Functional Safety for Implementation User Guide

Version T-2022.03-SP4, September 2022

SYNOPSYS®

Feedback

Contents

Preface

This preface includes the following sections:

About This User Guide

The Functional Safety Implementation User Guide describes how to use features provided by Synopsys tools in a complete end-to-end functional safety flow. This flow helps in implementing functional safety mechanisms and measures such as redundancy with registers or cores while also adhering to freedom from interference requirements such as tree splitting, placement, well and route separation, and so on.

This manual is intended for IC or sytem-on-chip design engineers who implement functional safety requirements for automotive applications and who are already familiar with Synopsys tools, such as Fusion Compiler, Design Compiler NXT, IC Compiler II, Formality, IC Validator, and TestMAX FuSa. Before using this manual, you should be familiar with the following topics:

- High-level design techniques
- ASIC design principles
- Functional partitioning techniques

This preface includes the following sections:

Related Products, Publications, and Trademarks

For additional information about the Fusion Compiler tool, see the documentation on the Synopsys SolvNetPlus support site at the following address:

https://solvnetplus.synopsys.com

You might also want to see the documentation for the following related Synopsys products:

- Design Compiler® NXT
- Formality®

- Fusion Compiler ™
- IC Compiler™ II
- IC Validator
- TestMAX™ FuSa

Feedback

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default value, such as `include_enclosing = true | false`
[ ]	Denotes optional arguments in syntax, such as `write_file [-format fmt]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*.
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Feedback

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

The SolvNetPlus site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNetPlus site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

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

https://solvnetplus.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

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

Contacting Customer Support

To contact Customer Support, go to https://solvnetplus.synopsys.com.

1

Design Flow Overview for Functional Safety Analysis and Implementation

Describes the design flow overview for functional safety analysis and implementation

Safety critical applications in the automotive industry have stringent demands for functional safety and reliability. The following Synopsys tools provide the functional safety flows for creating designs that are robust against failures:

- Design Compiler NXT
- Formality
- Fusion Compiler
- IC Compiler II
- IC Validator
- TestMAX FuSa

The TestMAX FuSa performs fast static analysis to calculate ISO 26262 functional safety metrics such as Single Point Fault Metric (SPFM). To improve functional safety, the tool

- Runs functional safety analysis at the RTL or gate level netlist for the full hierarchy
- Identifies and reports hotspots early in the design cycle
- Provides an ordered list of registers that can be replaced to improve SPFM

These tools provide support for automotive chip safety and reliability requirements according to ISO 26262 functional safety requirements and provides strategies to

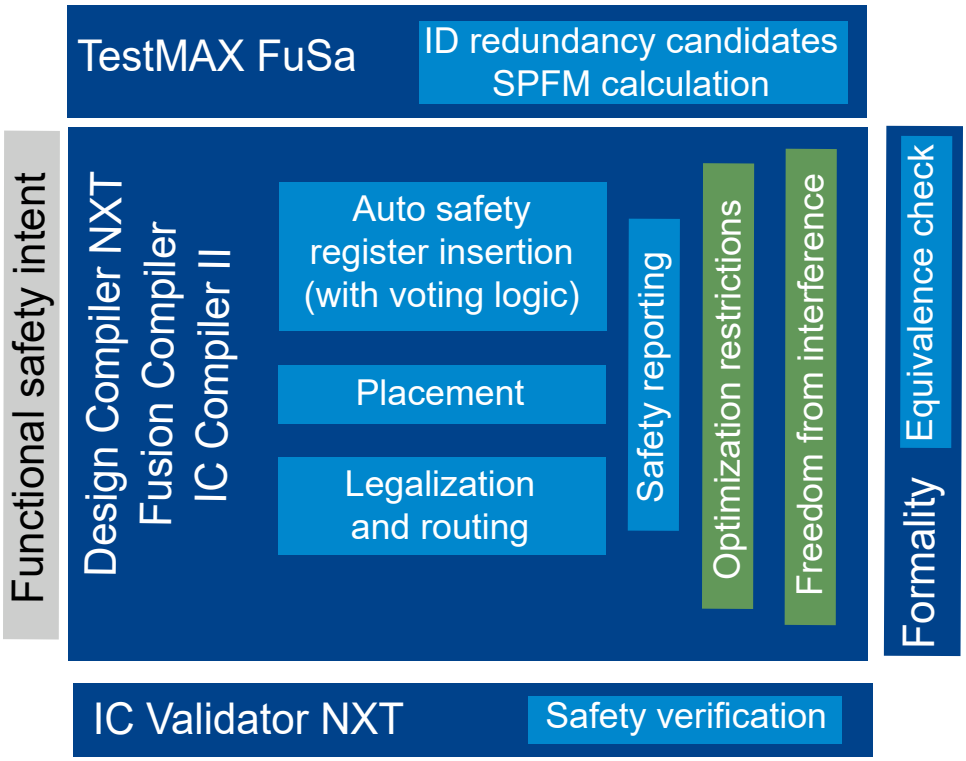
- Avoid transient failures such as a single-event upset (SEU) by using redundancy strategies such as triple-modular redundancy (TMR) and Dual Core LockStep (DCLS) flow
- Physically separate logic groups and have identical cores to operate in lockstep, such as Dual Core LockStep (DCLS)

This guide provides use models, functionalities, and limitations to support triple-modular redundancy (TMR), dual-modular redundancy (DMR), and fault-tolerant registers using these Synopsys tools that

- Identify registers that are safety critical
- Define and honor repelling bounds among the group of redundant registers
- Avoid common rail supplies or use tap cells to connect to the rail supplies

Figure 1 shows an overview of a design flow using Synopsys tools. These tools implement safety mechanism and report safety status.

Figure 1 Using Synopsys Tools for Functional Safety



2

Single Point Fault Metrics Analysis Using TestMAX FuSa

Describes the single point fault metrics analysis using TestMAX FuSa

The TestMAX FuSa tool performs functional safety analysis early in the design flow to provide guidance for changes resulting in ISO 26262 functional safety metrics improvements. ISO 26262 is the automotive industry standard for functional safety with the goal of avoiding unacceptable risks due to hazards caused by malfunctioning or unintended behavior of electrical or electronic systems. ISO 26262 defines automotive safety integrity level (ASIL), which is the level of risk reduction needed to achieve a tolerable risk for safety.

The functional safety engineers should perform a hazard and risk analysis based on the malfunction of critical safety functions in the design and determine the ASIL to achieve a tolerable risk. ISO 26262 defines the metrics to measure the functional safety targeted by ASIL as shown in Table 1.

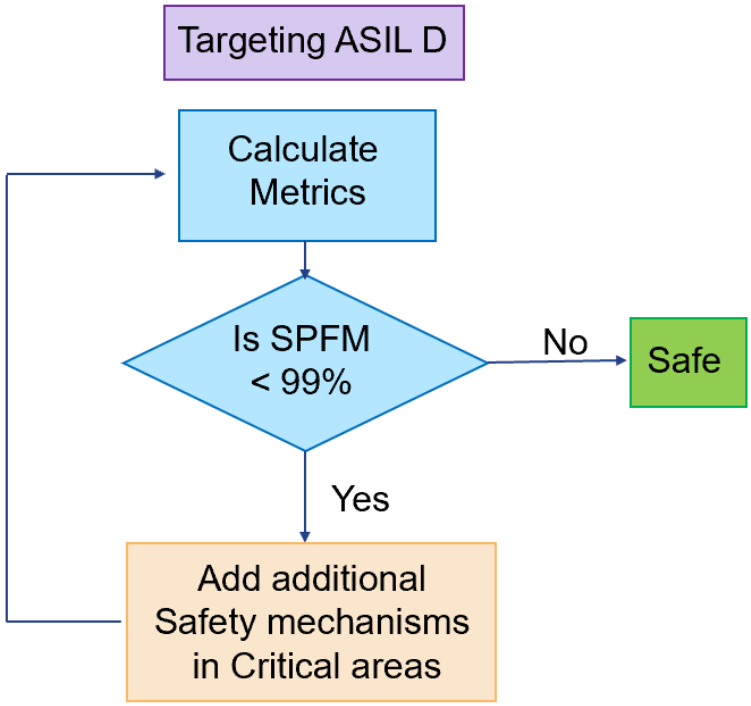
Table 1 Single Point Fault Metrics

Metric	ASIL A	ASIL B	ASIL C	ASIL D
Single point fault metric	NA	> 90%	> 97%	> 99%

Single point fault metric (SPFM) reflects the robustness of an item or function to an occurrence of a single fault that directly leads to the violation of a safety goal. The higher the ASIL, such as ASIL D, the more stringent level of safety reduction is required.

Functional safety analysis evaluates if the targeted safety level is achieved in a design. Otherwise, it determines which parts of the design should be enhanced for safety readiness. The workflow in Figure 2 shows the example for running functional safety analysis on ASIL D.

Figure 2 Workflow for Running Functional Safety Analysis on ASIL D



The TestMAX FuSa tool:

- Uses fast static analysis-based approach to calculate ISO 26262 functional safety metrics, such as Single Point Fault Metric (SPFM) either at RTL or gate netlist.
- Identifies blocks in the design that have the highest probability of causing functional safety failure.
- Reports a subset of registers to be replaced with error-tolerant equivalent registers.

This section includes the following topics:

Feedback

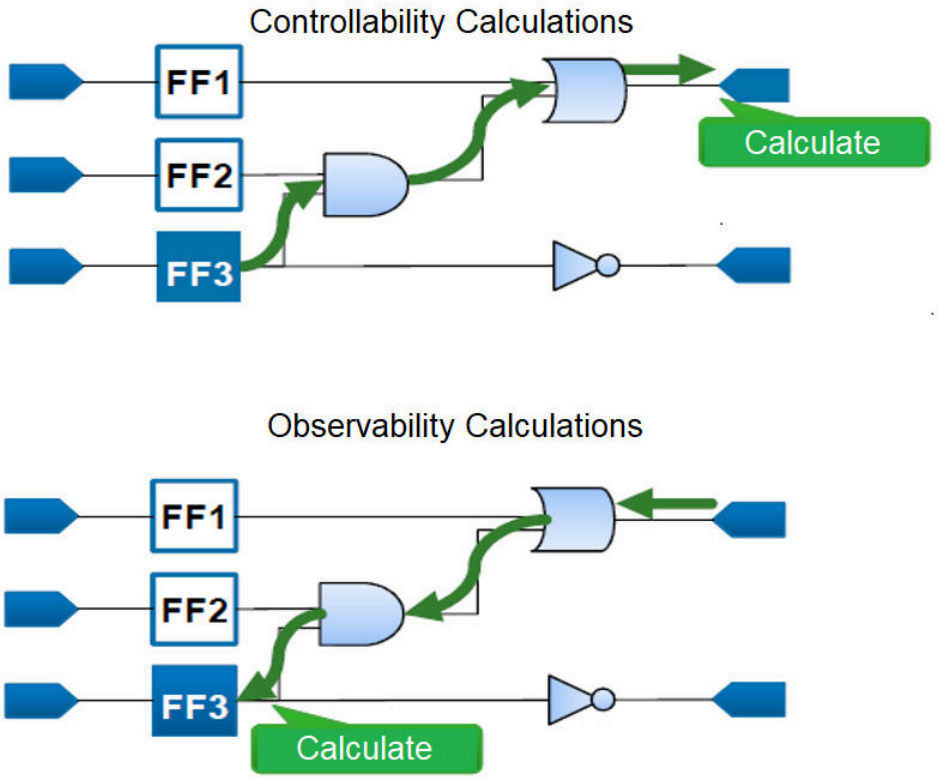
- Constraining DFT Control Signals
- Identifying Redundant Registers

Soft Error Analysis

Soft errors can cause catastrophic failures during operation of a safety critical design if they reach safety-related signals. The ISO 26262 SPFM calculation indicates if the risk of a soft error propagation through the design and impact on the critical functions is high. Then, the design must be modified to make it more tolerant to such errors.

The TestMAX FuSa tool performs soft error analysis on the complete design based on the probabilities of control and observe as shown in Figure 3.

Figure 3 Controllability and Observability Calculations



Calculating Controllability and Observability Metrics

All numerical values reported by the TestMAX FuSa tool depend on the probabilities of control and observe. The control and observe metrics are different for the TestMAX FuSa

Feedback

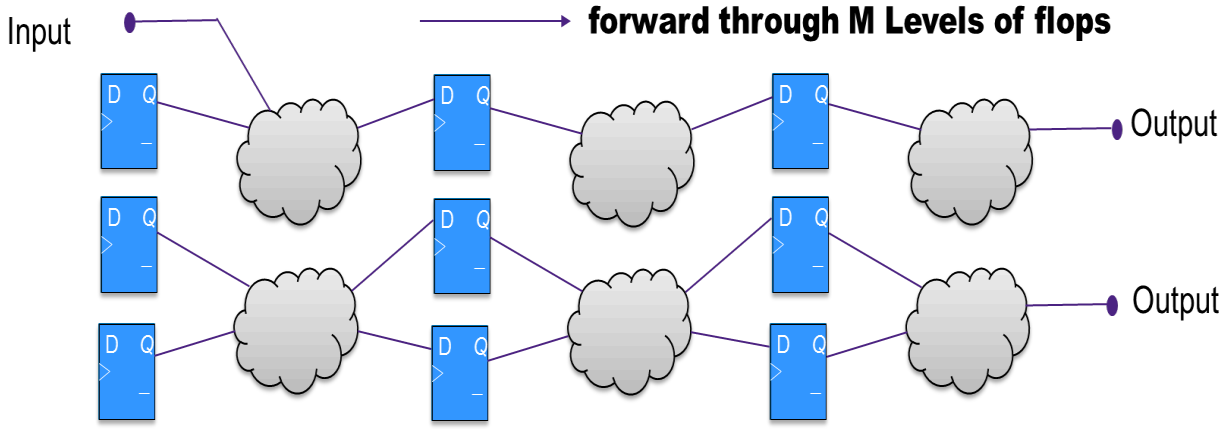
tool and the DFT tool, because the measurements are only for functional operation and not for scan operation in the DFT tool. So, the metrics is calculated using sequential circuit behavior and is not restricted to combinational logic analysis. The probabilities of 1 and 0 are calculated separately, but their sum is 100 percent. Other than testability calculation, the control and observe metrics must consider registers that are not scanned.

Controllability Calculation

The controllability calculation is performed as follows:

1. Initialize the probability of control to `1/0` on registers and primary inputs to `0.5`.
2. Propagate corrected values with the sequential SCOAP algorithm further until it reaches the number of levels set by the `ser_control_sequential_depth` parameter (the default is `100`) or by a primary output.
3. Produce the controllability values.

Figure 4 Multi-time-frame Controllability Calculation



Observability Calculation

Initial probabilities of observe are set as follows:

- Safety-critical registers and primary output ports are set to 1.
- Non-safety-critical primary output ports are set to 0.
- Non-safety-critical registers are set to `ser_observe_nsr_initial_value` (the default is 0.5).
- Non-safety-related registers are set to 0.

Feedback

By default, all primary output ports are safety-critical, and all registers are non-safety-critical. They can be changed by using the `safety_critical`, `non_safety_critical`, and `non_safety_related` attributes.

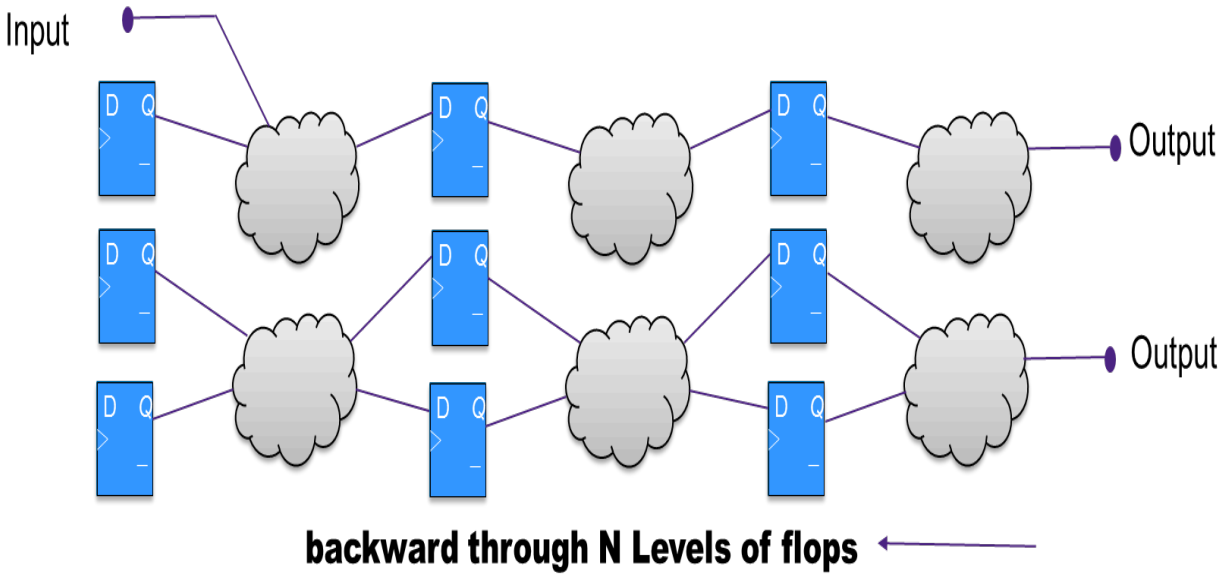
If a testbench exists and simulates the functional workflow of the design, the results in the FSDB file can be used to provide actual controllability values. In this case, the controllability calculations explained in these steps are not performed. To use the FSDB file, set the `ser_use_fsdb_for_control_probabilities` parameter and add the constraint to SGDC file, as follows:

```
activity_data -format fsdb -file <filename>
```

The observability calculation is performed as follows:

1. Use the probability values for controllability calculation.
2. Propagate corrected observe probabilities backward until they reach the number of levels set by the `ser_observe_sequential_depth` parameter (the default is `100`) or by a primary input or by a safety-related register.

Figure 5 Multi-time-frame Observability Calculation



The observability calculations can be further refined by setting either one of the following:

- Black boxes with the `ser_blackbox_control_0_probability` or `ser_blackbox_observe_probability` parameter
- Initial observe values on individual registers with the `force_probability` attribute

  Input ports of a design that do not change can be specified either with the `test_mode -functional` or `set_case_analysis` command.

Only the probability values for observe are used in the SPFM and diagnostic coverage calculations. The controllability values are used to get correct observability values but are not used in further calculations.

Soft Error Propagation

To perform soft error propagation analysis, set the `set_goal_option rules Info_soft_error_propagation` rule as part of the goal.

Commands are not required for soft error propagation analysis but might be specified to get required information. The commands that are used in the DFT tool, for example, the `clock` and `test_mode -scanshift` commands are not used for soft error propagation analysis.

For more information about the parameters and reports, see *TestMAX FuSa Rules Reference Guide*.

In the violations window, right-click on SPFM data *For Design Unit* as shown in Figure 6 to open the Fault Browser spreadsheet in the GUI. The `Info_soft_error_propagation` rule calculates SPFM using the following formulas:

$$SPFM = 1 - \frac{(\Sigma\ (\lambda_{sm}[i] \times (1 - DC[i]) \times (1 - P_{safe}[i])))}{(\Sigma\ \lambda_{reg}[i])}$$

$$LFM = 1 - \frac{((\Sigma\ (\lambda_{sm}[i] \times (1 - DCL[i]\ )\ )\ )}{(\Sigma\ (\lambda_{reg}[i] + \lambda_{sm}[i] - \lambda_{reg}[i] \times (1 - DC[i]) \times 1 - P_{safe}[i])))}$$

$$PMHF = \Sigma\ (\lambda_{reg}[i] \times (1 - DC[i]\ ) \times (1 - P_{safe}[i])) + \Sigma\ (\lambda_{sm}[i] \times (1 - DC[i]))$$

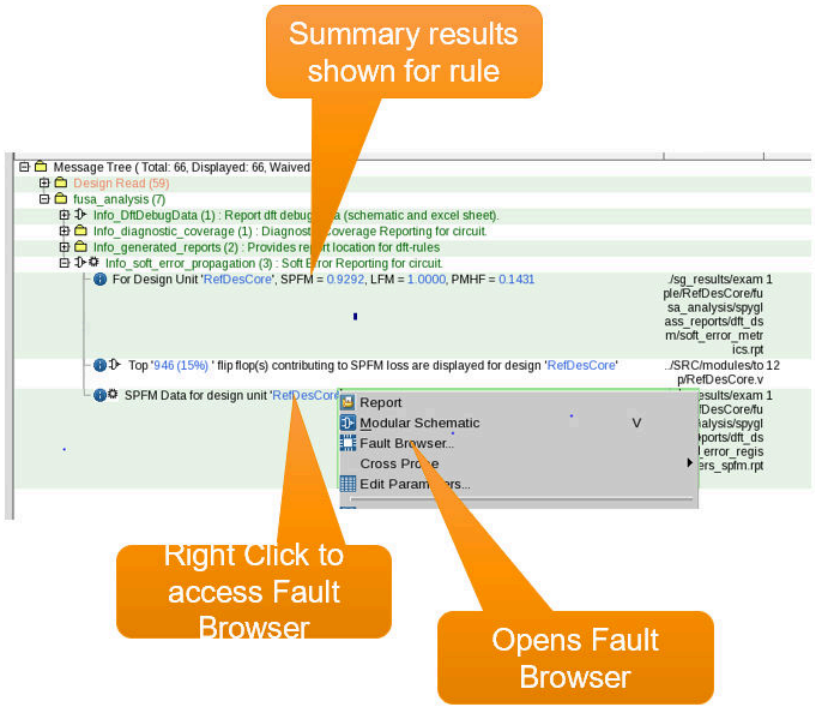
Where,

- SPFM is Single Point Fault Metric.
- LFM is Latent Fault Metric.

Feedback

- PMHF is Probability Metric for random Hardware Failures.
- λ_{reg}[i] is the soft error rate for each register [i].
- λ_{sm}[i] is the failure rate of the safety mechanism for each register [i]. The value is 0 if it has no safety mechanism.
- DC[i] is the diagnostic coverage for a fault on the same register. The value is 0 if it has no safety mechanism.
- DCL[i] is the diagnostic coverage for a latent fault on the same register. The value is 0 if it has no safety mechanism.
- P_{safe}[i] is 1-OBS[i]. OBS[i] is the probability that a fault effect can propagate from the same register to a safety-critical signal.

Figure 6 Fault Browser Spreadsheet



Separate failure rates are also reported for safety-critical registers, non-safety-critical registers, and safety mechanism registers, along with the total failure rate. The total failure rate is the sum of safety-critical registers, non-safety-critical registers, and safety mechanism registers.

Feedback

You must specify the per-register values used in the following formulas with the `ser_data` command:

`ser_data -name module/instance -`λ_{reg} `n [-`λ_{sm} `n] [-dc n] [-dcl n]`

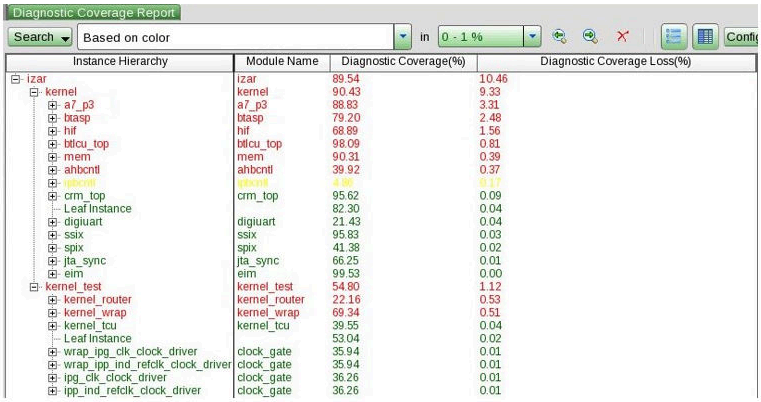
Where,

- `-name` specifies the module of the register or an instance name. If a block instance name is specified, the command applies to all registers within the block unless the block is overridden by a specific instance name.
- `-`λ_{reg} specifies λ_{reg} in units of Failure in Times (FITs). FIT is defined as one failure per billion (10^9) hours of operation). λ_{reg} is the rate of failure of a single register. The default depends on the following context:
  - If no register is assigned with an `-`λ_{reg} value, the default for all registers is `0.01`.
  - If any register is assigned with an `-`λ_{reg} value, the default for all unassigned registers is `0`.

Diagnostic Coverage Analysis

Diagnostic Coverage (DC) determines the proportion of dangerous faults or failures that can be detected or controlled by safety mechanisms implemented in the design. The TestMAX FuSa tool uses static analysis to calculate diagnostic coverage for all hierarchical instances in the design. The tool calculates diagnostic coverage for permanent faults such as aging related defects and measures the effectiveness of safety mechanisms for dangerous faults. This helps identify hotspot instances and their contribution to the loss in design diagnostic coverage. The Figure 7 shows Diagnostic Coverage Report browser:

Figure 7 Diagnostic Coverage Report Browser



Diagnostic coverage analysis is performed when the `set_goal_option rules Info_diagnostic_coverage` rule is part of the goal.

The commands are not required for diagnostic coverage analysis but might be used to get required information. The commands that are used in the DFT tool, for example, the `clock` and `test_mode -scanshift` commands, are not used for diagnostic coverage analysis.

For more information about the parameters and reports, see *TestMAX FuSa Rules Reference Guide*.

Note:

The report of probabilities by instance, which is useful for debugging low diagnostic coverage, is not printed by default. Use the `diagnostic_coverage_report_probability` parameter to print the report.

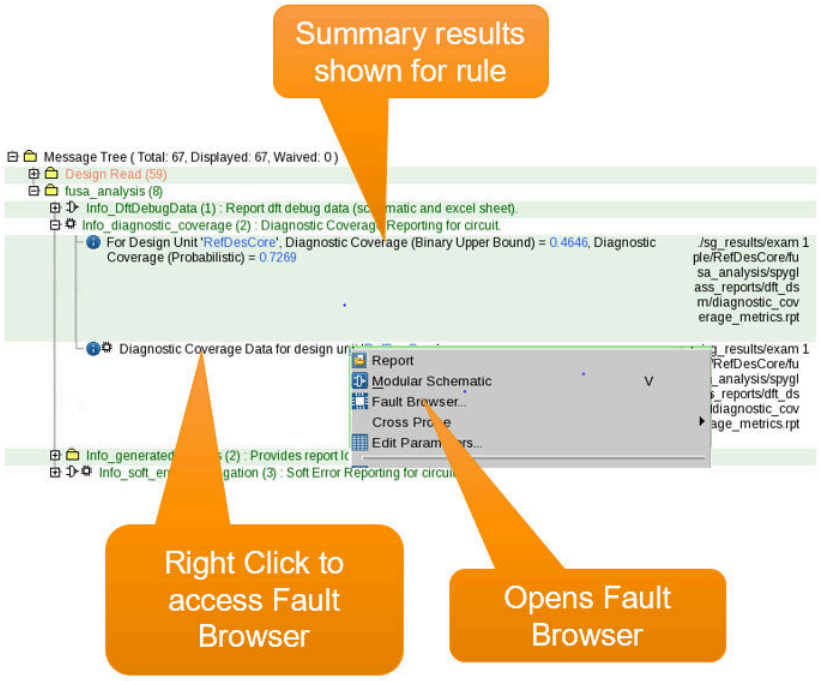
In the violations window, right-click on Diagnostic Coverage data *For Design Unit* as shown in the following figure to open the Fault Browser spreadsheet in the GUI. The `Info_diagnostic_coverage` rule calculates SPFM by using the following formula:

$$DC\ (Binary) = (1\ /\ n) \times \Sigma\ (1 - OBS_{SCP}[i] > 0\ \Sigma\ OBS_{SM}[i] = 0)$$

$$DC\ (Probabilistic) = (1\ /\ n) \times \Sigma\ (1 - OBS_{SCP}[i] \times (1 - OBS_{sm}[i]))$$

Where,

- `n` is the number of nodes in the design.
- $OBS_{SCP}[i]$ is the observability of net `[i]` with respect to any safety critical design point.
- $OBS_{SM}[i]$ is the observability of the same net with respect to any safety mechanism detection point.



The diagnostic coverage provides both the probabilistic and binary metrics:

- Probabilistic metric: Shows the coverage with a given number of clock cycles.

  To specify the number of clock cycles, run the `diagnostic_coverage_cycle_count_at_source_of_error` parameter with a nonnegative integer. The default is 1000.

- Binary metric: Shows the limit of the coverage with a given infinite number of clock cycles.

  Use the binary metric to qualify the effectiveness of the safety mechanisms. The probabilistic metric is more useful for debugging low-coverage areas of the circuit, so the Fault Browser shows the modules by coverage loss using this metric. This information allows you to identify the safety mechanism candidates.

To identify safety mechanism detection points, use the following command:

```
fusa_detection_point -name pin/port/net
```

To identify safety-critical observe points, use the following command:

```
fusa_observe_point -name pin/port/net
```

The `fusa_observe_point` command can also be used for soft error propagation analysis.

Feedback

Identifying TMR Candidates

The TestMAX FuSa tool can identify triple-modular redundancy (TMR) candidates for insertion by the Synopsys tools. This analysis might be done at the RTL or gate level. This section describes how to perform RTL analysis and provides a set of commands to accomplish RTL analysis. TestMAX FuSa is based on the SpyGlass technology and therefore it is run in the SpyGlass environment.

This section includes the following topics:

Project File

The project file is a Tcl file with a set of commands to use in the TestMAX FuSa tool. The following example shows a project file to identify TMR candidates:

```
#Global options
set_option nosavepolicy all
set_option dw no
set_option support_sdc_style_escaped_name yes
set_option enable_unified_naming_search yes
set_option allow_module_override yes

#Design
set_option top top_level_module_name

read_file -type sourcelist rtl_filelist.f
read_file -type sgdc project_name.sgdc

define_goal dft_ser_analysis -policy { dft_dsm }
{
    set_goal_option rules Info_soft_error_propagation
}
```

Source List File

The source list file is a Verilog file with a list of source directories and files. This can also be the same list of design files used for Verilog functional simulation of the RTL, excluding the testbench files. The following example shows the first few lines of a source list file:

```
+libext+.v
-y directory_path_name
```

```
+incdir directory_path_name
directory_path_name/module_name_1.v
directory_path_name/module_name_2.v
...
```

SpyGlass Design Constraints File

The SpyGlass Design Constraints (.sgdc) file contains constraints that are used to control analysis of a set of defined rules. For more information, see *TestMAX FuSa Rules Reference Guide*.

The following example shows a SGDC file specifying the same FIT rate for each register in the design, which is appropriate for RTL analysis:

```
current_design "top_level_module_name"
    ser_data -name top_level_module_name -lreg 0.000001
```

The specific FIT rate to be used is not important for identifying TMR candidates.

Commands to Identify TMR Candidates

The following command line creates a list of TMR candidates in a batch mode:

```
spyglass -shell -project project_file_name.prj -goal dft_ser_analysis
 -batch
```

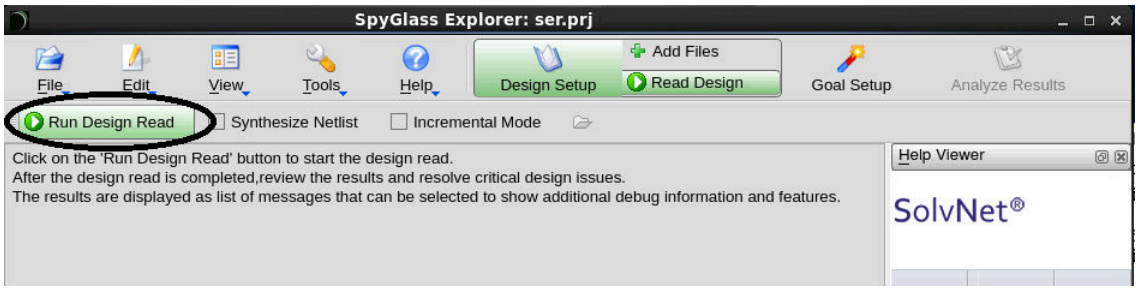
Alternatively, use the following command line to start the TestMAX FuSa tool in the GUI:

```
spyglass -project project_file_name.prj
```

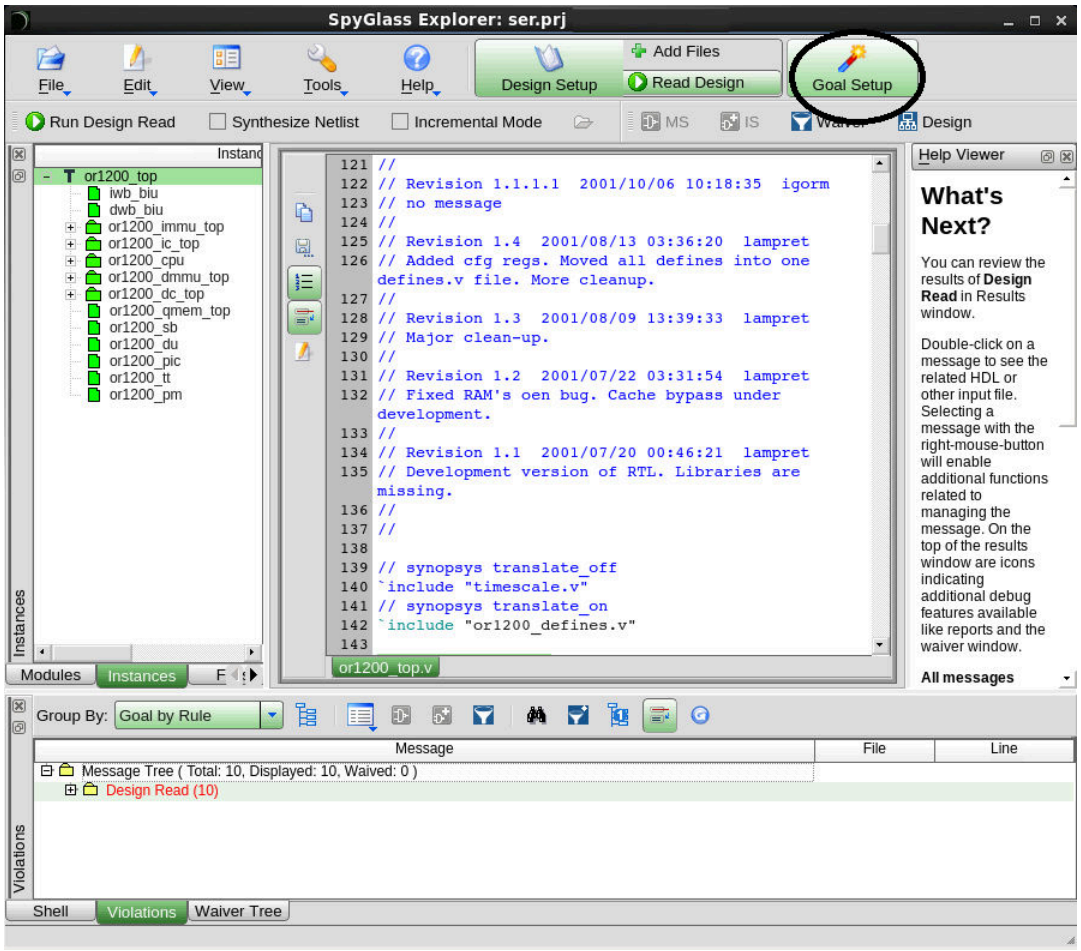
Feedback

In the SpyGlass Explorer application window,

1. Click the Run Design Read button.

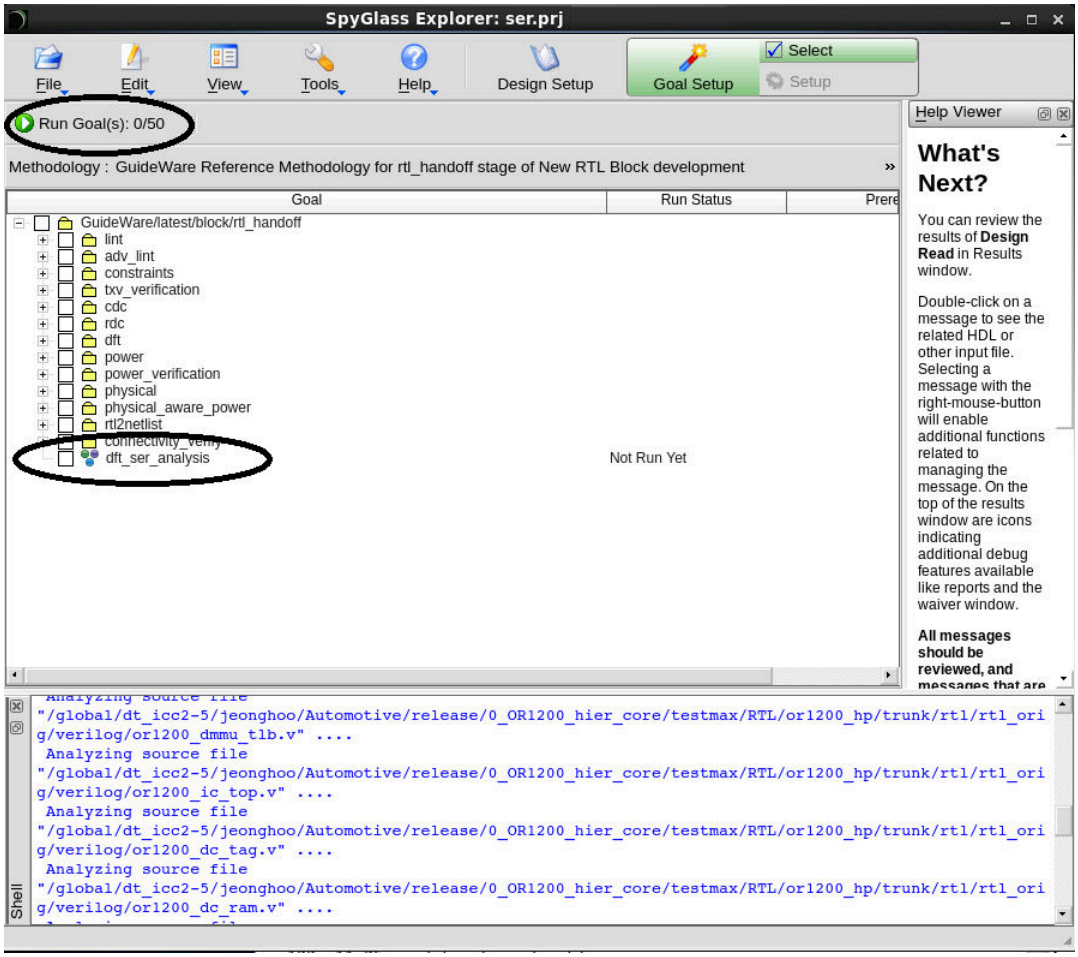


If the design is not read successfully, the tool reports an error message. Otherwise, click the Goal Setup button and select `dft_ser_analysis`.



