 workers, maximum 2k faults per fsim task - at an average of 5 tasks/ worker, initially with 2k faults each. The initial tasks will not be shuffled but later, tasks will be shuffled. Example2: 10k faults, 10 workers, max 2k faults per fsim task -> at an average of 1 task/worker, each with 1k faults. All tasks will be created during startup of fsim phase and none will use shuffling.</p>
-show_campaign_on_prompt <string>	The fdb_server configuration is shown at the prompt, default is true.
- show_cov_equations_layout <string>	Layout configuration for show_cov_equations command.
- show_det_pts_layout <string>	Layout configuration for show_detection_points command.
- show_fault_results_layout <string>	Layout configuration for show_fault_results command.
- show_faults_layout <string>	Layout configuration for show_faults command.
- show_fcs_layout_detail <string>	Layout configuration for detail data in show_campaigns command.
- show_fcs_layout_main <string>	Layout configuration for main data in show_campaigns command.

Option	Description
<code>-show_fdb_project_on_prompt <string></code>	The <code>fdb_project</code> configuration is shown at the prompt, default is <code>false</code> .
<code>-show_fdb_server_on_prompt <string></code>	The <code>fdb_server</code> configuration is shown at the prompt, default is <code>true</code> .
<code>-show_fms_layout_detail <string></code>	Layout configuration for detail data in show_failure_modes command.
<code>-show_fms_layout_main <string></code>	Layout configuration for main data in show_failure_modes command.
<code>-show_obs_pts_layout <string></code>	Layout configuration for show_observation_points command.
<code>-show_projects_layout <string></code>	Layout configuration for show_projects command.
<code>-show_promotion_tables_layout <string></code>	Layout configuration for show_promotion_tables command.
<code>-show_sms_layout <string></code>	Layout configuration for show_safety_mechanisms command.
<code>-show_status_groups_layout <string></code>	Layout configuration for show_status_groups command.
<code>-show_statuses_layout <string></code>	Layout configuration for show_statuses command.
<code>-show_tcs_layout <string></code>	Layout configuration for show_testcases command.
<code>-show_users_layout <string></code>	Layout configuration for show_users command.
<code>-show_worker_task_start <string></code>	Show/hide when a worker or task starts running on a machine, default <code>off</code> .
<code>-tsim_fsdb_dir <directory></code>	Path for FSDB storage, default is <code>tsim_fsdb_dir</code> in the current working directory.
<code>-tsim_layout <string></code>	Layout configuration for toggle simulation output (coats and fsim command).

Option	Description
-tsim_std_args <string>	Standard arguments for all toggle simulation tasks. To generate all value changes in toggle, set the command as follows: Syntax: <code>set_config -tsim_std_args '-tsim=limit_toggle:<option> <other tsim options like fsdb dumping limits>' where,limit_toggle:0 - All signals, no filtering (all value changes on all signals) - warning. Might produce bigger FSDB files, only use if a regular FSDB file is needed. limit_toggle:1 (Default) - All signals with all value changes for strobes, value changes on all other signals are filtered after 0->1, 1->0. limit_toggle:2 - All signals with all value changes filtered (also strobes). Use only with concurrent fault simulation, FSDB file will not be suitable for serial fault simulation. limit_toggle:3 - Keep all strobe signals, no value changes at all for other signals. Use only for serial fault simulation. The FSDB file is not suitable for dynamic testability (vc_coats). When running toggle sim, the FSDB dumping will start at the fault injection time. For example, if fault injection would happen at time 100ns, FSDB dumping starts at 100ns. There is no FSDB dumping up to 100ns. There is no fault injection in toggle simulation. Use this option to replace the default -tsim=limit_toggle:1. Example: <code>set_config -tsim_std_args -tsim=limit_toggle:0</code> will result in having only -tsim=limit_toggle:0 in the toggle sim command line.</code>
-use_absolute_path_task_details <true false>	By default, this option aids the <code>fsim</code> task to specify the <code>task.json</code> file (defines what the <code>fsim</code> task needs to do) with an absolute path and enables to just copy and paste the <code>simv</code> run command to execute it again in a shell. Possible values: True or False.
-update_interval <integer>	Time interval in seconds of periodic status updates. To disable the updates, set to 0.
-vc_coats_std_args <string>	<code>vc_coats</code> standard arguments.
-write_fault_results_from_fcm	Write fault results from FCM (instead of <code>simv</code>) to FDB. Possible values: True or False.
-write_fault_results_bulk_size <positive integer>	Maximum number of fault results to be written in one FDB transaction. Possible values: 0 - disabled any positive integer - enabled
-write_fault_results_interval <seconds>	Maximum interval in seconds before fault results are written to FDB. Possible values: 0 - disabled any positive integer - enabled

Example:

```
set_config -max_global_jobs 10
```

set_fc

Alias for [set_campaign](#).

set_format

Synopsis:

```
set_format  
-format <string>
```

Description:

Sets the format to read fault lists and write reports.

Parameters

Option	Description
-format <string>	The name of the format to use. Possible values include SFF, TetraMax.

Example:

```
set_format -format TetraMax
```

set_submit_cmd

Synopsis

```
set_submit_cmd  
-cmd <string>  
-disable_submit_cmd_check  
-grid_type <string>  
-hosts_file <string>  
-max_jobs <integer>  
-remote_host <string>  
-submit_timeout <integer>  
-task_type <string>  
-worker_submit_delay <integer>  
-worker_submit_size <integer>  
-worker_timeout <integer>
```

Description:

Sets the job submit command for grid/farm. This command is then used to create jobs for FCM tasks in the grid/farm. Returns an ID unique to the submit (`list_submit_cmd` and `show_submit_cmds` commands that can be used to view the submit commands (including this ID)).

Note:

The job submission commands are not stored in the FDB. When same configuration for multiple FCM sessions is required, a Tcl configuration script, which is sourced manually, or RC files are recommended.

Parameters

Option	Description
<code>-cmd <string></code>	The <code>-cmd</code> option is used to define the grid job submission command, for example, <code>qsub -P -light</code> option.
<code>-disable_submit_cmd_check</code>	Disable check and allow any user specified submit command; to manually test jobs can be submitted use the <code>test_grid_config</code> command.
<code>-grid_type <string></code>	The <code>-grid_type</code> option is used to identify the type of grid. Valid options are RSH, SSH, SGE, LSF, PBS, RTDA, and NB. Note: RSH and SSH do not represent grid protocols but can be used to connect and use a machine not on grid. In this case, the submission command can be 'rsh' or 'ssh' (with perhaps login credentials) and the <code>-remote_host</code> option is mandatory.
<code>-hosts_file <string></code>	File that lists machine names for RSH and SSH.
<code>-max_jobs <integer></code>	The <code>-max_jobs</code> option defines how many grid jobs in parallel can be run at a time for this host group. For multiple submit commands, that is, <code>set_submit_cmd</code> option, with same group name and different <code>max_jobs</code> , this value is updated for the defined group. Error is generated when <code>max_jobs</code> is less than 1 or <code>max_jobs</code> is greater than the global maximum (if set) for all groups (see configuration <code>-global_max_jobs</code> in <code>set_config</code>). When <code>max_jobs</code> option is not specified, the global maximum for all groups is used. When global maximum is also not specified, 1 job is created.
<code>-remote_host <string></code>	The <code>-remote_host</code> option is used only when <code>-grid_type</code> is either RSH or SSH or SH. It is used to identify the machine name where rsh/ssh logs in to run the FCM task. For other types of grids, this option is ignored.
<code>-submit_timeout <integer></code>	Maximum submission timeout for a grid job in seconds. Default value is 120, negative value means unlimited.

Option	Description
-task_type <string>	The <code>-task_type</code> takes a list of tools to identify which host groups the job submission command should be logically structured in. The host group(s) will be updated, or created if they do not exist yet, for every task type based on the needs of the task. For example, high memory, specific OS version, and so on. When <code>-task_type</code> is not specified, localhost submission command with host group name 'default' (with <code>work_dir</code> defined by the <code>-localhost_work_dir</code> configuration) is always available. In the presence of existing non-localhost commands, localhost command is not used unless explicitly mentioned (see <code>-localhost</code> option in <code>fsim</code> command in the example). When <code>-task_type</code> is not provided, the submit command is added to the default group. Valid options are <code>default</code> , <code>fcc</code> , <code>coats</code> , <code>tsim</code> , <code>fsim</code> , and <code>report</code> .
-worker_submit_delay <integer>	Period in seconds used to submit up to <code>worker_submit_size</code> tasks concurrently. The default value is 60.
-worker_submit_size <integer>	Maximum number of tasks to be submitted concurrently every period of length <code>worker_submit_delay</code> . Default value is 0 (unlimited).
-worker_timeout <integer>	Maximum timeout for a worker in seconds. Default value is 21600s, 0 means unlimited. That is, a worker is alive as long as tasks are available to be processed.

Example:

```
set_submit_cmd -grid_type SGE -cmd {qsub -b y -cwd -V -l
  os_bit=64,os_version='CS7.0'}
set_submit_cmd -grid_type SSH -cmd {ssh} -remote_host {hostname1
  hostname2 hostname3 ...}
```

Note:

All machines should be on the same network (IP address) so that they can all access the fdb server (started on machine where FCM runs).

show_attributes

Synopsis:

```
show_attributes
-key <list>
[-testcases | -tcs] <list of string>
```

Description:

Display a list of `ObjAttributes` objects from the FDB object in readable format. Requires the FDB to be connected (see [fdb_connect](#)) and the campaign option to be set (see [set_campaign](#)).

Parameters

Option	Description
<code>-key <list></code>	When <code>-key</code> option is provided, only the defined keys (if it exists in the FDB object) are returned. If the key is not found, then it is not the case of an error (in this case a warning is generated). An empty key, "", returns all the keys.
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>-testcases <list of string></code>	List of testcase(s) defined through the create_testcases command.

Example:

```
show_attributes -testcases TC_001
```

show_campaign_summary**Synopsis:**

```
show_campaign_summary
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-primesonly
-summarystyle <string>
[-testcases | -tcs] <list of string>
```

Description:

Prints fault campaign summary.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a <code>tcl</code> formatted list of failure modes defined through the SFF file.

Option	Description
-fc <string>	Alias for -campaign.
-fms <list of string>	Alias for -failure_modes.
-primesonly	Limit report to only prime faults.
-summarystyle <string>	Show the campaign summary in the specified format. Possible values are basic, prime, zoix, coverage.
-tcs <list of string>	Alias for -testcases.
-testcases <list of string>	List of testcase(s) defined through the create_testcases command.

Example:

```
show_campaign_summary -summarystyle coverage
```

show_campaigns

Synopsis:

```
show_campaigns
-brief
[-campaign | -fc] <string>
-verbose
```

Description:

Displays a list of fault campaign objects in readable format. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
-brief	Show a list of the names of possible campaigns.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-verbose	Display all possible details about the campaign.

Example:

```
show_campaigns
```

show_ces

Alias for [show_cov_equations](#).

show_cov_equations

Synopsis:

```
show_cov_equations
[-campaign | -fc] <string>
-name <string>
```

Description:

Prints a list of coverage equations objects.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-name <string>	Name of the equation.

Example:

```
show_cov_equations
```

show_detection_points

Synopsis:

```
show_detection_points
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
[-safety_mechanism | -sm] <string>
```

Description:

Prints a list of detection point objects.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-failure_modes <list of string></code>	Name of the failure mode.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-safety_mechanism <string></code>	Name of the safety mechanism.
<code>-sm <string></code>	Alias for <code>-safety_mechanism</code> .

Example:

```
show_detection_points -safety_mechanism SM_001
```

show_dps

Alias for [show_detection_points](#).

show_failure_modes

Synopsis

```
show_failure_modes
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-verbose
```

Description:

This command returns a list of failure mode objects. It requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.

Option	Description
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-verbose</code>	Show all possible information about the failure mode(s).

Example:

```
show_failure_modes -failure_mode FC_002
```

show_fault_results

Synopsis:

```
show_fault_results

-attributes
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-fids <list of int>
-only_collapsed
-only_prime
-raw
-result <list>
-status <list>
[-testcases | -tcs] <list of string>
-tool <list>
-verbose
```

Description:

Displays a list of `FaultResult` objects in readable format. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
<code>-attributes</code>	Print an additional column for attributes associated with a fault result.
<code>-campaign <string></code>	Name of the fault campaign.