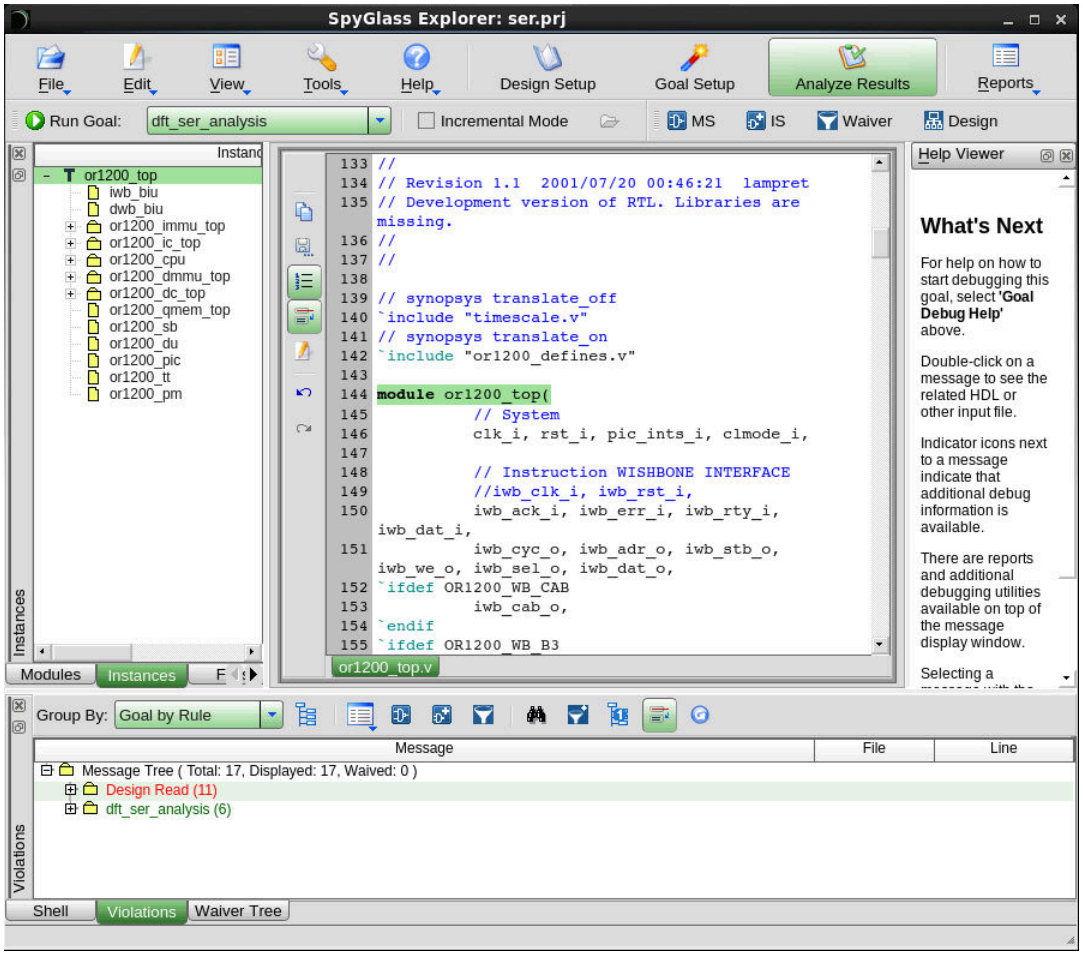
Feedback

2. Select `dft_ser_analysis` and then click the Run Goals (1/50) button.



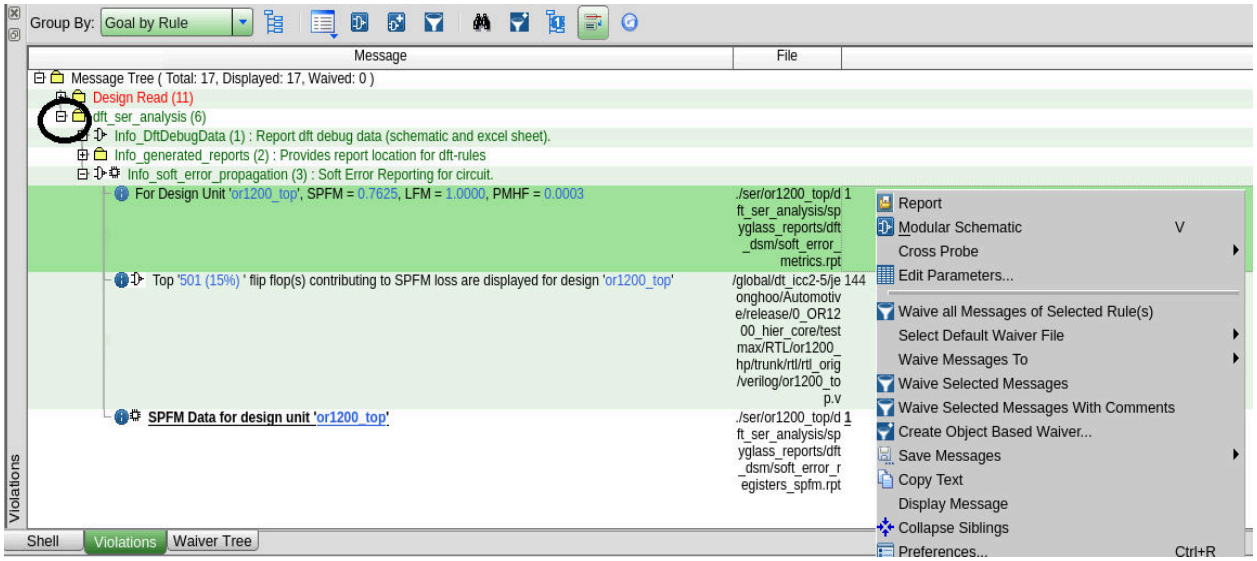
Feedback

The following results appear at the bottom of the window after completing running the goals:



Feedback

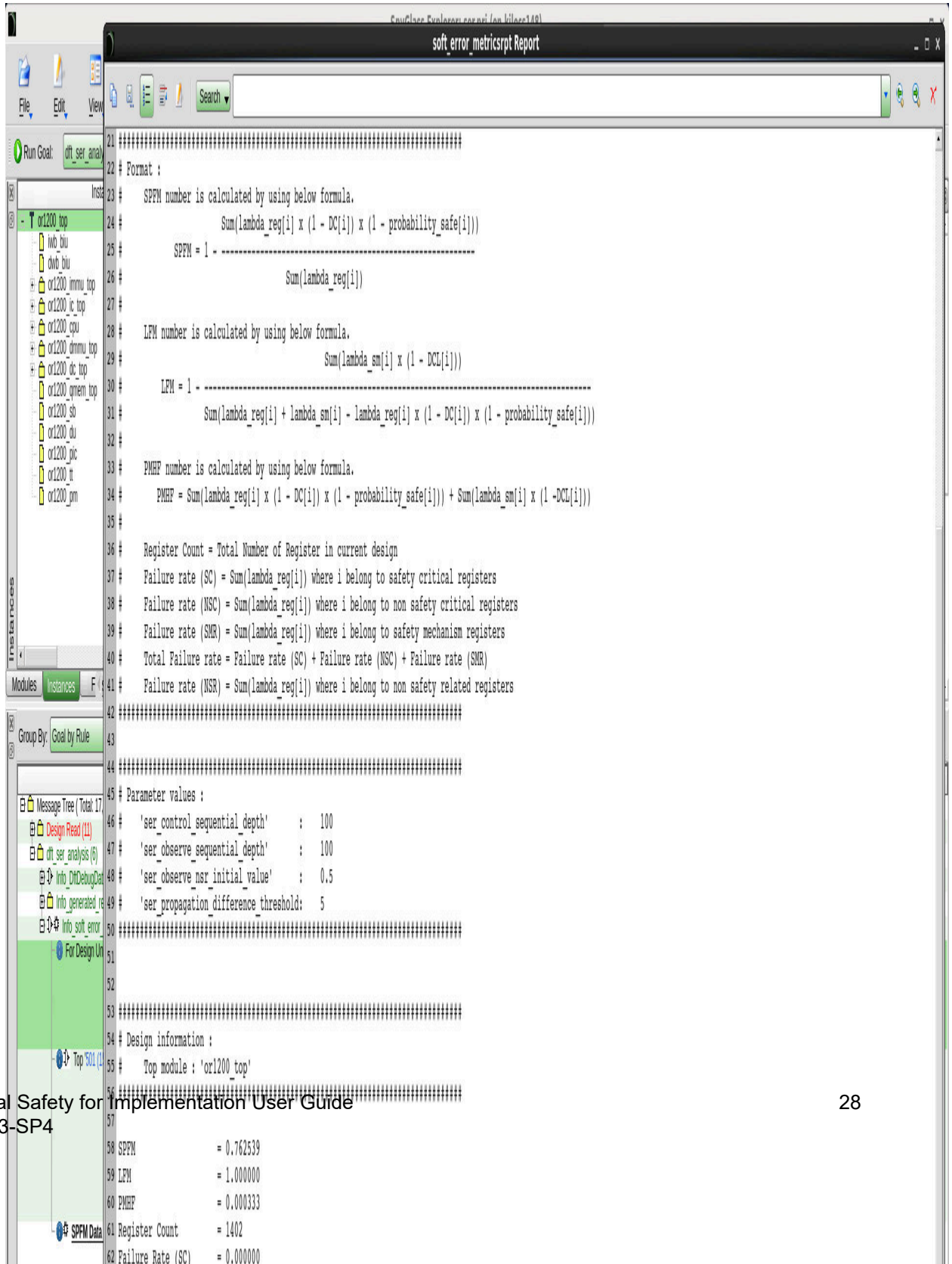
3. Click the plus (+) sign next to `dft_ser_analysis`.



Feedback

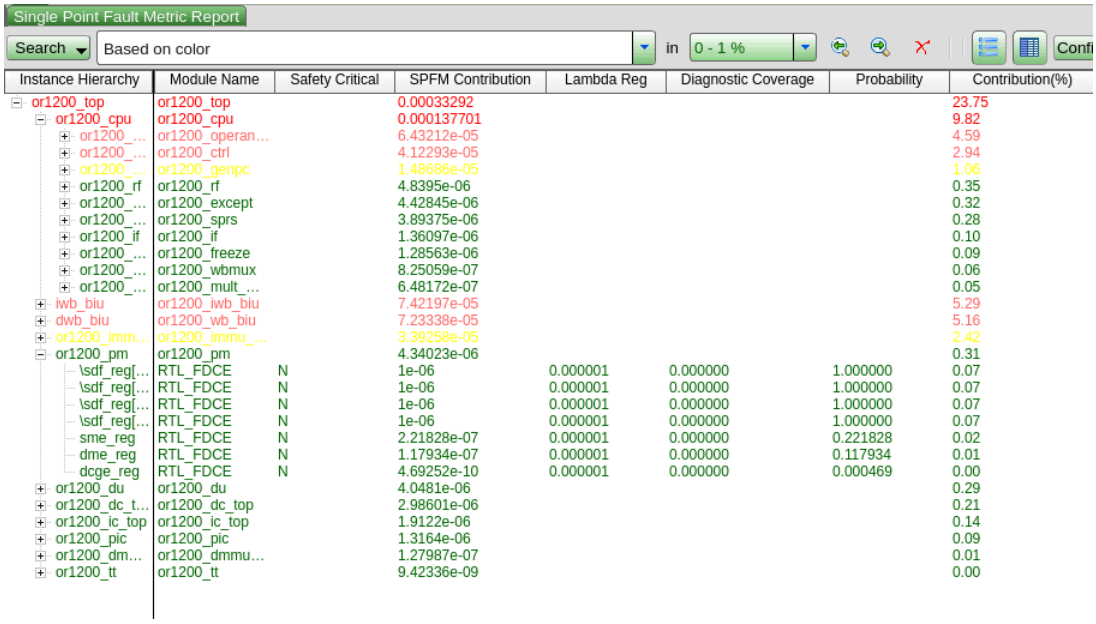
4. Click the plus (+) sign next to *Info_soft_error_propagation* to view the resulting messages. The paths to the report files are on the right-hand side.

   You can also select the report, left-click, and choose Report to view it, as shown in the following figure to view the SPFM report:



Feedback

6. Left-click on *Fault Browser* to view the SPFM contributions of the different design hierarchies:



The contribution of each module is color-coded by the percentage contribution to guide browsing. The contribution of each register in the design can be browsed.

The TMR candidates are stored in the following file:

```
project_file_name/consolidated_reports/top_level_module_name_dft_ser_anal
ysis  \
/soft_error_registers_spfm.rpt
```

The soft_error_registers_spfm.rpt file starts with about 50 lines of commented text that describes how the file was generated and what it is, because you can still interpret the file even if it is removed from its context. After the comments, a header is printed followed by the TMR candidates in order of priority, as shown in the following example with the first few TMR candidates:

```
Index         Safety SPFMContribution lambda_reg DC          Probability
 Contribution % Cumulative-Contribution %      Module
 Register Name
1              N      0.000001         0.000001   0.000000
 1.000000     0.071327       0.071327         "RTL_FDCE"
 "or1200_top.or1200_cpu.or1200_operandmuxes.\operand_b_reg[0] "
2              N      0.000001         0.000001   0.000000
 1.000000     0.071327       0.142653         "RTL_FDCE"
 "or1200_top.or1200_cpu.or1200_operandmuxes.\operand_b_reg[1] "
3              N      0.000001         0.000001   0.000000
 1.000000     0.071327       0.213980         "RTL_FDCE"
 "or1200_top.or1200_cpu.or1200_operandmuxes.\operand_b_reg[2] "
```

Feedback

Note:

The escaped instance names are followed by a white space before the double-quote symbol (” ”) ending the instance field. Non-escaped instance names do not include a white space before the double-quote symbol (” ”).

By default, the file contains 500 registers or 15 percent of the registers in the design, whichever is greater. If this number is larger than the budget for inserting TMRs, the tool lists the candidates starting with index number 1 and continue down the list until the budgeted number is reached.

Final Gate-Level Analysis

When design implementation is complete, you can use the TestMAX FuSa tool to report the final soft-error metrics with a gate-level netlist file. The following differences are observed between the RTL-level and gate-level netlist runs instead of reading the RTL source file, the gate-level run reads the final netlist as the HDL file and the Liberty source files of the synthesis libraries.

The SGDC file includes

- Constraints to force DFT control signals into functional mode.
- Constraints to prevent analysis of registers added during DFT insertion.
- Definitions for TMR registers to correctly identify redundant logics that are added during implementation.

Note:

After accounting all the differences, the SPFM values for the RTL-level and gate-level netlist runs must correlate closely.

Reading a Netlist

The project file uses the `read_file -type sourcelist` command at the RTL level. A gate-level netlist file is required for running the final gate-level analysis. In the project file, add the `read_file` command and specify the netlist file to read in. The synthesis libraries are required to interpret the gate-level netlist, but the DFT tool checks require the Liberty source to be read.

The design is read using the following commands:

```
read_file -type hdl top_level_design.v
read_file -type gateslib library1.lib
read_file -type gateslib library2.lib
...
```

The design netlist that is written by the synthesis tool is the one that should be read with the `-type hdl` option. The Liberty source files of all synthesis libraries that are read by the synthesis tool should be read with the `-type gateslib` option.

Constraining DFT Control Signals

The implementation process adds the DFT logic that is not part of the functional design. For the analysis, the DFT logic should be constrained so that the DFT functions are disabled. This allows the design functions to be analyzed without interference from DFT functions.

The signals used to configure the DFT logic can be found in the SPF files written after DFT insertion with the `write_test_protocol` command. The functional mode is the opposite of the state constrained in the SPF file. Some signals are *Test Mode* selects, that select different test modes with different selections. To identify these signals, all SPF files must be inspected to find the combination that is not used. These signals are listed in the `test_setup` macro, and the final states of all non-clock signals should be used.

Following are the constraints that are added to the SGDC file for the design example

```
test_mode -name "or1200_top.TM1" -value 0 -functional
test_mode -name "or1200_top.TM2" -value 0 -functional
test_mode -name "or1200_top.TM3" -value 0 -functional
test_mode -name "or1200_top.occ_testmode" -value 0 -functional
test_mode -name "or1200_top.scan_en" -value 0 -functional
test_mode -name "or1200_top.scan_en_icg" -value 0 -functional
```

Preventing Analysis of DFT Registers

The registers in the DFT logic must be constrained as non-safety-related registers to prevent from affecting the SPFM analysis. If the registers in the DFT logic are included in the analysis, the reported SPFM value increases. This is because the non-DFT logic is in functional mode so that the soft errors should not propagate in the DFT registers. The increased SPFM value does not reflect the exact functional circuit operation.

All modules inserted by the `insert_dft synthesis` command should be constrained as non-safety-related. These modules can be identified by comparing the top-level instances of the RTL-level and gate-level netlists. The constraints required for the design example are as follows:

```
non_safety_related -name "or1200_top.U_DFT_TOP_IP_0"
non_safety_related -name "or1200_top.occ_ctrl_clk_i"
non_safety_related -name "or1200_top.occ_ctrl_clk_cpu_i"
non_safety_related -name "or1200_top.rst_ctrl"

// All 368 wrapper cells must be constrained
non_safety_related -name or1200_top_WC_D1_test_0
```

Feedback

```
...
non_safety_related -name or1200_top_WC_D1_test_367
```

Identifying Redundant Registers

Describes the process to identify redundant registers.

The TMR registers substituted by the Synopsys tools must be identified so that their effect on soft error propagation can be calculated. To calculate soft error propagation, the TMR registers must be extracted by the synthesis tool. For example,

- In the Design Compiler NXT tool, use the following command to generate the safety data report.

  ```
  write_safety_register_script -output safety_data.rpt
  ```

- Write the following command for each TMR register:

  ```
  mark_safety_register -name  safety_register_group_0 -rule  rule1
   -registers \
  [get_cells {iwb_biu/wb_dat_o_reg[6] iwb_biu/wb_dat_o_reg[6]_tmr1
   iwb_biu/wb_dat_o_reg[6]_tmr2}] \
  -logic [get_cells {iwb_biu/tmr_voting_logic_hier_0/U1}]
  ```

- Convert each TMR register into a series of TestMAX FuSa constraints by using the following command:

  ```
  grep mark_safety_register safety_data.rpt | awk '{print $8,$9,$10}' |
   \
  sed 's/^/fusa_redundancy -instance /' | sed 's/\//./g' | sed 's/]$/
   -module  \
  or1200_top/' >> tmr.sgdc
  ```

- List the TestMAX FuSa constraints in the following format:

  ```
  fusa_redundancy -instance {iwb_biu.wb_dat_o_reg[6]
   iwb_biu.wb_dat_o_reg[6]_tmr1 \
  iwb_biu.wb_dat_o_reg[6]_tmr2} -module or1200_top
  ```

3

Safety Specification Format Files

Describes the Safety Specification Format Files for the implementation tools

Safety Specification Format (SSF) is a set of Tcl commands, which allow specification of the following safety mechanisms to be implemented in the design:

- Dual or triple module redundancy
- Safety registers
- Error protection schemes
- Safety Cores
- Failsafe finite state machines

Safety Specification Format (SSF) files describe the key safety mechanisms, such as safety register, safety error code, safety core, and failsafe FSM, to use with safety-aware implementation and verification flows. The following commands are examples of SSF:

- To describe the logical and physical properties for the implemented safety mechanism, use the following commands:
  - `create_safety_register_rule`, where the safety mechanism is safety register
  - `create_safety_error_code_rule`, where the safety mechanism is safety error code
  - `create_safety_core_rule`, where the safety mechanism is safety core
  - `create_failsafe_fsm_rule`, where the safety mechanism is failsafe FSM
- To associate the rule with the implemented safety mechanism, use the following commands:
  - `set_safety_register_rule`, where the safety mechanism is safety register
  - `set_safety_core_rule`, where the safety mechanism is safety core
  - `set_failsafe_fsm_rule`, where the safety mechanism is failsafe FSM

Feedback

- To specify an implemented safety mechanism and associate it with the rules, use the following commands:
  - `mark_safety_register`, where the safety mechanism is safety register
  - `mark_safety_error_code`, where the safety mechanism is safety error code
  - `mark_safety_core`, where the safety mechanism is safety core
  - `mark_failsafe_fsm`, where the safety mechanism is failsafe FSM
- `load_ssf`: Reads and interprets an SSF file in all the tools except the IC Validator tool. For details, see Reading Safety Specification Format Files.
- `write_ssf` or `save_ssf`: Writes an ASCII file in SSF format in the Design Compiler NXT, Fusion Compiler and IC Compiler II tools. For details, see Writing Safety Specification Format Files.

Note:

The complete syntax for the SSF language is not covered in this document.

Table 2 displays the read-only application options and application variables to see the SSF version information of the files in the Fusion Compiler, Design Compiler NXT, IC Compiler II, and RTL Architect tools respectively:

Table 2 SSF Version Information

Fusion Compiler, IC Compiler II, and RTL Architect	Design Compiler NXT	Description
`file.ssf.current_version`	`ssf_current_version`	Stores the current SSF version supported by the tools
`file.ssf.supported_versions`	`ssf_supported_version`	Stores the list of all the SSF versions supported by the tools

Reading Safety Specification Format Files

To read and interpret an SSF file, use the `load_ssf` command. This command reads and interprets the SSF commands contained in the specified file, translates them into the appropriate functional safety intent, and updates the design accordingly in all the tools except the IC Validator tool.

```
load_ssf file_name
```

The `file_name` argument is mandatory for the `load_ssf` command. The value of the `file_name` argument can be a relative path or absolute path to that file. The file must exist at the specified location and must be readable.

Handling Checks and Errors

The SSF reader performs syntactic as well as semantic checks for each entry in the input file. These checks include the following:

- Unrecognized command name or command option
- Missing `ssf_version` command at the beginning of the file
- Repeated `ssf_version` command
- Unsupported version of input file (marked using `ssf_version` command)
- All other checks done for command correctness, for example, correctness of object specification, option value check, range check, and so on.

When any syntax or semantic error is encountered while reading the file, the `load_ssf` command stops reading and returns to the prompt with a Tcl error.

```
fc_shell> load_ssf top.ssf
Information: Loading SSF file '/user/ssf/top.ssf' (FILE-007)
ssf_version 2.0
Error: Unsupported SSF version
Use error_info for more info. (CMD-013)
```

Writing Safety Specification Format Files

To write an ASCII file in the SSF format, use the `save_ssf` command . This command maps the functional safety intent in a design (possibly implemented) to the corresponding SSF commands, and writes these into the specified file.

```
save_ssf file_name
```

The `file_name` argument is mandatory for the `save_ssf` command. The value of the `file_name` argument can be a relative or absolute path to that file. The specified file location must be writable. If another file with the same name exists already, the `save_ssf` command overwrites the existing file.

Feedback

Safety Specification Format in Formality

During synthesis, the Design Compiler NXT and Fusion Compiler tools generate the guidance commands in the SVF file. The guidance commands in the SVF file specify the changes made in the implementation to specify SSF.

During preverification, the Formality tool reads the SSF file along with the SVF file and performs the following tasks:

- Independently implements the design changes on the reference design
- Reports inconsistencies between the SSF file and the guidance provided in the SVF file

You must read the SSF file after using the `set_top` command and before loading the UPF as shown in the following example:

```
set_svf   ${DESIGN}.svf
…
set_top ${DESIGN}
//Reads the SSF file
load_ssf -r  ${DESIGN}.ssf
load_upf -r ${DESIGN}.upf
```

After preverification, the Formality tool automatically generates a summary report of inconsistencies between the SSF file and the implementation of guidance commands in SVF file.

The Design Compiler NXT and Fusion Compiler tools perform the following tasks:

- Implement safety registers based on the safety targets specified in the SSF file
- Review the inconsistencies between the guidance in SVF file and SSF commands and take appropriate actions

If the guidance in SVF file is incorrect, it might be rejected. This might result in failing of compare points during verification.

To report the consistency checks after preverification, use the `report_safety_status` command. The following example shows the SSF consistency check summary report:

```
*********** SSF Consistency Checking Summary ***********
    16 FM-SSF-001 missing TMR in implementation

Please refer to the Formality log file for more details,
or execute report_safety_status

SSF file loaded:
 multiplier.ssf

************************************************************
```

4

Handling Functional Safety Registers

Describes the ways to avoid faults while handling the functional safety registers

Ion or electromagnetic radiation can cause a single event upset (SEU) in an affected logic, resulting in design faults such as a change of state in sequential elements or the output of a combination logic. If the faults propagate in the design, the faults can cause incorrect functionality in the overall design.

Transient faults in combinational logic might dissipate before the next clock cycle. But in sequential elements, they have high potential to propagate to the downstream logic, which should be of particular concern for functional safety designs.

Safety mechanisms can be added to a design to mitigate the impact of the SEU on the overall system in the following ways:

- Mitigate the SEUs successfully by allowing functionalities of a design to reach a safe state.
- Stop the SEUs from affecting circuit by allowing functionalities of a design to continue with the correct functionality.

This section includes the following topics:

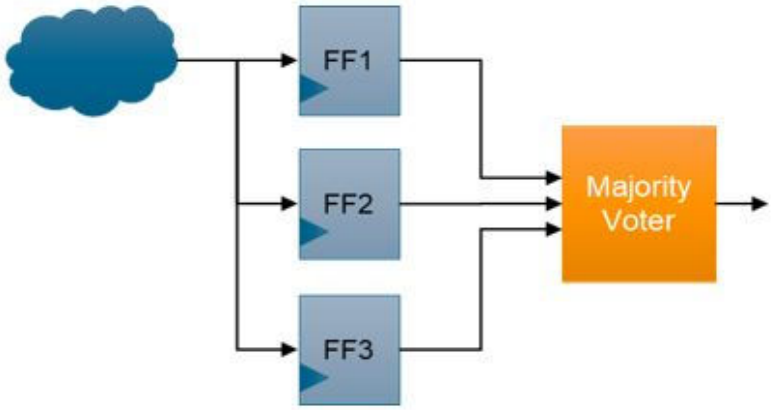
Safety Register Strategies

Synopsys tools support the following methods to mitigate SEUs on sequential elements:

- Triple-modular redundancy (TMR) register

  The most stringent designs rely on TMR register, which can correct a fault. Each identified register is replaced by three registers and its voting logic, as shown in Figure 8. If one of the registers experiences a failure, such as SEU, the functionality through the two replicated registers can prevail, negating the effects of the fault. Additional safety measures can be added to a TMR register structure, such as inserting wells on either side of the TMR register.

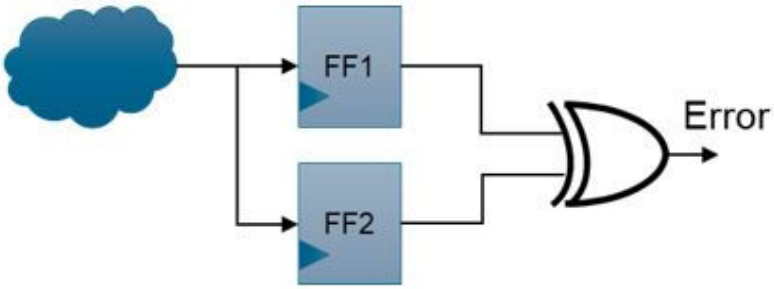
Figure 8 Three Registers and its Voting Logic



- Dual-modular redundancy (DMR) register

  Each identified register is replaced by two registers and can detect an error as shown in Figure 9, but cannot provide error correction like a TMR register. Additional logic is needed to determine what to do when an error occurs.

Figure 9 Two Registers and Detect an Error

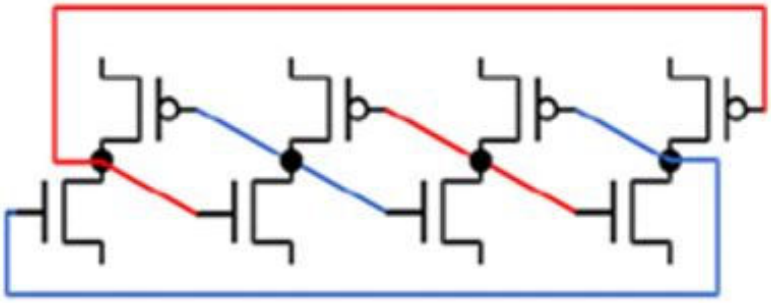


Feedback

- Fault-tolerant register

  Critical paths can resist SEUs with the use of fault-tolerant registers to reduce the chance of failure, where a fault is neither corrected nor detected, as shown in Figure 10. The fault-tolerant registers consume reduced area but are only available in special libraries. An identified register is mapped to a fault-tolerant register that is available in the library.

Figure 10 Fault-Tolerant Register



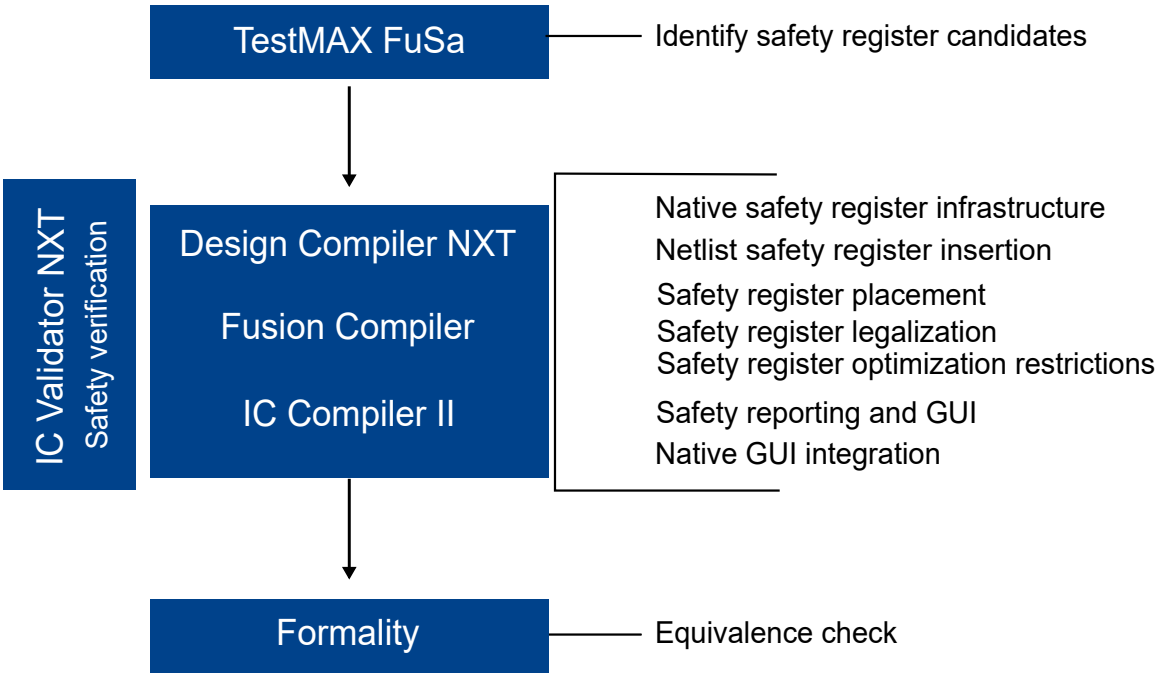
- Error protection schemes

  Using TMR or DMR or fault-tolerant registers for safety critical registers is expensive as each register is replicated in the process. Error protection schemes help in reducing the cost as the safety mechanism is applied per group of registers instead of individual register.

Feedback

Figure 11 shows the native RTL-to-GDSII flow to identify safety registers, insert safety registers, and for the Formality tool to verify that safety registers are inserted with RTL-to-gate or gate-to-gate-level during formal equivalence check.