Option	Description
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fids <list of int></code>	Get the list of Fault IDs. See report for a description of how to obtain fault IDs.
<code>-fms <list of string></code>	Alias for <code>-failure_mode..</code>
<code>-only_collapsed</code>	Show only collapsed faults. The status returned is the status of the prime fault this fault collapses to.
<code>-only_prime</code>	Show only prime faults.
<code>-raw</code>	Outputs individual results per test and per tool (results without promotion).
<code>-result <list></code>	Show results only for the results given. Valid results are <code>NOT_EXECUTED</code> , <code>PASSED</code> , <code>TEST_FAILED</code> , <code>TEST_ERROR</code> , <code>PROMOTION_ERROR</code> , <code>INVALID_FAULT_ID</code> and <code>INCONSISTENT_RESULT</code> .
<code>-status <list></code>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>-testcases <list of string></code>	List of testcase(s) defined through the create_testcases command.
<code>-tool <list></code>	Get results for the tool provided. Valid tools are <code>CERTITUDE</code> , <code>ZOIX</code> , <code>FCC</code> , <code>ZEBU</code> , <code>FTA</code> , <code>VCFORMAL</code> , <code>VERDI</code> , <code>VCS</code> , <code>REPORT</code> , <code>GLOBAL</code> , <code>VC_FSM</code> , <code>SER_CREATED_RESULT</code> , <code>DYNAMIC_TESTABILITY</code> and <code>ACSIM</code>
<code>-verbose</code>	Show all detail about the selected fault results.

Example:

```
show_fault_results -fids 12345 -fc FC_001 -testcases TC_001 -result PASSED
```

show_faults

Synopsis:

```
show_faults
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
```

Description:

Displays a list of fault objects. Requires the FDB to be connected (see [fdb_connect](#)).

When the `-campaign` option is not provided, the campaign configuration is used. When both `-campaign` and `-failure_modes` are not provided, all faults from all campaigns in the connected project are returned.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a tcl formatted list of failure modes defined through the SFF file.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .

Example:

```
show_faults -fc FC_001
```

show_fc_summary

Alias for [show_campaign_summary](#).

show_fcs

Alias for [show_campaigns](#)

show_fms

Alias for [show_failure_modes](#).

show_host_groups

Synopsis:

```
show_host_groups
[-host_groups | -hg] <list>
```

Description:

Displays a list of `GridHostGroup` objects in readable format.

Parameters

Option	Description
<code>-hg <list></code>	Alias for <code>-host_group</code> .
<code>-host_groups <list></code>	View only the specified host group. The default is to view all host groups.

Example:

```
show_host_groups
```

show_host_infos

Synopsis:

```
show_host_infos
-id <list>
```

Description:

Display a list of `GridHostInfo` objects in readable format.

Parameters

Option	Description
<code>-id <list></code>	Display the list of hosts specified. Input is an integer or <code>tcl</code> formatted list of integers corresponding to the host ID as assigned by FCM.

Example:

```
show_host_infos -id 0
```

show_metadata

Synopsis:

```
show_metadata
[-campaign | -fc] <string>
-index <list>
-name <string>
```

Description:

The `vc_fcm show_metadata` command allows users to display the metadata of a fault campaign.

Parameters:

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-index <list></code>	List of indexes corresponding to metadata entries.
<code>-name <string></code>	The name of the metadata block which is requested to be displayed.

Example:

```
show_metadata -name MD_003
```

show_observation_points

Synopsis:

```
show_observation_points
[-campaign | -fc] <string>
[-failure_modes | -fms] <string>
```

Description:

Prints a list of observation point objects.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-failure_modes <list of string>	Name of the failure mode.
-fc <string>	Alias for -campaign.
-fms <list of string>	Alias for -failure_modes.

Example:

```
show_observation_points
```

show_ops

Alias for [show_observation_points](#)

show_projects

Synopsis:

```
show_projects
-fdb_server <string>
```

Description:

Print a list of FdbProject objects in readable format.

Parameters

Option	Description
-fdb_server <string>	Specifies the name of the FDB server (host:port).

Example:

```
show_projects
```

show_promotion_tables

Synopsis:

```
show_promotion_tables
```

```
-all
[-campaign | -fc] <string>
-promoted_status <list>
-status <list>
```

Description:

Prints a list of promotion table objects.

Parameters

Option	Description
-all	List all built-in and user defined fault statuses.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-promoted_status <list>	List of promoted statuses to filter by.
-status <list>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.

Example:

```
show_promotion_tables
```

show_pts

Alias for [show_promotion_tables](#).

show_safety_mechanisms

Synopsis:

```
show_safety_mechanisms

[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
[-safety_mechanism | -sm] <string>
```

Description:

Display a list of `SafetyMechanism` objects in readable format. Requires the FDB to be connected (see [fdb_connect](#)).

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign. When <code>-campaign/-fm</code> arguments are not provided, safety mechanisms from all <code>-campaigns/-fms</code> in the project are included.
<code>-failure_modes <list of string></code>	Name of the failure mode.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-safety_mechanism <string></code>	Name of the safety mechanism. When <code>-safety_mechanism</code> or <code>-sm</code> option is provided, only the defined SM is returned to the list.
<code>-sm <string></code>	Alias for <code>-safety_mechanism</code> .

Example:

```
show_safety_mechanisms -campaign FC_001 -sm SM_001
```

show_sgs

Alias for [show_status_groups](#)

show_sms

Alias for [show_safety_mechanisms](#)

show_status_groups

Synopsis:

```
show_status_groups
-all
[-campaign | -fc] <string>
-name <string>
-redefined <list>
```

Description:

Prints a list of fault status group objects.

Parameters

Option	Description
-all	List all built-in and user defined fault status groups.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-name <string>	Name of the status group.
-redefined <list>	List of old names to filter by (only matches if status group has been redefined).

Example:

```
show_status_groups -all
```

show_statuses

Synopsis:

```
show_statuses
-all
[-campaign | -fc] <string>
-redefined <list>
-status <list>
```

Description:

Prints a list of fault status objects.

Parameters

Option	Description
-all	List all built-in and user defined fault statuses.
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for -campaign.
-redefined <list>	List of old statuses to filter by (only matches if status has been redefined).

Option	Description
-status <list>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.

Example:

```
show_statuses
```

show_task_failures

Synopsis

```
show_task_failures
-all_sessions
-help
-session <Session ID>
-task <integer>
-tc <test name>
-testcase <test name>
-type <type>
```

Description:

This command prints a list of previous failed tasks found in `fcm.dir`.

Parameters

Option	Description
-all_sessions	Select all sessions.
-help	Show this help.
-session <Session ID>	Select a session ID or latest.
-task <integer>	Specify the task ID.
-tc <test name>	Select a test. For example, -tc t1.
-testcase <test name>	Select a test. For example, -test t1.
-type <type>	Task type. Values: tsim, fsim, coats

Example:

```
show_task_failures -type fsim
```

show_tasks

Alias for `vc_fcm_show_tasks` command with the same options and capabilities.

Synopsis:

```
show_tasks

-active
-all_sessions
-campaign_name
-failed
-fc <name>
-help
-session <Session ID>
-success
-timeout
```

Description:

This command prints a list of previous tasks found in `fcm.dir`.

Parameters

Option	Description
<code>-active</code>	Select only tasks in the active state.
<code>-all_sessions</code>	Select all sessions.
<code>-campaign_name</code>	Name of the fault campaign.
<code>-failed</code>	Select failed tasks.
<code>-fc <name></code>	Name of the campaign.
<code>-help</code>	Show this help.
<code>-session <Session ID></code>	Select a session ID or latest.
<code>-success</code>	Select success tasks.
<code>-timeout</code>	Display only tasks that timed out.

Example:

```
show_tasks -timeout
```

show_tcs

Alias for [show_testcases](#)

show_testcases

Synopsis:

```
show_testcases
[-campaign | -fc] <string>
-coats_results
[-testcases | -tcs] <list of string>
```

Description:

Prints list of `Testcase` objects.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-coats_results</code>	Show <code>vc_coats</code> result summary.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>- testcase <list of string></code>	Name of the testcase as defined through the create_testcases command.

Example:

```
show_testcases -coats_results
```

show_users

Synopsis:

```
show_users
```

```
-fdb_project <string>
-fdb_server <string>
-user <string>
```

Description:

Prints the list of User objects.

Parameters

-fdb_project <string>	Set the name of the FDB project or fault database.
-fdb_server <string>	Specifies the name of the FDB server (host:port).
-user <string>	User name to search for.

Example:

```
show_users
```

start_server

Synopsis:

```
start_server
-fdb_server <string>
-fdb_storage_path <directory>
```

Description:

Start the SQL server for the fault database.

Parameters:

Option	Description
-fdb_server <string>	Specifies the name of the FDB server (host:port).
-fdb_storage_path <directory>	Path for FDB storage location. The default is fdb.

Example:

```
start_server
```

stop_server

Synopsis:

```
stop_server
-fdb_storage_path <directory>
-force
```

Description:

Stop the running SQL server for the fault database.

Parameters

Option	Description
-fdb_storage_path <directory>	Path for FDB storage location. The default is <code>fdb</code> .
-force	Force stop server.

Example:

```
stop_server
```

test_grid_config

Synopsis:

```
test_grid_config
-timeout <integer>
-retries <integer>
```

Description:

Launches a test application on the grid setup via [set_submit_cmd](#) command.

Parameters

Option	Description
-timeout <integer>	Time in seconds to wait for a job to be scheduled, default is 120.
-retries <integer>	Retry limit, default is 1.

Example:

```
test_grid_config
```

tsim**Synopsis:**

```
-help
-testcases <list of string>
-tcs <list of string>
-localhost
-verbose
```

Description:

Launches a test application on the grid setup via [set_submit_cmd](#) command.

Parameters

Option	Description
-help	Shows this help.
-testcases <list of string>	List of testcase names. Example: -test {t1 t2 t3}.
-tcs <list of string>	Alias for -testcases.
-localhost	Run analysis on local machine, instead of the cluster/farm.
-verbose	Print messages for worker start/stop and task start.

Example:

```
tsim -testcases
```

unset_campaign**Synopsis:**

```
unset_campaign
```

Description:

This command unsets the active campaign and it returns nothing.

Parameters:

None.

Example:

```
unset_campaign
```

unset_fc

Alias for [unset_campaign](#).

unset_format

Synopsis:

```
unset_format
```

Description:

Unsets the format to read fault lists and write reports.

Parameters:

None.

Example:

```
unset_format
```

unset_submit_cmd

Synopsis :

```
unset_submit_cmd  
-all  
-id <integer>  
-task_type <list>  
-task_type_all <list>
```

Description:

This command unsets the submit command set by [set_submit_cmd](#). To find out submit command ID, use [show_host_infos](#) command. When an ID that does not exist is provided, no error is generated. This command removes the groups that are not referenced by any submit command. The changed parameters of localhost submit command and default group are preserved.

Parameters

Option	Description
-all	If <code>-all</code> option is provided, all (except localhost) submission commands are removed.
-id <integer>	ID of the host info to unset.
-task_type <list>	The <code>-task_type</code> must be used with the <code>-id</code> option, only the specified task types for the given submission id are disabled. Valid options are <code>default</code> , <code>fcc</code> , <code>coats</code> , <code>tsim</code> , <code>fsim</code> , and <code>report</code> .
-task_type_all <list>	The <code>-task_type_all</code> disables all submissions for the given task types. It cannot be used with <code>-id</code> . Valid options are <code>default</code> , <code>fcc</code> , <code>coats</code> , <code>tsim</code> , <code>fsim</code> , and <code>report</code> .

Example:

```
unset_submit_cmd -id 1
```

user_mgmt

Synopsis:

```
user_mgmt
-add_role
-fdb_project <string>
-fdb_server <string>
-remove_role
-remove_user
-role <string>
-user <string>
```

Description:

This command manages users within the FDB environment. You can add or remove a user (as superuser) to an FDB project. The list of users is available in the project. When `-fdb_server/-fdb_project` command is not provided, `-fdb_server/-fdb_project` configuration is used. When none of these options are provided, the command gives an error. For the local FDB flow, `fdb_connect` must be used beforehand, that is, empty `fdb_server` is not allowed. When adding or removing options are used, the `-user` argument is mandatory. One of the operations `-add_role`, `-remove_role`, or `-remove_user` is required. When no operation is provided an error is returned.