Figure 11 Native RTL to GDSII Flow for Implementing and Verifying Safety Registers



The following features and benefits are supported in the RTL-to-GDSII flow:

- During synthesis:
  - Identify safety registers very early in the design flow to maximize Quality of Results (QoR) during placement and optimization
  - Ensure that the TMR registers or voting logics have the required optimization restrictions for optimization or mapping. For example, set registers with the `size_only` attribute, set registers without multibit, or set registers without merging, and so on
  - Transfer identified safety registers and safety requirements from synthesis to place and route
- During place and route:
  - Ensure that physical separation of safety registers is honored during placement and legalization
  - (Optional) Ensure that safety registers are isolated with tap cells and isolation is retained

- Ensure that safety register clock trees or resets are split with independent physical routes and buffering and that they are retained
- Report and view validity of functional safety requirements

Safety Register Rules and Groups

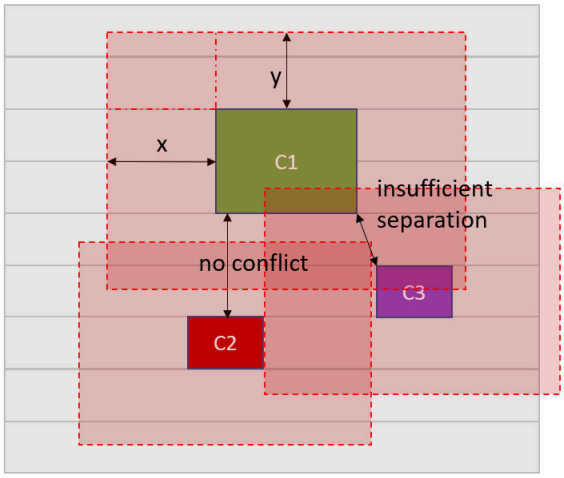
A safety register rule is an abstract grouping of constraints that are applied on safety-critical registers using the `create_safety_register_rule` command.

The safety register rules are of the following types:

- `tmr` for TMR register
- `dmr` for DMR register
- `ft` for fault-tolerant registers

The following constraints are applied to registers set with safety register rules:

- `-distance`: Specifies the distance values for separating the safety registers in one redundancy group. It can either be in the form of an {x y} pair or a single number denoting radial distance. The distances are measured from the edges of one register to the closest edges of another. For the `-type dual_mode` or `triple_mode`, this option must be specified. For `-type fault_tolerant`, this option is not required.
  - `-distance {x y}`: Specifies the *{x y}* distance pair means that either *x* or *y* separation must be honored. In other words, each pair of registers in the group should be either *x* distance apart measured on x-axis, or *y* distance apart measured on y-axis. This creates a virtual rectangular region around each register, in which another register of the same group must not be placed.

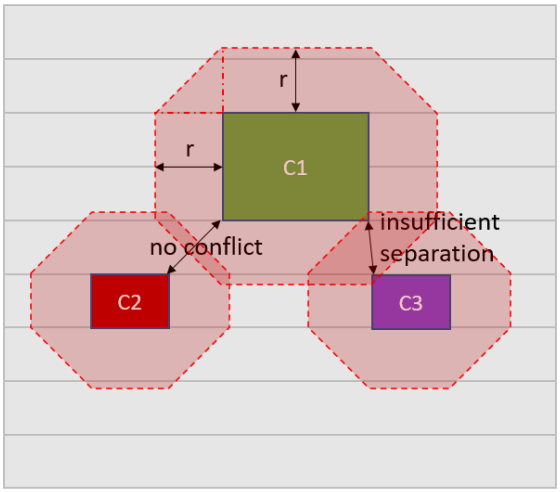


Feedback

Note:

You can specify a distance of zero for either *x* or *y*, but not for both simultaneously.

- `-distance {r}`: Specifying the Manhattan distance *r* between each pair of registers in the group. The Manhattan distance between two points is defined as the sum of the vertical and horizontal distance between their closest edges. The radial distance *r* creates an octagonal region around each register, wherein another register of the same group must not be placed.



Note:

The *r* value cannot be zero.

- `-tap_mapping`: Specifies a list of library cells that are used for instantiating tap cells for the safety registers.
- `-split_pin_types`: Specifies a list of pin types that are separately routed for replicated registers.
- `-update`: Specifies the existing safety register rule that should be updated.
- `-isolation`: Isolates safety registers with tap isolation.

The following example updates the `distance`, `logic_mapping`, `tap_mapping` and `split_pin_types` attributes of the existing safety register rule `RULE4`:

```
create_safety_register_rule -name RULE4 -update -logic_mapping
 LOGIC_LIB_CELL -distance {1 3} -split_pin_types reset -tap_mapping
 TAP_LIB_CELL
```

Use the `set_safety_register_rule` command to specify a register to be replaced by one of the following registers:

- Another fault-tolerant register
- A group of redundant registers

  The minimum separation between the redundant registers can be specified to be considered by the placement engine. If pins of the redundant registers should be split to different branches of trees, the safety register type to be considered can also be specified.

You can use the following application options with the `set_safety_register_rule` command:

- `-error_signal`: Specifies the pin or port to which the voting or compare error signal is connected.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

The following example associates a rule with the safety registers:

```
set_safety_register_rule -rule rule1 -registers {flip-flop1 flip-flop2
 flip-flop3 flip-flop4}
```

Use the `mark_safety_register` command to refer safety register groups to specific cells in a design. The safety register group consists of three registers and a few voting logic combinational cells. For example, three registers and voting logic cells form a safety register group, which can be used to implement TMR.

During placement, physical separation (repelling distance) is automatically applied to the three registers in a TMR group or to the two registers in a DMR group. Overlapping of repelling bounds is allowed for different safety register groups.

Use the `mark_safety_register` command with the following options:

- `-rule`: Specifies the safety register rule based on which the safety register group is created.
- `-name`: Specifies the name of the safety register group.
- `-registers`: Specifies the list of registers that should be part of the specified safety register group.
- `-logic`: Specifies the collection of logic cells that form a part of the voting logic circuit. These cells must be combinational cells.

- `-taps`: Specifies the list of tap cells for the safety registers in the safety register group.
- `-split_pins`: Specifies the list of pins that should be separately routed to be part of different branches of high-fanout tree.
- `-error_signal`: Specifies the pin or port to which the voting or compare error signal is connected.
- `-update`: Specifies the existing safety register group that should be updated.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.

  Note:

  `-correction signal` is valid only for type TMR. Therefore, the command issues an error message if the `-correction signal` option is specified for rules of type DMR or FT.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

The following example updates the taps, registers, logic and error signal attributes of the existing safety register group `group2`:

```
mark_safety_register -update -name group2 -taps {TAP3 TAP4} -registers
 {flop4/QN flop5/QN} -logic {Iand1} -error_signal err_blk/IN2
```

Reporting Safety Register Status and Violations

After creating and implementing safety register rules and groups, use the following commands to check and report issues:

- `report_safety_status`: Checks whether safety register rules and groups are defined in the top-level design through the complete Synopsys native functional safety flow.

  Use the command to create documentation of implemented safety measures that can be archived for ISO 26262 purposes.. The command performs all safety relevant checking, including reporting of Dual Core LockStep.

  The command provides the following options:

  - `-safety_register_rules`: Reports the specified safety register rule objects but restricts checking and reporting to the specified rules.
  - `-safety_register_groups`: Reports the specified safety register groups but restricts checking and reporting to the specified groups.

  The `report_safety_status` command reports all errors when performing consistency check on the safety register rules and groups in the current design. Example 1

illustrates the command checks for whether physical separation of registers are valid or inserted tap cells are honored according to the functional safety requirements. For more information, see Reporting DCLS Safety Status and Violations.

- `report_safety_register_rules`: Reports the specified safety register rules. If nothing is specified, reports all rules in the current design.
- `report_safety_register_groups`: Reports the specified safety register group objects. If nothing is specified, reports all groups in the current design.
- `report_bounds`: Reports the user-specified bounds constraints in the design.

Example 1 Textual Report by the report_safety_status Command

```
prompt> report_safety_status

---------- Safety status report ----------
Design : top
------------------------------------------
safety register rules: 3
safety register groups: 2
safety critical registers: 1
safety critical register with fault priority: 0
fault tolerant registers: 0
tree split buffers: 18
tap cells: 18
voting cells: 8
redundancy registers : 7

-------------- Messages ------------------
Information: Safety register related object counts for design top.
Warning:  Found safety_register_rule RULE2 without associated
 safety_register_group.
Error: Insufficient distance (0) between redundancy registers {R3_reg
 R2_reg} within safety_register_group group1, violating the minimum
 distance (100) required by safety_register_rule RULE1
Error: Insufficient distance (0) between redundancy registers {R3_reg
 R1_reg} within safety_register_group group1, violating the minimum
 distance (100) required by safety_register_rule RULE1
Error: Insufficient distance (0) between redundancy registers {R4_reg
 R5_reg} within safety_register_group group2, violating the minimum
 distance (100) required by safety_register_rule RULE1
Error: Insufficient distance (0) between redundancy registers {R4_reg
 R6_reg} within safety_register_group group2, violating the minimum
 distance (100) required by safety_register_rule RULE1
Error: The design still contains a safety critical register (REG4) that
 must be replaced by fault tolerant or redundancy logic to abide to
 safety_register_rule RULE1.
Error: The safety register groups {GRP1 GRP2} are not mutually exclusive.
 The shared cells are {TAP11 TAP10 TAP9 TAP3 TAP2 TAP1}.
Error: The safety_register_group GRP2 has 4 redundancy register(s), while
 3 are required by safety_register_rule RULE2.
```

Feedback

```
Error: Split pin REG5/RN in safety_register_group GRP2 still has to be
 buffered.
```

Enabling Support for Safety Registers

The Design Compiler NXT, Formality, and IC Validator tools do not require any application options to enable support for safety registers. Only the Fusion Compiler and IC Compiler II tools require application option settings to enable support for safety registers.

To enable support for safety registers in the Fusion Compiler and IC Compiler II tools, use the `set_app_options -name place.legalize.enable_advanced_legalizer -value true` application option.

Writing Safety Register Information

To write out a script file that can be sourced in another tool to re-create safety register rules and groups information, use the `write_safety_register_script -output ./create_safety_registers_info.tcl` command.

Supporting Voting Logic, Error Logic and Port Mapping

The voting logic, error logic, or both associated with the safety registers can be expressed as a VEL. A VEL can be either one of the following:

- (v1): an instance of a logic module
- (v2): an instance of a library cell
- (v3): connected leaf cells implementing the voting or error functionality; for simplicity, assume they belong to a virtual hierarchy

You can categorize a VEL as follows:

- Dual-modular redundancy (DMR) type
  - Without logic port map: one VEL with two inputs and one error output
  - With logic port map: one VEL, with corresponding logic port map specifying two inputs, and one error output
- Triple-modular redundancy (TMR) type
  - Without logic port map: one VEL with three inputs, and one voting output
  - With logic port map: one VEL, with corresponding logic port map specifying three inputs, one voting output, and optionally one error output
- Fault-tolerant (FT) type: logic port map is not supported

You can specify the safety register rule using the `create_safety_register_rule` command:

- With the `-logic_module` VEL: the VEL of the associated safety register group must have the `ref_module` VEL
- With the `-logic_mapping` {VEL1, …VELN}: the VEL of the associated safety register group must be an instance of the library cell VEL1, … or VELN
- Without either the `-logic_module` or the `-logic_mapping` VEL: the VEL of the associated safety register group is either of the following:
  - A single cell, optionally with logic port mapping
  - A set of leaf cells that together form a virtual hierarchical cell, never with logic port mapping

The `report_safety_status` command performs the following checks on user specified VEL:

- The VEL inputs and outputs, restricted to those specified by the optional logic port mapping, are connected
- The restricted input count matches with the type of the group: two inputs for type DMR and three inputs for type TMR
- The restricted output count matches with type of the group: one output for type DMR, one or two outputs for type TMR
- The VEL inputs are connected to identically named outputs of registers in the same safety register group
- Each of the VEL outputs have a unique load

Figure 12 shows the safety register group for safety register rule for type TMR with `-logic_module` or `-logic_mapping` VEL.

Feedback

Figure 12 Type TMR With -logic_module or -logic_mapping VEL

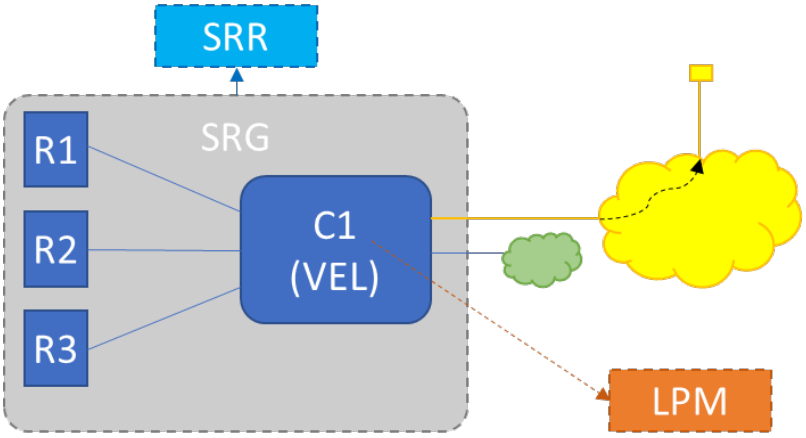
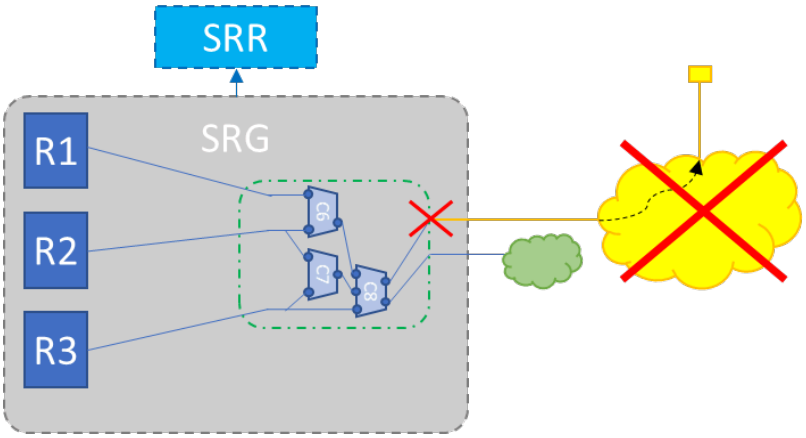


Figure 13 shows the safety register group for safety register rule for type TMR without the `-logic_module` or `-logic_mapping` VEL.

Figure 13 Type TMR Without -logic_module or -logic_mapping VEL



This section includes the following topics:

- Safety Registers With Multiple Functional Outputs
- Safety Registers With Only One Loaded Output
- Safety Registers With Loaded Complementary Outputs
- Connecting the Error Signal
- Supporting Manual Flow for Individual Leaf Cells
- Voting or Error Logic Protection

Safety Registers With Multiple Functional Outputs

A safety register rule or safety register group refers to the registers either on a cell or a pin basis. When the reference is a functional output pin of a register and the register has a complementary functional output pin (for the same bit in case of a multibit register), that output is also assumed to be included in the safety register group. When the reference is the single-bit register cell itself, all its functional outputs are assumed to be part of the safety register group.

Note:

Cell reference is not supported for multibit registers.

You cannot associate a functional output pin of a register with one safety register group and its complementary functional output (for the same bit in case of multibit register) with another safety register group.

Safety Registers With Only One Loaded Output

If only one of the two outputs of the safety registers are loaded, which must be consistent for each register in the group, then the loaded output is treated as the only output for the group. Therefore, only one VEL is expected.

Note:

While replacing the original register by a type DMR or TMR, ensure the same register output connectivity is maintained. For example, if the Q pin of a register is loaded, and the QN pin is not loaded, then the safety registers in the replacing type DMR should also only have their Q pins loaded, and QN pins unloaded. The `report_safety_status` command cannot check whether the situations before and after the replacement are consistent.

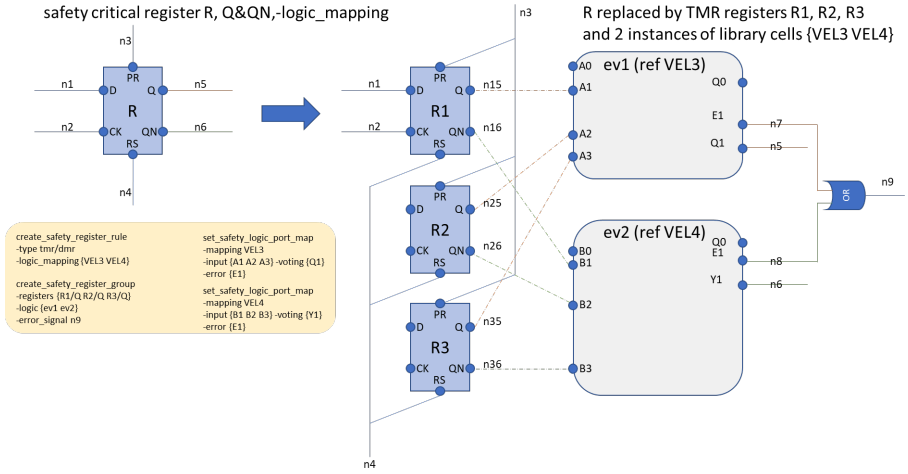
Safety Registers With Loaded Complementary Outputs

In a safety register group, if the safety registers have two loaded functional output pins (for the same bit, in case of multibit), each of the matching outputs must have unique corresponding logic. A voting logic should neither include a mixture of regular and complementary outputs, nor should it be connected to the two outputs of the same register. Figure 14 shows an example of a safety register group with loaded complementary outputs.

The outputs of the two VELs of the safety register group must not share the same net.

Feedback

Figure 14 Safety Register Group (Type TMR) With Two Loaded Outputs



Connecting the Error Signal

When you define the safety register group with the specified `-error_signal` pin or port, the VEL of the safety register group should have a known error output.

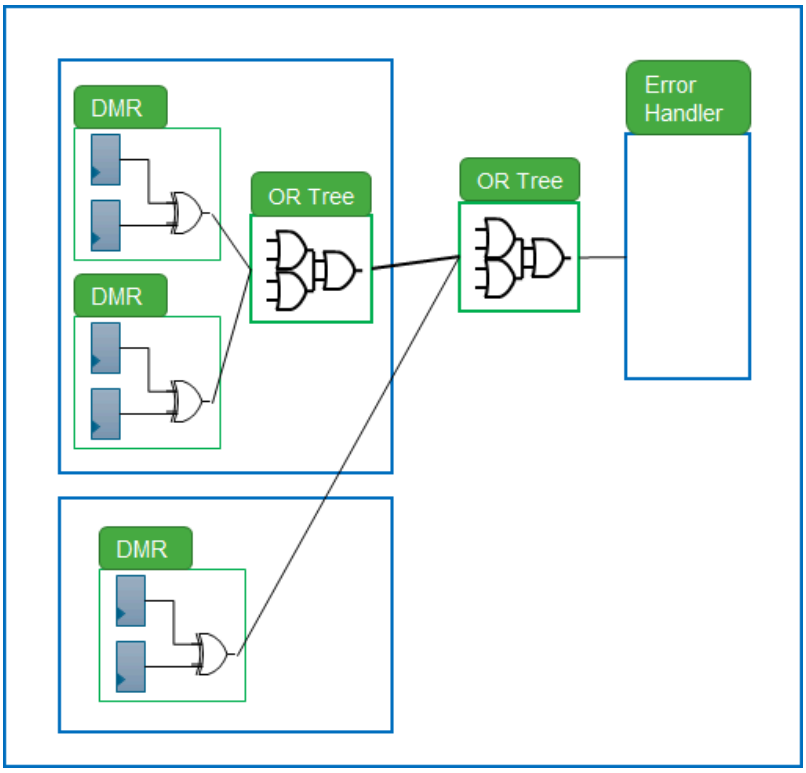
- In case of a type DMR, the error output is the single output of the VEL.
- In case of a type TMR, the associated logic port map must specify the `-error` option.

The name identified with the `-error_signal` pin or port can be either a top-level port in the design or a pin of another logical hierarchy in the design (for example a dedicated error handler module). When specified, the tools

- combine all the error signals associated with a specific safety register rule into a single signal by creating an OR-tree structure
  - The OR-trees are created at the lowest level of hierarchy to minimize port punching.
- create ports on any intervening levels of hierarchy to propagate the signal to specified target port or pin
- connect the fully combined signal to the specified port or pin

Feedback

Figure 15 OR-Tree Structure



The `-error_signal` should be connected to the error pin or port defined by the `set_safety_register_rule` command. For example, the input of the error handler as shown in the Figure 15.

Supporting Manual Flow for Individual Leaf Cells

For backward compatibility and supporting the manual flow, you must not specify the safety register rule with either the `-logic_module` or the `-logic_mapping` option. You can distinguish between the following two scenarios:

1. The `-logic` argument of the safety register group takes a hierarchical cell, optionally with an associated logic port map. The `-logic_module` option has the same requirement, but there is no restriction on the `ref_module` instance of the VEL of the safety register group.
2. The `-logic` argument of the safety register group takes a list of individual leaf cells comprising one or two virtual hierarchies (each representing a VEL). Because there is no actual hierarchical cell, there is no associated logic port map describing the inputs and outputs of the VEL. Therefore, you cannot distinguish between the output of the

VEL from its error output. For this reason, the virtual hierarchical is allowed to have only one output. Its input count should match the input of the safety register group (two inputs for type DMR and three inputs for type TMR).

The `report_safety_status` command analyzes the leaf cells of the safety register group by associating them to the uniquely named register outputs driving them. If cells are driven by distinctly named register outputs, the virtual hierarchies cannot be established, and the `report_safety_status` command reports this as an error. The pins of the virtual hierarchy are formed by picking representative pins on cells in the virtual hierarchy that connect to cells outside of the virtual hierarchy. The `report_safety_status` command checks requirements on the virtual hierarchy as on an actual hierarchical cell. All inputs and outputs of the leaf cells must be connected.

Voting or Error Logic Protection

The VEL, when part of a safety register group, must be protected from being modified during optimization as follows:

- If the VEL is a library cell, it is marked as `size_only` with reason safety.
- If the VEL is a hierarchical cell, its leaf cells are marked as `size_only` with reason safety.

The hierarchical cell itself is protected by the `freeze_ports` with reason safety to avoid modification during optimization. The required protection is added on association of the cell with the safety register group while using the `create_safety_register_group` command.

Safety Register Flow

The Synopsys tools support the following flows for handling safety registers:

- Manual Safety Register Flow: The safety measures are manually added to the design (either at netlist or RTL level) and should be marked for special handling by the synthesis tools, as shown in Figure 16.
- Automatic Safety Register Replacement Flows: The safety measures are added to the design automatically by the synthesis tools, as shown in Figure 17.

In the safety register flow,

- Any number of registers can be specified by using the `set_safety_register_rule` command.
- During synthesis, each SEQGEN with an applied safety rule is replaced with three registers and voting logic cells, if the mode is TMR (`triple_mode`). If the modes are

Feedback

`dual_mode` and `fault_tolerant`, SEQGEN with an applied safety rule is replaced with the relevant strategies. For more information, see Safety Register Strategies.

- A safety register group is created with set of three registers and voting logic cells for TMR. For DMR, a safety register group is created for set of two registers and compare logic cells.

  The Design Compiler NXT tool automatically handles TMR and DMR registers with the following group constraints:

  - Optimization
  - Register separation
  - Safety register groups are passed to the IC Compiler II tool to implement the remaining safety register rules

  The Fusion Compiler tool automatically handles TMR and DMR registers with the following group constraints:

  - Optimization
  - Register separation
  - Tap cells insertion
  - Wire splitting

This section includes the following:

- Manual Safety Register Flow
- Automatic Safety Register Replacement Flows
- Safety Register Synthesis Using the SPFM Target
- Using Multibit Registers

Manual Safety Register Flow

To create the following safety register structure in the manual safety register flow:

- Create safety register rules by using the `create_safety_register_rule` command. The `-mapping` option is not supported in the manual flow.
- Create safety register groups for existing cells in the netlist by using the `mark_safety_register` command. A safety register group contains a set of three registers plus a voting logic cell to implement TMR registers. The safety rule is provided to the safety register group to define how the tool should handle the safety

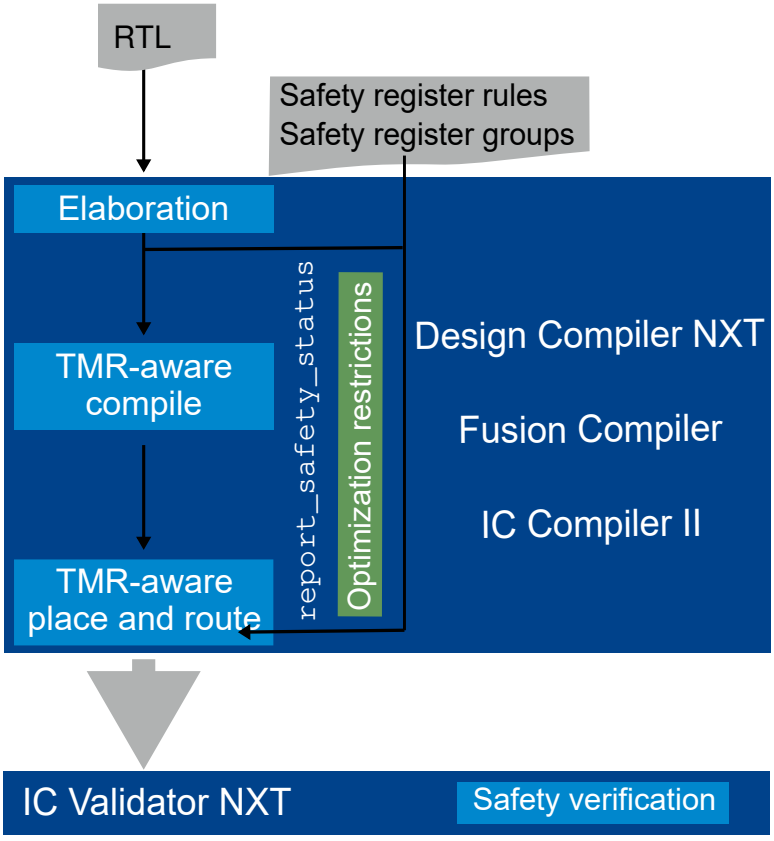
Feedback

register structure. The tool automatically handles the TMR registers with the following group constraints:

- Optimization restrictions, without removing redundant registers, multibit registers, and voting logic cells that are maintained in the design
- Register separation
- Signal separation
- Tap cell insertion

Figure 16 shows the manual safety register flow, which is from the existing RTL with RTL inferred TMRs or gate instantiated TMRs. The flow also works when you use the netlist with TMRs in the IC Compiler II and Fusion Compiler tools.

Figure 16 Manual RTL to GDSII Flow Starting Either From Netlist or RTL



Example 2 Implementing TMR Registers After Compile in Design Compiler NXT

```
analyze -f sverilog test.sv / elaborate test
source test.sdc
compile_ultra
```

Feedback

```
source user_manual_tmr_creation.tcl

create_safety_register_rule -name rule1 -type triple_mode -distance 10 \

mark_safety_register -name SRG1 -registers [get_cells -hier dout_1_reg*]
 \
-logic U30 -rule rule1

mark_safety_register -name SRG2 -registers [get_cells -hier dout_2_reg*]
 \
-logic U31 -rule rule1

compile_ultra -incremental
```

Example 3 Implementing TMR Registers After Mapping in Fusion Compiler

```
analyze -f sverilog test.sv / elaborate test
set_top_module test
source test.sdc
compile_fusion -to initial_map
source user_manual_tmr_creation.tcl

create_safety_register_rule -name rule1 -type triple_mode -distance 10

mark_safety_register -name SRG1 -registers [get_cells -hier dout_1_reg*]
 \
-logic U30 -rule rule1

mark_safety_register -name SRG2 -registers [get_cells -hier dout_2_reg*]
 \
-logic U31 -rule rule1

compile_fusion -from logic_opto
```

Automatic Safety Register Replacement Flows

The Design Compiler NXT and Fusion Compiler tools create the safety register structures during synthesis in the automatic safety register replacement flow, as follows:

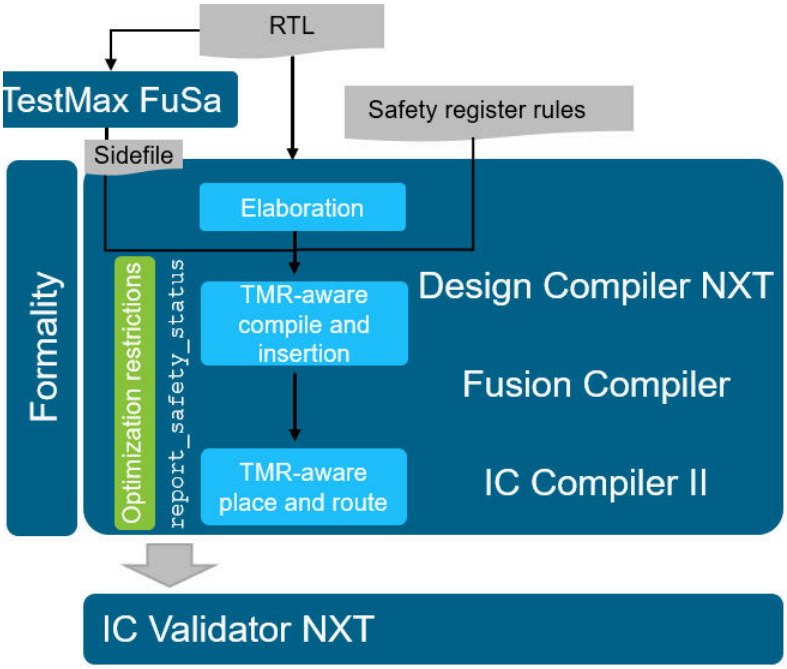
- (Optional) Imports the `spfm_loss` attribute settings from the TestMax FuSa tool by sourcing the `set_attribute` command output to the Design Compiler NXT and Fusion Compiler tools.
- Creates safety register rules with the `create_safety_register_rule` command.
- Applies safety register rules to the targeted sequential elements with the `set_safety_register_rule` command.

Feedback

- Inserts required safety register structures with the `set_safety_register_rule` command, during synthesis with the `compile_ultra` or `compile_ultra -incremental` command in the Design Compiler NXT tool or with the `compile_fusion` command in the Fusion Compiler tool.

Figure 17 shows the automated safety register flow when RTL does not have TMRs, and TMRs should be inserted during the compile stage.

Figure 17 Automated RTL to GDSII TMR Register Insertion Flow



Handling Error Signals

In the Design Compiler NXT and Fusion Compiler tools, the TMR and DMR register structures can produce error signals, when an SEU occurs on the chip. The error signals can be used to alert an external error monitor. You can locate the error monitor in the chip or off the chip by using the tool.

Pins or ports from the design can be set as the destination of error signals from all registers that are associated with a specified register rule. To specify such a pin or port, use the `-error_signal` option with the `set_safety_register_rule` command. All the individual error signals create an OR-tree and get connected to the specified pin or port. If the safety register rule applies to multiple hierarchies, the OR-tree for each module is created within the respective module to minimize port punching.

The following command replaces all the registers in the design according to the specified `DMR_rule1` safety register rule, and connects the output of the OR-tree to the

`control_unit/err_handler/DNR_err_sig` pin. Port punching occurs in the design, as required:

```
set_safety_register_rule -rule DMR_rule1 -registers [all_registers]
 -error_signal [get_pin control_unit/err_handler/DNR_err_sig]
```

The following command connects the output of the error OR-tree to the top-level `ERR_top` port :

```
set_safety_register_rule -rule TMR_rule1 -registers [all_registers]
 -error_signal [get_port ERR_top]
```

Inserting Safety Registers With Pre-Mapped Voting Logic Module

In the automatic safety register insertion flow, you might want to insert user-specified voting logic modules instead of using the automatically generated AOI structure. To insert the user-specified voting logic, use the `-logic_module` option with the `create_safety_register_rule` command, as shown in Figure 18.

You can insert safety registers by using RTL as a starting point with the pre mapped voting logic module. Use the `create_safety_register_rule -target` command when a specific structure or additional functionality is needed in a voting logic for TMR registers. The command allows you to use the pre mapped logic module as the voting logic. The pre mapped logic module must have three input ports and one output port.

To specify the voting logic module, use the `-logic_module` option as shown in the following example:

```
create_safety_register_rule -name myRule -type triple_mode -logic_module
 myVote
set_safety_register_rule -rule myRule -registers [get_cells *_reg]
```

Feedback

The following figure shows where the specified *myVote* module with the `-logic_module` option is used in the flow:

Figure 18 myVote Module With -logic_module Option

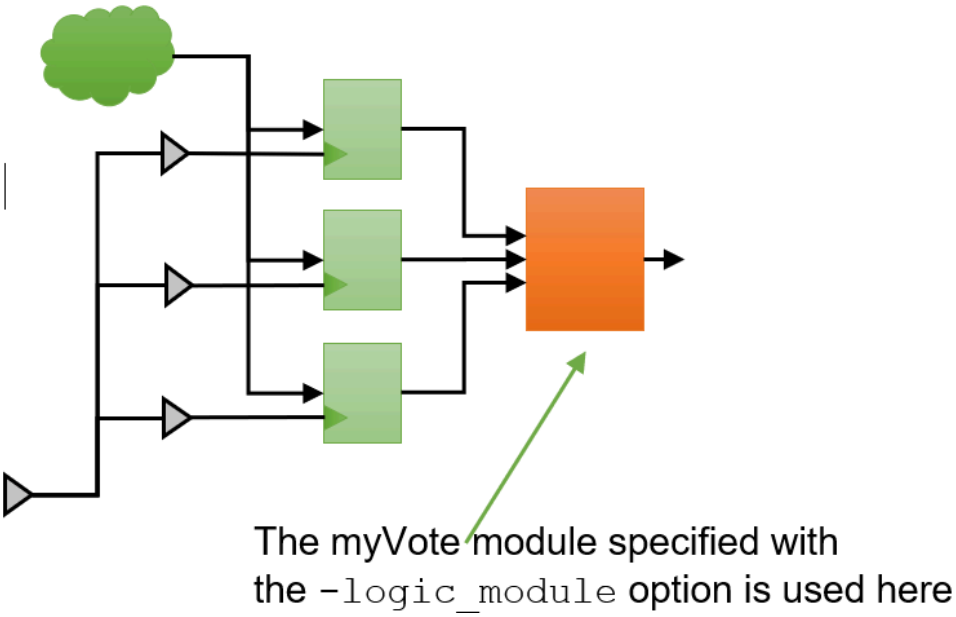
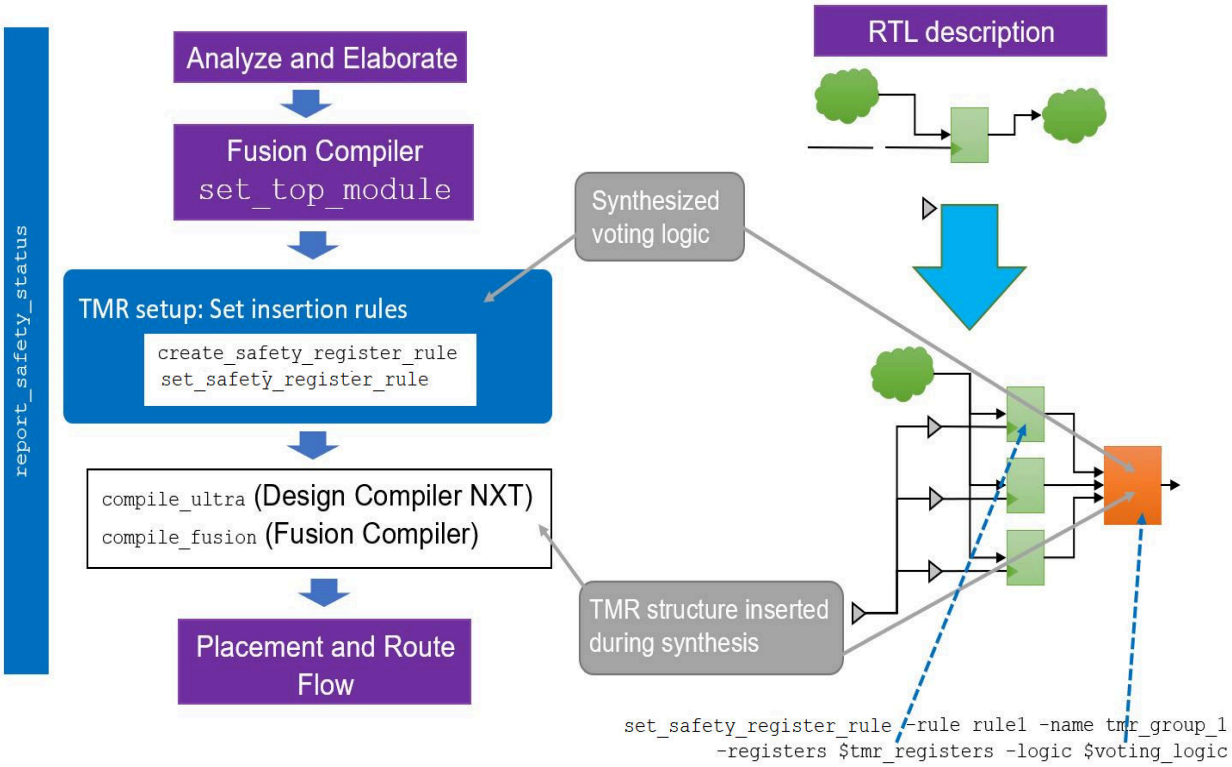


Figure 18 shows how the specified pre compiled logic module is inserted as the voting logic module for any TMR registers implementing the safety register rule. The safety

register rule is the one associated with the safety register group to which the TMR registers belong.

You can insert safety registers with the pre mapped voting logic as shown in the following example:

Figure 19 Inserting Safety Registers With the Pre-Mapped Voting Logic



Example 4 Inserting Safety Registers With the Pre-Mapped Voting Logic Module in Design Compiler NXT

```
analyze -f sverilog test.sv / elaborate test
source test.sdc

create_safety_register_rule -name rule1 -type triple_mode -distance 10

set_safety_register_rule -registers [get_cells -hier data1*_reg] -rule
 rule1

set_safety_register_rule -registers [get_cells -hier *curState*] -rule
 rule1

compile_ultra
```

Feedback

Example 5 Inserting Safety Registers With the Pre-mapped Voting Logic Module in Fusion Compiler

```
analyze -f sverilog test.sv / elaborate test
set_top_module test
source test.sdc

create_safety_register_rule -name rule1 -type triple_mode -distance 10

set_safety_register_rule -registers [get_cells -hier data1*_reg] -rule
 rule1

set_safety_register_rule -registers [get_cells -hier *curState*] -rule
 rule1

compile_fusion
```

Note:

- In the Design Compiler NXT tool, the voting logic module must be visible in the memory before synthesis. The voting logic module can be read from a Verilog netlist or from a pre compiled .ddc file, as shown in the following example:

  ```
  read_verilog myVote.gv
  ```

- In the Fusion Compiler tool, the pre mapped logic module to be synthesized must be visible to the current block. To make the pre mapped logic module visible, you must copy the module from a reference library to the current block, as shown in the following example:

  ```
  set_ref_libs -add voting/voting.nlib
  set currentBlock [current_block]
  open_block voting.nlib:voting1
  copy_module -design $currentBlock -from myVote -to myVote
  current_block $currentBlock
  ```

To specify a list of combinational library cells in the Fusion Compiler and Design Compiler NXT tools, use the `-logic_mapping` option with the `create_safety_register_rule` command. To synthesize the safety logic (for example, voting logic circuit) in the TMR or DMR register groups, the tool selects the required library cell from the list of combinational cells specified with the `-logic_mapping` option.

The logic block that follows either the DMR or TMR registers to identify error pins form other pins, or the dual core locked step (DCLS) blocks contains multiple output ports. To mark different ports of the specified voting logic module or library cells as

Feedback

input, output, or error ports, use the `set_safety_logic_port_map` command. The `set_safety_logic_port_map` command supports the following options:

- `-module`: Specifies one or more voting logic modules for which port mapping is applied.
- `-mapping`: Specifies one or more library cells for which port mapping is applied.
- `-input`: Specifies a list of input ports.
- `-voting`: Specifies the output of the voting logic.
- `-error`: Specifies an error port. This is optional because some logic might not have an error port.
- `-correction`: Specifies the name of an output pin of the module, which is the correction signal
- `-output`: Specifies a list of output pin names, which are used to mark the data output. For example, for decoder modules of safety error code mechanisms. The command issues an error message, if an input pin is provided. This option cannot be specified if the type is encoder.
- `-type`: Specifies the type of module or cell that is voting type, encoder type or decoder type.
- `-checkbits`: Specifies the output checkbits for encoder type, and the input checkbits for decoder type.

To list the port maps defined with the `set_safety_logic_port_map` command in the current block, use the `report_safety_logic_port_map` command.

Naming Automatically Inserted Safety Registers

The Synopsys tools handle naming of automatically inserted safety registers for a TMR register in the following ways:

- For Fusion Compiler,
  - The original register is renamed as `original_reg_name_tmr1`.
  - Replicated registers are named as `original_reg_name_tmr2`, `original_reg_name_tmr3`, and so on.
- For Design Compiler NXT,
  - The original register retains the same name.
  - Voting logic cells are placed in a hierarchy named `tmr_voting_logic_hier_<#>/*`.
  - Replicated registers are named as `original_reg_name_tmr1`, `original_reg_name_tmr2`, and so on.

The Synopsys tools handle naming of automatically inserted safety registers for DMR register in the following ways:

- For Fusion Compiler,
  - Original register is renamed as `original_reg_name_dmr1`.
  - Replicated registers are named as `original_reg_name_dmr2`.
- For Design Compiler NXT,
  - Original register retains the same name.
  - Error detection logic cells are placed in a hierarchy named `dmr_voting_logic_hier_<#>/*`. Error detection signals are left unconnected for you to connect.
  - Replicated registers are named as `original_reg_name_dmr1`.

Fault-Tolerant Register Insertion

You can swap regular registers with registers that have a failure in time (FIT) attribute defined as a lower area alternative for the TMR or DMR register structure insertion. Currently, FIT rate attributes are only supported with fault-tolerant registers.

Using fault-tolerant registers have the following benefits:

- Lower area usage per register when compared to insertion of TMR or DMR registers
- Reduced placement disruption and constraints

You might observe that the fault-tolerant register:

- Improves the tolerance against an SEU (as can also be done with TMRs). To achieve a specific targeted SPFM goal, the tool might need to replace more registers with fault-tolerant ones as compared to the number of TMR structures.
- Shows less resistance to common mode failures, such as an SEU on the clock line.

Identifying Fault-Tolerant Registers

Consult your library vendor to identify if fault-tolerant registers are available in the logic library.

Feedback

Specifying Rule for Fault-Tolerant Register Replacement

To create a rule for fault-tolerant register replacement, use the `create_safety_register_rule` command with the following options:

- `-type fault_tolerant`: Specifies the rule for fault-tolerant register replacement.
- `-register_mapping`: Specifies the technology library cells to be used for fault-tolerant register replacement. You must use the`-type` option with this option.

Note:

- The `-distance` and `-split_pin_types` options are not supported for fault-tolerant register replacement.

Identifying Registers for Replacement

To set registers with the rule created by the `create_safety_register_rule` command, use the `set_safety_register_rule` command.

Note:

The `-error_signal` option is not supported for the fault-tolerant register replacement.

Safety Register Synthesis Using the SPFM Target

To automatically add safety registers to a design, use the synthesis flow with a SPFM target.

You can specify a SPFM target with the `set_spfm_target` command. This is useful if a design is targeted at multiple ASIL targets or if a list of registers is not available

When you specify an SPFM target with the `set_spfm_target` command, the Design Compiler NXT and Fusion Compiler tools replace the registers until they meet the specified SPFM target based on the rule specified using the `safety_register_rule` command. The automatic safety register replacement flow and the safety register synthesis flow using the SPFM target can be combined in the design.

The SPFM value is specified in percentage. Use the `set_current_spfm` command to specify the current value of the SPFM of the design with the current safety measures inserted in the design.

```
set_current_spfm <float> [-cells <hier_inst>]
```

To specify the current SPFM value for the design hierarchy at the specified instance, use the `-cells` option with the `set_current_spfm` command.

To specify the SPFM loss for a register or a group of registers, use the `set_spfm_loss` command if the registers are not converted to TMR or DMR registers. After SPFM

analysis, the Design Compiler NXT tool selects the appropriate registers to convert the registers to TMR or DMR registers based on the SPFM loss values.

```
set_spfm_loss <float> -registers <register_names>
```

The tool issues a warning message if the current SPFM value is between 0 and 1 to avoid confusion between decimal or fractional settings and percentages. You can set or import the `set_spfm_loss` command using a supplemental SSF file from a fault analysis tool, such as the TestMAX Fusa.

Determining the SPFM Target

The ISO 26262 standard specifies the SPFM target for a design depending on the ASIL target (A,B,C and D), as shown in Table 1. For example, the `set_spfm_target 99` SPFM target is specified for an ASIL-D rated design.

Specifying Constraints for Synthesis

To set up a design for SPFM targeted synthesis,

1. Import the `spfm_loss` attribute settings from the TestMax FuSa tool by sourcing the `set_attribute` command output to the Design Compiler NXT and Fusion Compiler tools.
2. Use the `set_spfm_target` command with the appropriate ASIL target.
3. Create safety register rules using the `create_safety_register_rule` command with the type of register structures and other constraints (same as other flows).
4. Specify the rule to be used during synthesis with the `-target` option of the `set_safety_register_rule` command. For example,
   - The `set_safety_register_rule -target -rule TMR_rule1` applies the rule to the entire design.
   - The `set_safety_register_rule -target -rule TMR_rule1 -registers $myRegs` command applies the rule only to the specific registers as candidates for replacement.

     Also, the tool decides whether all or a few of the registers should be replaced to meet the SPFM target.
5. To replace the registers, use the `compile_ultra` command in Design Compiler NXT or the `compile_fusion` command in Fusion Compiler.
6. Verify the results in the TestMAX FuSa tool at the gate-level netlist.

Feedback

Prerequisites to Synthesize the Safety Registers

The TestMAX FuSa tool generates the `spfm_loss` settings in a standalone file. This file is then added to the RTL or gate netlist using one of the following flows:

- Elaborated Netlist Flow
- Gate-Level Netlist Flow

You can choose the required flow based on the following parameters:

- Overall tool flow
- The degree of difficulty in achieving the SPFM target
- The number of registers expected to be replaced
- Criticality of the impacted areas in the design

Elaborated Netlist Flow

The TestMAX FuSa tool analyzes the `spfm_loss` attribute settings at the RT level. The Design Compiler NXT and the Fusion Compiler tools annotate the settings to the elaborated netlist. The elaborated netlist flow,

- adds optimization restrictions, area impact, and placement constraints at the initial stage of the flow
- provides better QoR
- does not require additional optimization steps to exit and restart the tool

The elaborated netlist flow might not provide the correct correlation with the final gate-level netlist SPFM result. You might have to run the tool or use the flow multiple times and reapply the SPFM target to meet the final target.

Gate-Level Netlist Flow

The TestMAX FuSa tool does not analyze the `spfm_loss` attribute settings at the RTL, but only during early gate-level netlist.

The gate-level netlist flow provides better correlation with the final SPFM result in a single run. However, the gate-level netlist flow uses more registers for replacement and can disrupt the overall placement congestion or timing results. Also, it might require the synthesis tool to rerun the TestMAX FuSa tool after the initial optimization stage.

Feedback

Using Multibit Registers

To use multibit registers in the safety register flow, you must set the value of the `spfm_loss` attribute only on the output pins using the `set_attribute` command. The default is `0.0` percent. The value must be between `0.0` and `100.0` percent.

Note:

Make sure not to set the `spfm_loss` attribute on multibit registers.

By default, the tool assigns a value to all the output pins, excluding the direct scan-out pin if the `spfm_loss` attribute is not set on any output pin.

To remove any attributes that are set on the output pins with the `spfm_loss` attribute, use the `remove_attribute` command.

Creating Safety Register Groups for Multibit Registers

To create safety register groups for individual bits of a multibit register, use the `-registers` option with the `mark_safety_register` command to specify individual cells or output pins of a multibit or a single-bit register:

```
mark_safety_register [-registers <cells or output_pins>]
```

To display all the registers associated with the safety register group irrespective of whether the group is created using registers or pins, use the `grouped_registers` attribute with the `get_attribute` command:

```
get_attribute [get_safety_register_group grp1] grouped_registers
```

The `grouped_registers` attribute contains `{r1 r2 r3}` and the `grouped_register_pins` attribute is not populated as shown in the following example:

Example 6 Using grouped_registers Attribute With the get_attribute Command

```
fc_shell> get_attribute [get_safety_register_group grp1]
 grouped_registers
{r1 r2 r3}
```

To list the collection of output pins in a safety register group, use the `grouped_register_pins` attribute of the `get_attribute` command as follows:

```
get_attribute [get_safety_register_group grp1] grouped_register_pins
```

The `grouped_registers` attribute contains `{r1 r2 r3}` and `grouped_register_pins` attribute contains `{r1/Q r2/Q r3/Q}` as shown in the following example:

Example 7 Using grouped_register_pins Attribute With the get_attribute Command

```
fc_shell> get_attribute [get_safety_register_group grp1]
 grouped_register_pins
{r1/Q r2/Q r3/Q}
```

Note:

The `grouped_register_pins` attribute gets populated only when you have specified at least one pin while creating the group either manually or automatically by the tool.

You can create a group with register outputs using the `-registers` option. To list the output pins of a register for the specified group, use the `get_pins -of_objects` command.

If you have specified pins in addition to the cells, use the `get_safety_register_groups -of_objects` command to display the groups containing these pins.

Example 8 Using the get_safety_register_groups -of_objects Command

```
fc_shell> get_safety_register_groups -of_objects r1/Q
{grp1}
```

Note:

If you delete a pin, which is part of a safety register group, the pin automatically gets deleted from the group to ensure database validity.

Applying Safety Register Rules on Multibit Registers

To associate a rule with the output pins of registers instead of the registers themselves, use the `-registers` option of the `set_safety_register_rule` command to specify individual output pins of a multibit register:

```
set_safety_register_rule [-registers <cells or output_pin_list>]
```

Note:

The `set_safety_register_rule` command can have a mix of registers and output pins. The pins must be the output pins of multibit registers.

You can use the `-of_objects` option of the `get_pins` command to list the collection of register output pins, if the `register_outputs` command has a rule associated with the register outputs.

Feedback

Verifying Triple-Modular Redundancy With Formality

The Formality tool verifies the designs containing TMR registers in the following scenarios:

- TMR Registers Already Exist in RTL
- TMR Registers Inferred From RTL Registers
- TMR Registers Inferred by the Synopsys Implementation Tools
- TMR Registers Inserted With Third-Party Tools

TMR Registers Already Exist in RTL

The Synopsys tools use the `create_safety_register_rule` and `mark_safety_register` commands to understand the existing TMR registers during synthesis and place and route. Because the TMR registers already exist in both the RTL and the netlist, there is no additional setup required in the Formality tool.

TMR Registers Inferred From RTL Registers

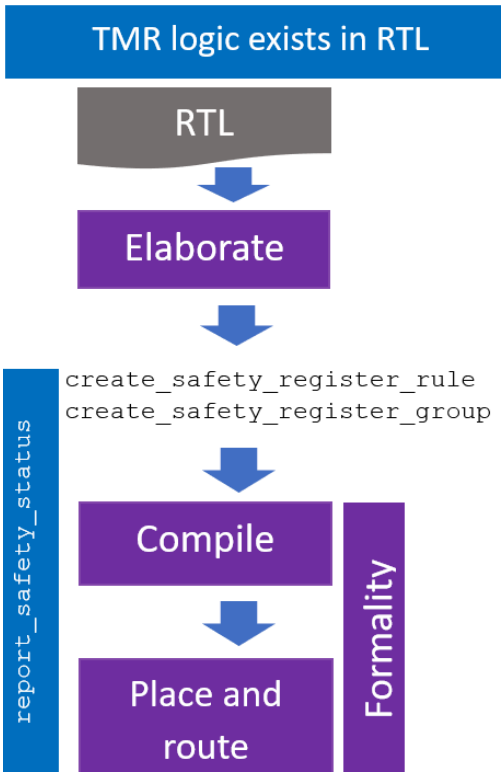
The Synopsys tools use the `create_safety_register_rule` and `set_safety_register_rule` commands to specify the `triple_mode` type for the RTL registers.

During synthesis, the `guide_safety_reg_group` commands are included in the SVF file. In the SVF file, the `guide` commands indicate which RTL registers are replicated along with the names of the new registers.

During pre verification, when the Formality tool processes the SVF file, the `guide_safety_reg_group` commands are processed. During verification, the Formality tool creates replicated registers and majority voting logic in the reference design and verifies the implemented netlist, as shown in Figure 20.

Feedback

Figure 20 TMR Registers Inferred From RTL Registers During Synthesis



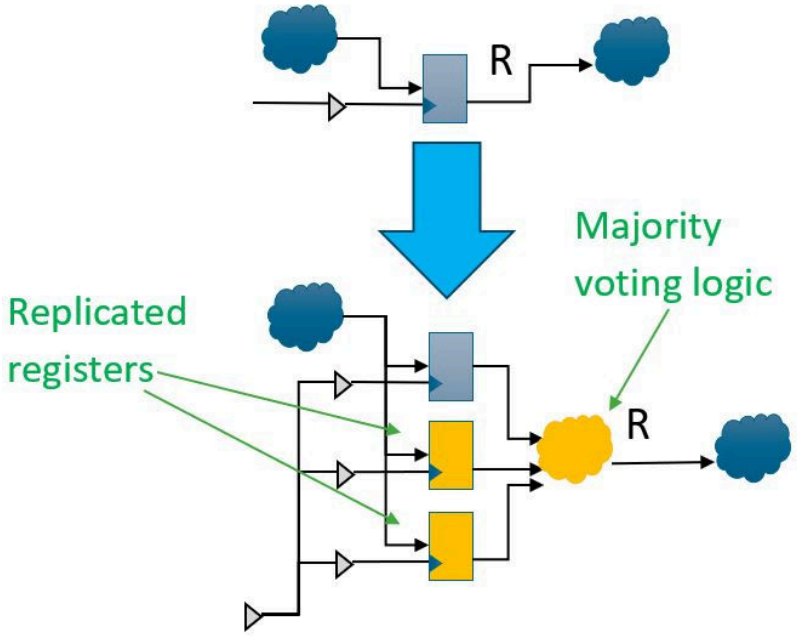
TMR Registers Inferred by the Synopsys Implementation Tools

The Synopsys implementation tools inserts the `guide_safety_reg_group` commands into the SVF file created during implementation. The Formality tool processes the SVF file during the pre verification step, and makes changes in the reference design by replicating registers and inserting voting logic, as shown in the following example:

```
guide_safety_reg_group -design {design_name} -from {R_reg} -to {R_reg
 R_tmrA R_tmrB}
```

Feedback

The replicated register names are specified using the SVF file, and the Formality tool creates the majority voting logic, as shown in the following figure:



After pre verification is completed, the TMR registers and voting logic exist in both the RTL and the netlist and then the Formality tool verifies the designs normally.

TMR Registers Inserted With Third-Party Tools

If synthesis is performed by third-party tools or TMR registers exist in a netlist but the associated SVF file is not available, the Formality tool inserts the TMR register in the RTL without the SVF file with the `replicate_safety_register` command. Use the command to specify a list of registers. You can also use the `guide_safety_reg_group` commands, but information from the SVF file is applied only to single registers.

The following example shows how to use the `replicate_safety_register` command when there are multiple registers that need TMR. Every register named state_reg[*] in the cpu design is applied with the TMR register. Make sure that the `-to` option specifies three TMR register names to create the TMR registers.

```
set regs [get_cells -quiet {r:/WORK/cpu/state_reg[*]}]
replicate_safety_register ${regs} -from {\(.*\)_reg\(.*\)} -to {{\1_A\2}
 {\1_B\2} {\1_C\2}}
```

Where, the `-from` and `-to` fields use unix sed type regular expression syntax for name substitution from the original register name to the new TMR register names. In this case, assuming the names of the original RTL registers are `state_reg[0]` and `state_reg[1]`,

the name of the new TMR registers created by the command might be `state_A[0]`, `state_B[0]`, `state_C[0]`, `state_A[1]`, `state_B[1]`, and `state_C[1]`.

5

Error Protection Schemes

To reduce area and performance costs, use error detection and correction techniques for a group of registers. The error protection schemes include the following:

- Error protection rules
- Implemented error protection schemes

Error Protection Rules

Error protection rules describe the requirements for insertion and handling of error protection logic on groups for registers. You can use the `set_safety_error_code_rule` command to apply error protection rules to the registers with no existing error protection logic. You can mark registers that have existing error protection logic using the `mark_safety_error_code` command.

This section includes the following:

Creating Error Protection Rules

You can create a new safety error code rule using the `create_safety_error_code_rule` command. This command supports the following options:

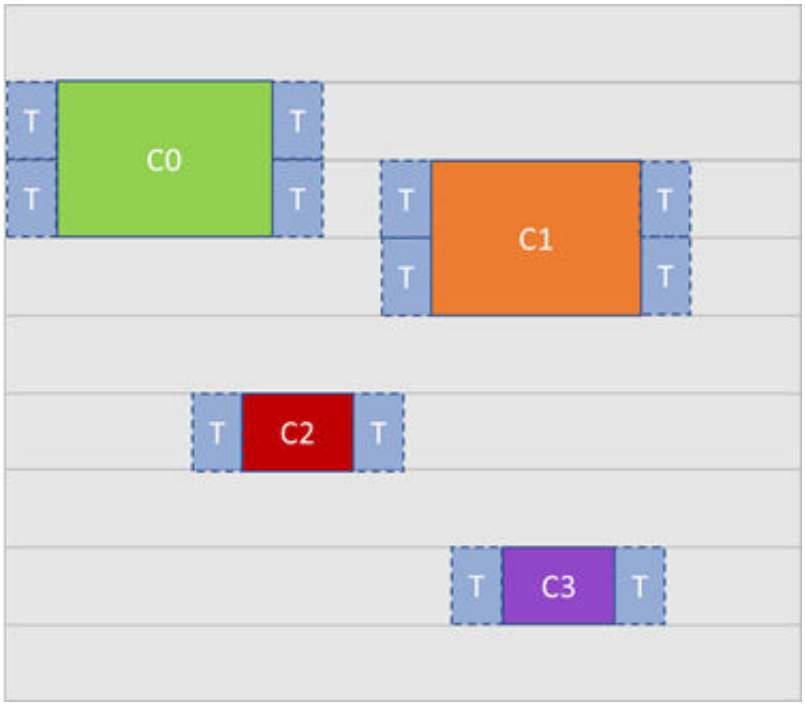
- `-name`: Specifies the name of the rule to be created. This is optional. If not specified, a tool-generated name is assigned.
- `-update`: Updates any existing rule. This is optional. The `-name` option must be specified along with the `-update` option. You can update a rule only if it is not applied to any data using the `set_safety_error_code_rule` command or there is no group referring to it.

- `-type`: Specifies the type of encoding to be performed. The encoding options are as follows:
  - Even or odd parity
  - Error correction code
  - Error detection code
- `-slice_size`: Specifies the number of bits of data to be encoded as a group. For example, if the data width is 32 and the `slice_size` is specified as 8, then for every 8 bits, one group is created. This option is optional and when not specified, all data bits are encoded as a single group.
- `-sequential`: Specifies the sequential encoding. This is a mandatory option, as only sequential encoding is supported.
- `-isolation`: Ensures that the data bit and check bit registers have adjacently placed well-taps. This option can be used only with the `-sequential` option.
- `-tap_mapping`: Specifies a list of reference modules for the well-taps to refer. This option can be used only with `-sequential` option.

During placement, space is reserved next to registers of safety mechanisms that require isolation.

Feedback

Figure 21 Well-tap (T) planned adjacent to registers (C0, C1, C2, C3) of a safety error code group




```
create_safety_error_code_rule -name RULE -type edc -mode encode_decode
 -sequential
```

Setting Error Protection Rules

You can use the `set_safety_error_code_rule` command to apply a safety error code rule to a set of registers with no existing error protection scheme if

- No other rule is already applied for given data bits
- The given data bits are not a part of any existing safety error code group
- The registers are not part of any safety mechanisms such as TMR, DMR, fault tolerant, and failsafe finite state machine

Feedback

You need to execute this command for each set of data bits on which the safety mechanism needs to be applied. The `set_safety_error_code_rule` command supports the following options:

- `-rule`: Specifies the safety error code rule to be applied. This rule must be valid or existing rule object or rule name.
- `-data`: Specifies the data signal which needs to be encoded or decoded. This list is an ordered set of pins or ports. In case of sequential encoding, these data signals are output pins of the registers.
- `-checkbits`: Specifies an ordered list of pins or ports, where the checkbits from the encoder output are connected.
- `-error_signal`: Specifies the pin or port to which the decoder error output is connected to. It is the net load to which the error output of the group must be connected to using an OR-tree. It indicates that a data error occurs while decoding and reconciling with the checkbits.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement Id string for tracking in the requirement management system.

```
set_safety_error_code_rule -rule RULE -data [get_pins {reg1/Q reg2/Q
 reg3/Q reg4/Q}]
```

Getting and Reporting Error Protection Rules

The `get_safety_error_code_rules` command gets a collection of the safety error code rules based on the specified options and filters. The `get_safety_error_code_rules` command supports the following options:

- `-of_objects`: Identifies the constraints associated with these data signal pins or ports.
- `-filter`: Filters collection with expression.
- `-quiet`: Does not print any messages.
- `-regexp`: Identifies patterns as regular expressions.
- `-exact`: Considers wildcards as plain characters.
- `-nocase`: Performs case-insensitive matching.
- `-expect`: Expects exactly the specified number of matching objects. The value of count must be greater than or equal to 0.
- `-expect_at_least`: Expects at least the declared number of matching objects

Feedback

- `-expect_each_pattern_matches`: Ensures that each pattern must match at least one object.
- `patterns`: Finds constraints with names matching these patterns.

Use the `report_safety_error_code_rules` command to print a report of the specified rule objects. Use the `-objects` option to report the rules specified using the objects or names.

```
Safety Error Code Rule
                Description
-------------------------------------------
error_rule1     Slice size:       8
                Encoding:         odd_parity
                Sequential:       true
```

Removing Error Protection Rules

To remove the specified safety error code rule objects from the current design, use the `remove_safety_error_code_rules` command . The `remove_safety_error_code_rules` command supports the following options:

- `-objects`: Specifies the safety error code rules to be removed. You can either specify the objects or names of the safety error code.
- `-all`: Removes all the safety error code rule objects defined in the current design.
- `-from`: Disassociates the safety error code rule objects from the specified data that deletes the rule-data association.

To delete a pin or port on which a rule is applied, use the `remove_pins` or `remove_ports` command. These commands dissociate the rule from the data bits.

Parity Schemes

You can use parity schemes to detect and fix the errors, which prevent incorrect data propagation.

The types of parity schemes are as follows:

- Parity
- Error Correction Code
- Error Detection Code

Feedback

Parity

You can use parity for hamming2 re-encoding to detect a 1-bit error in data and parity registers, and generate an error signal.

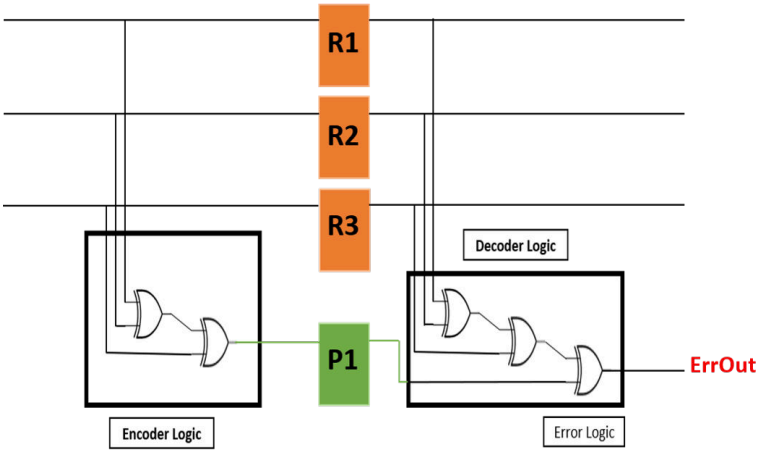
There are two types of parity:

- Even parity: For even parity, the tool synthesizes XOR gates.
- Odd parity: For odd parity, the tool synthesizes XNOR gates.

Insert parity as shown in the following example:

```
create_safety_error_code_rule  -name SEQ_ENC_DEC_RULE1 -sequential -mode
 encode_decode -type even_parity -slice_size 3
set_safety_error_code_rule -name SEQ_ENC_DEC_RULE1 -data {R1/Q R2/Q R3/Q}
 -error_signal {ErrOut}
```

Figure 22 Parity Insertion



Note:

`ErrOut` displays the error in the data or parity registers.

Parity encoder has the input nets of the data registers as input and computed parity as outputs. This computed parity acts as the input to the parity register. Parity decoder has the output nets of the data registers and parity register as input, and generates errors.

The hierarchical cell contains the encoder and decoder logic. These hierarchical cells must be marked using the `mark_safety_error_code` command, with the `-encoder` or `-decoder` option.

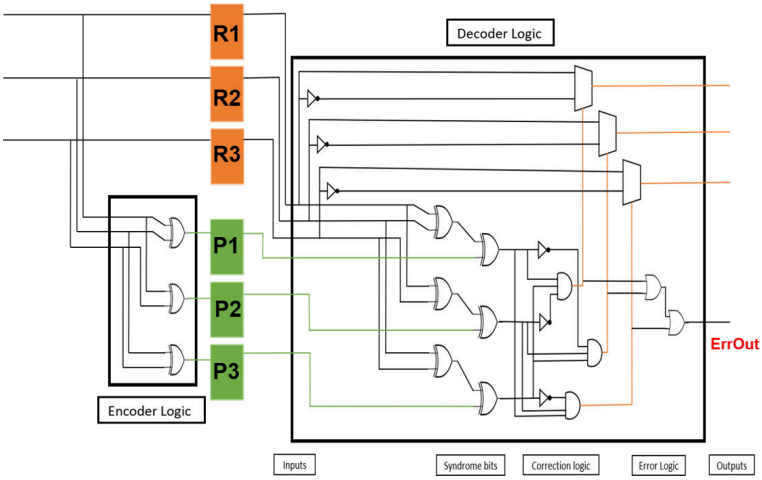
Feedback

Error Correction Code

During synthesis, the safety error code groups include encoder and decoder logic cells to encode and decode the input data signal. The error correction code (ECC) for hamming3 re-encoding ensures that the decoder detects, and fixes 1-bit error in data registers. The decoder generates an error signal, as shown in the following example:

```
create_safety_error_code_rule -name SEQ_ENC_DEC_RULE1 -sequential -mode
 encode_decode -type ecc -slice_size 3
set_safety_error_code_rule -name SEQ_ENC_DEC_RULE1 -data {R1/Q R2/Q R3/Q}
 -error_signal {ErrOut}
```

Figure 23 Error Correction Code



Note:

`ErrOut` displays the error in the data registers.

ECC encoder has the input nets of the data registers as input and computed parities as outputs. These computed parities are the input to the parity registers.

ECC decoder has the output nets of the data registers and parity register as input. It generates the corrected output along with the error signal.

Error Detection Code

Use error detection code (EDC) for hamming3 re-encoding, which detects 1-bit error in the data or parity registers and generates an error signal, as shown in the following example:

```
create_safety_error_code_rule -name RULE1 -type edc -mode encode_decode
 -slice_size 3 -sequential
set_safety_error_code_rule -rule RULE1 -data {{R1/Q R2/Q R3/Q}} -clock
 {EDC_CLK} -error_signal {ErrOut}
```

Note:

For EDC rule, the encoder cell is same as ECC encoder, but decoder cell does not contain correction logic. The error generation is same as in ECC rule.

The input nets of the data registers are the input of the EDC encoder. The computed parities are outputs of EDC encoder, which are then input to the parity registers.

EDC decoder has the output nets of the data registers and parity register as input. It generates the error signal as output.

Implemented Error Protection Schemes

You can mark the registers, with error protection schemes, that have already been implemented, for proper tool handling throughout the flow.

Marking Safety Error Codes

The `mark_safety_error_code` command creates a new safety error code group. Use this command when you need to mark the entire design that has an existing encoding or decoding, as a group. It groups one set of data bits, checkbits, clock, enable pins and correction or error pins, and stores the safety error code rule of the group. You can also specify a group name.

You can run this command after the design is linked and execute this command for each set of data bits for which a group must be created.

The `mark_safety_error_code` command supports the following options:

- `-name`: Specifies the name of the group to be created. This option is optional, and if it is not specified, the tool generates a name automatically.
- `-rule`: Specifies the safety error code rule which is assumed to be applicable for this group. The rule must be a valid or an existing rule. You can specify using an object or a rule name.
- `-data`: Specifies that the data signal is encoded or decoded. This list is an ordered set of register output pins. All the objects must be register output pins.
- `-checkbits`: Specifies an ordered list of pins or ports, where the checkbits from the encoder output are connected. This option is mandatory for all modes. All the objects must be pins or ports. Specify the checkbits in the same order as that of data bits when forming slice groups.
- `-error_signal`: Specifies the pin or port to which the decoder error output is connected to. It is the net load to which the error output of the group must be

connected to using an OR tree. It indicates that a data error occurs while decoding and reconciling with the checkbits. This option can be specified for all modes.

- `-encoder`: Specifies the hierarchical or leaf-level cell, which is the encoder logic for the safety error code group. The `report_safety_status` command checks if the:
  - encoder's input count matches the rule's data bit count
  - encoder's output count matches the rule's check bit count
- `-decoder`: Specifies the hierarchical or leaf-level cell, which is the decoder logic for safety error code group. The `report_safety_status` command checks if the:
  - decoder's input count matches the safety error code rule's data bit and check bit count
  - decoder has as many outputs as data bits, an optional error output and an optional correction output
- `-taps`: Specifies the collection of tap cells for the registers in this group.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

```
mark_safety_error_code -name GROUP -rule RULE -data [get_pins {reg1/Q
 reg2/Q reg3/Q reg4/Q}]
-checkbits [get_pins {sec1/reg5/Q reg6/Q reg7/Q}] -encoder encdr -decoder
 decdr
-error_signal [get_ports out6]
```

During synthesis, the safety error code groups support the encoder and decoder logic cells to encode and decode the input data and to detect errors using the error signal.