Adding Users:

```
vc_fcm> ::fcm::user_mgmt -add_role -user dummy1-role SUPERUSER
-fdb_server <Host>:<Port>-fdb_project default
```

Where, <Host>:<Port> specifies the server/machine on which the FDB is connected and Port number of the FDB server. These details can be take out from *fcm_debug.log* inside *fcm.dir*.

Removing Users:

```
vc_fcm> ::fcm::user_mgmt -remove_user -user abc1
```

::fcm::add_user and **::fcm::remove_user** commands are an alias to **::fcm::user_mgmt** while adding and removing users respectively.

Parameters

Option	Description
-add_role	Add a user role.
-fdb_project <string>	The name of the FDB project or fault database.
-fdb_server <string>	Specifies the name of the FDB server (host:port).
-remove_role	Remove a user role.
-remove_user	Remove a user with all its roles.
-role <string>	Specify the role to manage. Possible values are SUPERUSER, PROJECT_ADMIN, PROJECT_FULL_ACCESS, PROJECT_RESULT_WRITE, PROJECT_READ_ONLY. Where, SUPERUSER - Provides full cluster access. That is, PROJECT_ADMIN access for all projects. PROJECT_ADMIN - database admin: Provides full access/ control over the database and can add users to this group. PROJECT_FULL_ACCESS - Provides full access to the database. PROJECT_RESULT_WRITE - Can create tests, tasks and fault results. Can read the entire Database. PROJECT_READ_ONLY - Can read the entire database.
-user <string>	Specifies the user to operate on.

Example:

```
user_mgmt -user USER_001 -remove_user
```

vcf

Synopsis:

```
vcf
```

```
-check_control  
-check_detect
```

```
-check_observe
-check_structural
-dut_path <string>
-f <filename>
-fault_summary
-gen_checks
-gen_model
-lic_elite
-lic_zoix
-map_status <filename>
-max_mem <string>
-max_time <string>
-report <string>
-save_fdb
-status <list>
-top <string>
-vcs_cmd_line <string>
-verdi
```

Description:

Run formal analysis to analyze faults. VC Formal can be run to find additional safe faults through structural, controllability, observability, and detectability analysis.

Parameters

Option	Description
-check_control	Run controllability analysis checks. Faults found to be not controllable will be marked UT as a result of this analysis.
-check_detect	Run detectability analysis.
-check_observe	Run the observability stage. Faults found to be not observable will be marked as UB.
-check_structural	Analyze the structural testability of the design. Faults found to be structurally untestable will be marked with the appropriate faults status from the UG fault status group.
-dut_path <string>	Optional argument to FCC. This option sets the top-level scope where VC Z01X looks for default strobe locations. -dut_path is not connected to -fsim=dut:<> specified at compile. If DUT is provided at VCS compile time though -fsim=dut:<>, then FCC uses this as DUT. If DUT is also provided on the FCC command line, then we must ensure that DUT hierarchical paths are equal.
-f <filename>	Provide the Tcl file name for VC Formal to run. The file will be passed to the underlying command vcf -f <filename>.
-fault_summary	Generate the fault summary.

Option	Description
-gen_checks	Generate the checks necessary for formal analysis.
-gen_model	Generate the VC Formal model.
-lic_elite	Use the licenses from VC Formal Elite package.
-lic_zoix	Use the licenses from the VC Z01X package.
-map_status <filename>	Provide a file with the status mapping to equate SFF statuses with VC Formal built-in statuses.
-max_mem <string>	Set the maximum amount of memory for each stage of VC Formal.
-max_time <string>	Set the maximum runtime for each stage. The runtime is defined with a time unit.
-report <string>	Define the list of options to pass to the VCF <code>fusa_report</code> command. Add <code>-summary</code> for a summary report.
-save_fdb	Save data to the FDB.
-status <list>	Provide a list of statuses to be considered by each stage.
-top <string>	Define the top module name.
-vcs_cmd_line <string>	Pass a list of command line arguments to VCS.
-verdi	Open Verdi.

Example:

```
% vc_fcm -tcl_script fcm.tcl -connect -fc riscdemo
set_config -global_max_jobs 1
create_testcases -name test1 -exec "$::env(PWD)/simv" -args
  "+TESTNAME=user_test1" -fsim_args ""
create_testcases -name test2 -exec "$::env(PWD)/simv" -args
  "+TESTNAME=user_test2" -fsim_args ""
vcf -gen_model -top test -dut_path test
vcf -gen_check
vcf -report "-summary"
vcf -check_structural
fsim
report -campaign riscdemo -report fsim_v.rpt -overwrite
```

verdi

Synopsis:

```
verdi
[-campaign | -fc] <string>
-args <string>
-log <filename>
[-foreground | -fg]
```

Description:

Start Verdi.

Parameters

Option	Description
-campaign <string>	Name of the fault campaign.
-fc <string>	Alias for <code>-campaign</code> .
-args <string>	Further <code>verdi</code> command-line arguments.
-log <filename>	Log file capturing <code>stderr</code> and <code>stdout</code> of Verdi.
-foreground	Run Verdi in the foreground, return after it exits.
-fg	Alias for <code>-foreground</code> .

Example:

```
verdi -fc FC001
```

write_fault_results

Synopsis:

```
write_fault_results
[-campaign | -fc] <string>
[-failure_modes | -fms] <list of string>
-fault_class <list>
-fault_locations <filename | list>
-fault_type <string>
-fids <list of int>
-sff_faultlist <filename>
-status <string>
[-testcases | -tcs] <list of string>
```

Description:

Writes fault results to FDB as tool `USER_CREATED_RESULT`.

Parameters

Option	Description
<code>-campaign <string></code>	Name of the fault campaign.
<code>-fc <string></code>	Alias for <code>-campaign</code> .
<code>-failure_modes <list of string></code>	List of failure mode(s). The list is a <code>tcl</code> formatted list of failure modes defined through the SFF file.
<code>-fms <list of string></code>	Alias for <code>-failure_modes</code> .
<code>-fault_class <list></code>	List of fault classes. Possible values are <code>ARRY</code> , <code>PORT</code> , <code>PRIM</code> , <code>VARI</code> , <code>WIRE</code> , <code>FLOP</code>
<code>-fault_locations <filename list></code>	Either (a) a path to a file containing fault locations (one per line) or (b) list of fault locations. Can contain wildcards.
<code>-fault_type <string></code>	0 or 1 for stuck-at faults.
<code>-fids <list of int></code>	Write the list of Fault IDs. See report for a description of how to obtain fault IDs.
<code>-sff_faultlist <filename></code>	Path to SFF file with FaultList tag. Other tags are ignored.
<code>-status <string></code>	Show results for only the given list of two-character statuses. The default is to use all statuses. The list of status can include built-in fault statuses, user defined fault statuses, and status groups.
<code>-tcs <list of string></code>	Alias for <code>-testcases</code> .
<code>-testcasess <list of string></code>	List of testcase(s) defined through the create_testcases command.

Example:

```
write_fault_results -sff_faultlist input.sff
```

16

Appendix B: Command Syntax

This appendix summarizes options and usage for VC Z01X standalone tools.

Available Options

- Fault Campaign Compiler - [vc_fcc](#)
- Coverage Report - [vc_fdb_report](#)
- Fault Simulation - [vcs](#)
- Compilation - [vcs](#)
- Logic Simulation - [simv](#)

Special Verilog directives, system tasks, and system functions

- [\\$fs_compare](#) (system function)
- [\\$fs_observe](#) (system function)
- [\\$fs_detect](#) (system function)
- [\\$fs_set_attribute](#) (system task)
- [\\$fs_add_attribute](#) (system task)
- [\\$fs_inject](#) (system task)
- [\\$fs_inject](#) (system task)
- [\\$fs_verify](#) (system task)
- [\\$fs_verify_onevent](#) (system task)

System Tasks for User-Controlled Fault Detection

- [\\$fs_compare](#) (system function)
- [\\$fs_drop_status](#) (system task)
- [\\$fs_set_status](#) (system task)
- [\\$fs_default_status](#) (system task)

- [\\$fs_get_status \(system function\)](#)
 - [\\$fs_strobe_onevent\(\) \(system task\)](#)
-

Available Options

The following options may be used on the command line of some of the VC Z01X-related programs and utilities described in this chapter.

- `-help` - Sends usage information to standard out.

Program and utilities list:

[vc_fcc](#), [vc_fdb_report](#), [vcs](#), [vcs](#), [simv](#)

vc_fcc

Name

`vc_fcc`

SYNOPSIS

```
$VCS_HOME/bin/vc_fcc
[-arry_faults_on_mda]
[-campaign <string>]
[-change_faultlist_status]
[-collapse <string>]
[-collapse_untestable_faults]
[-cplite]
[-daidir <directory>]
[-dut_path <string>]
[-excludedcoloff]
[-fault_test_coverage]
[-faultlist <filename>]
[-fc_update_lock_timeout <integer>]
[-fdb_path <directory>]
[-fdb_project <string>]
[-fdb_timeout <integer>]
[-fm_result_sharing <string>]
[-format <string>]
[-full64]
[-gen_tab <tab_file>] [-hier_summary <integer>]
[-hier_summary_faultlimit <integer>]
[-ignore_suppress]
[-inmdir <directory>]
[-log <filename>]
[-no_design]
[-no_fgen]
```

```

[-no_flop_array]
[-no_flop_info]
[-nolibcell]
[-nometadata]
[-no_systematic_sampling]
[-overwrite]
[-progress_timeout <integer>]
[-prune <string>]
[-quiet]
[-report <filename>]
[-rt_elab_args <filename>]
[-sample <string>]
[-seed <integer>]
[-ssf <filename>]
[-summary]
[-suppress_cell]
[-uncontrollability]
[-warn <list>]

```

DESCRIPTION

The VC Z01X fault campaign compiler creates a fault campaign in FDB containing all of the faults that have been defined for a given campaign, as well as their locations, collapsing data, and pre-simulation detectability status if assigned by the fault campaign compiler.

OPTIONS

-arry_faults_on_mda - Generates only ARRY fault for multidimensional array objects (MDA) and generates only VARI and/or WIRE fault for non-MDA objects.

-campaign <string> - Sets the name of the fault campaign.

-change_faultlist_status - Changes the fault status of a faults in an input FaultList.

Examples:

-change_faultlist_status NA,UU=DD - Changes all NA and UU fault statuses to DD.

-change_faultlist_status ---NA - Changes fault status of all collapsed faults to NA.

-change_faultlist_status ---prime:UT=NA - Changes fault status of all the collapsed faults to be equal to their corresponding prime faults. Changes all UT fault statuses to NA.

-collapse <string> - Enable or disable fault collapsing. Valid values are **on** (default) and **off**.

-collapse_untestable_faults - Enable collapsing of untestable faults. Effective only if global pruning is enabled.

-cplite - Do reduced constant identification/propagation.

-daidir <directory> - Define the location of *simv.daidir* directory. The default is *simv.daidir* in the directory where *vc_fcc* is invoked.

-dut_path <string> - Optional argument to FCC. This option sets the top-level scope where VC Z01X looks for default strobe locations. **-dut_path** is not connected to **-fsim=dut:<>** specified at compile. If DUT is provided at VCS compile time though **-fsim=dut:<>**, then FCC uses this as DUT. If DUT is also provided on the FCC command line, then we must ensure that DUT hierarchical paths are equal.

-excludedcolloff - Do not exclude collapsed faults that belong to an excluded prime fault. Default behavior is to exclude all its collapsed faults if the corresponding prime fault is excluded. Excluding a collapsed fault in both cases just excludes the single collapsed fault. When this option is enabled, the collapsed fault will be promoted to a prime fault if the corresponding prime fault is excluded.

-fault_test_coverage - Create test and fault coverage formulas.

-faultlist <filename> - Provide a comma or space-separated list of fault list files (non-SFF) to be parsed.

-fc_update_lock_timeout <integer> - Set the timeout value (in seconds) to wait to get a write access to the fault campaign.

-fdb_path <directory> - Define the path to the fault database directory. The path can be specified as either an absolute directory path or a local path from where *vc_fcc* is invoked.

-fdb_project <string> - Name of the project in the fault database.

-fdb_timeout <integer> - Timeout value (in seconds) when writing to fault database.

-fm_result_sharing <string> - Enable sharing of results for different failure modes. Valid values are **on**, **off**, or **auto**.

Where,

On: Results are always shared between failure modes. Sharing is forced, even if the observation and/or detection points are different.

Off: (Default) Results are never shared between failure modes.

Auto: Results are shared between failure modes if the observation and detection points are the same.

It is all or none with **auto**. Only if the observation and detection points are the same for all failure modes, then sharing is enabled. To force sharing, use **on**.

-format <string> - Format of manufacturing fault list file(s) specified with **-faultlist**. The supported formats are *tetramax*, *fastscan*, and *verifault*.

-full64 - Runs 64-bit executable.

-gen_tab <tab_file> - Generate the PLI table file while running the fault campaign compiler. The PLI table can be used to recompile the design before running fault simulation to provide debug capabilities only on a specific set of locations in the design to reduce the memory consumption and improve execution speed.

-hier_summary <integer> - Write hierarchical summary in hybrid flow. Optional argument provides number of levels from root scope downwards to report statuses for (default is 0, meaning write full hierarchy).

-hier_summary_faultlimit <integer> - Total number of faults below which scope is merged with parent scope when writing hierarchical summary data (default [if not specified or for negative values] = 1% of total faults but at most 1000, inactive = 0, fixed limit for positive values).

-ignore_suppress - Ignore the compile time option `-fsim=suppress+cell` option and generate faults inside cells.

-incdir <directory> - Specifies a directory to search when the input SFF file includes additional files.

-log <filename> - Specifies a name for a log file `vc_fcc` should create. The default is `vc_fcc.log`.

-no_design - Instructs `vc_fcc` to skip checking the correctness of the unrolled fault location against design database (daidir) and store fault locations into the FDB. This option is useful if there is no dadir or the user is certain if the locations are correct. With a very large unrolled FaultList, this option improves the runtime. This option cannot be used with FaultGenerate or wildcard expansion. It is only used if the input SFF fault list is fully unrolled.

During fault simulation, if a fault location does not exist, the fault injection fails and is skipped for any non-existing locations. Fault simulation continues and the other faults with correct locations are injected. Thus, check for this error message if using this option:

`FSIM Error: <Object> not found in the design.`

-no_fgen - Skip fault generation in hybrid mode.

-no_flop_array - Do not generate *FLOP* faults on arrays.

-no_flop_info - Do not use unified inferencing to identify flops.

-nolibcell - Only modules with `celldefine/endcelldefine` should be treated as cells.

-nometadata - Ignore the *Metadata* section of input SFF files.

-no_systematic_sampling - Use pseudo-random sampling selection instead of systematic sampling.

-overwrite - Overwrite existing fault campaign in the fault database, instead of merging.

-progress_timeout <integer> - Issue progress message every given number of seconds

-prune <string> - Fault pruning (on | off | local | global). Local pruning does only the local load tracing.

Global pruning does additional testability analysis based on cone-of-influence from observation and detection points. The default is local pruning.

-quiet – Do not print progress messages.

-report <filename> - Generates report in the form of SFF file in the *FileList* section, which contains a list of all generated faults.

-rt_elab_args <filename> - Provides a file which contains runtime elaboration options. Typically used for items like values of VHDL generics.

-sample ci:+<float>+cl+<integer> - The ci and cl settings are used in conjunction to perform confidence interval sampling, which allows you to randomly select faults using standard statistical confidence models.

ci - Defines the confidence interval and is a floating point or integer value greater than 0 but less than or equal to 100.

cl - Defines the confidence level and is defined from a specific set of values (70, 75, 80, 85, 90, 92, 95, 96, 98, and 99, usually 95 or 99).

Example:

```
+sample+ci+0.5+cl+95
```

-sample num:<integer> - Use for fixed number sampling. The <integer> field represents the number of faults that are selected.

-sample percent:<float> - Samples a specified percentage of faults from the total possible faults. The <float> field specifies the percentage of total faults to simulate. Default is 100.0. If you specify a percentage less than 100 without specifying **-seed**, the fault generator inserts faults on a pseudo-random basis throughout the design hierarchy. This option supports decimal precision.

-seed <integer> - For use when fault sampling is enabled and you want a fault set other than the one resulting from the default seed value, which is 1000000. To change the seed value, specify <integer> to be any unsigned 32-bit integer value.

The method used is pseudo-random, meaning that the fault set will be consistent from run to run provided neither the circuit nor the seed value is modified.

This option enables you to generate multiple different sets of randomly selected faults, to help you evaluate how consistent (and thus how representative overall) the random fault sets may be.

- sff <filename>** - Comma or space-separated list of SFF files to be parsed.
- summary** - Write hierarchical summary data to the FDB.
- suppress_cell** - Does not generate faults in cells, except PORT faults.
- uncontrollability** - Enables UC (uncontrollable) fault status marking during VC Z01X untestable analysis. In this mode, VC Z01X propagates constant values (including 'x') to determine whether faults can be marked UC. If all fault locations receive a static 'x', the fault is marked UC.
- warn <string>** - Comma-separated list of error message IDs that should be downgraded to warnings.

RETURN VALUE

0 - Successful Operation

1 - Fatal error or Error encountered during operation

SEE ALSO

[vc_fdb_report](#), [vcs](#), [simv](#)

vc_fdb_report

NAME

vc_fdb_report

SYNOPSIS

```
$VCS_HOME/bin/vc_fdb_report

[-aggregatedsummary]
[-campaign <string>]
[-collapseoff]
[-csv]
[-failure_mode <string>]
[-faultlimit <integer>]
[-faultstatus <string>]
[-fdb_path <string>]
[-fdb_project <string>]
[-filterattributes <string>]
[-flat]
[-format <standard|tetramax|fastscan|verifault>]
[-gz]
[-help]
[-hierarchical <integer>]
[-log <filename>]
[-metadataonly]
[-nostatusdefinitions]
```

```

[-overwrite]
[-primesonly]
[-print]
[-promotedsummary]
[-quiet]
[-report <filename>]
[-showattributes <string>]
[-showexcludedfaults]
[-showfaultid]
[-showfub]
[-showmetadata]
[-showmetadataid]
[-showsimdetails]
[-showtestresults <string>]
[-showtimingid]
[-sort <hier|status|desc>]
[-summaryonly]
[-summarystyle <basic|prime|zoix>]
[-targeted <count>]
[-test <string>]
[-tool <z01x|certitude|vcformal|zebu|Verdi|vcs|coats|fcc>]
[-unselected <fault|location|observability|controllability>]
[-verbose]

```

DESCRIPTION

Generates a text coverage report file showing the fault campaign results contained in the fault database.

OPTIONS

-aggregatedsummary - Create a combined summary across all fault lists. By default, duplicate faults across fault lists have each status counted once for each fault list, if only one status is desired per unique fault see **-promotedsummary**.

-campaign <string> - Sets the name of the fault campaign.

-collapseoff - For collapsed faults, shows the status of the prime fault instead of --.

-csv - Creates CSV tables of `FaultList` summaries. The files will be created in the directory `<report_name>_csv_files`.

For more information, see [Comma Separated Value \(CSV\) Fault Coverage Summary](#).

-failure_mode <string> - Set failure modes to be reported. The default is to report all failure modes. To specify the unnamed failure mode, use the string `DEFAULT`. The argument may be a comma separated list of failure modes.

Example:

```
-failure_mode fm1,fm2,fm3
```

-faultlimit <integer> - Hide child instances that contain less than the specified fault count. This argument is valid only in combination with creation of a hierarchical coverage report (**-hierarchical**). Useful for readability when hierarchical reports are dominated by large areas of the design with few faults.

-faultstatus <string> - Display only faults with specified statuses. Use a comma separated list of individual statuses or status groups. When a status group is encountered all statuses in that status group is marked for display.

Example:

`-faultstatus OD,NG,PD`

-fdb_path <string> - Define the path to the fault database directory. The path can be specified as either an absolute directory path or a local path from where `vc_fdb_report` is invoked.

-fdb_project <string> - Name of the project in the fault database.

-filterattributes <string> - Display faults in the fault list that contains attributes that match the specified filter.

Syntax:

`-filterattributes [<key>[=<value>] [+...]]`

For example, to display only faults that have attributes with keys `someKey1`, and `someKey3`, but only if the `someKey1` attributes value matches `some_value_1`:

`-filterattributes someKey1=some_value_1+someKey3`

-flat - Print a single descriptor for each transient fault, rather than combine transients that only differ by cycle number

-format <standard|tetramax|fastscan|verifault> – Create the coverage report in the requested format. The supported formats are SFF (`standard`, `tetramax` (TestMAX ATPG), `fastscan`, and `verifault`. The default setting is to report in standard (SFF) format.

-gz – Create the requested coverage report in compressed gz (GNU zip) format. The resulting file will be named according to **-report** argument with the .gz suffix appended.

-hierarchical <integer> - Create a hierarchical report. Optional argument specifies the number of levels of hierarchy to report on. The default is to report on 1 level. Specify 0 to report on the entire hierarchy.

-help - Prints command help text and exits.

-log <filename> - Specify a filename for the logfile. The default is `vc_fdb_report.log`.

-metadataonly - Prints only the metadata of the SFF report. Useful to quickly review the history of the campaign.

- nostatusdefinitions – Omit the StatusDefinitions block from the coverage report.
- overwrite - Overwrite an existing file if a name conflict occurs with the report to be created.
- primesonly - Limit the report to show only the prime faults.
- print - Print output of report to screen and log file.
- promotedsummary - Optional argument to change methodology of total summaries. By default, when the same fault spans multiple fault lists, vc_fdb_report counts a status for each fault per fault list. This option is for enabling the methodology of the counting faults that span failure modes only once and to use the promoted status between the fault lists.
- quiet - Disable progress messages.
- report <filename> - Specify the filename for creation of the coverage report. By default, vc_fdb_report uses <campaign>_report.sff, unless -test is specified, then it will be <campaign>_<test>_report.sff
- showattributes <string> - Print fault attributes in reports. Takes an optional filter to only show specified attributes, no filter will show all.

Example:

```
-showattributes [<key>[=<value>] [+...]]
```

For example, to show all attributes with keys *exampleKey1*, *exampleKey2*, and *exampleKey3*, but only if the *exampleKey3* attributes value matches *some_value_3*:

```
-showattributes exampleKey1+exampleKey2+exampleKey3=some_value_3
```

-showexcludedfaults - Show faults with excluded statuses in the fault list. By default, faults with excluded statuses are not displayed.

-showfaultid - Display a comment after each descriptor showing its associated fault ID.

-showfub- Optional argument to display FunctionalBlock. FunctionalBlock was replaced with TestList in SFF reports. However, it may be useful to get the old style for importing to other SFF compilers that still require the functional block information.

-showmetadata - Print fault campaign metadata in the report. The Metadata block contains a logged history of the campaign.

-showmetadataid - Optional argument to print fault campaign metadata in report with commented indices. The Metadata block contains a logged history of the campaign. This index can be used to add or remove individual entries.

-showsimdetails – Optional argument to display *Attempt* count, and *NumPot* count (if applicable) for each fault. Note: NumPot is only displayed when no user defined fault statuses were used in the campaign. Attempt count is the number of times the fault was

attempted for fault simulation. *NumPot* is the number of potential results (e.g. PD,PT) that occurred in this campaign run for each fault.

-showtestresults <string>— Option to display the non-promoted test results in a comment following the fault description. Specifies the status, or status group of results to display.

-showtimingid - Used to display every transient fault's timing ID in its descriptor.

Example:

```
ND ~ ("timing1", 25) {PORT "testbench.dut.blk1.port1"}
```

-sort <hier|status|desc> - Set the sorting type for the coverage report. Valid arguments: *hier* (sort by hierarchy), *status* (sort by status), and *desc* (sort by descriptor starting from fault type (~,0,1)).

-summaryonly - Prints only the summaries of the SFF report. This option is useful for reporting quick counts and may be used with **-print** option to display the summaries to the screen.

-summarystyle <basic|prime|coverage|zoix> - Use an alternative summary format. Different summaries are *basic*, *prime*, *coverage*, and *zoix*. The default summary style is *basic*.