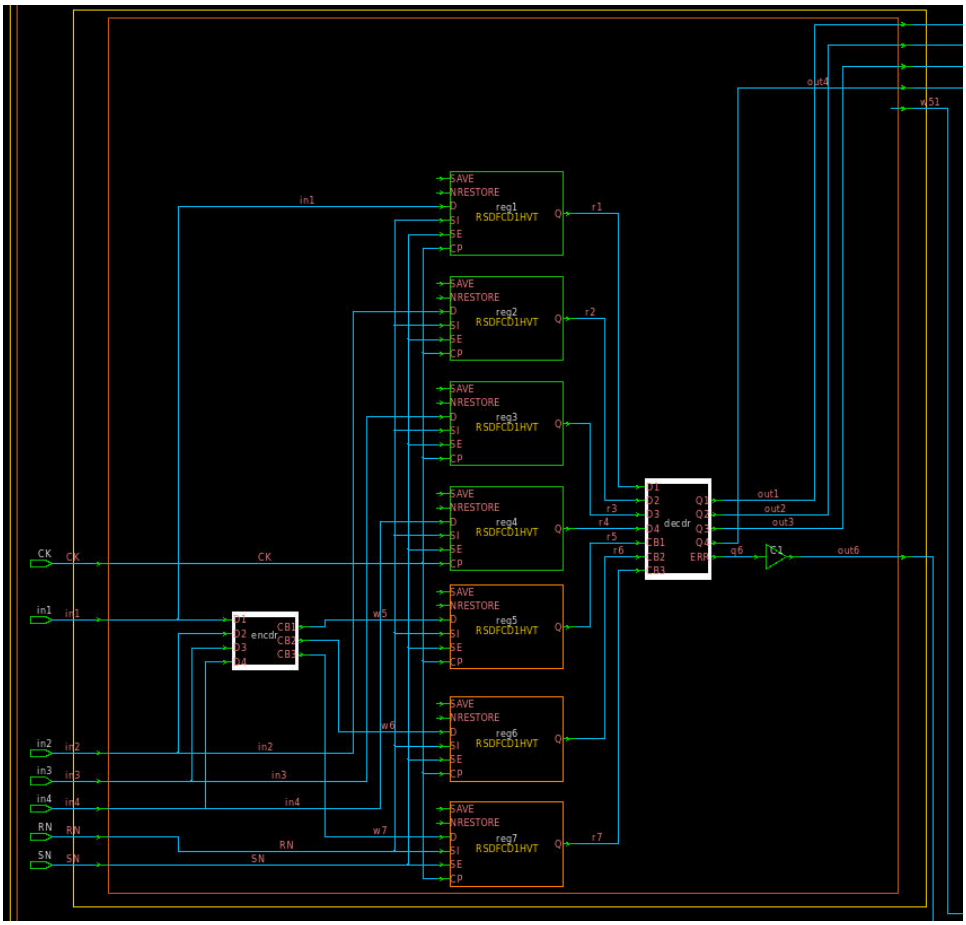
Encoder or decoder cells must be marked on the related safety error code group to safeguard their interface during optimization. The marked cells are protected automatically by the `safety_size_only` attribute for leaf cells and by the `safety_freeze_ports` attribute for hierarchical cells. The restrictions are removed after dissociating the encoder and decoder cells from the safety error code group.

The encoder and decoder cells must be connected properly. Their module or library cell must have the pin type that should be identified by a logic port map. The `set_safety_logic_port_map` command creates the logic port map, similar to creating voting logic of TMR as shown in the following example:

```
set_safety_logic_port_map -module Decoder -type decoder -input {D1 D2 D3
 D4} -checkbits {CB1 CB2 CB3} -error ERR
```

Feedback

Figure 24 Error Code Protected Register Using Encoding Type EDC



Getting and Reporting Safety Error Code Groups

You can use the `get_safety_error_code_group` command to get a collection of safety error code group objects based on the specified options and filters. If no option is specified, it displays all the safety error code groups of the current design. The `get_safety_error_code_group` command supports the following options:

- `-of_objects`: Identifies the groups which refer to the specified rule.
- `-filter`: Filters collection with the specified expression.
- `-quiet`: Does not print any messages.
- `-patterns`: Identifies the groups with names matching the specified patterns.
- `-regexp`: Identifies patterns as regular expressions.

- `-exact`: Considers wildcards as plain characters.
- `-nocase`: Performs case-insensitive matching.
- `-expect`: Expects the exact number of matching objects. The value of `count` must be greater than or equal to 0.
- `-expect_at_least`: Expects at least the specified number of matching objects
- `-expect_each_pattern_matches`: Ensures that each pattern must match at least one object.

Use the `report_safety_error_code_groups` command to print a report of the specified group objects. This command supports the following options:

- `-objects`: Reports the groups, specified using objects or names. If not specified, all groups are reported.

```
SAFETY_ERROR_CODE_GROUP_30
Rule Name:       error_rule3
Data:            reg[6] reg[5] reg[4] reg[3] reg[2] reg[1] reg[0]
Checkbits:       parity_reg_0 parity_reg_1 parity_reg_2 parity_reg_3
Error Signal:    opa_safety_err
Encoder cell:    or1200_tt/edc_encoder_logic_hier_7
Decoder cell:    or1200_tt/edc_decoder_logic_hier_8
```

Removing Error Code Groups

Use the `remove_safety_error_code_groups` command to remove the specified safety error code group objects from the current design. This command supports the following options:

- `-objects`: Removes the safety error code groups, specified using objects or by names.
- `-all`: Removes all the safety error code group objects defined in the current design.

Feedback

Reporting Safety Error Codes

The safety specification format files support the safety error code to specify parity, error detection code, error correction code encoding and decoding, and the three stages of logic are as follows:

- Encoder stage: Logically computes the checkbits for the group's data bits
- Sequential stage: Logically computes the data bit and checkbit registers in case of sequential encoding, absent in case of combinational encoding
- Decoder stage: Logically compares the data bit and checkbits with an error output and correction output (if available).

The following are three values of mode types:

- `encode`: Encoder stage with optional sequential stage
- `decode`: Decoder stage
- `encode_decode`: Encoder stage with optional sequential stage and decoder stage

The `report_safety_status` command checks the consistency between the safety error code groups and its safety error code rules and examines the safety error code related issues using the generated report.

- Use the `-safety_error_code_rules` option with the `report_safety_status` command to report the specified safety error code rules.
- Use the `-safety_error_code_groups` option with the `report_safety_status` command to report the specified safety error code groups.

6

Handling Safety Cores

You can apply the safety core concept to duplicate an entire module (for example a processor), associate it with voting or error logic and associate to error handlers. Such module duplication (as done for DCLS, that is Dual Core Lockstep) protects the design against impact of SEU by comparing the computations done by the cores and issuing an error if those computations differ. A core is a single logical hierarchy. DCLS can detect an error in the core but cannot fix it.

This section describes the following:

Dual Core LockStep

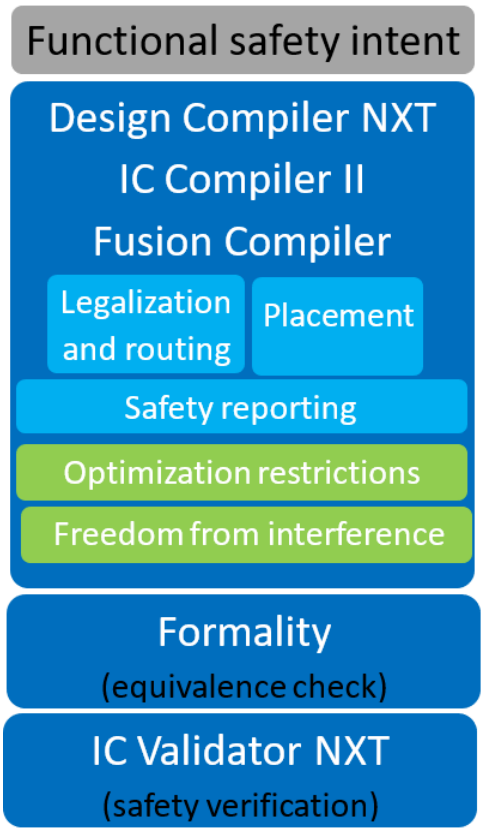
Describes the Dual Core LockStep process

DCLS improves system reliability. It runs two identical logic cores in parallel and parses their outputs through a comparator logic. If the output values are not same, it indicates an error and raises a signal for the system to take a corrective action, such as moving to a known safe state.

The layouts of the two cores must be physically independent. In addition. the internal cells, buffers, and nets from one core must be physically separated from the other core.

Feedback

The following figure shows an overview of a native DCLS flow using functional safety intent with Synopsys tools.



This section includes the following topics:

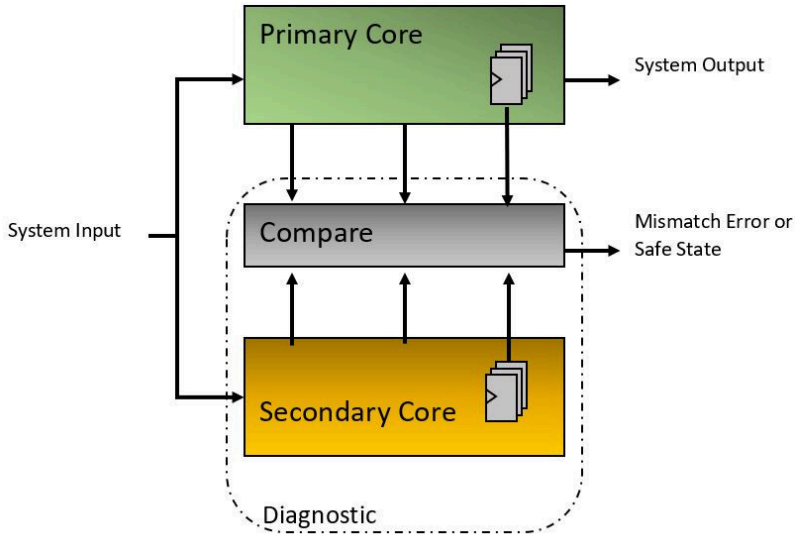
- Safety Consideration During DCLS
- Dual Core LockStep Flow

Feedback

Safety Consideration During DCLS

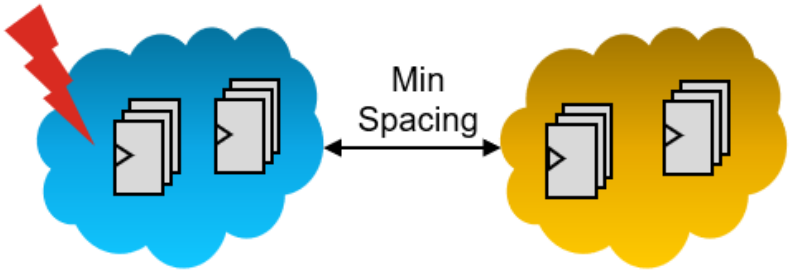
Figure 25 shows the layout of the DCLS system.

Figure 25 Dual Core LockStep System



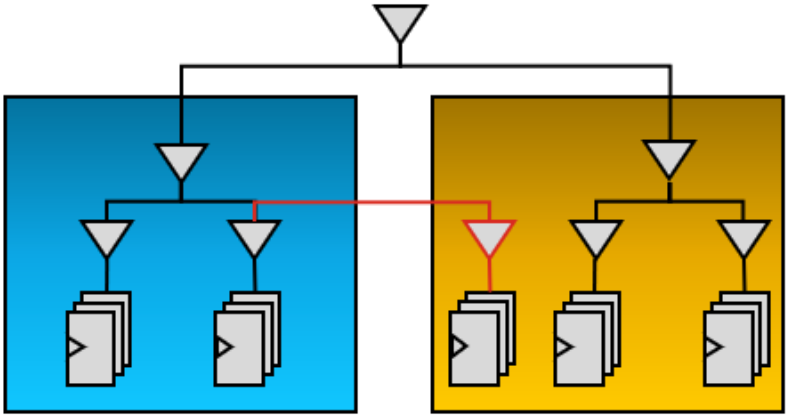
In the DCLS system, as shown in Figure 25,

- Safety critical core (for example, processor core) is replicated to form a secondary core. Internal or external signals from each core are compared cycle by cycle.
- Random hardware failures in one of the cores are detected using the comparator logic that flags an error state or a safe state.
- During placement and legalization, the tool creates a physical separation, as shown in the following figure, between the primary and secondary cores to reduce possibility of common mode failures due to soft errors.

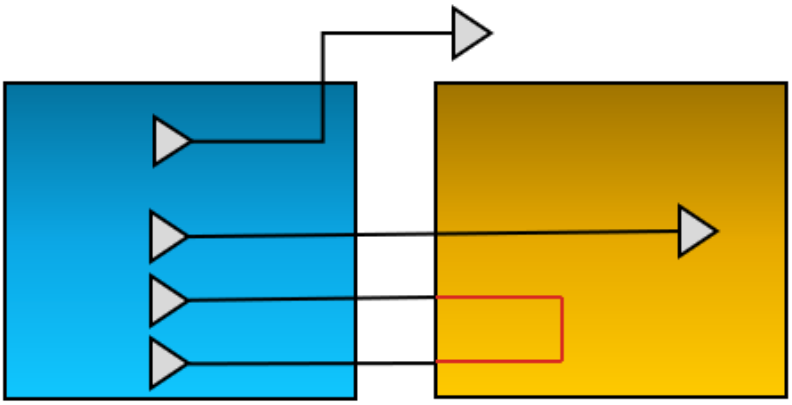


Feedback

- During clock tree synthesis, the tool avoids clustering of the registers from both the cores, as shown in the following figure, reducing the possibility of common mode failures.



- During routing, the tool avoids overlapping of physical routes between two cores if possible, as shown in the following figure. As the signal transitions in one of the cores can affect the other core through crosstalk and create the common mode failure.



Dual Core LockStep Flow

The DCLS flow includes the following steps, as shown in Figure 26:

- Safety Core Rules

  Perform the DCLS safety core rule setup before synthesis. You need to create safety core rules to enable physical separation between cores. Placement and legalization honor repelling distance requirements throughout the DCLS flow.

Feedback

- Safety Core Groups

  Perform the DCLS safety core group setup before placement optimization. You need to create safety core groups to enable physical separation between cores. Placement and legalization honor repelling distance requirements throughout the DCLS flow.

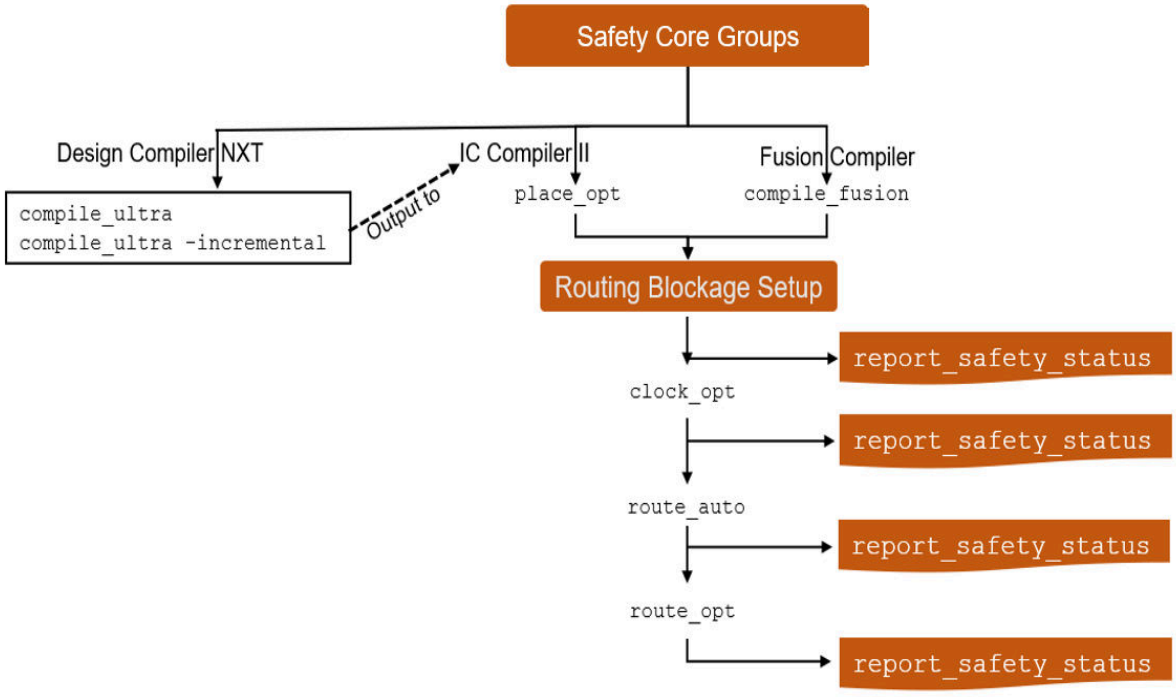
- Routing Blockage Setup

  Perform the DCLS routing blockage setup before clock optimization. Blockages ensure that nets from each core are separated by the repelling distance defined in the safety core groups, and blockages are automatically created throughout the flow as needed by the tool.

- Reporting DCLS Safety Status and Violations

  Use the `report_safety_status` command to report violations related to safety registers and DCLS. The command checks the safety status and reports violations on any placement or routing separation and on all safety cores, as placement is legalized with the IC Compiler II tool.

Figure 26 Steps in the DCLS Flow



Feedback

Perform the following steps, as shown in Figure 26:

- In the Design Compiler NXT tool (note that the Design Compiler NXT DCLS flow improves correlation with the IC Compiler II tool), perform the safety core setup before compile.
- In the IC Compiler tool,
  - Create the safety core rule to specify physical separation between cores logical hierarchies.
    - The safety core groups mark the implementation of safety cores as specified by a given safety core rule. The safety cores in one group are physically separated by the distance specified by the rule.
  - The remaining steps in the DCLS flow are performed by the IC Compiler II tool, as shown in Figure 26.

Defining Cell and Net Types

The following cell and net types are defined in the DCLS flow, as shown in Figure 27.

Cells in safety cores are classified as follows:

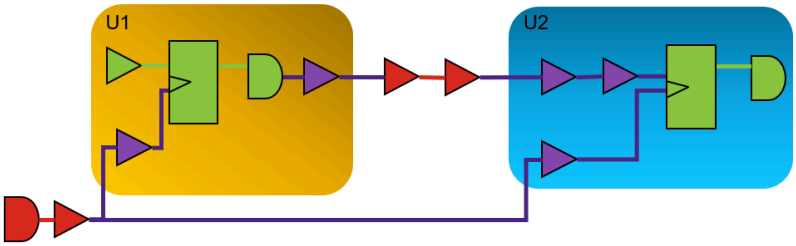
- External cells (red): Cells not belonging to a core.
- Transit cells (purple): Repeaters belonging to a core connecting directly or through another repeater to a cell not belonging to that core.
- Internal cells (green): Cells belonging to a core and not being transit cells.
- Ignored cells: The application fixed cells belonging to the core, which are not transit cells.

Nets in safety cores are classified as follows:

- External nets (red): Nets that are not connected to cells that belong to a core.
- Transit nets (purple): Any other nets except external nets or internal nets.
- Internal nets (green): Nets connected to internal cells that belong to a core which is not connected to any transit cell.
- Ignored nets: Nets that are connected to the ignored cells

Feedback

Figure 27 Cell or Net Separation Between Internal Cells or Nets Belong to Distinct Cores of the Same Safety Core



Safety Core Rules

You can create a safety core rule using the `create_safety_core_rule` command and such a rule can be applied to a safety critical core using the `set_safety_core_rule` command. The rule is a specification of the intent to duplicate the safety core and indicates physical separation requirements for placement and routing. The `create_safety_core_rule` command supports the following options:

- `-name`: Specifies the name of the rule to be created. If not specified, the tool automatically generates and assigns a name.
- `-update`: Updates any existing core rule. The `-name` option must be specified along with the `-update` option.
- `-num_cores`: Specifies the number of copies of core. Default is 2.
- `-distance`: Specifies the distance values between the cores. It can be either in the form of {x y} pair or a single number denoting the radial distance. The distance cannot be 0.
- `-routing_separation`: Enables routing blockage creation. Specifies if routing separation is required or not. Default is false.
- `-routing_guardband`: Specifies single number denoting radial distance. Default is 0.
- `-logic_module`: Specifies the reference module to be inserted for voting or comparison.
- `-logic_mapping`: Identifies the library cells, which can be used to implement the voting or compare logic.
- `-tap_mapping`: Specifies the list of library cells, out of which the appropriate one is used for instantiating tap cells for the safety cores. If this option is specified, the `-isolation` option is also automatically assumed even if it is not specified. This is optional.

Feedback

- `-isolation`: Specifies that the safety core group needs isolation. If this option is given, it is not necessary to specify the tap cells using the `-tap_mapping` option. If tap library cells are not available, the tool automatically selects the tap cells from the available reference libraries. This option is optional.

The following safety core rule defines a required Manhattan distance of 8u between the cores that should follow this rule:

```
prompt> create_safety_core_rule -name rule3 -distance 8
```

The `set_safety_core_rule` command associates a safety core with a rule. This command supports the following options:

- `-rule`: Specifies the safety core rule to be applied to the cores.
- `-error_signal`: Specifies the pin or port to connect the comparison of error outputs.
- `-split_pins`: Identifies which pins require tree splitting.
- `-cores`: Specifies the list of core instances on which safety core rule need to be set.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

The following example creates an association of a rule with cores and pins:

```
set_safety_core_rule -rule rule1 -cores {core1 core2}
```

The `report_safety_status` command reports such associations and provide counts for the number of associated safety cores. Also, the unused safety core rules are reported. If a specified core already has another rule associated with it, that core is skipped, and the original rule remains applied to it.

You can use the `get_safety_core_rules` command to find and create a collection of safety core rules based on the specified options and filters. If no options are specified, it displays all the safety core rules of the current design.

You can use the `remove_safety_core_rules` command to remove the specified safety core rule from the current design. If the specified safety core rule is associated to any safety core group, the command displays a warning message.

You can use the `report_safety_core_rules` command to display all the specified rules.

Feedback

Safety Core Groups

Figure 26 shows the safety core groups setup step in the DCLS flow. To setup safety core group in the DCLS flow,

- Enable advanced legalization to set legalization for the safety core groups.
- Define the safety core groups with the `mark_safety_core` command.

  Safety core separation is honored during placement and legalization stages (`place_opt`, `clock_opt`, and `route_opt`). You can use the `report_safety_status` command after the `place_opt` stage to ensure that physical separation is honored as needed.

The `mark_safety_core` command creates a safety core group that replaces the existing implementation that used repelling group bounds. The `report_safety_status` command counts the safety register groups. Use the `mark_safety_core` command to specify the group as a safety core group.

The `mark_safety_core` command supports the following options:

- `-rule`: Specifies the safety core constraint based on which the safety core group is created.
- `-cores`: Specifies the list of core instances.
- `-name`: Specifies the name of the safety core group.
- `-logic`: Specifies the cells that constitute the voting logic.
- `-error_signal`: Specifies the pin or port to connect the comparison of error outputs.
- `-split_pins`: Specifies the set of pins, which should be split, to be a part of different branches of clock or high-fanout net tree. When used, all cores must have the same pin names.
- `-taps`: Specifies the collection of tap cells for the registers in this group.
- `-error_output`: Specifies the output pin of the logic circuit, which represents the error output. This signal indicates that an error is detected in the state registers. It might be an output pin of a logical hierarchy or a leaf cell.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

Feedback

The following example updates the `split_pins`, `error_signal`, `logic`, and `taps` attributes of existing safety core group `group2`:

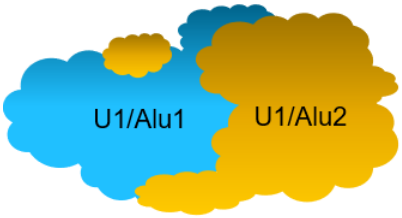
```
prompt> mark_safety_core -update -name group2 -split_pins {flop/Q}
 -error_signal err_blk/IN1 -logic {Iand1} -taps {TAP3 TAP4}
```

You can use the `get_safety_core_groups` command to find and create a collection of safety core group objects based on the specified options and filters. If no options are specified, it displays all the safety core groups defined in the current design.

The `remove_safety_core_groups` command removes the specified safety core group from the current design. The `report_safety_core_groups` command displays a report of the specified safety core group objects.

Supporting Coarse Placement

By default, hierarchical modules are placed optimally for wire length, timing, congestion, power, and so on. Placement of hierarchies might be fragmented as shown in Safety Core Groups, which is not ideal for functional safety.



If safety cores are set up as a safety core group as shown in Figure 28, the tool

- Performs initial core separation during coarse placement, where the tool allows a few spacing violations that can be removed during the subsequent legalization step (see Legalization Support).
- Avoids U-shapes or other complex shapes of the safety cores.

  This reduces the possibility of internal wire routing crossing from one module to another module.

Coarse placement supports separation of the safety cores belonging to the safety core groups defined with the `mark_safety_core_groups` command.

Feedback

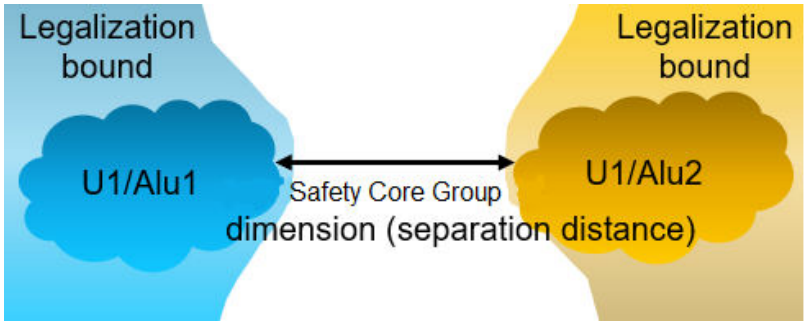
Figure 28 Hierarchies Set Up as a Part of Safety Core Group

Legalization Support

Legalization enforces separation of safety cores belonging to a safety core group, as shown in Figure 28. The tool, as shown in the Figure 29,

- Enforces separation by creating legalization bounds
- Automatically calculates legalization bounds for each core of the safety core group before performing legalization
- Enforces separation of a safety core's internal cells during legalization but cannot enforce separation on external and transit cells

Figure 29 Separation of Safety Core Group With Legalization Bounds



List of Commands to Manage Safety Core Groups

This section provides examples that explain how to create safety core groups and collections for safety core groups and safety cores.

The following example shows how to create a safety core group with distinct separation types (rectangular and diamond shape), which is represented in Figure 30.

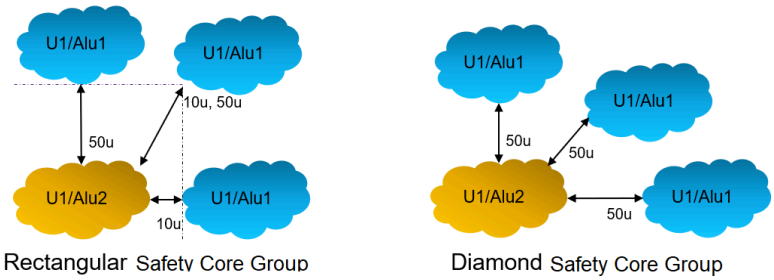
```
create_safety_core_rule -name SCR1 -distance {10 50};
mark_safety_core -name SCG1 -rule SCR1 [get_cells {U1/Alu1 U1/Alu2}]

create_safety_core_rule -name SCR1 -distance {50};
mark_safety_core -name SCG1 -rule SCR1 {U1/Alu1 U1/Alu2}
```

Feedback

Figure 30 shows the two distinct safety core groups. For a given placement location of the yellow safety core,

- Three potential placement solutions for the blue safety core are shown for core separation.
- Coarse placement selects the solution based on the objectives of placement.
- Distances are measured from leaf cell edge to leaf cell edge.

Figure 30 Safety Core Group With Rectangle and Diamond Shapes Respectively



The following examples show how to create various collections for safety core groups and safety cores:

- `set SCG [get_safety_core_groups]`: Creates a collection of all safety core groups in a design.
- `set SCGCells [get_attribute -objects $SCG -name cells]`: Creates a collection of all cells or groups defined in a safety core group.
- `set internalCells [get_attribute [get_cells $coreX] -name rgb_core_internal_cells]`: Creates a collection of all internal cells in a design.
- `set transitCells [get_attribute [get_cells $coreX] -name rgb_core_transit_cells]`: Creates a collection of all transit cells in a design.

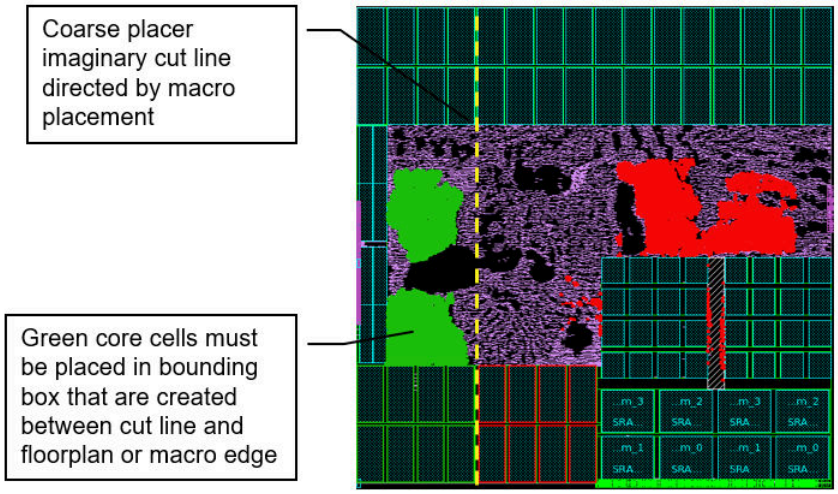
Handling Macros

To handle macros in the DCLS flow for DCLS core separation,

- The DCLS coarse placer produces horizontal or vertical cut lines to separate cores.
- Fixed macro placement affects cut lines that are selected by placer.
- Cut lines produce bounding box against floorplan edge or other blockages to include standard cell placement.

Feedback

- The following method is recommended to check macro placements:
  - Start with the floorplan design by using fixed macro placement.
  - Set up safety core groups.
  - Perform coarse placement with the `create_placement` command.
  - View the standard cell core separation and the imaginary cut lines in GUI, as shown in the following figure:



In the DCLS flow, you might have to ignore either a few or all macros, such as SDC constraints, non production macro placement, and so on. To ignore macros while implementing DCLS, set the `dcls_ignore` attribute on macro cell objects, as shown in the following example:

```
define_user_attribute -type boolean -classes cell -name dcls_ignore
set macros [get_cells -physical_context -filter "is_hard_macro==true"]
set_attribute -objects $macros -name dcls_ignore -value true
```

After you set the `dcls_ignore` attribute on the macro cell objects, the macro cells are ignored during

- Coarse placement DCLS separation
- Creation of legalization bounds
- Creation of routing blockages
- Reporting safety status and violations

  Cells are ignored and the attached nets are added to the ignored section of the safety status report. For more informations, see Reporting DCLS Safety Status and Violations.

Feedback

Routing Blockage Setup

The routing blockage setup step in the DCLS flow is performed in the IC Compiler II and Fusion Compiler tools, as shown in Figure 26. The tool automatically creates the routing blockages throughout the DCLS flow for safety core rules for which `-routing_separation` option is specified:

To manually create the routing blockage, use the `create_safety_core_group_shapes` command.

To create a collection of internal and transit nets, use the following commands:

```
set internalNets [get_attribute [get_cells $coreX] -name
 rgb_core_internal_nets]
set transitNets  [get_attribute [get_cells $coreX] -name
 rgb_core_transit_nets]
```

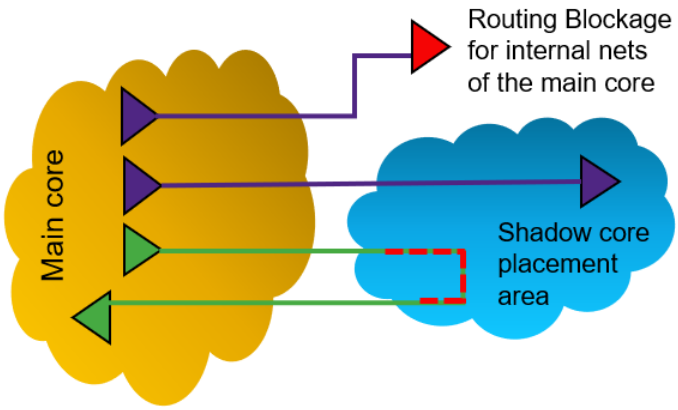
Routing Separation

This section describes the requirements for routing separation.

To ensure routing separation, as shown in Figure 31.

- Purple transit nets are allowed to cross over the shadow core placement area (indicated by purple).
- Green internal nets should be blocked from crossing over the shadow core placement area (indicated by green and dotted red).

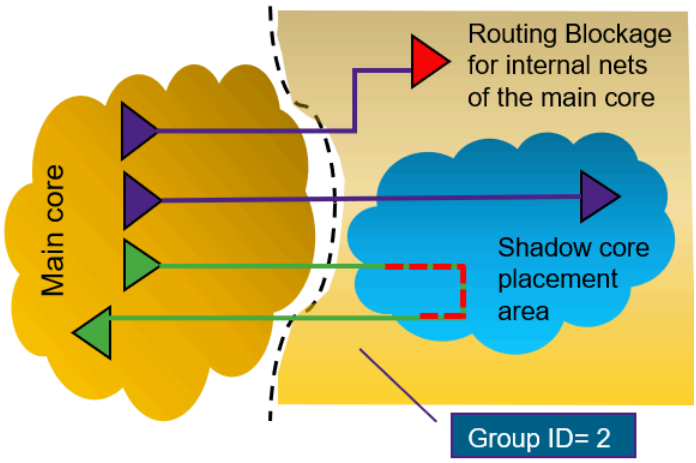
Figure 31 Routing Separation



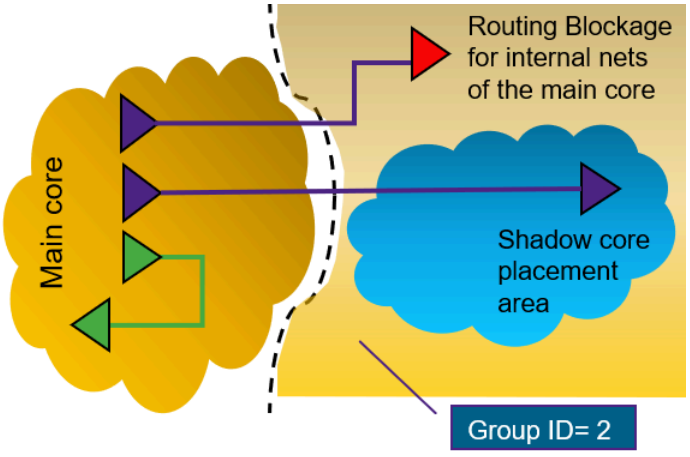
Feedback

To ensure routing blockages are automatically created after legalization,

- Expanded routing blockages are created around each safety core and assigned with an unique ID with the `routing_blockage_group_id` attribute (indicated by yellow square with Group ID).
- The `routing_blockage_group_id` attribute of internal nets of a safety core is set to the routing blockage ID of the other safety core in the bound. The following figure shows the routing blockages covering the shadow core with `routing_blockage_group_id == 2`. The internal nets of the main core are also assigned with `routing_blockage_group_id == 2` to direct the router not to route the internal nets over routing blockages of the shadow core.



- The global and detail routers ensure that the internal nets of the main core are physically separated from the other safety core in the group. Transit nets are allowed to cross over.



Feedback

See Also

- Defining Cell and Net Types

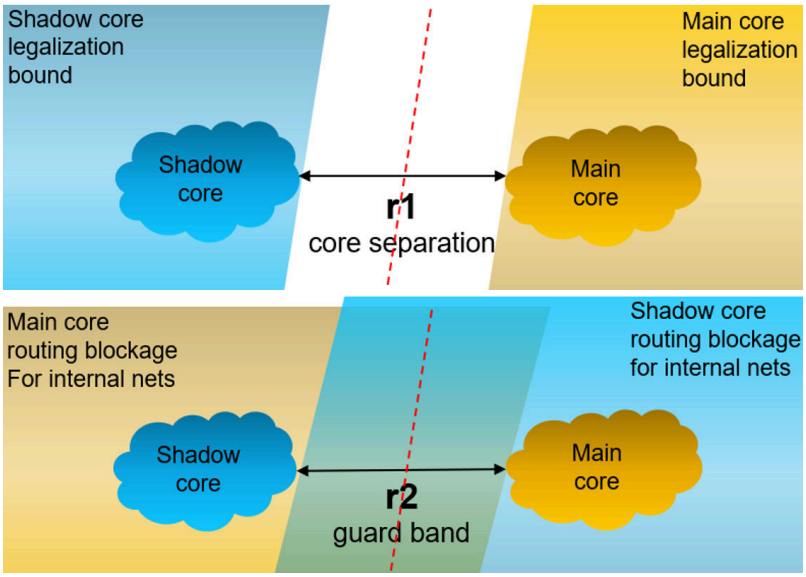
Routing Guard Band

The routing guard band defines and enforces the required internal net routing separation between the cores. By default, a routing guard band equal to the core separation distance is applied to the routing blockages. For example, the following command defines a cell separation distance of r1:

```
create_safety_core_rule -name SCR1 -distance {r1} -routing_separation;
mark_safety_core -name SCG1 -rule SCR1 {main_core shadow_core}
```

In Figure 32, the routing guard band r1 is the core separation distance. By default, the r2 routing guard band is equal to the r1 core separation distance.

Figure 32 Routing Guard Band With Core Separation Distance



You can reduce the routing guard band in a few instances and give more space for the router to get an access to cell pins that are very close to the legalization bound edge. To reduce the routing guard band, specify the routing guard band while creating manual routing blockage with the following command:

```
create_safety_core_group_shapes -safety_core_groups SCG1 [-guard_band
 {r2}]
```

You can also define the routing guard band by setting the `-routing_guard_band` option to the`create_safety_core_rule` command. The routing guard band r2 (as shown in

Figure 32) must satisfy `0 <= r2 <= r1`. The specified format must be the same as the format used for the rule's `-distance` option:

- Rectangle: The guard band should have the `{x y}` format (`-distance {width height}`).
- Diamond: The guard band should have the `{r}` format (`-distance {extent}`)

Reporting DCLS Safety Status and Violations

You can generate the safety status report with the `report_safety_status` command at any point in the DCLS flow after defining the safety core groups. You can view the DCLS safety status report in the following ways:

- Textual Safety Report
  - Summary section in the textual report: Includes user-defined header section for customization, the summary section with pass and fail information for each safety core group.
  - Messages section in the textual report: Provides detailed information using the error and information messages, listing any safety requirement violations such as potential placement or routing issues.
- GUI Safety Report

  View information, warning, or error messages in the GUI message browser. Highlight to view detailed information of each message and then use the GUI hyperlinks to highlight cores, cells, and nets for debugging.

Textual Safety Report

The `report_safety_status` command generates the safety status textual report for DCLS that includes the summary and messages section.

Example 9 shows the summary section in the textual report that lists the following details (also, see Example 10):

- The total number of safety core groups that are passed or failed.
- The total number of safety core group cores that are passed or failed.
- The total number of safety core group cells and nets that are internal, transit, or ignored.
- The total number of safety core group routing blockages.
- The total number of safety core group with or without routing blockages.
- The total number of safety core group cores with or without routing blockages.

Feedback

Example 9 Summary Section of the Textual Report

```
---------- Summary ----------------------
safety core groups (total/passed/failed): 6/3/3
safety core group cores (total/passed/failed): 12/9/3
safety core group cells (total/internal/transit/ignored):
 88906/87672/0/1234
safety core group nets (total/internal/transit/ignored):
 92115/84367/6342/1406
routing blockages: 169
safety core groups (total/with/without) routing blockages: 6/6/0
safety core group cores (total/with/without) routing blockages: 12/12/0
------- end of Safety status report ------
```

Example 10 shows the messages section in the textual report with the following messages:

- Information messages.
- Warning messages that you must review and debug.
- Error messages that indicate violations of functional safety constraints.

Example 10 Messages Section of the Textual Report

```
---------- Messages ---------------------
Information: Safety core group related object counts for block ChipTop.
 (SR-132)
Error: Found an internal net of core dcls1 in safety core group MySCG1
 has a blockage_group_id 3 which is not associated to the core. (SR-147)
Error: The internal nets of core dcls1 in safety core group MySCG1 are
 referring to multiple blockage_group_ids {3 1}. (SR-150)
Error: Found an internal net (dcls2/C1/n1) of core dcls2 in safety core
 group MySCG1 is missing a blockage_group_id associated to the core.
 (SR-149)
Error: Found an internal net of core dcls2 in safety core group MySCG1
 has a blockage_group_id 1 which is not associated to the core. (SR-147)
Error: Found an internal net of core dcls2 in safety core group MySCG1
 which has a blockage_group_id 1 that is associated to hostile core
 dcls1. (SR-148)
Error: Missing routing blockages on layer METAL1 with blockage_group_id 1
 for core dcls2 in safety core group MySCG1. (SR-129)
Error: Missing routing blockages on layer METAL2 with blockage_group_id 1
 for core dcls2 in safety core group MySCG1. (SR-129)
Warning: Found 3 internal cells of core dcls2 blocked by routing blockage
 (blockage_group_id 1) on layer METAL3 in safety core group MySCG1.
 (SR-145)
Warning: Found 3 exposed hostile cells from core dcls1, not fully covered
 by routing blockage (blockage_group_id 1) on layer METAL3 for core dcls2
 in safety core group MySCG1. (SR-146)
Information: Core dcls2 in safety core group MySCG1 has 3 protruding
 transit nets that overlap routing blockage (blockage_group_id 2, layer
 METAL8). (SR-144)
...
```

Feedback

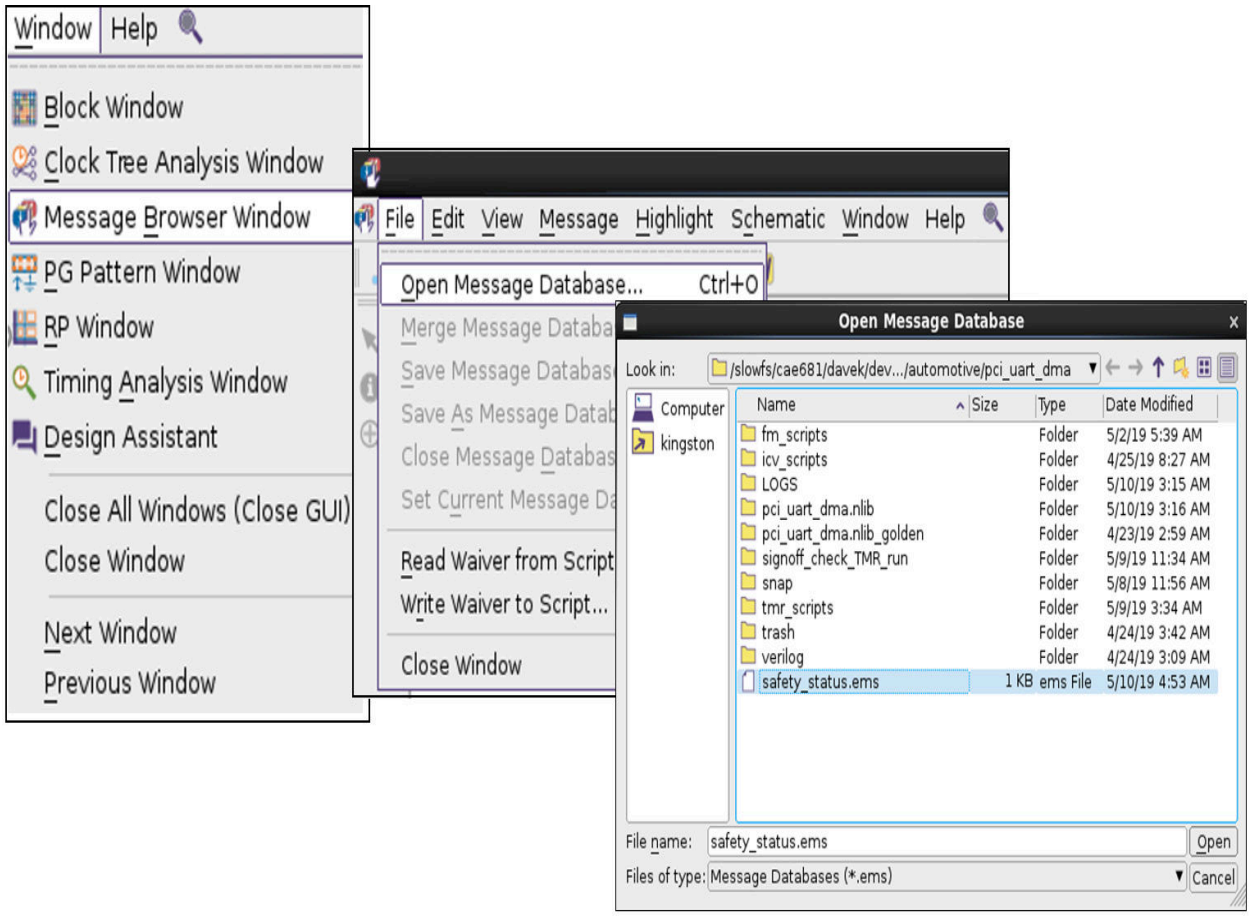
```
Information: Details of core dcls2 in safety core group MySCG1. (SR-151)
---------- Summary ----------------------
Statistics - internal/transit/ignored
-----------------------------------------------------
| type  category  total  internal  transit  ignored |
| ====  ========  =====  ========  =======  ======= |
| SC       cells      6         6        0        0 |
| SC        nets     10         4        6        0 |
-----------------------------------------------------

Statistics - with/without
-------------------------------------------------------------------------
| type                                 category  total  with  without |
| ====  ===================================  =====  ====  ======= |
| SC    groups, considering routing blockages      1     1        0 |
| SC     cores, considering routing blockages      2     2        0 |
-------------------------------------------------------------------------

Statistics - passed/failed
-------------------------------------------------------------
| type                      category  total  passed  failed |
| ====  ==========================  =====  ======  ====== |
| SC               safety core rules      1       1       0 |
| SC              safety core groups      1       0       1 |
| SC                    safety cores      2       0       2 |
-------------------------------------------------------------
------- end of Safety status report ------
```

GUI Safety Report

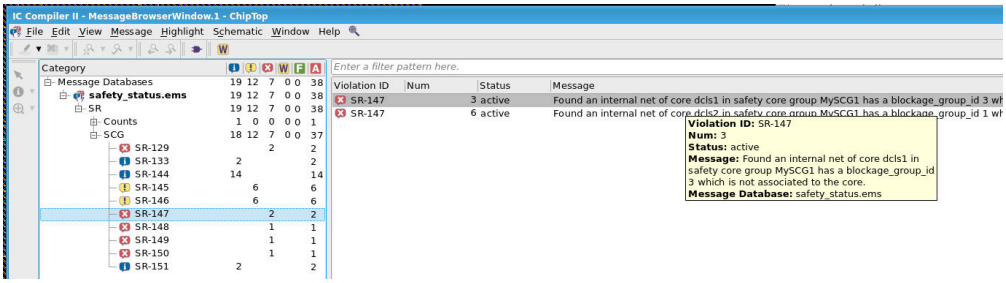
You can view the DCLS safety status report by using the GUI message browser. The GUI safety report uses message databases for the Enhanced Messaging System (.ems). This section explains how to view the messages in detail in different windows.

Feedback

To view messages in the GUI message browser, choose Window > Message Browser Window > File > Open Message Database. Select the file that you want to view from the Open Message Database window and click open, as shown in the following figure:

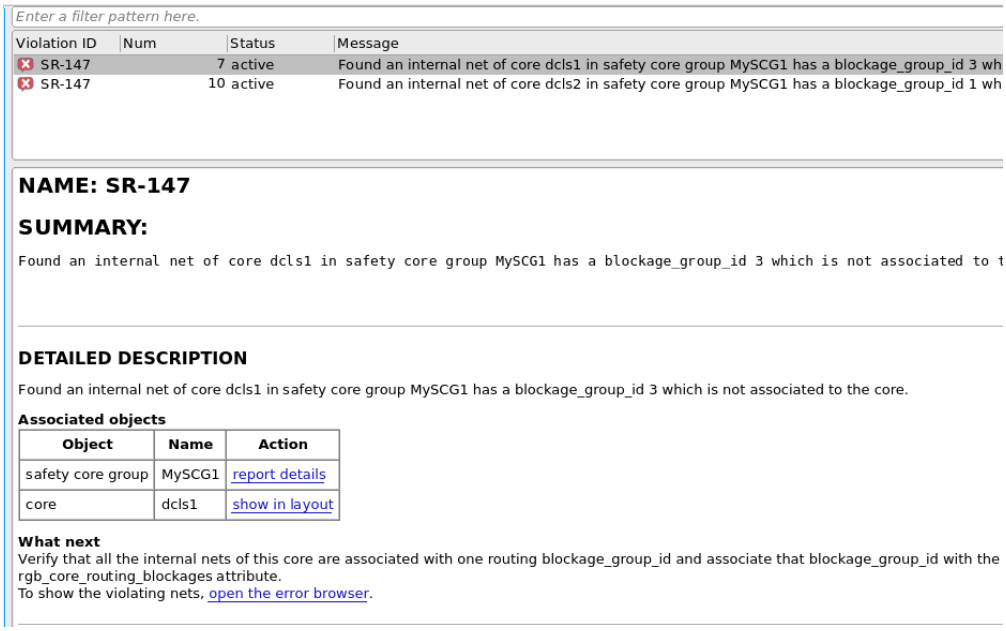


The following figure shows how to highlight each message to view the complete description of error, warning, and information messages. Hover using the mouse to view additional information of a message.

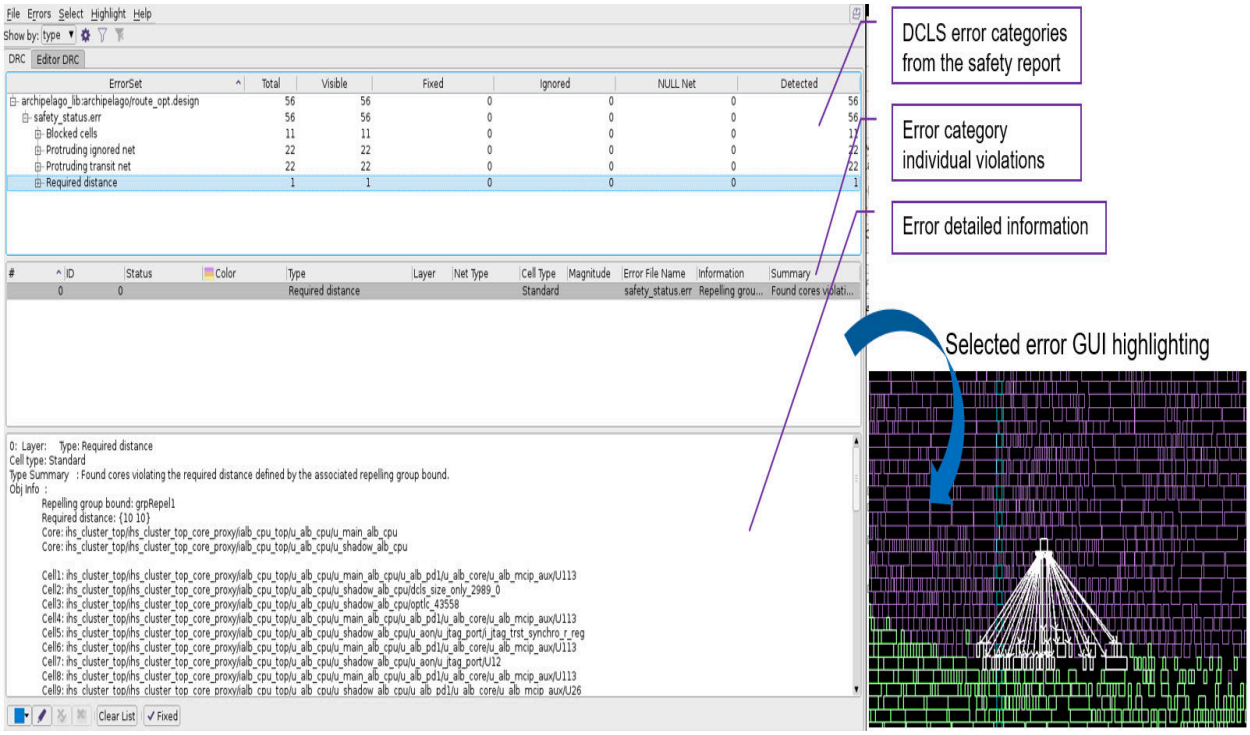


Feedback

When you highlight a warning or an error message, it displays detailed information with the hyperlinks, as shown in the following figure:



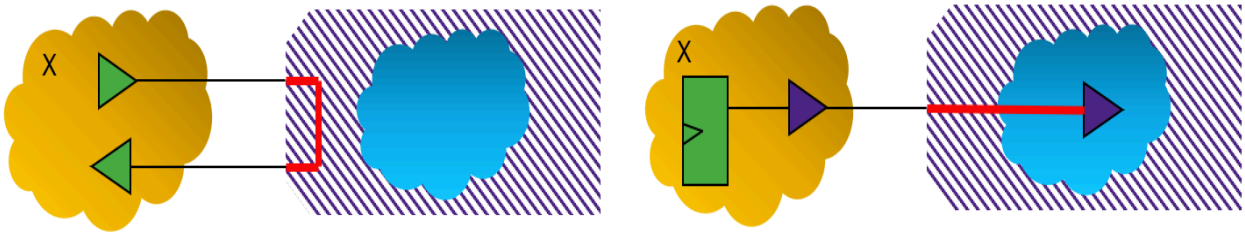
The following figure shows the Error browser:



Feedback

The following error messages and figure show issues with protruding nets:

```
Error: (info) Core X in safety core group grpRepel has 1 protruding
 internal nets
overlapping routing blockage (blockage_group_id 10, layer M11). (SR-043)

Information: (info) Core X in safety core group grpRepel has 1 protruding
 transit nets
that overlap routing blockage (blockage_group_id 9, layer M2). (SR-044)
```

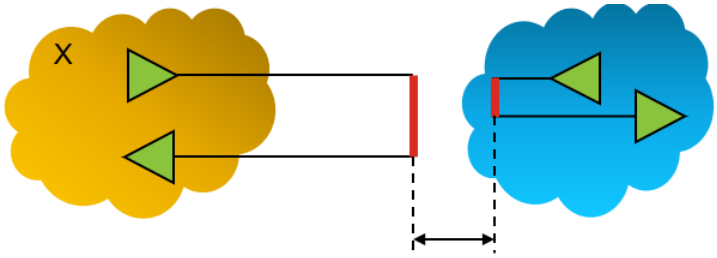


Protruding internal net error Message (SR-043)

Protruding transit net information message (SR-044)

The following error message and figure show issues with internal routing core separation:

```
Error: (error) Core X in safety core group grpRepel has 1 internal nets
 that
violate the required core separation. (SR-042)
```



Violates the required core separation for routing guard band

Supporting Voting Logic, Error Logic and Port Mapping

The voting logic, error logic, or both associated with the safety cores can be expressed as a VEL. A VEL can be either one of the following:

- (v1): an instance of a logic module
- (v2): an instance of a library cell

You can specify the safety core rule using the `create_safety_core_rule` command:

- With the `-logic_module` VEL: the VEL of the associated safety core group must have the `ref_module` VEL for each of the safety core's outputs
- With the `-logic_mapping` {VEL1, …VELN}: the VEL of the associated safety core group must be an instance of the library cell VEL1, … or VELN for each of the safety core's outputs. It is not required that each output uses the same library cell.
- Without either the `-logic_module` or the `-logic_mapping` VEL: the safety core group does not need to have an associated VEL. This can be used for designs in which you have already incorporated voting or error logic in one of the cores itself. If the safety core group is associated with VELs, all the core's outputs are expected to have a unique VEL per unique output name. The `report_safety_status` command reports the missing VELs.

When the number of cores of safety core rules is two, the VEL is expected to have two inputs and one error output. If the VEL has more inputs or outputs, an associated logic port map must exist to identify the two inputs and the error output, but no voting output. The `report_safety_status` command issues an error message if the (restricted) input or output count is not met.

The VELs associated with a safety core group must be protected by safety `size_only` in case of leaf cells, and safety `freeze_ports` in case of hierarchical cells. The `report_safety_status` command reports if VELs are missing restrictions.

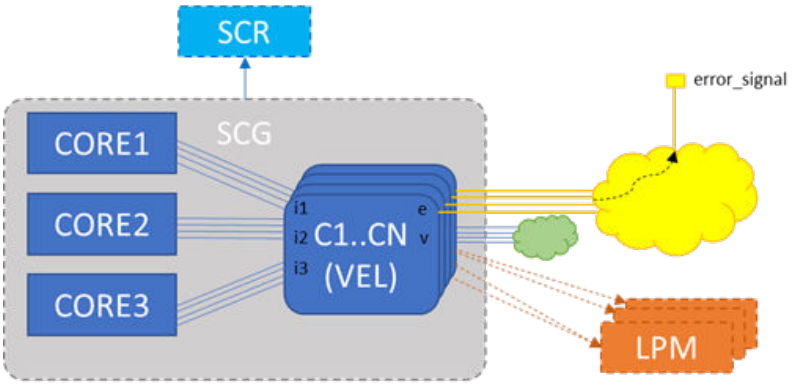
Checking Error Signals

The safety core group needs to have an associated `-error_signal` pin or port when the core count is two, but also if the underlying safety core register specifies a VEL with error output using the `-logic_module` or `-logic_mapping` options with optional logic port map.

The `report_safety_status` command displays an error message if the expected `error_signal` is not annotated on the safety core group. When the safety core group specifies an `-error_signal` pin or port, all of the VELs' error outputs need to feed into an OR-tree that's part of that pin or port's input cone. The `report_safety_status` command checks whether this condition is met, but does not perform any functional check on the combinatorial cells in the input cone to check that they are indeed OR-gates.

Feedback

Figure 33 Safety Cores With Voting or Error Logic and Error Output



Safety Core Isolation

A high-energy charged particle passing through pn-junction diodes might cause a failure. This failure is referred as battery effect or Multi-Coupled Bipolar Interaction (MCBI) and it affects the well that might cause several cells to fail simultaneously. To protect the neighboring registers and cores from this issue, the isolation is applied to a few registers of the safety cores to add well-taps adjacent (left and right side) to the registers. Note that you should apply isolation to only those registers that are next to a register of a hostile core in the same row.

To control inserting of the well-tap, use the `-isolation` option with the `create_safety_core_rule` command.

```
create_safety_core_rule -isolation
```

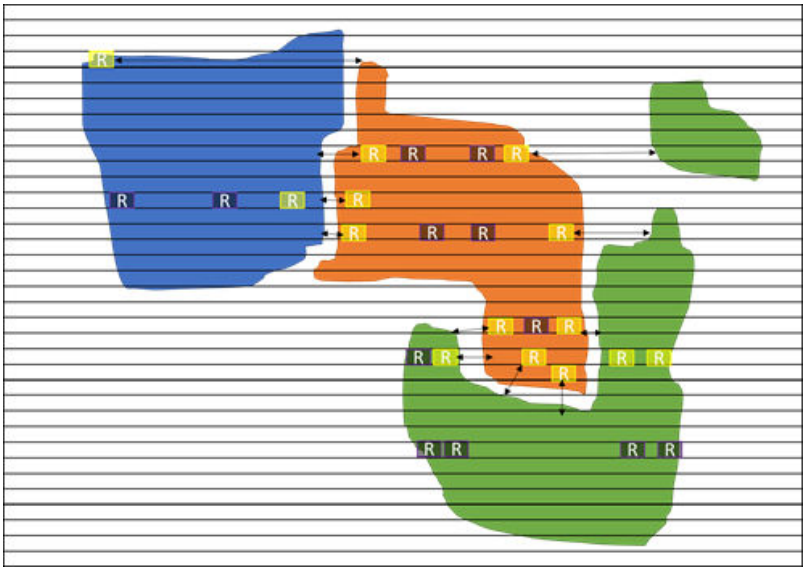
To specify a list of reference modules and limit the modules the well-tap cells should refer, use the `-tap_mapping` option with the `mark_safety_core` command. If not specified, any well-tap library cell is allowed. To mark the inserted well-taps for a given safety core group, use the `-tap` option with the `mark_safety_core` command.

The legalizer and placer reserve space to place well tap throughout the implementation flow, but the insertion takes place after `route_opt`, for example, during chip-finishing. The well-taps are inserted using the `create_safety_tap_cells` command. For more details on inserting tap cells, see the Inserting Tap Cells.

For a safety core group with isolation, the `report_safety_status` command checks that all registers neighboring and sharing a register of a hostile core have appropriate well-taps in place, and these taps are marked on the safety core group. If the underlying safety core rule specifies reference modules using the `-tap_mapping` option, the well-taps must refer one of the specified modules. The marked taps must have appropriate type well-tap. All the site rows occupied by those isolated register must have adjacent well-tap cells.

Feedback

Figure 34 Well-tap isolation for registers neighboring a register of a hostile safety core in a shared row



7

Failsafe Finite State Machine

Finite State Machines (FSM) are used as control logic in digital designs. A Single Event Upset (SEU) in an FSM has the potential to corrupt large portions of logic controlled by a state machine. This type of failure is classified as a derived failure as per ISO 26262 standards and can have a large effect on the reliability calculations for the overall design.

To mitigate this risk of failure, FSM must be implemented in such a way that the state machine always has a known behavior for every possible state. The setting of this minimum requirement is known as Failsafe finite state machine. Along with setting of the minimum requirement, the state machine might require additional safety measures to meet specific safety goals for a design. These safety measure might include functional safety oriented state encodings for SEU detection or correction, or even redundant versions of the state machine with voting logic.

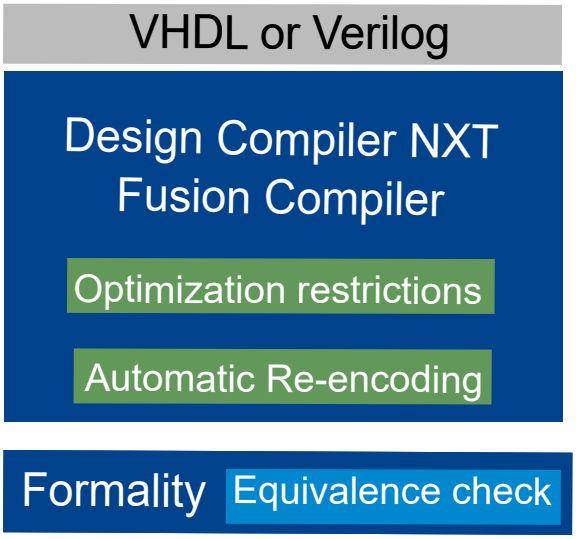
The Failsafe FSM feature in the Design Compiler NXT and the Fusion Compiler tools enables the following features:

- Optimization restrictions to implement all functionality written in RTL without changes, including default and unreachable states.
- Automatic re-encoding of the state machine from a binary encoding in RTL into a functional safety encoding to detect or correct an SEU

Feedback

The following figure shows the failsafe FSM flow with the Synopsys tools:

Figure 35 The Failsafe FSM Flow



This section includes the following topics:

- Structure of a RTL FSM
- Controlling Optimization
- Automatic Re-encoding of Failsafe FSMs
- FSM Rules and Groups
- Manual Failsafe FSM Flow
- Reporting FSM Information
- Visualizing SEU Handling
- Using Formality to Verify a Re-encoded Failsafe FSM

Structure of a RTL FSM

A finite state machine in a SystemVerilog or VHDL file consists of the following elements as shown in Example 11 and Example 12:

- Combinational decode logic to determine the next state and the state machine outputs.
- Sequential elements to store the state.
- An enumerated type to capture state names and possibly encodings.

The syntax for a FSM is slightly different depending on the RTL language. SystemVerilog and VHDL examples are shown in Example 11 and Example 12.

Example 11 shows the structure of a SystemVerilog FSM that includes

- An enumerated type for the state vector: The `typedef` definition with different states enumerated. The enumeration (enum) can contain the encoding for the various states, or the default is sequential binary encoding starting at 0.
- `always_ff`: This process specifies the registers for the FSM.
- `always_comb`: This process specifies the next state logic.

Example 11 Structure of a SystemVerilog FSM

```
typedef enum logic [4:0] {
  IDLE = 5'b00000,
  S1   = 5'b00011,
  S2   = 5'b00101,
  S3   = 5'b00110,
  S5   = 5'b01001,
  S6   = 5'b01010
} state_ham2_t;

always_comb
begin
  nextState = IDLE;
  case (currState)
    IDLE: if (go) nextState = S1;
    S1: nextState = S2;
    S2: nextState = S3;
    S3: nextState = S5;
    S5: nextState = S6;
    S6: nextState = IDLE;
    default: nextState = IDLE;
  endcase
end

always_ff @(posedge clk or negedge rst)
  curState <= rst ? nextState : IDLE;
```

Example 12 shows the structure of a VHDL FSM that includes

- A type declaration
- Process specification
- A sequential process that specifies the registers for the FSM
- A combinational process that specifies the next state logic

Feedback

Note:

VHDL FSM does not need an enumerated encoding.

Example 12 Structure of a VHDL FSM

```
type STATE_TYPE is (S0, S1, S2, S3, S4);
 attribute enum_encoding : string;
 attribute enum_encoding of STATE_TYPE :
     type is "00000, 00011, 00101, 00110,
              01001, 01010";

 signal state, next_state: STATE_TYPE;

process (state)
   begin
        case state is
            when S0 =>
                    next_state <= S1; outp <= 0;
            when S1 =>
                    next_state <= S2; outp <= 1;
            when S2 =>
                    next_state <= S3; outp <= 2;
            when S3 =>
                    next_state <= S4; outp <= 3;
            when S4 =>
                    next_state <= S0; outp <= 4;
            when others =>
                    next_state <= S0; outp <= 5;
        end case;
    end process;

 process (CLK, RST)
 begin
   if RST = '0' then
     state <= s0;
   elsif ( CLK'event and CLK = '1' )  then
     state <= next_state;
   end if;
 end process;
```

Controlling Optimization

For a state machine to be considered failsafe, it must always have a known behavior during an SEU. To consider an SEU for an FSM, you need to check the single-bit flip from its current state.

In many FSMs, the single-bit flip can take the state machine into a state that is not explicitly defined by the state enumeration variable. To handle this type of SEU, you must set a behavior for any state that is not explicitly defined. Set the FSM registers in the RTL with a default behavior or a defined behavior, as shown in Figure 36.

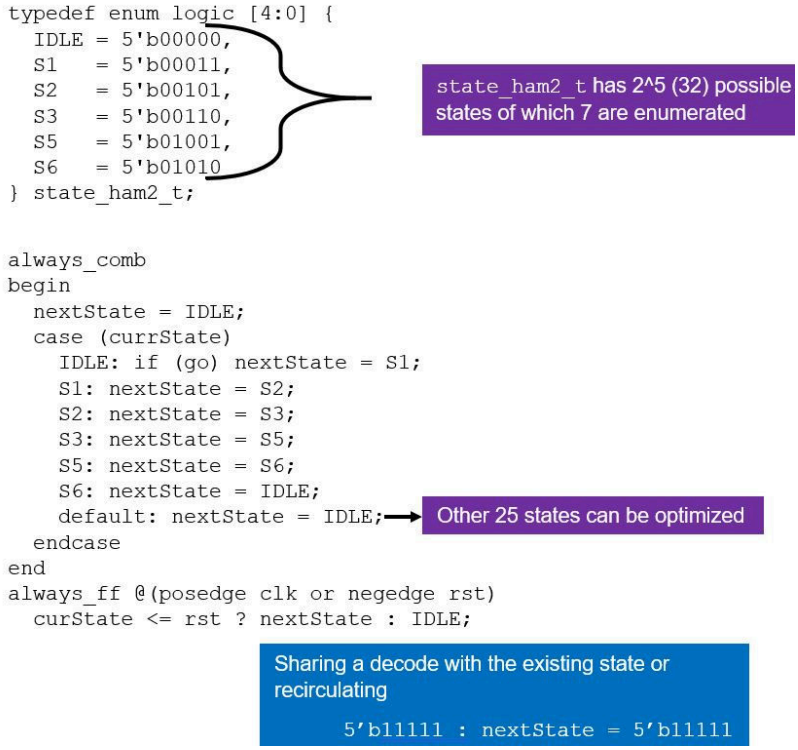
Feedback

In a default synthesis flow, the unreachable states, which include the default or other clauses in the RTL are treated as `dont_care` space optimizations.

To restrict the Design Compiler NXT and Fusion Compiler tools from changing the state transition behavior during elaboration and synthesis, add the `fsm_complete` pragma to the RTL code. See Example 13 for SystemVerilog and Example 14 for VHDL.

In Figure 36, all states out of the 32 possible states are in the default with an unknown behavior except for the 7 states that are enumerated. The remaining 25 states can be optimized because the tool can share a decode with the existing state or recirculate the states as the FSM registers with the `dont_care` attribute can be optimized.

Figure 36 Example to Show Optimization of FSM Registers



See Also

- Single Event Upset (SEU)

Automatic Re-encoding of Failsafe FSMs

In the Design Compiler NXT and the Fusion Compiler tools, use the functional safety friendly state encodings to provide resilience for a state machine to recover from an SEU.

Feedback

Hamming distance is the number of bits needed to change from one state encoding to the next state encoding.

In ISO 26262, the assumption is that only one SEU happens in one cycle. A hamming distance of two helps to detect a corrupt state that represents invalid encoding. The encoding must meet the following hamming distance:

- `- n+1`: Encoding detects `n` errors
- `2n+1`: Encoding corrects `n` errors

Note:

The hamming2 encoding in the constraints is the minimum area implementation of a Hamming distance 2 encoding. One-hot encoding, which is an FSM encoding with only one active bit for each state, is also the Hamming distance 2 encoding that is available for re-encoding.