-targeted <count> - Creates the targeted report using the unselected report. Requires the **-unselected** option and an argument for the number of blocked or uncontrolled locations to include in the report. 0 means all the locations will be included.

-test <string> - Create the coverage results from the listed test. The argument may contain a single test name or a comma separated list of tests. If multiple tests are listed, the promoted results for those tests will be calculated and reported.

-tool <zoix|certitude|vcformal|zebu|verdi|vcs|coats|fcc> - Show results from faults where the status was set by the given tools. The valid tools are: *zoix*, *certitude*, *vcformal*, *zebu*, *verdi*, *vcs*, *coats*, and *fcc*. When **-tool** is not used then *vc_fdb_report* retrieves global promoted results of all tools

-unselected <fault|location|observability|controllability> - Create an unselected report. The optional argument selects the mode of the report including *fault*, *location*, *observability*, and *controllability*. The default mode is *fault*.

-verbose - Output verbose messages

RETURN VALUE

0 - Successful Operation

1 - Fatal error or Error encountered during operation

SEE ALSO

[vc_fcc](#), [vcs](#), [simv](#)

See [Working with Results](#) for details about fault reports and fault dictionary reports.

VCS

NAME

vcs

SYNOPSIS

```
$VCS_HOME/bin/vcs -fsim <file_list>
[-debug_region=lib+cell]
[-fsim=class]
[-fsim=dut:<string>]
[-stim=forcelist]
[-fsim=iddq]
[-fsim=inprim]
[-fsim=portfaults]
[-fsim=serial_flow]
[-fsim=suppress+cell]
[-hsopt=gates]
[-stim=module:<string>]
[+nolibcell]
[+notimingcheck]
```

DESCRIPTION

VC Z01X is an extension of the VCS simulator. During the compilation of the design, Verilog and VHDL files are compiled and linked into an executable program, and data files are created for use by the rest of the fault simulation process.

The options documented here are those specific to fault simulation. To see a full list of compile time switches for VCS, please consult the *VCS User Guide*.

OPTIONS

-debug_region=lib+cell - Applies debug capabilities to the libraries, real cell modules and the ports of real cell modules.

-fsim=class - Supports class in concurrent `faultsim`. Without this switch, FCM might report illegal access when fault propagates through class structure.

-fsim=dut:<string> - Identify the DUT to be used for fault simulation. Faults will be blocked from propagating faults effects outside the identified DUT.

-stim=forcelist - Informs the compiler to build the design for general `forcelist` stimulus capability. Requires `forcelist_file` option at run time.

- fsim=iddq - Enables IDDQ fault.
- fsim=inprim - Enables implementation of PRIM fault injection on gates.
- fsim=portfaults – Enables fault injection on cell ports by default. This option is recommended for RTL fault simulation.
- fsim=serial_flow - Enables serial fault simulation mode during compile. Required switch for SDF usage in the simulation.
- fsim=suppress+cell- Restricts generation of port faults to modules defined as cells. Recommended for gate level fault simulation.
- hsopt=gates - Improves runtime performance on gate-level designs (both functional and timing simulations with SDF). You may see some compile-time degradation when you use this switch.
- stim=module:<string> - Compile the specified DUT for stimulus capability. Required for primary external stimulus (eVCD and FSDB).
- nolibcell – Does not define a cell module as defined in libraries unless they are under the `celldefine compiler directive.
- +notimingcheck: Required switch for fault simulation. VC Z01X fault simulation does not allow timing checks.

RETURN VALUE

- 0 - Successful Operation
- 1 - Fatal error or Error encountered during operation

SEE ALSO

[vc_fcc](#), [vc_fdb_report](#), [simv](#)

simv

NAME

simv

SYNOPSIS

```
simv
[-stim=clk:<string>]
[-stim=file:<filename>]
[-stim=forcelist_dut:<string>+<string>]
[-stim=forcelist_file:<filename>]
[-stim=forcelist_mode:<language|external|all>]
[-stim=inst:<string>]
```

```
[-stim=type:<string>]
[-stim=verify]
[-stim=verify_all]
[-stim=verify_mismatch_limit=<int>]
[-stim=verify_off]
```

DESCRIPTION

Invokes the VC Z01X simulator in logic simulation mode.

OPTIONS

-stim=clk:<string> - Lists signals in the design that should be treated as clock. To avoid race conditions between clocks and data, clock signals are driven in the non-blocking assign region. You may optionally use `-deraceclockdata` compile switch to help resolve design race conditions.

-stim=file:<filename> - Specifies the path of external stimulus file to be used.

-stim=forcelist_dut:<string>+<string> - Determines the paths for the design's DUT and forcelist's DUT. The first string is the path to the design DUT and the second string is the path to the forcelist DUT.

-stim=forcelist_file:<filename> - Path to the forcelist stimulus file.

-stim=forcelist_mode:<language|external|all> - Describes which force types will be applied. Possible values are `language`, `external`, or `all`. By default, the mode is set to `language` forces.

-stim=inst:<string> - The argument specifies the name of the DUT in the external stimulus file. It is not the DUT instance in design where external stimulus is being applied.

-stim=type:<string> - Defines the type of external stimulus being used.

-stim=verify - The switch enables the stimulus verification mode. This mode verifies signals whenever system tasks such as `$fs_strobe`, `$fs_compare`, or `$fs_verify` are called by logic simulation. When verifying signals, the mode will compare the end-of-timestep values of any listed signal in such system task against the stimulus value in that timestep. If there is a difference, a mismatch warning will be reported. The purpose of this mode is to verify that relevant signals are correctly simulated at important times, such as `$fs_strobe` calls.

-stim=verify_all - The switch enables verification of logic simulation when using eVCD or FSDB as stimulus by comparing the current simulation with the stimulus being used.

-stim=verify_mismatch_limit:<int> - Sets limit on number of mismatches to display before killing the simulation. Used in conjunction with `-stim=verify_all` switch.

-stim=verify_off - Disables verification of logic simulation against the external stimulus even if `$fs_verify` system task is used.

RETURN VALUE

- 0 - Successful Operation.
- 1 - Fatal error or Error encountered during operation.
- 1 - When `-stim=verify` is in use and mismatches are found, unless `-stim=verify +mismatch+limit` is used.
- 255 - When simulation ends due to a TERM or QUIT signal.

SEE ALSO

[vc_fcc](#), [vc_fdb_report](#), [vcs](#)

Special Verilog directives, system tasks, and system functions

This section describes usage of the following Synopsys-specific keywords:

- [\\$fs_inject \(system task\)](#)
- [\\$fs_verify \(system task\)](#)
- [\\$fs_verify_onevent \(system task\)](#)

Also see, [System Tasks for User-Controlled Fault Detection](#).

Note:

VC Z01X allows you to select module instances using `$fs_set_status`, `$fs_drop_status`, `$fs_set_status_onevent`, `$fs_drop_status_onevent`, and `$fs_disable_onevent` system tasks and applies a system task to each output and inout port of an instance. For example, you can select the module instances using the `$fs_set_status` task as follows: `$fs_set_status("DD", test.dut.out1);`

\$fs_inject (system task)

For fault simulation.

```
initial begin
#100 $fs_inject();
end
```

Specifies the time at which faults are injected into the fault simulation. By default, faults are injected at time 0. The time can alternatively be specified with the command line plus argument `-fsim=fault+inject+<float>`.

If a design contains `$fs_inject()` but it never executes, toggle reports a failure. Fault Campaign Manager marks the test as failing, and the log file indicates `$fs_inject() was not executed.`

Fault injection via `$fs_inject()` occurs at the end of the time step.

\$fs_verify (system task)

For logic simulation.

```
always @(<trigger condition>) begin
    $fs_verify(var1,var2,var3);
end
```

Provides a verification point for VC Z01X when using the `-stim=verify` or `-stim=verify_all` option. When called, the simulation compares the specified simulation values with the expected eVCD or FSDB value. The expected stimulus values come from the eVCD or FSDB stimulus file. `$fs_verify` verifies signals for logic simulation and during the good machine signals of a fault simulation. The signals are not used for fault machine simulation.

Note:

VC Z01X automatically compares the set of signals used for fault detection when `$fs_compare` is called. You can use `$fs_verify` to set up alternative set of signals to be compared and you need to set up the appropriate trigger condition to call `$fs_verify` based on the knowledge of the design and test.

\$fs_verify_onevent (system task)

The `$fs_verify_onevent()` system task compares the specified simulation values with the expected simulation values from either the eVCD verification file or the eVCD stimulus file whenever one of the signals in the signal list changes. Use this system task with the `-stim=verify` option to provide verification points without providing specific timing information, to continuously monitor and verify value changes in the signal list. You must provide a signal list to specify the points to monitor and verify. `$fs_verify_onevent()` verifies signals for logic simulation and during the good machine signals of a fault simulation. The signals are not used for fault machine simulation.

Example syntax:

```
initial $fs_verify_onevent( <signal_list> );
```

System Tasks for User-Controlled Fault Detection

This set of system tasks provides flexibility in fault detection. They allow users to determine what constitutes a detect and to set the status of a fault based on more than just good machine/faulty machine signal value comparisons.

Capabilities

- Allow users to determine fault detection criteria
- They do not block the module from diverging
- Allow a user to control dropping or continuing to simulate a fault

For more details, see the following system tasks:

- [\\$fs_compare \(system function\)](#)
- [\\$fs_observe \(system function\)](#)
- [\\$fs_detect \(system function\)](#)
- [\\$fs_set_attribute \(system task\)](#)
- [\\$fs_add_attribute \(system task\)](#)
- [\\$fs_drop_status \(system task\)](#)
- [\\$fs_set_status \(system task\)](#)
- [\\$fs_default_status \(system task\)](#)
- [\\$fs_get_status \(system function\)](#)
- [\\$fs_strobe_onevent\(\) \(system task\)](#)

\$fs_compare (system function)

Causes the simulator to compare the good machine (GM) and faulty machine (FM) values for the listed signals. When you use this function, the simulator automatically handles all `$fs_compare()` calls as `#0 $fs_compare #0` to ensure the GM and FM values are synchronized during the value compare.

- If the good machine and faulty machine are the same value for all signals `$fs_compare()` will return 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs_compare()` will return 1.

- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_compare()` will return 2.
- If both the second and third condition are met in one `$fs_compare()` signal list, the function will return 1.

This function is only effective in the faulty machine. If this function is called from within the good machine, return value will be 0.

Following a good machine call to `$fs_compare()` a number of fault simulation checks occur at the end of the time step:

- **Oscillation fault checking:** For more information, see [Controlling Oscillation Detection](#).
- **Hyperfault checking:** For more information, see [Controlling Hyperfault Detection](#).

Syntax:

```
int compare;
compare = $fs_compare(sig1, sig2, sig3);
```

\$fs_observe (system function)

The `$fs_observe` system function causes the simulator to compare the (good machine) GM and faulty machine (FM) for the signals defined in the SFF file as observation points. See [FailureMode Section](#) for information about defining the observation points in the SFF file. The `$fs_observe` system function does not require a list of signals as an argument to the function like `$fs_compare`, as the signals to compare are defined in the SFF file.

Causes the simulator to observe the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs_observe()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs_observe()` returns 1.
- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_observe()` returns 2.
- If both the second (that is, return 1) and third conditions (return 2) are met in one `$fs_observe()` signal list, the function returns 1.

The `$fs_observe()` function is only effective when called from the faulty machine (FM). If this function is called from within the good machine (GM), the return value is 0.

The FailureMode associated with the FM determines the observation points when making a call to the `$fs_observe()` function.

Syntax:

```
$fs_observe();
```

Example:

Input SFF:	Strobe:	Coverage Results:
<pre>FailureMode fml { Observe { "tb.dut.sig1" "tb.dur.sig2" } SafetyMechanism(SM1) } SafetyMechanism SM1 { Detect { "tb.dut.sig3" } } FaultList fml { NA 0 { PORT "tb.dut.cell1.Z" } NA 0 { PORT "tb.dut.cell2.Z" } }</pre>	<pre>always @(negedge clock) if (\$fs_observe() == 1) \$fs_set_status("ON"); always @(sim_detect_sig) if (\$fs_detect() == 1) \$fs_drop_status("OD");</pre>	<pre># Set through \$fs_observe ON 0 { PORT "tb.dut.cell1.Z" } # Set through \$fs_detect OD 0 { PORT "tb.dut.cell2.Z" }</pre>

\$fs_detect (system function)

The `$fs_detect` system function causes the simulator to compare the (good machine) GM and faulty machine (FM) for the signals defined in the SFF file as detection points. See [SafetyMechanism Section](#) for information about defining the Detection points in the SFF file. The `$fs_detect` system function does not require a list of signals as an argument to the function like `$fs_compare`, as the signals to compare are defined in the SFF file.