To provide robust functionality in the presence of SEUs, implement state machine re-encoding to support hamming distance 2 and hamming distance 3 encodings.

- To specify the re-encoding goal for an FSM in the RTL, use the following examples:
  - See Example 13
  - See Example 14

Example 13 Adding a New Compiler Directive in SystemVerilog

```
always_comb
begin
  // synopsys fsm_complete "currState"

  case (currState)
    IDLE: if (go) nextState = S1;
    S1: nextState = S2;
    S2: nextState = S3;
    S3: nextState = S5;
    S5: nextState = S6;
    S6: nextState = IDLE;
    default: nextState = IDLE;
  endcase
end

always_ff @(posedge clk or negedge rst)
  currState <= rst ? nextState : IDLE;
```

Example 14 Adding a New Compiler Directive in VHDL

```
  attribute FSM_COMPLETE : BOOLEAN;
  attribute FSM_COMPLETE of state : signal is TRUE;

 process (state)
```

Feedback

```
    begin
         case state is
             when S0 => next_state <= S1; outp <= 0;
             when S1 => next_state <= S2; outp <= 1;
             when S2 => next_state <= S3; outp <= 2;
             when S3 => next_state <= S4; outp <= 3;
             when S4 => next_state <= S0; outp <= 4;
             when others => next_state <= S0; outp <= 5;
         end case;
     end process;

  process (CLK, RST)
  begin
    if RST = '0' then
      state <= s0;
    elsif ( CLK'event and CLK = '1' )  then
      state <= next_state;
    end if;
  end process;
```

- To define the encoding style in the FSM rule, use the `create_failsafe_fsm_rule` command with the `-encoding_style` option.

  ```
  prompt> create_failsafe_fsm_rule -encoding_style hamming2...
  ```

FSM Rules and Groups

You can specify functional safety intents for FSM using the FSM rules and groups to recover from deadlock due to SEU. FSM rules and groups help you to define a mechanism encoded in the FSM to move to a recovery state. A safe state machine automatically transitions to a safe state after an SEU.

Creating FSM Rules and Groups

To create a new failsafe FSM rule, use the `create_failsafe_fsm_rule` command with the following options:

- `-name`: Specifies the name of failsafe FSM rule to be created.
- `-encoding_style`: Specifies the encoding style for failsafe FSM rule.
- `-update`: Updates the existing failsafe FSM rule.
- `-isolation`: Specifies if registers need to be isolated with TAP isolation.
- `-tap_mapping`: Specifies the list of library cells, out of which the appropriate one is used for instantiating tap cells for the registers of failsafe FSM group.

```
create_failsafe_fsm_rule -name rule1 -encoding_style hamming2
```

Feedback

Use the `set_failsafe_fsm_rule` command to set a failsafe FSM rule on specified registers. If the specified register already has another rule associated with it, the original rule remains applied to it. Also, specified list of registers should be part of existing FSM. You can use the `set_failsafe_fsm_rule` command with the following options:

- `-rule` to specify the failsafe FSM rule to be applied to the registers.
- `-error_signal` to specify the pin or port to connect the comparison of error outputs.
- `-registers` to specify the list of output pins of FSM registers on which failsafe FSM rule needs to be applied.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

The following example sets a rule for the FSM and specifies an error signal for creation:

```
prompt> set_failsafe_fsm_rule -rule rule1 -registers {state[0]/Q
 state[1]/Q \    state[2]/Q} -error_signal {err_port}
```

To create a FSM group for specifying FSM registers along with synchronizer and parity registers, which are added for a failsafe FSM, use the `mark_failsafe_fsm` command with the following options:

- `-rule`: Specifies the failsafe FSM rule based on which failsafe FSM group is created.
- `-name`: Specifies the name of the failsafe FSM group to be created. If not specified, the tool assigns a name to the failsafe FSM group automatically.
- `-registers`: Lists register pins, which are part of FSM group.
- `-parity`: List parity registers or pins, which are part of parity registers for the FSM group.
- `-synchronizer` to list synchronizer registers or pins, which are part of synchronizer registers for the FSM group.
- `-tap`: Specifies the collection of tap cells for the registers in this group.
- `-update`: Updates existing failsafe FSM group.
- `-error_signal`: Specifies pin or port to connect the comparison of error outputs.
- `-correction_signal`: Specifies the pin or port in the design to which the correction outputs are connected.
- `-requirement_id`: Specifies the requirement ID string for tracking in the requirement management system.

Feedback

The following example creates a failsafe FSM group with the `-synchronizer` and `-error_signal` options:

```
prompt> mark_failsafe_fsm -rule rule2 -registers {flop1/Q flop2/Q
 flop3/Q} \    -parity {flop4/Q} -synchronizer {flop5/Q} -error_signal
 {err_port} \
-name group2
```

Querying FSM Rules and Groups

To get all the failsafe FSM rules of the current design, use the `get_failsafe_fsm_rules` command with the following options:

- `-of_objects`: Gets failsafe FSM rules associated with the specified registers.
- `-filter`: Filters failsafe FSM rules with the specified expression.
- `-quiet`: Does not print messages.
- `-regexp`: Displays patterns with full regular expressions.
- `-exact`: Considers wildcards as plain characters.
- `-nocase`: Performs case-insensitive matching.
- `-expect`: Expects exactly the specified count of matching objects: count greater than or equal to 0.
- `-expect_at_least`: Expects at least the specified count of matching objects: count greater than or equal to 1.
- `-expect_each_pattern_matches`: Expects each pattern must match at least one object.
- `-pattern`: Finds failsafe FSM rules with names matching the specified pattern.

To print a report of the specified failsafe FSM rules, use the `report_failsafe_fsm_rules`command with the following option:

- `-objects`: Specifies the failsafe FSM rule names to be reported.

You can use the `get_failsafe_fsm_groups` command to list all failsafe FSM groups defined in the current design with the following options:

- `-of_objects`: Gets failsafe FSM groups associated with specified failsafe FSM rules.
- `-filter`: Filters failsafe FSM groups with the specified expression.
- `-quiet`: Does not print messages.
- `-regexp`: Displays patterns with full regular expressions.

Feedback

- `-exact`: Considers wildcards as plain characters.
- `-nocase`: Performs case-insensitive matching.
- `-expect`: Expects exactly the specified count of matching objects: count greater than or equal to 0.
- `-expect_at_least`: Expects at least the specified count of matching objects: count greater than or equal to 1.
- `-expect_each_pattern_matches`: Expects each pattern must match at least one object.
- `-pattern`: Finds failsafe FSM groups with names matching the specified pattern.

To print a report of the specified failsafe FSM groups, use the `report_failsafe_fsm_groups`command with the following option:

- `-objects`: Specifies the failsafe FSM group names to be reported.

Removing FSM Rules and Groups

To remove the specified failsafe FSM rules from the current design, use the `remove_failsafe_fsm_rules` command. If the specified rules are referenced by any failsafe FSM group, the command issues an appropriate warning message. You can use the `remove_failsafe_fsm_rules` command with the following options:

- `-objects`: Specifies the failsafe FSM rule names, which need to be removed.
- `-all`: Removes all failsafe FSM rules defined in the current design.
- `-from`: Dissociates the failsafe FSM rules from the specified registers represented by output pins. It deletes the rule association with FSM.

To remove the specified failsafe FSM groups from the current design, use the `remove_failsafe_fsm_groups` command with the following options:

- `-objects`: Specifies the names of the failsafe FSM groups, which need to be removed.

Feedback

Manual Failsafe FSM Flow

In the Design Compiler NXT and the Fusion Compiler tools, you can specify functional safety intents for FSM by manually creating and marking failsafe FSMs in the design using one of the following flows:

- RTL based flow: To use RTL based flow for creating and marking failsafe FSMs in the design, perform the following steps:

  1. Use the `analyze` command to analyze the specified HDL source files and store the resulting templates into the specified library in a format ready to specialize and elaborate to form linkable cells of a full design.

     ```
     prompt> analyze top
     ```

  2. Use the `elaborate` command to elaborate the top-level module.

     ```
     prompt> elaborate top
     ```

  3. Use the `set_top_module` command to set the specified module to be the top-level design, link the entire design, and create a single block to be used for the remainder of the synthesis flow.

     ```
     prompt> set_top_module top
     ```

  4. Use the `create_failsafe_fsm_rule` command to create failsafe FSM rules. As failsafe FSM must be implemented at RTL level, specify synchronizer and parity registers in the RTL with the `mark_failsafe_fsm` command to create and mark failsafe FSMs. This command creates one FSM group per FSM in the design, irrespective of whether it is of type failsafe or not. For example, the FSM corresponding to `R1`, `R2`, `R3` is associated with `rule1`. If the registers are already associated with the failsafe FSM rules, use the `load_ssf` command to read the specified failsafe FSM rules.

     ```
     prompt> load_ssf top.ssf
     The top.ssf file contains the following information:
     ssf_version 1.0
     create_failsafe_fsm_rule -name rule1 -encoding_style hamming2
     mark_failsafe_fsm -name group1 -registers {FF1 FF2 FF3 FF4} -parity
      {FF_parity}
     ```

  5. Use the `compile_fusion` command to run the flow of synthesis, placement, and optimization.

     ```
     prompt> compile_fusion
     ```

Feedback

- Netlist flow: To use netlist flow for creating and marking failsafe FSMs in the design, perform the following steps:

  1. Run the `report_rtl` command to verify if a design already has FSMs. Netlist creation using the `read_verilog` command is not supported for this flow as this happens only when running `analyze` and `elaborate` commands.
  2. Instantiate the synchronizer and parity registers manually using the `create_cell` command and perform the encoding and connections in the netlist.
  3. Use the `mark_failsafe_fsm` command to mark a FSM as a failsafe FSM.

Note:

For both RTL based and netlist flows, ensure that failsafe FSM has already been synthesized.

Reporting FSM Information

To report any FSM related information, use the `report_fsm` command. The command reports the current state of vectors and encoding of FSM.

After re encoding, the `report_fsm` command report shows the following information related to the FSM during compile:

- The style and bit length of the encoding used
- New registers added to the state of the vector
- New state encodings

Note:

In the Design Compiler NXT tool, the `report_fsm` command does not display information for clock and asynchronous reset signals.

Example 15 and Example 16 show the reports generated by the `report_fsm` command before and after synthesis.

Example 15 FSM Information Reported by report_fsm Command Before Synthesis

```
****************************************
Report : FSM
Design : top
Version: S-2021.06-DEV
Date   : Tue Apr 20 12:07:31 2021
****************************************

Design : fsm_hamming2
Filename : /test_v.v - 26
Clock             : FSM1/clk
```

```
Asynchronous Reset: FSM1/rst_e


Current Encoding (length/style)     : 2 / binary
State Vector: { current_state_e_reg[1] current_state_e_reg[0] }


FSM_COMPLETE : FALSE
Re-Encoding style     : NONE


State Encodings and Order:

S0          : 00
S1          : 01
S2          : 10
S3          : 11
Default Next State : S0
```

When you use the `report_fsm` command post-synthesis, the command displays the following information:

Example 16 FSM Information Reported by report_fsm Command Post-Synthesis

```
****************************************
Report : FSM
Design : top
Version: S-2021.06-DEV
Date   : Tue Apr 20 12:07:41 2021
****************************************

Design : fsm_hamming2
Filename : /test_v.v - 26
Clock             : FSM1/clk
Asynchronous Reset: FSM1/rst_e


Current Encoding (length/style)     : 3 / hamming2
State Vector: { current_state_e_reg[1] current_state_e_reg[0]
 parity_reg_hamming2_current_state_e }
Synchronizer register: FSM1_synchronizer_hamming2


FSM_COMPLETE : TRUE
Re-Encoding style     : hamming2


State Encodings and Order:                (Original Encoding) :

S0          : 000                           (00)
S1          : 011                           (01)
S2          : 101                           (10)
S3          : 110                           (11)
```

Feedback

```
Default Next State : S0

Tool inferred States (Mapping : State encoding)
Default:
Tool Inferred Error: 100 010 001 111
Design : fsm_hamming2
Filename : /test_v.v - 35
Clock              : FSM1/clk
Asynchronous Reset: FSM1/rst_t
```

During synthesis, an FSM can be transformed into a failsafe FSM. Optional parity and synchronization registers, and optional error detection or correction logic are added, next to the original state registers,depending on its encoding style:

- `fsm_complete`: FSM has no undefined states, and `dont_care` optimization is prevented by restrictions on the state registers and next state combinational logic.
- hamming2: The state vector is re encoded by adding a parity bit as least significant bit, and combinational error detection or correction logic is added. You can add an optional synchronizer register to the error signal to reduce metastability of the state registers.
- hamming3: The state vector of `d` bits are re encoded adding `k` parity bits (registers) as least significant bits such that `2^k≥d+k+1`. Combinational error detection or correction is added. No synchronizer register is added.

You can use the `-failsafe_fsm_rules` and `-failsafe_fsm_groups` options with the `report_safety_status` command to report the specific failsafe FSM rules and groups with respect to the specified block.

Options of the report_fsm Command

You can use the following options of the `report_fsm` command at different stages of synthesis:

- `-all`: Reports all the available FSM modules in the design. See Example 17 and Example 18 for the `report_fsm -all` command reports before and after synthesis.

Example 17 The report_fsm -all Command Report Before Synthesis

```
dc_shell> report_fsm -all

****************************************
Report : Failsafe FSM
Version: Q-2019.12-SP1
Date   : Wed Feb 26 10:07:19 2020
****************************************
mid1:f1  binary  state[2:0]       fc, h2
bot1:f1  binary  myState[3:0]     fc, h3
bot2:f1  binary  state1[2:0]      notfusa
```

Feedback

Where,

- `fc` represents the `fsm_complete` pragma
- `h2` represents hamming2 re-encoding
- `h3` represents hamming3 re-encoding
- `notfusa` represents not completely enumerated and without the `fsm_complete` pragma

Example 18 The report_fsm -all Command Post-Synthesis

```
**************************************
Report : Failsafe FSM
Version: Q-2019.12-SP1
Date   : Wed Feb 26 10:07:19 2020
**************************************
mid1:f1  hamming2  state[2:0]        fc, h2
bot1:f1  hamming3  myState[3:0]      fc, h3
bot2:f1  binary  state1[2:0]         notfusa
```

- `-verbose`: Reports additional information for the following encodings post synthesis:
  - For `fsm_complete` and hamming2 re-encoding, see Example 19.

Example 19 Using the report_fsm -verbose Command for fsm_complete and hamming2 re-encoding

```
**************************************
Report : FSM
Design : top
Version: S-2021.06-DEV
Date   : Tue Apr 20 12:07:41 2021
**************************************

Design : fsm_hamming2
Filename : /test_v.v - 26
Clock             : FSM1/clk
Asynchronous Reset: FSM1/rst_e
Design : fsm_hamming2
Filename : /test_v.v - 35
Clock             : FSM1/clk
Asynchronous Reset: FSM1/rst_t

Current Encoding (length/style)     : 4 / hamming2
State Vector: { current_state_t_reg[2] current_state_t_reg[1]
 current_state_t_reg[0] parity_reg_hamming2_current_state_t }
Synchronizer register: FSM1_synchronizer_hamming2


FSM_COMPLETE : TRUE
```

```
Re-Encoding style     : hamming2


State Encodings and Order:                (Original Encoding) :

T0          : 0000                          (000)
T1          : 0011                          (001)
T2          : 0101                          (010)
T3          : 0110                          (011)
T4          : 1001                          (100)
T5          : 1010                          (101)
T6          : 1100                          (110)
T7          : 1111                          (111)
Default Next State : T0

Tool inferred States (Mapping : State encoding)
Default:
Tool Inferred Error: 1000 0100 0010 1110 0001 1101 1011 0111
```

- For `fsm_complete` and hamming3 re-encoding, see Example 20.

Example 20 Using the report_fsm -verbose Command for fsm_complete and hamming3 re-encoding

```
****************************************
Report : FSM
Design : top
Version: S-2021.06-DEV
Date   : Tue Apr 20 12:07:41 2021
****************************************

Design : fsm_hamming3
Filename : /test_v.v - 128
Clock             : FSM2/clk
Asynchronous Reset: FSM2/rst_e


Current Encoding (length/style)      : 5 / hamming3
State Vector: { current_state_e_reg[1] current_state_e_reg[0]
 parity_reg[2]_current_state_e parity_reg[1]_current_state_e
 parity_reg[0]_current_state_e }


FSM_COMPLETE : TRUE
Re-Encoding style     : hamming3


State Encodings and Order:                (Original Encoding) :

S0          : 00000                         (00)
S1          : 01101                         (01)
S2          : 10110                         (10)
S3          : 11011                         (11)
```

Feedback

```
Default Next State : S0

Tool inferred States (Mapping : State encoding)
Default:
Tool Inferred Error: 10000 01000 11000 00100 10100 01100 11100 00010
 10010 01010 11010 00110 01110 11110 00001 10001 01001 11001 00101
 10101 11101 00011 10011 01011 00111 10111 01111 11111
```

- `-show_error_states`: Shows that the following new states are inserted by the tool for hamming2 or hamming3 re-encoding:

```
State Encodings and Order (Original Encoding)(Error Encoding):
S0          : 000000  (000)           (000001, 000010, …)
S1          : 001011  (001)      (000001, 000010, …)
S2          : 010101  (010)      (000001, 000010, …)
```

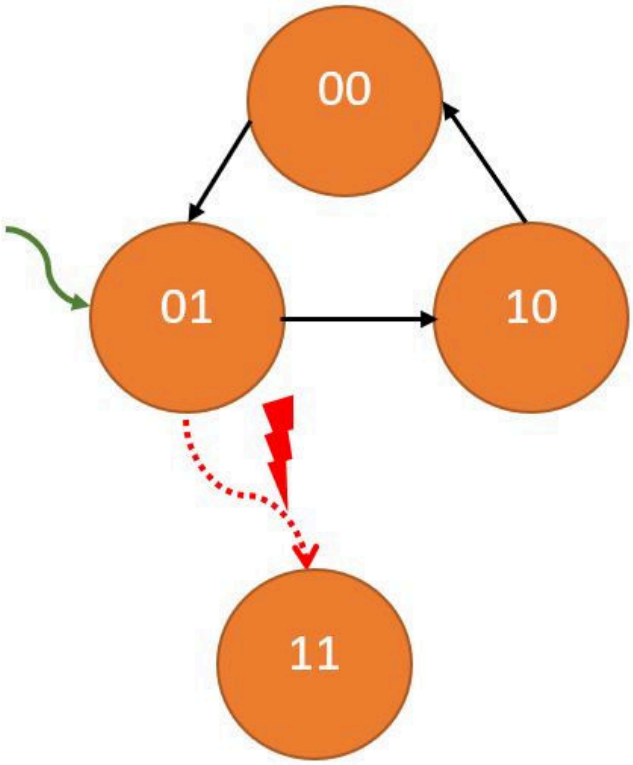
Visualizing SEU Handling

This section describes the behaviour of Failsafe FSM when an SEU occurs with different encodings.

- Figure 37
- Figure 38
- Figure 39
- Figure 40

Feedback

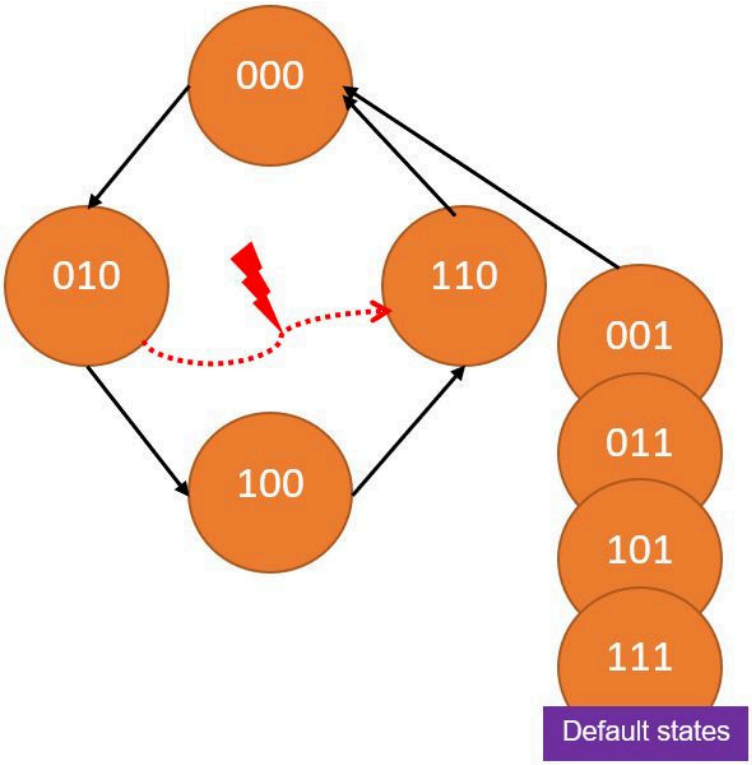
Figure 37 FSM is Incompletely Specified



In Figure 37, the SEU event might cause unpredictable behavior including a locked up machine if the FSM is incompletely specified. For example, the FSM might be bumped from the FSM state `01` to `11` by the external event (SEU) that leads to a machine getting locked and remaining in the state `11`.

Feedback

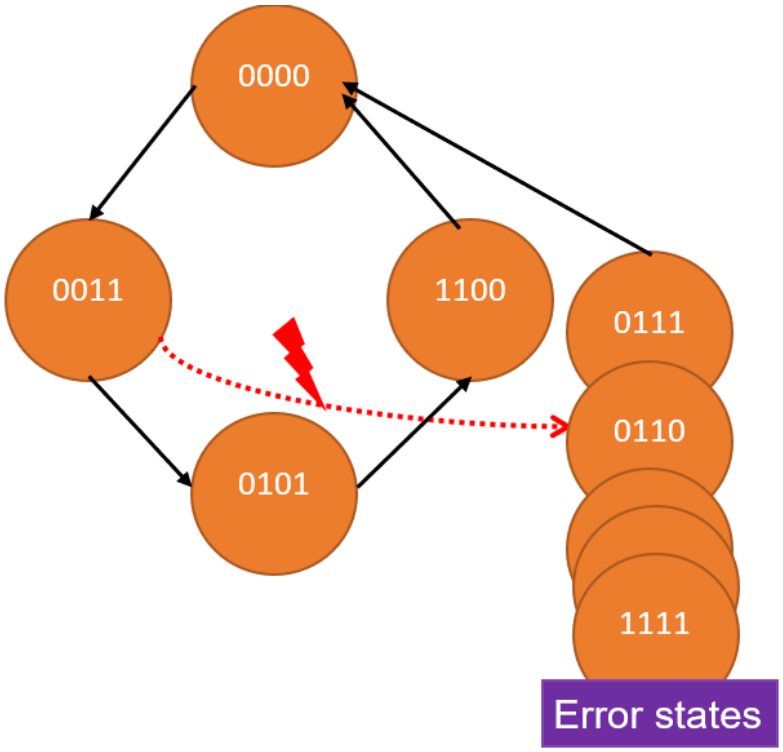
Figure 38 FSM is Completely Specified in Binary Encoding



You can use the `fsm_complete` pragma with a binary encoded state machine. If the state machine is fully specified but does not have functional safety encoding, an SEU might take the state machines to a different state but a valid state. In this case, the state machine continues to function, but can have an unexpected behavior resulting in a hamming distance of `1` for transitions.

Feedback

Figure 39 Hamming2 Re-Encoding

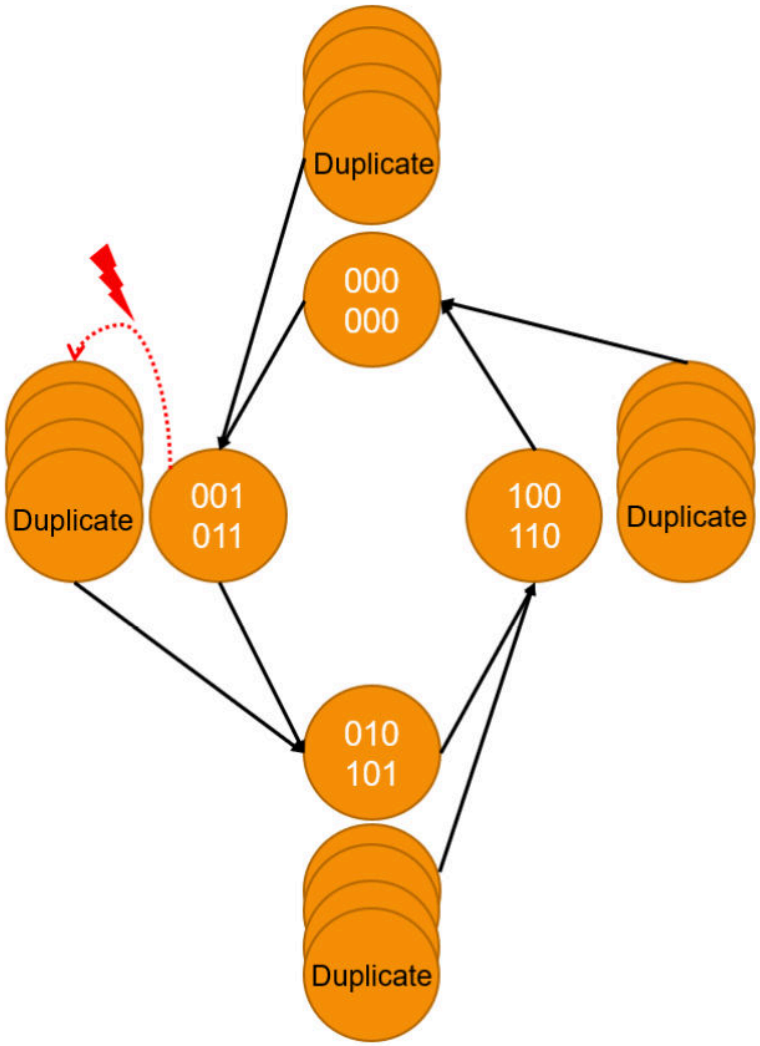


If the state machine is re-encoded into a hamming2 encoding, an SEU always takes the state machine to an error state.

For example, the FSM is bumped from state `0011` to `0111` by an SEU, so the FSM detects the error state, flags the error, and returns the FSM to the reset state, as shown in Figure 39.

Feedback

Figure 40 Hamming3 Re-Encoding



If the FSM is re-encoded into a hamming distance 3 encoding, an SEU sends the FSM into a duplicate of the current state (error state) and the FSM recovers and functions normally.

For example, the FSM is bumped from state `001011` to `001111` by an SEU, and the next state logic for `001111` is a duplicate of the next state logic for `001011`. Therefore, the FSM proceeds to the correct next state `010101`, as shown in the Figure 40.

Using Formality to Verify a Re-encoded Failsafe FSM

During synthesis, the Design Compiler NXT and the Fusion Compiler tools generate the `guide_safety_fsm` guidance command in the SVF file. The `guide_safety_fsm` command is generated in FSM re-encodings during synthesis.

During preverification, the Formality tool reads the SVF file and generates the parity and error checking registers and logic for the failsafe FSM hamming 2 and hamming3 encoding types. The hamming2 encoding type uses the recovery state. The recovery state is applied to the next state of the state vector when the tool detects a parity error.

Note:

The hamming3 encoding type has error recovery built into it. Therefore, a recovery state is not required.

8

Inserting Tap Cells

You can specify the `-isolation` option for various types of safety intent such as safety registers, safety cores, failsafe finite state machine, and safety error codes. The legalizer and placer reserve space to place well-tap throughout the implementation flow, but the insertion takes place after `route_opt`, for example, during chip-finishing.

The `create_safety_tap_cells` command adds tap cells for all functional safety related groups with a rule that prescribes isolation, following optional `tap_mapping` instructions, or any well-tap in the library.

The `create_safety_tap_cells` command can operate on the following specified groups by using options:

- Safety register groups: `-safety_register_groups`
- Failsafe FSM groups: `-failsafe_fsm_groups`
- Safety error code groups: `-safety_error_code_groups`
- Safety core groups: `-safety_core_groups`

You can limit the available well-tap modules by using the `-tap_lib_cells` option. This is additional filtering on top of optional filtering provided by the group's rule.

The tap cells are inserted into the gaps reserved for taps by the legalizer, and no other cells are moved. The inserted tap cells are marked on the group they were inserted for.

9

Inserting Redundant Trees

You can use the `insert_redundant_trees` command to reduce the common path in shared signals by adding redundant trees. To reduce the chances of SEU affecting both registers simultaneously, the `insert_redundant_trees` command places the guide buffers close to the common driver to ensure that each guide buffer drives at most one register of this safety register group (or safety core of a safety core group). The `insert_redundant_trees` command always operates within the scope of a single physical block.

The `insert_redundant_trees` command supports the following options:

- `-safety_register_groups` or `-safety_core_groups`: Specifies the group objects either provided by the safety register groups using the `-safety_register_groups` option or safety core groups using the `-safety_core_groups` option. Either the `-safety_register_groups` or the `-safety_core_groups` option must be specified.
- `-buffer_lib_cell`: Specifies the new buffers, introduced by the command, instantiated from the provided library cells.
- `-inverter_lib_cell`: Specifies the new inverters, introduced by the command, instantiated from the provided library cells. Inserting the inverters is optional. If no inverter library cell is specified, but an inverter needs to be inserted for polarity correctness, the command issues an error message.
- `-pins`: Specifies the pins of the safety core group or safety register group.
- `-pin_types`: Specifies the type of the pins such as clock, reset or scan
- `-suppress_initial_buffering`: Suppresses the first-time behavior of the `insert_redundant_trees` command

Figure 41 Inserting Redundant Tree to Reduce SEU for Common Paths

Feedback

The `insert_redundant_trees` command automatically applies `dont_touch` restriction reason `safety_redundancy` on the net between each identified driver and guide. The net is also identified as a high-net weight. Based on the availability of the location of the driver, the inserted guides are placed at their associated driver's location.

- If the driver of a guide has no assigned location, the guide is placed near its direct loads.
- If the location of the driver cannot be established, the guide is placed at coordinate (0, 0).

Splitting Trees

Each group object has a set of redundant pins that require splitting. The pins operated on by inserting redundant trees are selected per safety register group as follows:

- If neither `-pin_types` nor `-pins` option is specified, the `insert_redundant_trees` command selects the safety register group register's pins that match any of the associated safety register rule's split pin types. For safety cores groups, the split pins specified using the `set_safety_core_rule -split_pins` command are selected.
- If the `-pin_types` option is specified, the `insert_redundant_trees` command uses those pin types for selection instead.
- If the `-pins` option is specified, the `insert_redundant_trees` command computes the specified names of the pins on the registers of the safety register group. For safety core groups, the listed hierarchical pins on the safety core's hierarchy are selected.

  The `insert_redundant_trees` command automatically extends the pins select on one register or core to include similar pins on the other registers or cores of their group.

Note:

If either `-pin_types` or `-pins` option is specified, the pins operated on are different from the split pins, identified by the `set_safety_code_rule` or `safety_register_rule` command or marked on the safety register group or the safety core group.

For each of the selected pins of one redundancy group (safety register group or safety core group), the `insert_redundant_trees` command processes a set of peer pins (those with equal base name) one at a time before moving on to the next set of peer pins. Such a set of pins is called as a redundant pin set.

The `insert_redundant_trees` command operates on one redundant pin set (set of peer split pins) at a time, performing the transformation as shown in Figure 41.

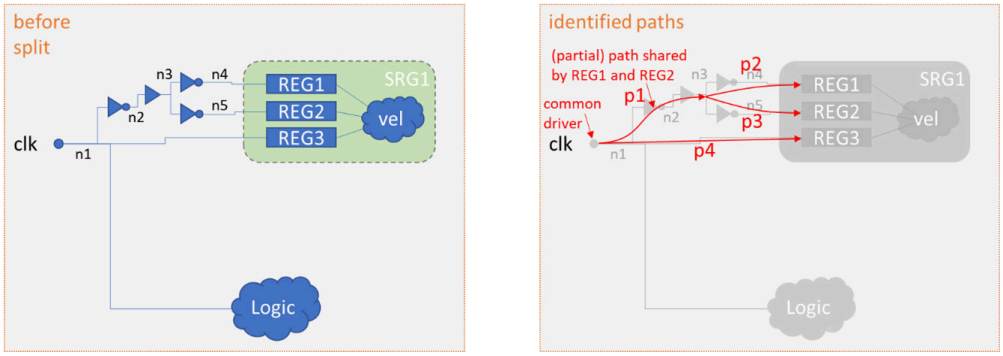
Feedback

The command first identifies driver and paths to the split pins of the safety register groups and safety core groups (see Driver Collection and Path), and then inserts guide buffers to reduce common path length (see Inserting Guide Buffer).

Driver Collection and Path

The driver collection, which is a set of common drivers, is established through backward tracing starting at the split pins. The driver collection drives the split pins using paths containing repeaters only, starting at the split pins of all the groups that are being processed, as shown in Figure 42. The tool performs the path tracing through repeaters but stops at the integrated clock gates (ICG) and at frozen hierarchical ports. The nets marked as `dont_touch` do not stop the tracing.

Figure 42 Driver Identification and Path Analysis



Inserting Guide Buffer

The `insert_redundant_trees` command introduces the guide buffers marked with the `safety_redundancy_size_only` option and ensures that each guide buffer drives at most one split pin. Optimization reduces the common net between the common drivers and their guide buffers using a high net weight.

If the `insert_redundant_trees` command is run on the driver for the first time, it does not drive any previously inserted guide buffers. You can insert a guide buffer between the driver output and its current net. The latter net is given a high net weight to instruct optimization to physically shorten the common net. The `insert_redundant_trees -suppress_initial_buffering` command suppresses the first-time behavior.

If the `insert_redundant_trees` command needs to add a guide buffer behind a driver with an output net protected by the `dont_touch` attribute, other than `safety_dont_touch` attribute, the command issues an error message because the net cannot be modified.

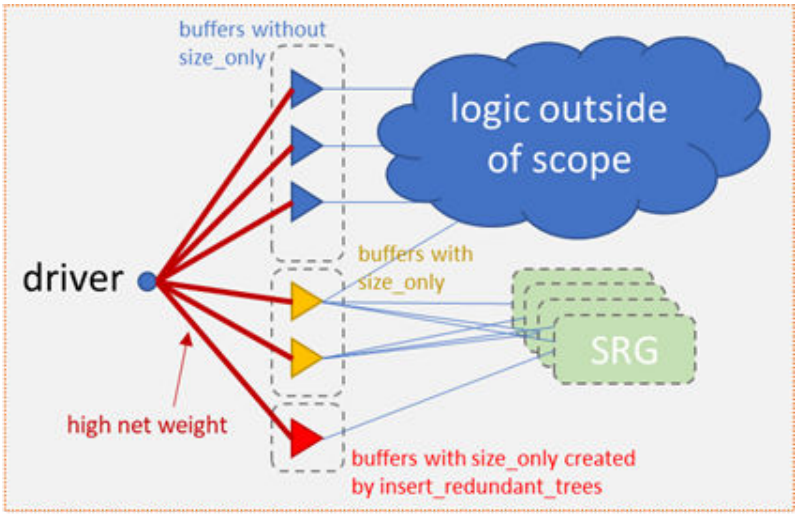
Feedback

You can introduce the nets with the `safety_dont_touch` attribute by running the boundary protection using the attribute on the boundary nets of safety cores. You can insert the guide buffers, and the `safety_dont_touch` attribute is transferred to the driver net.

You can run the `insert_redundant_trees` command multiple times. It collects directly driven guide buffers per driver, as shown in Figure 43. They might be inserted during a previous run of the command or inserted for a previously processed redundant pin set.

The guide buffers are marked with `safety_size_only` attribute and tagged by the `redundancy_tree_guide` attribute to distinguish them. To ensure that the driver has a unique guide buffer for all split pins belonging to one group, the command creates and connects new guide buffers in the driver's hierarchy (considered as a new load for the driver). The guide buffer is an instance of the library cell provided using the `-buffer_lib_cell` option of the `insert_redundant_trees` command.

Figure 43 Drivers Showing Direct Loads After Tree Splitting



Removing Common Path

For each split pin in the redundant pin set, the path between the driver and the split pin should start with a unique guide buffer for the split pin. If no such buffer exists, it is added.

Feedback

Figure 44 Tree Splitting for Single Driver and Multiple Groups

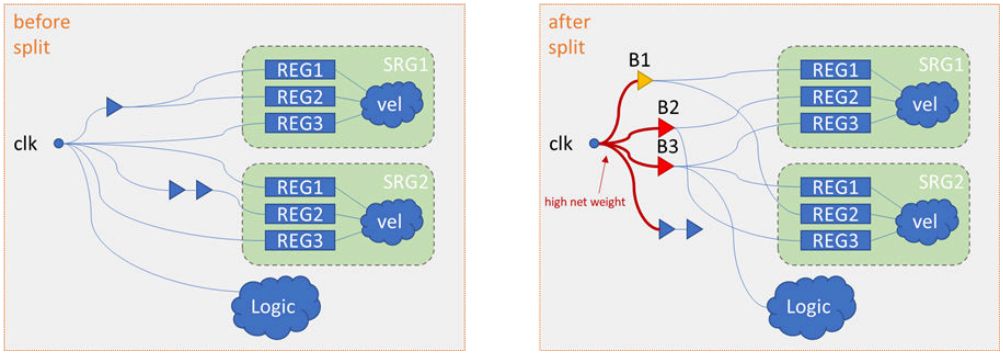
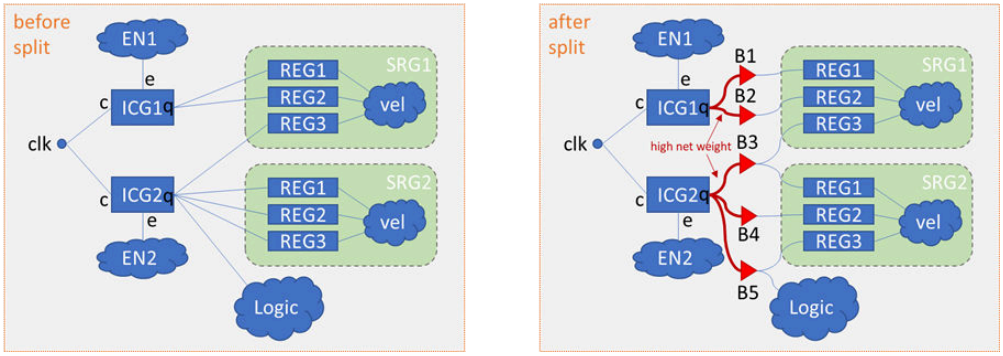


Figure 45 Tree Splitting for Clock With Integrated Clock Gates



10

Redundant Via Insertion

Describes Redundant Via Insertion and its flows

Redundant via insertion is an important design-for-manufacturing (DFM) feature that is supported by Zroute throughout the routing flow. The design target for redundant via insertion should either be 90 percent or close to 100 percent to have a reliable design. The goal is to achieve 100 percent redundant via insertion by inserting double-cut (1x2 or 2x1) vias on selected via layers.

The flows for the redundant via insertion are:

- Traditional Redundant Via Insertion Flow
- 100 Percent Redundant Via Insertion Flow

Traditional Redundant Via Insertion Flow

Traditional redundant via flows replace single-cut vias with redundant vias by using the `add_redundant_vias` postroute command. The traditional redundant via flow takes the DRC convergence as the highest priority. You can replace the single-cut vias with double-cut vias or design-for-manufacturing (DFM) vias after the DRC of signal routing has converged. To maintain similar routing DRCs, use the `add_redundant_vias` postroute command to convert the maximum possible single-cut vias to redundant vias.

100 Percent Redundant Via Insertion Flow

The 100 percent redundant via insertion flow is different from the traditional redundant via insertion flow. You should not use traditional redundant via insertion options in the 100 percent redundant via insertion flow.

To enable 100 percent redundant via insertion, you must prepare a technology file with only double-cut contact codes as default.

You can insert double-cut vias by constructing double-cut vias and driving Zroute through only the selected double-cut vias to meet the 100 percent target. Insert the double-cut vias during the global routing, track assignment, and detail routing phases to force the router to

consider double-cut vias all the time. If you insert only single-cut vias, there might not be enough resources for the router to swap them to double-cut vias later. Double-cut vias are larger in size and need more spacing.

The 100 percent redundant via insertion flow requires the following changes:

1. Use the technology file containing only the double-cut vias as the default contact code.
2. Enable the following Zroute option to improve the routing quality of results (QoR).

```
set_app_options -name route.common.treat_via_array_as_big_via \
-value true
```

Set the technology file with double-cut vias and the routing option at the beginning of the place and route flow. All the signal route nets, including clock nets, use double-cut vias as default contact codes.

Note:

To use these commands to insert double-cut (1x2 or 2x1) vias on selected via layers, you must have the IC Compiler II or Fusion Compiler add-on licenses. Redundant via insertion is only applicable to IC Compiler II and Fusion Compiler.

Updating the Technology File

The traditional routing flow defines the single-cut vias as the default contact code. For 100 percent redundant via insertion flow,

1. Update the technology file to include only double-cut vias as the default contact code by setting the `isDefault` attribute to `1`, as shown in Example 21. For the single-cut vias with contact code, set the `isDefault` attribute to `0`.
2. Use the updated technology file for the complete place and route flow.

Example 21 Setting the Double-Cut Vias as the Default Contact Code

```
ContactCode "via12_array" {
     ….
        minNumCols              = 1
        minNumRows             = 2
        isDefaultContact        = 1
     …..
    }
    ContactCode "via12" {
     ….
        minNumCols              = 1
        minNumRows             = 1
        isDefaultContact        = 0
     …..
    }
```

Feedback

Improving Redundant Via Insertion Rate

For each via layer, create only one double-cut via array that includes

- Two small-sized square via cuts
- The default metal width enclosure

The double-cut via array should not trigger any design rule checks (DRC). If the double-cut via array is not correctly set to the default contact code, the tool might generate single-cut vias and report them as the `need fat contact` violations. Zroute might change the double-cut via array to insert single-cut vias because design rule check convergence has higher priority.

For information about improving design rule check convergence, see Improving Design Rule Check Convergence.

Adding minCut for the Via Layer

The `minCuts` parameter is used to specify the minimum number of cuts for a particular layer. For example, setting `minCut` as 2 for a via layer defines a double-cut via array for that layer. Example 22 shows how to add `minCuts` for the via layer in the layer section.

Example 22 Adding minCut for the Via Layer in the Technology File

```
Layer "XX" {
     ...
        cutTblSize              = 3
        cutNameTbl            = (Vsm, Vv, Vh)
        minCutsTbl              = (2, *, xxx, -1, -1, -1,
                                   2, *, xxx, -1, -1, -1,
                                   2, *, xxx, -1, -1, -1,
...
     }
```

Improving Design Rule Check Convergence

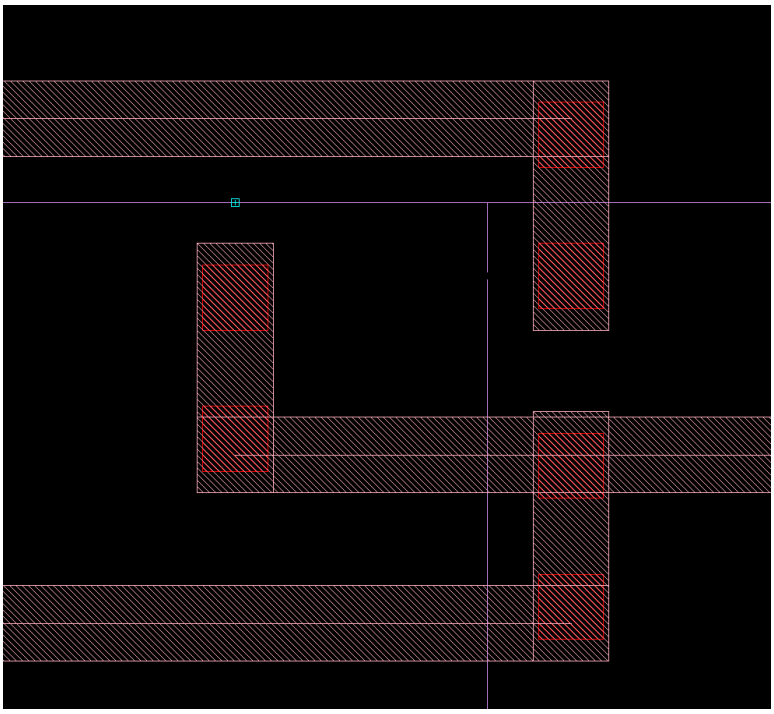
To improve design rule check convergence, enable the following application option:

```
set_app_options -name route.common.treat_via_array_as_big_via \
 -value true
```

The default is `false`. If you enable the `route.common.treat_via_array_as_big_via` application option, the global router uses a shifted double-cut via model to compute the required routing resource for double-cut vias. To save the routing resources or to fix DRC, track assignment and detail routing shifts the double-cut vias to the positive or negative side to form the L-shape. Figure 46 shows the results after detail routing of the double-cut via, and the detail routing wire forms the L-shape with the `route.common.treat_via_array_as_big_via` application option.

Feedback

Figure 46 Double-Cut Via and Detail Routing Wire Forms L-Shape



To disable all the redundant via insertion options, use the following application option:

```
set_app_option -list {route.common.concurrent_redundant_via_mode off}
```

Example 23 shows the double-cut vias from the via array report. Use this report to check the results of the 100 percent redundant via insertion flow, and make sure not to use the output of the Zroute report to check the results of the 100 percent redundant via insertion functionality.

Example 23 Report of Double-Cut Vias

```
Via    VIA56_1x2 :      50843
Via    VIA56_2x1 :        563
Via    VIA45_1x2 :     132125
Via    VIA45_2x  :       3382
Via    VIA34_1x2 :     238729
Via    VIA34_2x1 :      15928
```

A

Dual Core LockStep Flow

Note:

Repelling Group Bound Setup section is deprecated and is replaced by the Safety Core Groups section.

The DCLS flow includes the following steps, as shown in Figure 47:

- Repelling Group Bound Setup

  Perform the DCLS repelling group bound setup before placement optimization. You need to create repelling group bounds to enable physical separation between cores. Placement and legalization honor repelling distance requirements throughout the DCLS flow.

- Routing Blockage Setup

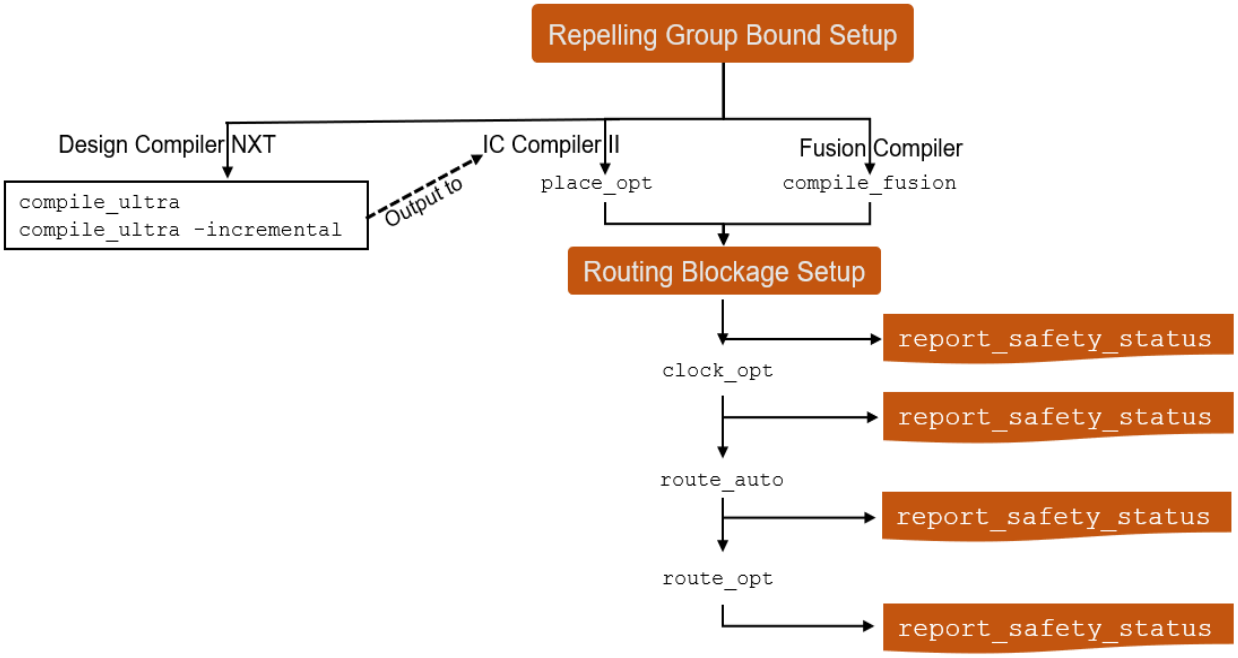
  Perform the DCLS routing blockage setup before clock optimization. Blockages ensure that nets from each core are separated by the repelling distance defined in the repelling group bounds, and blockages are automatically created throughout the flow as needed by the tool.

- Reporting DCLS Safety Status and Violations

  Use the `report_safety_status` command to report violations related to safety registers and DCLS. The command checks the safety status and reports violations on any placement or routing separation and on all repelling bounds, as placement is legalized with the IC Compiler II tool.

Feedback

Figure 47 Steps in the DCLS Flow



Perform the following steps, as shown in Figure 47:

- In the Design Compiler NXT tool (note that the Design Compiler NXT DCLS flow improves correlation with the IC Compiler II tool), perform the DCLS repelling group bound setup before compile.
- In the IC Compiler tool,
  - Create the repelling group bounds to enable physical separation between cores logical hierarchies.
    - The repelling group bounds are created to enable physical separation between core logical hierarchies.
    - The repelling distance requirements are honored during placement in the IC Compiler II tool.
  - Perform the remaining steps in the DCLS flow are performed by the IC Compiler II tool, as shown in Figure 47.

Defining Cell and Net Types

The following cell and net types are defined in the DCLS flow, as shown in Figure 48.

Feedback

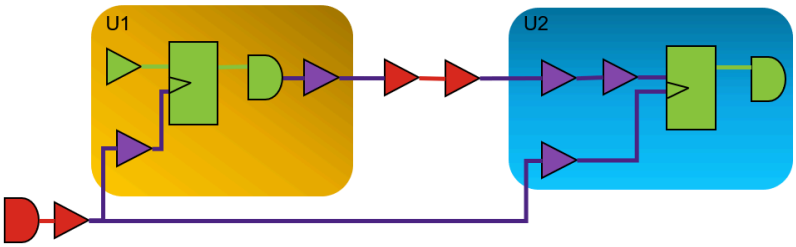
Cells are defined during Repelling Group Bound Setup as follows:

- External cells (red): Cells not belonging to a core.
- Transit cells (purple): Repeaters belonging to a core connecting directly or through another repeater to a cell not belonging to that core.
- Internal cells (green): Cells belonging to a core and not being transit cells.

Nets are defined during Routing Blockage Setup as follows:

- External nets (red): Nets that are not connected to cells that belong to a core.
- Transit nets (purple): Any other nets except external nets or internal nets.
- Internal nets (green): Nets connected to internal cells that belong to a core which is not connected to any transit cell.

Figure 48 Cell or Net Separation Between Internal Cells or Nets Belong to Distinct Cores of the Same Repelling Ground Bound



Repelling Group Bound Setup

Note:

Repelling Group Bound Setup section is deprecated and is replaced by the Safety Core Groups section.

Figure show the repelling group bounds setup step in the DCLS flow. To setup repelling group bounds in the DCLS flow,

- Enable advanced legalization to set legalization for repelling group bounds.
- (Optional) Define repelling bound groups with the `create_group` command.
- Define repelling group bounds with the `create_bounds` command.

  Repelling group bounds separation is honored during placement and legalization stages (`place_opt`, `clock_opt`, and `route_opt`). You can use the `report_safety_status` command after the `place_opt` stage to make sure that physical separation is honored as needed.

Feedback

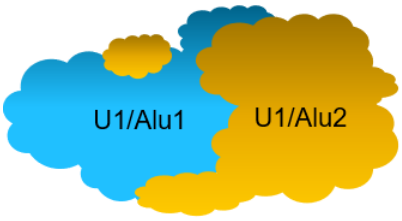
To create repelling group bounds that enable physical separation between cores, use the `create_bounds` command with the following options:

- `-repelling diamond | rect`: Specifies a bound to be repelling rather than attractive to separate cores using the diamond or rectangular cell for cell spacing. A repelling bound specifies a floating area between a pair of objects to keep them apart in the bound.
- `-dimension`: Specifies the physical separation distance in microns between the safety cores in the repelling group bound. For specifying the required separation,
  - Use `-dimension {X Y}` on a rectangular bound.
  - Use `-dimension {R}` on a diamond bound.
- `-groups`: Specifies a list of two safety cores to adhere to the repelling bound separation distance. Safety core is a single hierarchical cell or a repelling bound group with hierarchical cells. The safety cores must be mutually exclusive.

To support coarse placement, create repelling group bounds. In the IC Compiler II tool, create a group of hierarchical or leaf cells to define a single core to use in a repelling group bound setup with the `create_group` command. The single core is treated as a single entity during physical separation. Use the `-repelling` option of the `create_group` command to specify the group as a repelling bound group.

Supporting Coarse Placement

By default, hierarchical modules are placed optimally for wire length, timing, congestion, power, and so on. Placement of hierarchies might be fragmented as shown in Repelling Group Bound Setup, which is not ideal for functional safety.



If safety cores are set up as a repelling group bound as shown in Figure 49, the tool

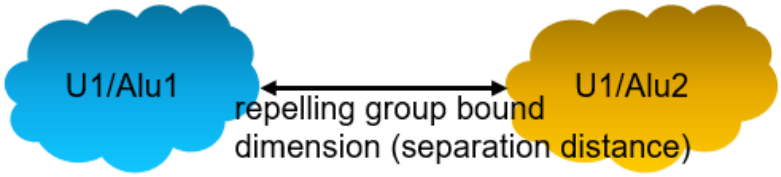
- Performs initial core separation during coarse placement, where the tool allows a few spacing violations that can be removed during the subsequent legalization step (see Legalization Support).
- Avoids U-shapes or other complex shapes of the safety cores.

  This reduces the possibility of internal wire routing crossing from one module to another module.

Feedback

Coarse placement supports separation of the safety cores belonging to the repelling group bounds defined with the `create_bounds` command.

Figure 49 Hierarchies Set Up as a Part of Repelling Group Bound

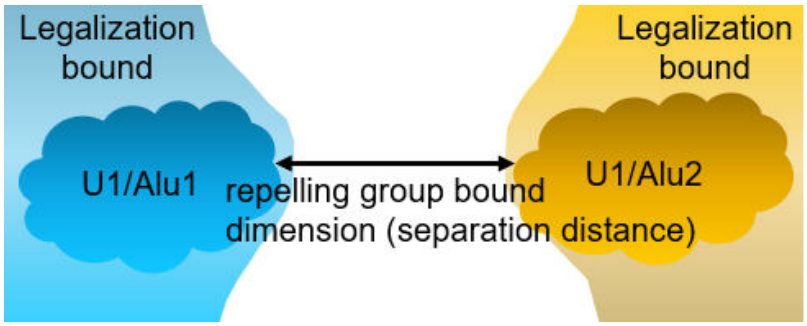


Legalization Support

Legalization enforces separation of safety cores belonging to a repelling group bound. The tool, as shown in the Figure 50,

- Enforces separation by creating legalization bounds
- Automatically calculates legalization bounds for each core of the repelling group bound before performing legalization
- Enforces separation of a safety core's internal cells during legalization but cannot enforce separation on external and transit cells

Figure 50 Separation of Repelling Group Bound With Legalization Bounds



List of Commands to Manage Repelling Group Bounds

This section provides examples that explain how to create repelling group bounds and collections for repelling group bounds and repelling bound groups.

The following example shows how to create a repelling group bound with distinct separation types (rectangular and diamond shape), which is represented in Figure 51.

```
create_bounds -name RGB1 -repelling rect -dimension {10 50} [get_cells
 {U1/Alu1 U1/Alu2}]
```

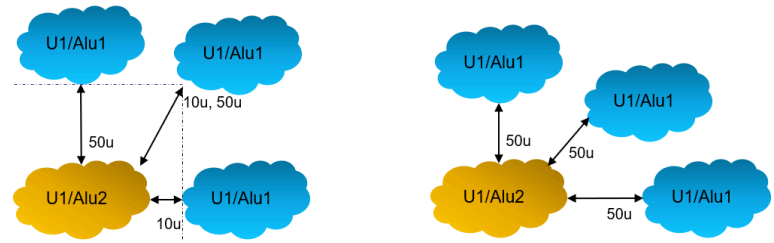
Feedback

```
create_bounds -name RGB1 -repelling diamond -dimension {50} {U1/Alu1
 U1/Alu2}
```

Figure 51 shows the two distinct bounds. For a given placement location of the yellow safety core,

- Three potential placement solutions for the blue safety core are shown for bound separation.
- Coarse placement selects the solution based on the objectives of placement.
- Distances are measured from leaf cell edge to leaf cell edge.

Figure 51 Repelling Group Bound With Rectangle and Diamond Shapes Respectively



Rectangular repelling group bound

Diamond repelling group bound

The following example shows how to create a repelling group bound for two groups, which is represented in Figure 52.

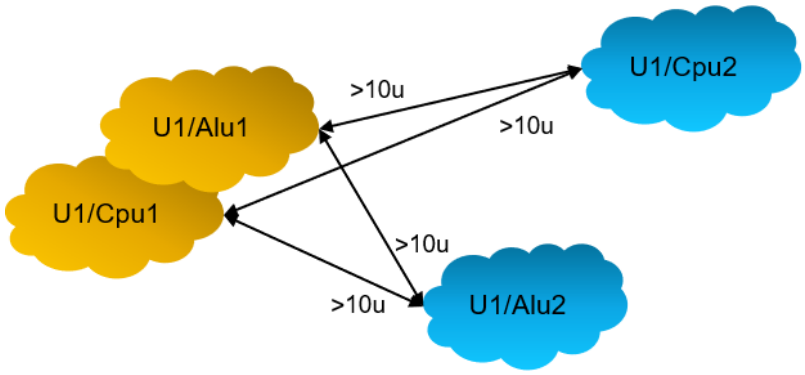
```
create_group -name group_1 -repelling [get_cells {U1/Alu1 U1/Cpu1}]

create_group -name group_2 -repelling [get_cells {U1/Alu2 U1/Cpu2}]

create_bound -name RGB1 -dimension {10} -repelling diamond [get_groups
 {group_1 group_2}]
```

Figure 52 shows the hierarchies of the yellow core and the hierarchies of the blue core.

- Shows a potential placement of the blue core that satisfies the separation required by the bound.
- Coarse placement selects the solution based on the objectives of placement.
- Distances are measured between leaf cells of distinct safety cores

Feedback

Figure 52 Repelling Group Bound With Two Groups



The following examples show how to create various collections for repelling group bounds and repelling bound groups:

- `set RGB [get_bounds -filter is_repelling==true]`: Creates a collection of all repelling group bounds in a design.
- `set RGBCells [get_attribute -objects $RGB -name cells]`: Creates a collection of all cells or groups defined in a repelling group bounds.
- `set RBG [get_groups -filter is_repelling_bound_group==true]`: Creates a collection of all repelling bound groups in a design.
- `set RBGCells [get_attribute -objects $RBG -name objects]`: Creates a collection of all cells or groups defined in a repelling bound groups.
- `set internalCells [get_attribute [get_cells $coreX] -name rgb_core_internal_cells]`: Creates a collection of all internal cells in a design.
- `set transitCells [get_attribute [get_cells $coreX] -name rgb_core_transit_cells]`: Creates a collection of all transit cells in a design.

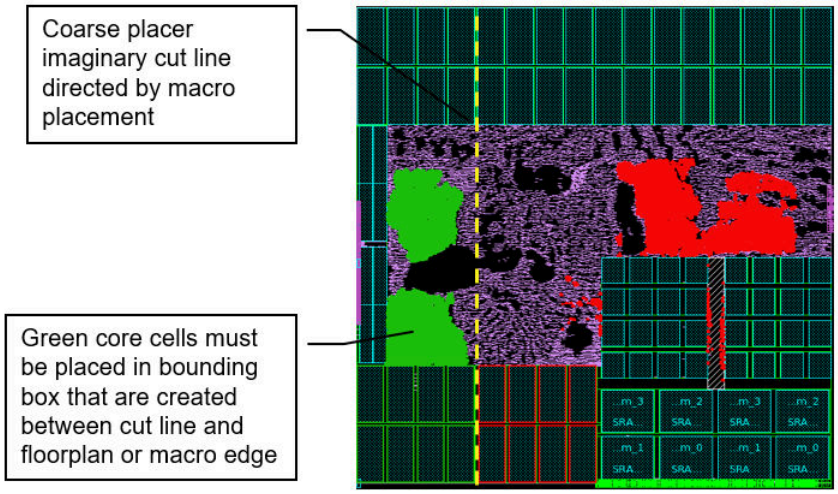
Handling Macros

To handle macros in the DCLS flow for DCLS core separation,

- The DCLS coarse placer produces horizontal or vertical cut lines to separate cores.
- Fixed macro placement affects cut lines that are selected by placer.
- Cut lines produce bounding box against floorplan edge or other blockages to include standard cell placement.

Feedback

- The following method is recommended to check macro placements:
  - Start with the floorplan design by using fixed macro placement.
  - Set up repelling group bounds.
  - Perform coarse placement with the `create_placement` command.
  - View the standard cell core separation and the imaginary cut lines in GUI, as shown in the following figure:



In the DCLS flow, you might have to ignore either a few or all macros, such as SDC constraints, non production macro placement, and so on. To ignore macros while implementing DCLS, set the `dcls_ignore` attribute on macro cell objects, as shown in the following example:

```
define_user_attribute -type boolean -classes cell -name dcls_ignore
set macros [get_cells -physical_context -filter "is_hard_macro==true"]
set_attribute -objects $macros -name dcls_ignore -value true
```

After you set the `dcls_ignore` attribute on the macro cell objects, the macro cells are ignored during

- Coarse placement DCLS separation
- Creation of legalization bounds
- Creation of routing blockages
- Reporting safety status and violations

  Cells are ignored and the attached nets are added to the ignored section of the safety status report.

Feedback

Routing Blockage Setup

The routing blockage setup step in the DCLS flow is performed in the IC Compiler II and Fusion Compiler tools, as shown in Figure 26. The tool automatically creates routing blockages throughout the DCLS flow with the following application option:

```
set_app_options -name report.safety.auto_generate_rgb_routing_blockages
 -value true
```

To manually create the routing blockage, use the `create_repelling_group_bound_shapes` command.

To create a collection of internal and transit nets, use the following commands:

```
set internalNets [get_attribute [get_cells $coreX] -name
 rgb_core_internal_nets]
set transitNets  [get_attribute [get_cells $coreX] -name
 rgb_core_transit_nets]
```

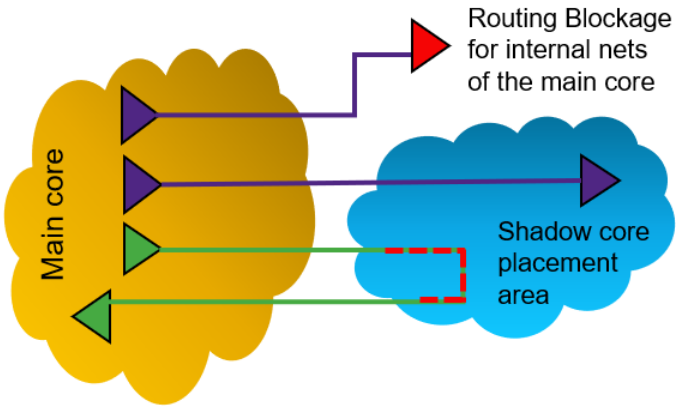
Routing Separation

This section describes the requirements for routing separation.

To ensure routing separation, as shown in Figure 53.

- Purple transit nets are allowed to cross over the shadow core placement area (indicated by purple).
- Green internal nets should be blocked from crossing over the shadow core placement area (indicated by green and dotted red).

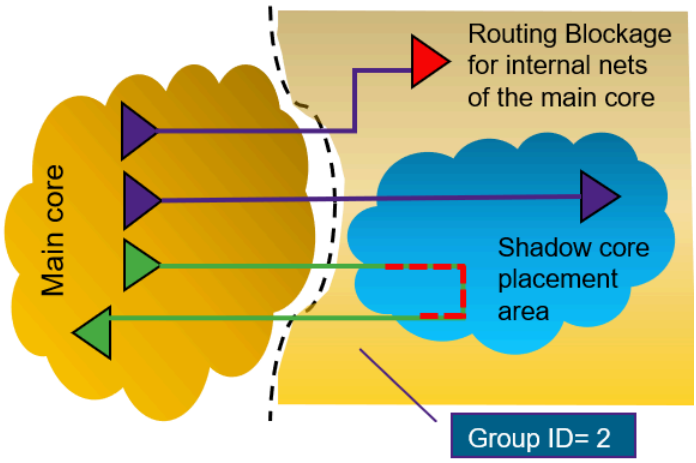
Figure 53 Routing Separation



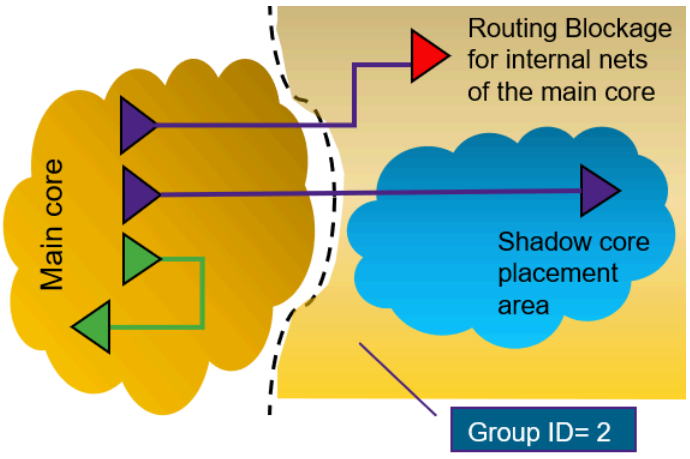
Feedback

To ensure routing blockages are automatically created after legalization,

- Expanded routing blockages are created around each safety core and assigned with an unique ID with the `routing_blockage_group_id` attribute (indicated by yellow square with Group ID).
- The `routing_blockage_group_id` attribute of internal nets of a safety core is set to the routing blockage ID of the other safety core in the bound. The following figure shows the routing blockages covering the shadow core with `routing_blockage_group_id == 2`. The internal nets of the main core are also assigned with `routing_blockage_group_id == 2` to direct the router not to route the internal nets over routing blockages of the shadow core.



- The global and detail routers ensure that the internal nets of the main core are physically separated from the other safety core in the bound. Transit nets are allowed to cross over.



Feedback

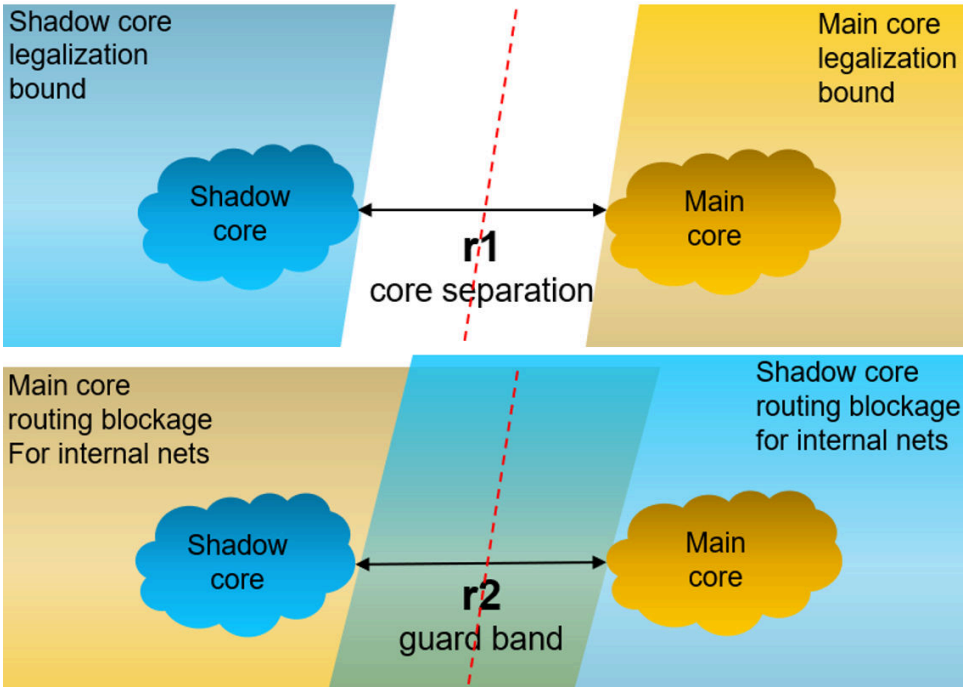
Routing Guard Band

The routing guard band defines and enforces the required internal net routing separation between the cores. By default, a routing guard band equal to the core separation distance is applied to the routing blockages. For example, the following command defines a cell separation distance of r1:

```
create_bound -name RGB1 -repelling diamond -dimension {r1} {main_core
 shadow_core}
```

In Figure 54, the routing guard band r1 is the core separation distance. By default, the r2 routing guard band is equal to the r1 core separation distance.

Figure 54 Routing Guard Band With Core Separation Distance



You can reduce the routing guard band in a few instances and give more space for the router to get access to cell pins that are very close to the legalization bound edge. To reduce the routing guard band, specify the routing guard band while creating manual routing blockage with the following command:

```
create_repelling_group_bound_shapes -repelling_group_bounds RGB1
 [-guard_band {r2}]
```

Feedback

You can also define the routing guard band by setting the `rgb_guard_band` attribute on the repelling group bound. The routing guard band r2 (as shown in Figure 54) must satisfy `0 <= r2 <= r1` if the repelling group bound is of the following types (`-type`):

- Rectangle: The guard band should have the `{x y}` format (`-dimension {width height}`).
- Diamond: The guard band should have the `{r}` format (`-dimension {extent}`)