Causes the simulator to detect the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs_detect()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs_detect()` returns 1.
- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_detect()` returns 2.
- If both the second (that is, return 1) and third conditions (return 2) are met in one `$fs_detect()` signal list, the function returns 1.

The `$fs_detect()` function is only effective when called from the faulty machine (FM). If this function is called from within the good machine (GM), the return value is 0.

The FailureMode associated with the FM determines the detection points when making a call to the `$fs_detect()` function.

Syntax:

`$fs_detect()`

Example:

Input SFF:	Strobe:	Coverage Results:
<pre> FailureMode fml { Observe { "tb.dut.sig1" "tb.dut.sig2" } SafetyMechanism(SM1) } SafetyMechanism SM1 { Detect { "tb.dut.sig3" } } FaultList fml { NA 0 { PORT "tb.dut.cell1.Z" } NA 0 { PORT "tb.dut.cell2.Z" } } </pre>	<pre> always @(negedge clock) if (\$fs.observe() == 1) \$fs_set_status("ON"); always @(sim_detect_sig) if (\$fs_detect() == 1) \$fs_drop_status("OD"); </pre>	<pre> # Set through \$fs_observe ON 0 { PORT "tb.dut.cell1.Z" } # Set through \$fs_detect OD 0 { PORT "tb.dut.cell2.Z" } </pre>

\$fs_detect (system function)

The `$fs_detect` system function causes the simulator to compare the (good machine) GM and faulty machine (FM) for the signals defines in the SFF file as detection points. See [SafetyMechanism Section](#) for information about defining the Detection points in the SFF file. The `$fs_detect` system function does not require a list of signals as an argument to the function like `$fs_compare`, as the signals to compare are defined in the SFF file.

Causes the simulator to detect the good machine (GM) and faulty machine (FM) values for the listed signals.

- If the GM and FM are the same value for all signals, `$fs_detect()` returns 0.
- If the GM is '0' or '1' and the FM is the opposite value for any signal, `$fs_detect()` returns 1.
- If the GM is '0' or '1' and the FM is 'x' or 'z' for any signal, `$fs_detect()` returns 2.
- If both the second (that is, return 1) and third conditions (return 2) are met in one `$fs_detect()` signal list, the function returns 1.

The `$fs_detect()` function is only effective when called from the faulty machine (FM). If this function is called from within the good machine (GM), the return value is 0.

The FailureMode associated with the FM determines the detection points when making a call to the `$fs_detect()` function.

Syntax:

`$fs_detect()`

Example:

Input SFF:	Strobe:	Coverage Results:
<pre> FailureMode fml { Observe { "tb.dut.sig1" "tb.dur.sig2" } SafetyMechanism(SM1) } SafetyMechanism SM1 { Detect { "tb.dut.sig3" } } FaultList fml { NA 0 { PORT "tb.dut.cell1.Z" } NA 0 { PORT "tb.dut.cell2.Z" } } </pre>	<pre> always @(negedge clock) if (\$fs.observe() == 1) \$fs_set_status("ON"); always @(sim_detect_sig) if (\$fs.detect() == 1) \$fs_drop_status("OD"); </pre>	<pre> # Set through \$fs.observe ON 0 { PORT "tb.dut.cell1.Z" } # Set through \$fs.detect OD 0 { PORT "tb.dut.cell2.Z" } </pre>

\$fs_set_attribute (system task)

This system task enables an active key, value attribute pair. The given key, value pair is stored to any fault detected while the key, value pair is active. This replaces any key, value already associated with the fault (not the "active" key, value). For more information, See [Specifying Custom Fault Attributes Using Key, Value Pairs](#).

Syntax

```
$fs_set_attribute(<verbose>, <key>[, <format>, <value>]);
```

\$fs_add_attribute (system task)

This system task enables an active key, value attribute pair. The given key, value pair is stored to any fault detected while the key, value pair is active. This appends the current value of key, if one already exists for that fault. System tasks are used in both the GM and FM. For more information, See [Specifying Custom Fault Attributes Using Key, Value Pairs](#).

Syntax

```
$fs_add_attribute(<verbose>, <key>[, <format>, <value>]);
```

\$fs_drop_status (system task)

This system task is added to the testbench at the points where the testbench is checking the results of a simulation. It causes the simulator to assign the specified status to a fault and drop the fault from further simulation. The status argument is a string constant containing a two character VC Z0IX status code. (See [Interpreting Fault Status](#).)

The signal list argument is an optional list of sense points for consideration by testability. Required for use with Testability and Fault Campaign Manager.

Syntax

```
$fs_drop_status(<"status">[, <signal_list>]);
```

\$fs_set_status (system task)

This system task is added to the testbench at the points where the testbench is checking the results of a simulation. It causes the simulator to assign a status to a fault and continue to simulate the fault. The status argument is a string constant containing a two character VC Z0IX status code. (See [Interpreting Fault Status](#).)

The signal list argument is an optional list of sense points for consideration by testability. Required for use with Testability and Fault Campaign Manager.

This task is only effective in the faulty machine.

Syntax

```
$fs_set_status(<"status">[, <signal_list>]);
```

\$fs_default_status (system task)

Causes the simulator to assign a status to all faults for which a status was not set by \$fs_drop_status() or \$fs_set_status(). This is typically the undetected status (ND). The status argument is a string constant containing a two character VC Z0IX status code. (See [Interpreting Fault Status](#)) This task is only effective in the good machine.

If no default status is set using \$fs_default_status() the default status is ND.

Syntax:

```
$fs_default_status("status");
```

\$fs_get_status (system function)

Causes the simulator to get the current status of a fault. A string constant containing a two character VC Z0IX status code is returned. This function is only effective in the faulty machine. If this function is called from within the good machine, an empty string will be returned.

Syntax:

```
#100 status = $fs_get_status();
```

\$fs_strobe_onevent() (system task)

The \$fs_strobe_onevent() system task compares the values of the good machine (GM) to the faulty machine (FM) for the listed strobe points whenever the signal changes in either of the machines. If `-stim=verify` is enabled, the values are also compared whenever the values in the stimulus file changes. Use this system task to specify nets, variables, or module instances as strobe points.

Syntax:

```
initial $fs_strobe_onevent( <signal_list> );
```

Example in Signal Value Strobing

The following example compares three signals. If the faults are 1 (the typical case of detection of an unsafe fault), the faults are set to DD and dropped from simulation. If the faults are 2, the faults are set to PT and continue simulating.

```
int compare;
always @(valid)
begin
  compare = $fs_compare(sig1,sig2,sig3);
  if (1 == compare)
    $fs_drop_status("DD");
  else if (2 == compare)
    $fs_set_status("PT");
end
```

Appendix C: Language Support

This appendix lists the supported SystemVerilog (IEEE 1800-2005, IEEE 1800-2009) and unsupported Verilog (IEEE 1364-1995, 1364-2001, 1364-2005) language constructs and system tasks and functions.

This section contains the following subsections:

- [Supported Language Constructs](#)
 - [Supported VCS Technologies](#)
-

Supported Language Constructs

All Language constructs supported by VCS are also supported for VC Z01X. Some may not be conducive for concurrent fault simulation.

Supported VCS Technologies

The following table lists the supported VCS features and the support level or error checks if the technology is used for VC Z01X.

Table 17 Supported VCS Technologies

VCS Technology	Error Check	Comments
XPROP	Yes	
Code/Assertion/Functional coverage	No	VC Z01X will disable internally the coverage
Partition Compile/Pre-Compile IP	Yes	
VHDL	No	Dropped as IA in concurrent and then run IA in serial mode.
AMS/MSV	No	Dropped as IA in concurrent and then run IA in serial mode.

Table 17 Supported VCS Technologies (Continued)

SDF	No	Serial mode only.
NLP	Yes	
Scalene	Yes	
Distributed Simulation	No	
Nettype	Yes	
FGP	Yes	
Simprofile	Yes	
Radify (+rad)	Yes	
Assertions/PSL	No	VC Z01X will disable internally
System C	No	Dropped as IA on boundary in concurrent and then run in serial mode.
SAIF		
Aspect-Oriented Extensions	No	
Open Vera	No	

18

Appendix D: Using VC Z01X with Verdi

The chapter consists of the following subsections:

- [Generating Verdi KDB](#)
- [Viewing eVCD Simulation Output in Verdi](#)
- [Enabling FSDB Results](#)
- [VC Z01X FSDB Options](#)
- [Supported Verilog System Tasks](#)
- [Generate Block Display](#)
- [Fault Analysis and Debug](#)
- [Supported Environment Variables and Simulation Command Line Options](#)
- [Limitations](#)

Note:

For more information on FSDB, see *Linking Novas Files with Simulators and Enabling FSDB Dumping* guide in Verdi documentation.

Generating Verdi KDB

Verdi Knowledge Database (KDB) is supported in VC Z01X ,add the `-kdb` option in compilation step, When you specify the `-kdb` option, Verdi creates the KDB and generates the design.

Example:

```
// Compile the design using VCS and generate Verdi KDB
% vcs -fsim -kdb <compile_options> <source files>
```

Viewing eVCD Simulation Output in Verdi

Many Verdi features are useful for viewing VC Z01X simulation results, including nTrace, nWave, and nRegister. See Verdi documentation from Synopsys for information about using these and other aspects of the tool.

Any waveform dump (eVCD) output of VC Z01X logic simulation is suitable for conversion to Verdi's FSDB (.fsdb) format. For example, you can open logic simulation results file directly from the Verdi waveform tool, which performs an automatic conversion. Alternatively, you can use Verdi's vfast utility; you can also generate the results of your VC Z01X logic simulation directly in FSDB.

Enabling FSDB Results

Ensure `VERDI_HOME` is defined and that `VERDI_HOME` is set to the same or newer version as VC Z01X. If VC Z01X does not find a defined `VERDI_HOME`, it will produce an error.

Command-line options have the highest priority, then environment variables, then system tasks. For example, if you have `$fsdbDumpfile("myfile.fsdb")` in your Verilog but put `+fsdbfile+newfile.fsdb` on the command-line, the FSDB file will be *newfile.fsdb*.

VC Z01X FSDB Options

The following option is available during compile:

The following options are available during simulation time:

<code>+fsdbfile+<filename></code>	Specifies a name for the FSDB output file for use with Verdi.
<code>+fsdb+strength[=on off]</code>	Overrides strength settings in the design when generating output for use with Verdi. For example, if <code>\$fsdbDumpStrength()</code> was not used when the design was compiled, <code>+fsdb+strength[=on]</code> would allow strength.

Supported Verilog System Tasks

The following Verilog system tasks are supported for use in designs that will be simulated for use with Verdi:

- `$fsdbDumpOn`, `$fsdbDumpOff`

The following options are supported:

- `+fsdbfile+filename`
- `+glitch`
- `$fsdbSwitchDumpfile`
- `$fsdbAutoSwitchDumpfile`
- `$fsdbDumpfile`
- `$fsdbDumpFinish`
- `$fsdbDumpflush`
- `$fsdbDumpvars`

The following options are supported:

- `+IO_Only`
- `+Reg_Only`
- `+mda`
- `+packedmda`
- `+packedmda+struct`
- `+struct`
- `+skip_cell_instance=mode` (`mode = 0~2`)
- `+all`
- `+generic`
- `+functions`
- `+fsdbfile+filename`
- `$fsdbDumpvarsByFile`

The following options are supported:

- +fsdbfile+filename (This option is used to generate FSDB file with given filename. It is used at runtime.)
- option_in_file (supported options in text file):
 - +IO_Only
 - +Reg_Only
 - +mda
 - +packedmda
 - +packedmda+struct
 - +struct
 - +skip_cell_instance=mode (mode = 0~2)
 - +all
 - +generic
 - +functions
- \$fsdbSuppress
- \$fsdbDumpMDA

Note:

The +all option does not include the +functions option.

For more information on the options, see *Linking Novas Files with Simulators and Enabling FSDB Dumping Guide* in Verdi documentation.

The following options are supported:

- +fsdbfile+filename
- +skip_cell_instance=mode (mode = 0~2)
- \$fsdbDumpMDAByFile

The following options are supported:

- +fsdbfile+filename
- option_in_file (supported options in text file):
 - +skip_cell_instance=mode (mode = 0~2)

The following example demonstrates the use of `$fsdbDumpfile` and `$fsdbDumpvars` Verilog system tasks:

```
`timescale 1ns/1ns
module test;
initial
begin
$fsdbDumpfile("test.fsdb")
$fsdbDumpvars(0,test);
end
endmodule
```

The usage model to compile and simulate the above design is as follows:

```
$VCS_HOME/bin/vcs test.v
```

Generate Block Display

For nested generate blocks, VC Z01X creates an artificial hierarchy that is easier to understand and navigate than the flattened name. For example, the following code creates the name `row1[1].col1[1].bit_ff`. This would normally be represented as one level of hierarchy, but in VC Z01X FSDB output, it shows up as three levels.

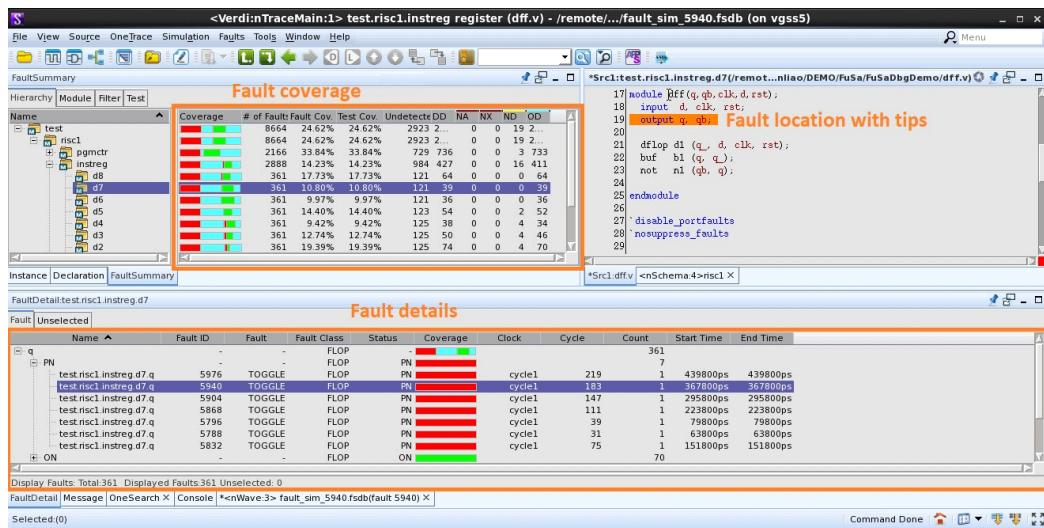
```
generate
begin
    genvar i;
    for (i = 0; (i < NUM_WORDS_STATUS); i = (i + 1))
        begin : row1
            genvar j;
            for (j = 0; (j < WORD_SIZE); j = (j + 1))
                begin : col1
                    FDE bit_ff();
                end
            end
        end
    endgenerate
```

Fault Analysis and Debug

Verdi® is used for debug purpose and conducting fault analysis by using both interactive command line options and environment variables. These commands perform some of the key functions of the tool and allow you to get information about the fault details and view GM/FM waveforms.

Using FDB Database

With the existing FDB flow, you can load FDB with fault campaign, FDB server and project name into Verdi. Both command line and environment variable works for Verdi.



ENV:

- SNPS_FDB_PROJECT
- SNPS_FDB_STORAGE_PATH
- SNPS_FDB_FAULT_CAMPAIGN

For more details on command line options, see [Supported Environment Variables and Simulation Command Line Options](#).

You can generate FSDB for the faulty machine using the above methodology and compare against the good machine (see [Logic Simulation](#)).

Dumping GM/ FM into the same FSDB

VC Z01X by default dumps FM fault. For both FM and GM, you can dump with "gmfm" mode.

Fault Campaign Manager provides a way to debug a fault location under question by adding the following information to the TCL script used.

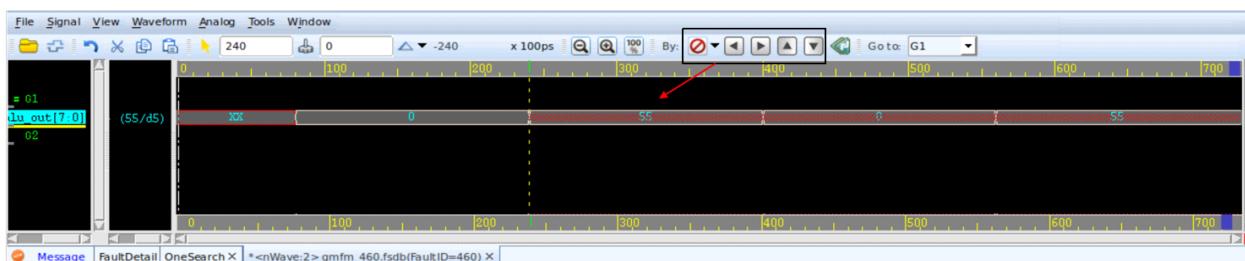
```
dump -fids <list> [-failure_modes <list of string>] -tc <string> -mode gmfm -fsdb <filename>
```

Usage: `dump` - Dumps an FSDB file with signal value changes of a single faulty machine.

- [-fids] - List of fault IDs.
- [-fm] - Name of the failure mode.
- [-tc] - Name of the testcase.
- [-mode] - Optional. Specify the dumping mode: 'fm' or 'gmfm' (default is 'fm').
- [-fsdb] - Path (relative to CWD or absolute) to fsdb file.

The following shows a sample portion of how the TCL script can be written to dump FM and GM with FID 47.

```
create_testcases -name user_test1 -exec "simv" \
-args "+TESTNAME=user_test1" -tsim_args "+fsdb+struct +fsdb+mda" \
-fsim_args "-fsim=limit+hyperactive+0"
create_testcases -name user_test2 -exec "simv" \
-args "+TESTNAME=user_test2" -tsim_args "+fsdb+struct +fsdb+mda" \
-fsim_args "-fsim=limit+hyperactive+0"
fsim
dump -fids 47 -tc user_test1 -mode gmfm -fsdb <target_filename>
```



For more information, see *Verdi User Guide*.

Supported Environment Variables and Simulation Command Line Options

The following table shows the environment variable and their equivalent command line option that are supported in the FSDB Dumper for VC Z01X.

Table 18 Environment Variables and Simulation Command Line Options

Environment Variable	Simulation Command Line Option
FSDB_DUMP_LIMIT	+fsdb+dump_limit=size
FSDB_FAULT_DUMP_MODE	+fsdb+fault_dump_mode+[gmfm fm]
FSDB_FILE	+fsdbfile+filename

Table 18 Environment Variables and Simulation Command Line Options (Continued)

Environment Variable	Simulation Command Line Option
FSDB_ALL	+fsdb+all
FSDB_FUNCTIONS	+fsdb+functions
FSDB_GLITCH	+fsdb+glitch=num
FSDB_IGNORE_VHDL_COMPLEX	+fsdb+ignore_vhdl_complex
FSDB_IO_ONLY	+fsdb+io_only
FSDB_MDA	+fsdb+mda
FSDB_PACKEDMDA	+fsdb+packedmda
None	+fsdb+no_all_msg
None	+fsdb+no_error
None	+fsdb+no_warning
FSDB_NO_PARALLEL	+fsdb+no_parallel
None	+fsdb+parallel=off
	FSDB_REG_ONLY
None	+fsdb+sequential
FSDB_SKIP_CELL_INSTANCE	+fsdb+skip_cell_instance=mode (mode = 0~2)
None	+fsdb+cell Note: This option will be ignored if skip_cell_instance is specified.
FSDB_STRENGTH	+fsdb+strength
FSDB_STRUCT	+fsdb+struct
None	+fsdb+writer_mem_limit=num
FSDB_GENERIC	+fsdb+generic

Limitations

- Verdi's interactive mode is not supported.
- Verdi's source debug features are not supported.

19

Appendix E: VC Z01X Command Map

The chapter consists of the following subsections:

- [Z01X to VC Z01X Command Map](#)
 - [Z01X to VC Z01X Fault Generation Command Map](#)
 - [Z01X to VC Z01X Simulation Command Map](#)
 - [Z01X to VC Z01X Reporting Command Map](#)
-

Z01X to VC Z01X Command Map

The following tables will use these abbreviations:

- N/A - Not applicable.
- NYI - Not Yet Implemented

Table 19 Z01X to VC Z01X Compile Command Map

Z01X Commands	VC Z01X Commands
-boundscheck	N/A
-cc '<string>'	-cc <string>
-cellaware	NYI
-cflg (<options>)	-CFLAGS <string>
-cpp <string>	-cpp <string>
-design	-o <filename>
-f <filename>	-f <filename>
-F <run.f>	-F <run.f>
-fcc	N/A
-flop	N/A

Chapter 19: Appendix E: VC Z01X Command Map
 Z01X to VC Z01X Command Map

-fsdb	-debug_access
-fsdb_legacy	N/A
-fsdb_reader	N/A
-gcc_path <path>	N/A
-h	-help
-i <filename>	-i <filename>
-ignore <keyword_argument>	-ignore <keyword_argument>
-l <filename>	-l <filename>
-lflg <option>	-LDFLAGS <options>
-legacy_zoix	N/A
-Mdir=<name>	-Mdir=<name>
-memdeposit	N/A
-mem2pac	N/A
-mem2pac+<module1>+<module2>	N/A
-mem2pacmax+<maxsize>	N/A
-mem2pacdiag	N/A
-nonbaudpsched	N/A
-o <name>	-o <name>
-onlyzoioxopts	N/A
-portfaults	-fsim=portfaults -debug_region=cell+lib
-portfaultarrays	NYI
-propagate	N/A
-q	-q
-svnet	N/A

Chapter 19: Appendix E: VC Z01X Command Map
 Z01X to VC Z01X Command Map

-svnetport	N/A
-svnet_noreg	N/A
-u	-u
-v <filename>	-v <filename>
-verdi_home	N/A
-version	-id
-w	N/A
-y <directory>	-y <directory>
+alwaystrigger	N/A
+ams	-ams
+amsext+<ext1>+<ext2>+<. ..>	N/A
+assert+disable	-assert disable
+cli+force	N/A
+decompile+<filename>	N/A
+define	+define
+defparam+<hiref>=<value>	-pvalue+<string>=<string>
+delay_mode_distributed	+delay_mode_distributed
+delay_mode_distributed_p ath+ (min max avg)	NYI
+delay_mode_distributed_p ath	NYI
+delay_mode_fault+valueti mescale	NYI
+delay_mode_path	+delay_mode_path
+delay_mode_unit	+delay_mode_unit
+delay_mode_zero	+delay_mode_zero
+design+<design>	-o <filename>

Chapter 19: Appendix E: VC Z01X Command Map
 Z01X to VC Z01X Command Map

+fault+array+r +w +rw	NYI
+fault+transient_hold_udp	NYI
+fault+var	N/A
+inmdir+<directory>	+inmdir+<directory>
+iopath+edge+match	N/A
+libext+<string>	+libext+<string>
+liborder	+liborder
+librescan	+librescan
+libverbose	+libverbose
+lic_wait+<time>	+vcsllic+wait
+max_err_count+<number>	NYI
+maxdelays	+maxdelays
+maxlogs+<num>	N/A
+mindelays	+mindelays
+modelsim	N/A
+neg_tchk	+neg_tchk
+noautonaming	NYI
+noautoudp	N/A
+nolibcell	+nolibcell
+nomodportcheck	N/A
+noneg_tchk	+noneg_tchk
+no_notifier	+no_notifier
+no_path_cond	+no_path_cond
+noprunef+vars +cell +gates +prims +tasks +funcs +<moduleName> +<filename> +combine	N/A

+nosource	N/A
+nospecify	+nospecify
+no_tchk_msg	+no_tchk_msg
+notimingcheck	+notimingcheck
pli +load<filename>:<function>+<filename>:<function>	N/A
+pli+main+<func_name>	-e <string>
+pli+table+<filename>{+<filename>}	-P <filename>
+pli+vcstable+<filename>{+<filename>}	N/A
+precision+<value>	N/A
+profile	N/A
+protect	+autoprotect
+sdfverbose	+sdfverbose
+sdf_file+<file_name.sdf>	-sdf <file_name.sdf>
+sdf_path+<instance_name>	NYI
+suppress+cell	-fsim=suppress+cell
+sv	-sverilog
+svext	+systemverilogext+<string>
+task+noiochange	N/A
+timescale+<unit>/<prec>	-timescale <unit/pres>
+timescale+override+<unit>/<prec>	-override_timescale=<string>/<string>
+timescale+precision+<time>	N/A
+top+<module>{+<module>}	-top <module>
+transport_int_delays	+transport_int_delays

+transport_path_delays	+transport_path_delays
+typdelays	+typdelays
+undriven+<value>	NYI
+vcs	N/A
+verbose+cpu	N/A
+verbose+optimized	N/A
+verbose<+prim><+behav><+assign><+directive>\<+scope><+parameter><+generate><+array>	N/A
+verbose+svnet	N/A
+verbose+udp	N/A
+verbose+undriven	N/A
+verbose+unused	N/A
+wdir+<dir>	N/A
+wreal+<default 4state sum avg min max>	N/A

- **Note:**

The default compile time unit for Z01X is ns and VC Z01X is s.

Z01X to VC Z01X Fault Generation Command Map

Note:

Synopsys recommends running fault generation through Fault Campaign Manager.

Table 20 Z01X to VC Z01X Fault Generation Command Map

Z01X Commands	VC Z01X Commands
+collapse	-collapse on
+design+<design>	N/A

Table 20 Z01X to VC Z01X Fault Generation Command Map (Continued)

+dut+<module>+<integer> +<string>	-dut_path <hierarchical path of dut instance>
+format+<string>	-format <string>
+ignore+force	NYI
+ignore+suppress+<PRIM>+<PORT>+<WIRE>+<FLOP>+<VARI>+<ARRY>+<BRID>	-ignore_suppress
+lic_wait+<time>	NYI
+lic+verbose	NYI
+max_err_count+<integer>	NYI
+maxfault+<integer>	NYI
+maxlogs+<integer>	N/A
+model+stuck<<+prim><+port><+net><+assiggn><+in><+out><+var><+array>>	N/A
+nocollapse	-collapse off
+sample+ci+<float>+cl+<float>	-sample ci:<float>,cl:<float>
+sample+num+<integer>	-sample num:<integer>
+sample+pct+<float>	-sample pct:<float>
+scramble	N/A
+status+old_fault_status1+old_fault_status2+...=<new_fault_status1> <old_statuses_group1>+<old_status_group2>...=<new_fault_status1>	NYI
+template+<filename>	NYI
+udppins	NYI
+verbose<<+scalefactor><+conflict>>	NYI
+warnreturn+<exit_status>	N/A
-collsampling	NYI
-designdir <directory>	-daidir <directory>

Table 20 Z01X to VC Z01X Fault Generation Command Map (Continued)

-excludedcoloff	-excludedcoloff
-faultlistwildcard	NYI
-fdb_load_fms <string>	N/A
-fdef <filename>	-fdb_path <directory>
-fr <filename> <filename> ...	-faultlist or -sff
-fstrobe <filename>	NYI
-full_reconvergence_check	NYI
-fullsampling	NYI
-inmdir <directory>	NYI
-l <logfile>	-log <logfile>
-maintainflop	N/A
-nofloparray	NYI
-seed <float>	-seed <float>
-statusimport	NYI
-systematicsampling	-systematic_sampling
-tracerandomsystask	NYI
-uncontrollability	-uncontrollability

Z01X to VC Z01X Simulation Command Map

Note:

Synopsys recommends running fault simulation through Fault Campaign Manager.

Table 21 Z01X to VC Z01X Simulation Command Map

Z01X Commands	VC Z01X Commands
---------------	------------------

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

+cli+noecho	N/A
+cli+vcs	N/A
+countoncse	-fsim=countoncse
+debug+<level>	N/A
+design+<design>	N/A
+dictionary+values+all	NYI
+err_line_length+<num>	N/A
+fault+disable+<status option>	-fsim=fault+disable+<status>
+fault+inject+<time>+sync	-fsim=fault+inject+<string>
+fault+limit<+testbench,+all>	(Compile) -fsim=dut:<string>
+fault+machine+"<fault id>" <fault_id>+<fault_id2>	-fsim=fault+machine+<id>
+fault+messages	-fsim=fault+messages
+fault+stats<+hypercheck+progress+strobe>	NYI
+fault+status+<<status options> <status_group>>	-fsim=fault+status+<string>
+fault+update+<time>	NYI
+finishonsave+<filename>	NYI
+fmd+file+<file_name>	N/A
+forcelist+dut+<string>[+<string>]	(Compile) -stim=forcelist-stim=forcelist_dut:<string>[+<string>]
+forcelist+file+<filename>	(Compile) -stim=forcelist-stim=forcelist_file:<filename>
+forcelist+mode+<string>	(Compile) -stim=forcelist-stim=forcelist_mode:<string>
+fsdb+autoname+<file_name_template>	N/A

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

+fsdb+cell	+fsdb+cell
+fsdb+clk+<string>[+<string>]	-stim=type:fsdb,clk:<string>
+fsdb+drive+x+0	NYI
+fsdb+drive+x+1	NYI
+fsdb+drive+z+0	NYI
+fsdb+drive+z+1	NYI
+fsdb+dut<instance>+fsdb_instance	NYI
+fsdb+fault_dump_mode+gmfm fm	NYI
+fsdb+file+<filename>	-stim=type:fsdb,file:<filename>
+fsdb+init+file<filename>	NYI
+fsdb+limit+mismatch+<integer>	- stim=type:fsdb,verify_mismatch_limit=<integer>
+fsdb+nowarn	NYI
+fsdb+strength=(on off)	
+fsdb+undriven+0	NYI
+fsdb+undriven+1	NYI
+fsdb+verify	-stim=type:fsdb,verify
+fsdb+verify+<filename>	NYI

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

+initvar+0 1	+vcs+initreg+0 1. To enable initialization for an entire design, the +vcs+initreg+random option must be specified at compile time and one of the following options must be specified at runtime: +vcs+initreg+0 +vcs+initreg+1 +vcs+initreg+random +vcs+initreg+seed_value Example 1: % vcs +vcs+initreg+random [other_vcs_options] file1.v file2.v file3.v % simv +vcs+initreg+random [simv_options] All Verilog variables, registers and memories are assigned random initial values. Example 2: % vcs +vcs+initreg+random [other_vcs_options] file1.v file2.v file3.v % simv +vcs+initreg+0 [simv_options] All Verilog variables, registers and memories are assigned with initial value as 0. For more details, see VCS User Guide.
+lic_wait+<time>	+vcs+lic+wait
+lic+verbose	N/A
+limit+hyperact+<real>	-fsim=limit+hyperactive+<int>
+limit+hyperact+immediate	NYI
+limit+hypercheck+<#>	NYI
+limit+hyperfault+0	NYI
+limit+hypermem+<int>	NYI
+limit+hypermem+immediate	NYI
+limit+hypertrophic+<real>	NYI
+limit+hypertrophic+immediate	NYI
+limit+loop+<#>	-fsim=limit+loop+<integer>
+limit+maxstrokes+<#>	NYI
+limit+mindetects+<#>	NYI
+limit+osc+<num>	(Disable) -fsim=noosc

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

+limit+paging+<num>	NYI
+max_err_count+<num>	NYI
+no_cancelled_e_msg	+no_cancelled_e_msg
+no_notifier	+no_notifier
+no_pulse_msg	+no_pulse_msg
+no_show_cancelled_e	+no_show_cancelled_e
+no_tchk_msg	+no_tchk_msg
+noc2	N/A
+noreschedule+gate	N/A
+osc_info	NYI
+osct+messages+<num>	NYI
+oscd+forcex	NYI
+oscd+ignore	NYI
+pathpulse	+pathpulse
+profile+hierarchical	NYI
+progress+<num>	+progress+<num>
+pulse_e/<num>	+pulse_e/<num>
+pulse_r/<num>	+pulse_r/<num>
+pulse_e_style_ondetect	+pulse_e_style_ondetect
+pulse_e_style_onevent	+pulse_e_style_onevent
+pulse_r/<num>	+pulse_r/<num>
+show_cancelled_e	+show_cancelled_e
+toggle+countx	NYI
+toggle+nopropagation	NYI
+toggle+start+<time>	NYI

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

+transport_int_delays	+transport_int_delays
+transport_path_delays	+transport_path_delays
+vcd+autoclk	NYI
+vcd+clk+<signals>	(eVCD only) -stim=type:evcd -stim=clk:<string>
+vcd+dumpfile+<filename>	-vcd <filename>
+vcd+dumppon+<time>	+vcs+dumppon+<t>+<ht>
+vcd+dumpvars+<filename>	+vcs+dumpvars+<filename>
+vcd+dut+<instance>+<vcd_instance>	NYI
+vcd+dut+level+<lowest_stim_level>	NYI
+vcd+file=<filename>	-stim=type:evcd,file:<filename>
+vcd+init+file+<filename>	NYI
+vcd+limit+mismatch+<num>	- stim=type:evcd,verify_mismatch_limit=<integer>
+vcd+verbose	NYI
+vcd+verifyall	(eVCD) -stim=type:evcd,verify_all
+vcd+verifyall+<verification_file>	NYI
+vcs+initreg+0 +1 +x +z +random	+vcs+initreg+0 +1 random
+vcs+initregsync+0 +1 +x +z +random	N/A
+verify+priority+<stimulus verify>	NYI
+verify+tolerance+<time>	(FSDB) -fsim=verify_tolerance
+weight+dcecount+<weight>	NYI
-cdfout <file.cdf>	NYI
-detectoa	NYI
-fdef <file.fdef>	N/A

Table 21 Z01X to VC Z01X Simulation Command Map (Continued)

-fpp <num>	N/A
-fppmax <num>	(FCM) set_config -max_faults_per_fsim_task
-i <filename>	-i <filename>
-ignore_stimulus_switches	N/A
-l <filename>	-l <filename>
-maxhard <num>	-fsim=maxhard:<num>
-maxpot <num>	-fsim=maxpot:<num>
-maxt <num>	+vcs+finish+<time>
-osc	NYI
-path <unix_path>	N/A
-q	-q
-restart	N/A
-s	-ucli
-stim_ignore_scalar_ranges	NYI
-stim_nostrobe	NYI
-stimulus_strength <integer string>	NYI
-toggle <togfile>	N/A
-uniq_prior_drop_fault=<status>	N/A
-vcspli	N/A
-zdetect	NYI

Z01X to VC Z01X Reporting Command Map

Table 22 Z01X to VC Z01X Reporting Command Map

Z01X Commands	VC Z01X Compile Commands
+collapseoff	-nocollapse
+definition	NYI
+design+<name>	N/A
+dictionary+values+base+<string>	NYI
+dut+add+<string>	NYI
+dut+delete+<string> <integer>	NYI
+fault+status+<string>	-faultstatus <string>
+format+<string>	-format <string>
+group+<none summary detail>	NYI
+hierarchical+<level>	-hierarchical <integer>
+maxlogs+<num>	N/A
+model+<string>	NYI
+pins+format+zoix	NYI
+sort+<sort option>	-sort <string>
+summary	NYI
+csv	-csv
+togg+file+<.vtog file>	NYI
+udppins	NYI
+verbose	-verbose
+weight+ <<status> <status_group>>=<value> ...	NYI
-designdir <path>	N/A
-dict <name>	NYI

Table 22 Z01X to VC Z01X Reporting Command Map (Continued)

-dictout <name>	NYI
-faultlimit <integer>	-faultlimit <integer>
-fdef <file.fdef>	-fdb_path <directory>
-filterattributes	-filterattributes <string>
-filterproperties	NYI
-fulltotal	NYI
-gz	-gz
-help	-help
-l <filename>	-log <filename>
-long	NYI
-nogrouptransient	NYI
-out <filename>	-report <filename>
-path <string>	NYI
-reportmode<1 2>	N/A
-showattributes	-showattributes <string>
-showproperties	NYI
-showtestfaultinfo	NYI
-tf <file>	N/A
-v	N/A